On the use of Parameterization in the Implementation of Geometry Object-classes

Kjell Kjenstad

Kongsberg Defence & Aerospace a.s,
P.O.Box 1003, N-3601-Kongsberg, Norway.
kjell.kjenstad@kongsberg.com

Telemark University College,
Department of Computer Science and Mathematics,
N-3800-Bø, Norway.
kjell.kjenstad@hit.no

Abstract. Parameterization of object-classes is an attractive approach during implementation of an object-oriented software system. This text discusses different cases of possible parameterization in the implementation of geometry object-classes when used to describe location and extent of geographic features. This approach has obvious advantages and disadvantages and should be used with prudence. When the external conditions are favorable, and when the implementation has been made properly, parameterization is considered to be a powerful tool for implementing high quality software.

Introduction

Parameterization of object-classes is an attractive technique during design and implementation of object-oriented data-models. This is a powerful mechanism by offering an additional level of abstraction compared to normal object-oriented data modeling. However, it adds as well an additional level of complexity compared to traditional object-oriented programming, and hence, it should be used with prudence. This is particularly true for complex object-class models with a strict mathematical foundation.

Geometry is an example this type of object-class model. This text shows some examples and experience on the use of parameterized object-classes, mainly in the design of the data-model for implementation of geometry. Geometry models shown in different international GIS standards [1], [2] do not use parameterized object-classes. However, these are mainly conceptual models, and for these types of models parameterization is less useful.

The main idea behind parameterization

A parameterized object-class is basically an "almost ready" object-class. It is turned into a normal object-class by giving the parameter(s) a specific value. This process is called *instantiation*, and instantiation may be performed either during initial compilation (*explicit instantiation*) or during subsequent use (*implicit instantiation*) of the object-class. Similarly, functions (such as global functions or methods of object-classes) may also be parameterized.

This means that one parameterized object-class or function may form the basis for different instantiated object-classes or functions. This may lead to very compact source code and powerful mechanisms. However, it may also lead to the well-known effect of run-time "code-explosion" which in many cases may easily be overcome.

Explicit instantiation is preferable during the implementation phase of the development project. This leads to immediate discovery of compile errors. According to our experience, explicit instantiation should be maintained even when releasing the software; in particular when producing high quality software is a primary issue. By doing so, the object-classes or functions may not be instantiated in other variants than those having been tested and supplied as explicit instances by the developer of the software.

The realization of the parameterized geometry object-class model

The proposed geometry model has partially been realized during the implementation of geometry object-classes in a project using C++ as the programming language environment, and UML as the modeling language environment. The C++ language offers parameterization in the terms of *templates*. There are two basic types of template mechanisms: *class templates* and *function templates* and both mechanisms may be mixed. In this implementation, class templates have been used as the main mechanism, and function templates have been used as a supplementary mechanism. Subsequent chapters show four different cases where parameterization could be used in connection with the implementation of geometry object-classes.

In the case of the C++ language, the data-type of the parameters may either be object-classes or one of the basic language data types (int, float, etc.). The C++ language offers as well the possibility to have different implementations for different parameters for some selected functions. This possibility permits the hiding of different structures and algorithms behind a common abstract interface. In some circumstances, this may be a very powerful mechanism.

The code has been compiled on different compilers. Support for templates is a rather recent part of the C++ standard, and hence, all compilers do not yet support all these mechanisms with sufficient stability. This means that there has been some struggling against cryptic compiler error messages during the implementation phase.

In addition, the uneven status and quality of compilers have forced this particular project to support both implicit and explicit instantiation, where the selection of the type of instantiation is set as a compile-time flag.

Case 1: Parameterization of co-ordinate value data-types

Geometry object-class models often represent mathematical geometry in the Euclidean space. The space is often metric, and the measurement units represent real world dimensions. The first natural choice for parameterization is the data-type of the co-ordinate value representation. Fig. 1 shows an UML-representation of this case.

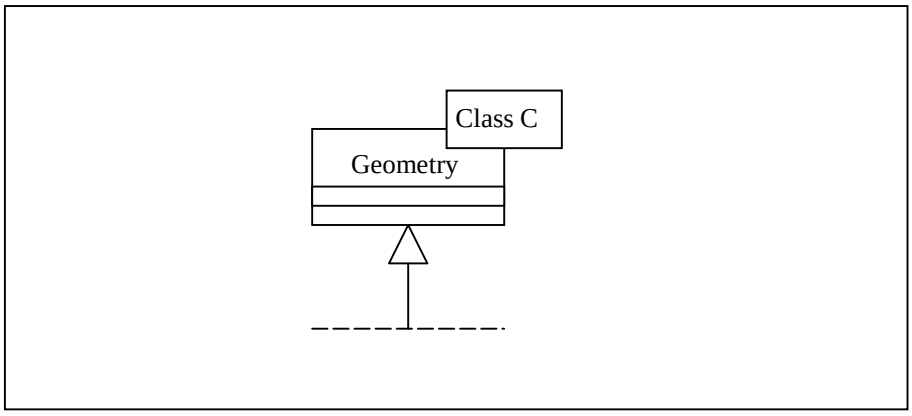

Fig. 1. UML representation of case 1. The geometry object-class parameter *C* represents the data-type of the co-ordinate value of a geometry location.

The different alternatives

In this particular case, there are at least four different alternative choices for the parameter. Each of them is listed in a subsequent chapter.

Alternative 1: Floating point co-ordinate value data-type representation

The traditional representation of such co-ordinate values is by using *floating point* data-types. However, two different types of floating point representation have traditionally been used, i.e. a 4-byte floating point data-type often called *float32*, and a 8-byte floating point data-type often called *float64*. The first one has traditionally been used when saving of memory is the primary issue, while the latter has been used when accurate representation of co-ordinate values in co-ordinate system with a global origin (such as for instance UTM) is the primary issue. This example shows already a potential use of the parameterization of the co-ordinate value data-type. However, these are not the only examples.

Alternative 2: Integer co-ordinate value data-type representation

Sometimes there is a need to represent geometry in a discrete Euclidean space, i.e. a space with integer co-ordinate values. An example is the representation of elevation values in a regular grid-space co-ordinate system. Of cause the real-world co-ordinates are represented by floating point value data-types, while the indexes of the grid-nodes lives in a discrete index-space. Standard geometry object-classes and properties may be given meaningful definitions in this type of discrete space. Hence, supporting parameterization of geometry with integer value data-types is a possible solution. Similar to floating point data-types, two (or more) different types of integer representation could be used, i.e. a 2-byte integer data-type often called *int16*, a 4-byte integer data-type often called *int32*. Even *int8* and *int64* could be used in some cases.

In some GIS systems *int32* have been used as the internal representation of world co-ordinates even if the external representation have been a floating type one. Co-ordinates have been transformed between the two representations using a local origin and a local co-ordinate unit. This type of representation could be called a *fixed-point* value representation of co-ordinates. The reason for this choice is the possibility to save memory with less loss of accuracy in the co-ordinate representation compared to floating point representation.

Alternative 3: Unsigned-integer co-ordinate value data-type representation

Index spaces may also use the so-called unsigned integer representation. This variant allows only for positive integer co-ordinate values, which is often the case in index-spaces. The advantage of this representation is the exclusion of illegal negative co-ordinate values, but the disadvantage is the danger of underflow arithmetic near the co-ordinate axis. In theory this problem is present in all Euclidean spaces, but we are closer to the edge in this unsigned integer case. Unsigned integer values have representation analogue to the signed integer values, i.e. *uint8*, *uint16*, *uint32*, *uint64*.

Alternative 4: Other co-ordinate value data-type representations

Sometimes the previously mentioned co-ordinate value data-type representation using the basic data-types of the programming language is inappropriate. For instance representing geographic co-ordinates in an Euclidean space (i.e. a Simple Cylinder Projection) using floating point value data-type representation (i.e. as radians or decimal degrees) is not ideal when representing the latitude and longitude of geographic location. In our project, a collection of *Angle* object-classes have been used in order to overcome some of the shortcomings of the floating point representation, and this solution has shown to be successful. This means that geometry object-classes have been instantiated using the *Angle* object-class as the parameter for the co-ordinate representation.

The lessons learned

Implementing a dozen of the most common geometry types and instantiated them with the co-ordinate value data-types *int16*, *int32*, *float64* and *Angle* have given some painful lessons, and some of the major lessons learned will be treated in subsequent chapters.

Lesson learned no. 1: The need for formal ValueTypes

When implementing geometry algorithms, there is a need for a set of constants connected to the co-ordinate value data-type arithmetic. These are constants such as *MaximumValue*, *MinimumValue*, *IllegalValue*, *ResolutionValue*, *ZeroValue*, *Unit-Value*, etc. These constants have to be formalized in a way neutral to the parameter-type. In this case, a formal parameterized ValueType object-class instantiated for each co-ordinate value data-type has been implemented. Each of these object-classes has default initial value for each constant, but the application may subsequently change the value for the constant. This is particularly important for the *ResolutionValue* value data-type. This constant is for instance used as the epsilon value in the implementation of the geometry point comparison operator.

However, the latter example pinpoints one of the fundamental weaknesses of the proposed parameterization model, by indicating that the co-ordinate representation parameter does not uniquely characterize the spatial reference system of the geometry. For instance, several geometry spaces using *float64* co-ordinate representation (for instance UTM and geographic co-ordinates represented as radians) would like to use different values for the resolution value. The proposed model of case 1 does not offer this possibility. The only solution is to move the appropriate constants from the geometry co-ordinate value-type level to the spatial reference system level.

Lesson learned no. 2: The need for operators

The formulas involved in geometry algorithms cover a rich set of operators. It is important that these operators have appropriate support for all data-types and object-classes used as co-ordinate value data-type parameter. If not, the instantiation process will end up with compiler errors, and the remedy is to implement the operator or to avoid the operator by a casting operation to one of the basic data-types (often *float64*). The latter solution is rather dirty and should be avoided if possible.

Lesson learned no. 3: The need for conversion functions

The previously mentioned casting operation requires a formal casting function between the different possible parameter data-types. In our case, a parameterized function called *c_to_c* has been implemented for all possible combinations of casting conversions in order to convert a co-ordinate value. In each case, a well-defined convention has to be defined. For instance, what is the meaning of the conversion from *Angle* to *float64*? Does it mean the angle represented as radians or decimal degrees? The latter problem once more pinpoints one of the fundamental weaknesses of the proposed model of case 1. In other words, there are several spatial reference systems for Euclidean geometry represented by the same co-ordinate value data-type.

In addition to the *c_to_c* function, there is a need for a similar parameterized function on the geometry object-class level called *ConvertType* in order to convert the entire geometry from one co-ordinate representation to another. This is necessary in connection with conversions between different spatial reference systems such as for instance between geographic co-ordinates and UTM co-ordinates. In this case, a parameterized function of the parameterized geometry object-class has been made in order to perform the conversion. Originally this conversion has been made as an integrated part of the spatial reference system conversion operation, but for practical reasons, the operation has been divided into two different functions.

Lesson learned no. 4: The problem of co-ordinate representation overflow

The problem of possible *overflow* during arithmetic operations on the co-ordinate value is always a problem and should always be treated seriously during implementation. The problem is even more important when working with normal co-ordinate values that are close to the overflow limit. It should however be mentioned that *overflow* in this context does not necessarily mean digital representation overflow, but also overflow connected to normal modulo arithmetic such as for instance the modulo-two-pi arithmetic for longitude co-ordinates. This overflow limit is the discontinuity where the representation is wrapping from the biggest to the smallest legal co-ordinate value enforced by the respective modulo of the representation of the data-type.

The unsigned integer case has already been mentioned as a possible problem. The *Angle* representation has also a *modulo-two-pi* discontinuity value that is always a problem for geographic longitude values on the global level. On the local level, however, a user-defined discontinuity point may push the discontinuity as far as possible from the area of interest. The geographic latitude value discontinuity on the poles is even a worse problem.

All these examples show the difficulty when working with a parameterized co-ordinate value. All discontinuity point checking have to be given a neutral generic implementation taking advantage of the previously mentioned value-type constants. This may complicate the code, and may represent a possible source of coding error. This fact pinpoints the advantage of, and need for explicit instantiation of geometry object-classes with formal object-class tests for each explicit parameter value.

Lesson learned no. 5: The need for co-ordinate range checking

The previous examples also show the need for a co-ordinate range checking as for instance in the geographic latitude co-ordinate case. The same is also true for most map projections. Defining this co-ordinate range as a user-defined value-type constant is not a good solution because of the same reasons as for the resolution value constant. Instead the range definition should be moved to the spatial reference system level. This solution also implies formal co-ordinate type parameter on spatial reference system object-classes.

Case 2: Parameterization of the Spatial Reference System Relation

The lessons learned during the previous case show that some of the constants on the co-ordinate value-type level should be moved to the spatial reference system level. This means that the geometry function algorithms should access spatial reference system information during run-time. This is in accordance with some of the international standard geometry models [1], [2] which define these types of operations as spatial reference system constraints checking.

An even simpler example of the same problem is linked to geometry operators, such as for instance the *+-operator*. Adding two geometry points for instance, requires that the operator should check the constraint that both operands belongs to the same spatial reference system in order to perform the operation. This type of constraint will work perfectly when the exception is handled in a proper way. The major problem is however linked to the fact that this constraint is check during run-time. This approach is satisfactory for most applications, but for high quality software systems like the ones used in the aerospace industry; compile-time constraint checking is preferred over run-time constraint checking because of effectiveness and security.

Adding the reference system object-class as a class parameter is a well-known object-oriented pattern in order to achieve compile-time checking in the case of singleton reference system object-class. This approach has for instance been used in connection with time reference systems. This approach is *not* directly applicable to geometry because there is often an indefinite number of possible spatial reference systems object-instances for one specific type of spatial reference systems object-class. For instance the *TransverseMercatorProjection* spatial reference system object-class may have an indefinite number of object-instances dependent on the choice of the projection parameters such as central meridian and offset of origin. This means that the *TransverseMercatorProjection* object-class can not be used as a parameter of a geometry object-class in order to obtain constraint checking, because the operator will not distinguish two different object-instances of this particular projection during compile-time.

How can this problem be overcome? The obvious and direct solution of using an object-instance as the parameter for the object-class is not possible in languages like C++ because instances do not exist before run-time. However, an alternative indirect approach using an enumerated parameter is feasible, assuming that there is one instantiation of geometry for spatial reference system no. 1, one instantiation of geometry for spatial reference system no. 2, etc. Fig. 2 shows an UML-representation of this case. By letting this enumerator be the parameter of the geometry, the compiler inhibits operation on geometries belonging to two different spatial reference systems. This means that a geometry object-class is instantiated for each of the spatial reference systems used simultaneously. This could lead to serious code-explosion if not appropriate actions are taken.

This approach has another advantage. Instead of letting each geometry object-instance point to its associated spatial reference system object-instance, as usually

modeled, the spatial reference system pointer could be lifted to the object-class level as an object-class attribute. This is due to the fact that all geometries of the same enumerated object-class parameter by definition points to the same spatial reference system. This saves memory space and simplifies spatial reference system pointer management.

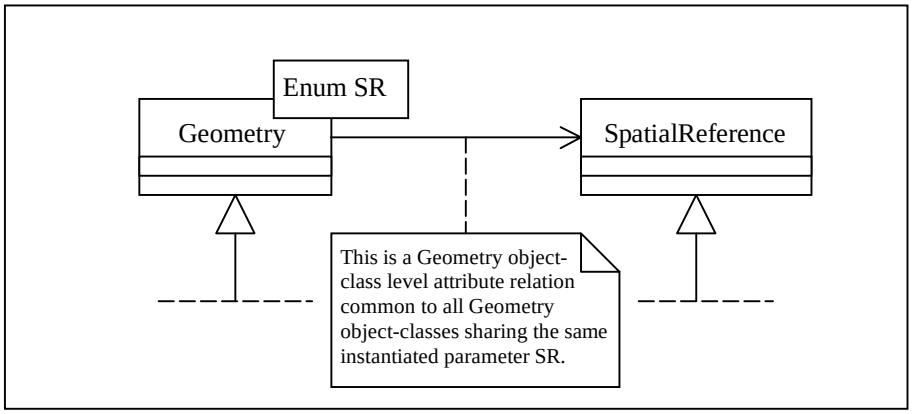

Fig. 2. UML representation of case 2. The geometry object-class parameter *SR* represents an enumerated value, one for each instance of spatial reference system used. The relation between Geometry and SpatialReference object-classes is an object-class level (i.e. static) attribute of the Geometry. This means that all Geometry object-classes instantiated with one specific enumeration value are associated with a common SpatialReference object-instance.

Case 3: Alternative parameterization of the Spatial Reference System Relation

In case no. 2 the spatial reference system relation has been parameterized as an enumerated value instead of an object-class for compile-time constraints checking reasons. However, using the spatial reference system object-class as a parameter may still be used for other reasons.

The fact that functions in a parameterized object-class may have different implementations for different parameter values have already been mentioned in the introduction. This technique could for instance be used in the case of the spatial reference system relation if there is a different geometry arithmetic for different spatial reference systems. Most spatial reference systems represent projected co-ordinate systems, and hence, all of them obey the laws of Euclidean geometry and trigonometry.

However, in the case of a spherical or ellipsoidal geo-centric spatial reference system (often called a geographic co-ordinate system), the geometry and trigonometry arithmetic, and hence the algorithms, are quite different. For instance, calculating the

distance between two points in an Euclidean spatial reference system and the (great-circle) distance between two points in a geodetic spatial reference system is quite different. By having these two (and possibly other) alternative spatial reference object-classes as parameter to the geometry object-classes, these different algorithms could be hidden behind a common object-class signature or interface. Fig. 3 shows an UML-representation of this case.

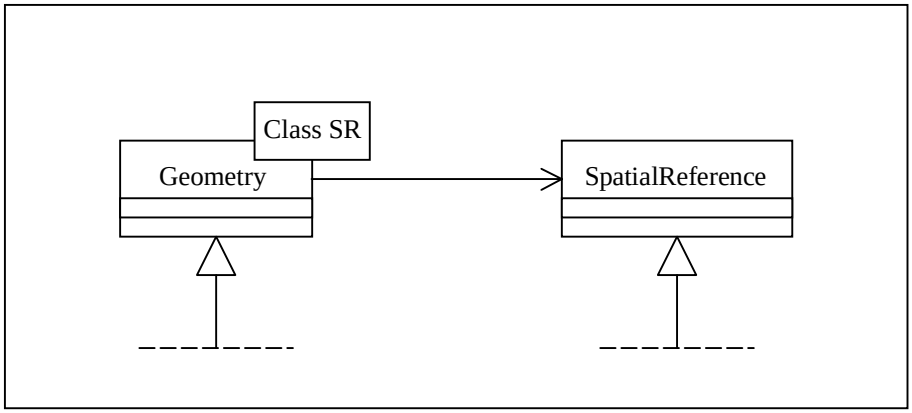

Fig. 3. UML representation of case 3. The Geometry object-class parameter *SR* represents one specific subclass of the spatial reference system object-class.

This case has not been implemented, which means that there is no evidence on the eventual difficulties encountered during this type of approach. It is believed that the major challenge is not linked to software implementation, but rather to the formulation of a common mathematical description of the geometry of these different spatial reference systems. Turned into software language this means that the challenge is to define the common syntax and semantics of the parameterized object-class.

Case 4: Parameterization of the dimensionality of geometry

The same approach as the one used in case no. 3 could in principle be used in hiding implementation differences between geometries with different dimensionality behind a common object-class signature. This is not a common approach in the case of coexistence of 2D and 3D. The reason is that the mathematical formulation of these two types of geometry is often different. This means that it is inconvenient to give these two types of geometry a common object-class signature. Instead the *2D-as-a-subset-of-3D* or *separate-inheritance-trees* are better approaches.

However, different variants of 2,5D type geometry may be given more efficient implementation using object-class parameters. For instance, a 2,5D *polyline* may have the following possible cases for the elevation value:

1. No elevation value (which is a 2D polyline).
2. One common elevation value.
3. One elevation value for each point.
4. Etc.

The code for maintaining the structure and algorithms for all these different cases may be very complex, which in turn may influence performance and quality of the software. A parameterized alternative will split the differences into independent units of code with better performance and quality as a consequence. Fig. 4 shows an UML-representation of this case.

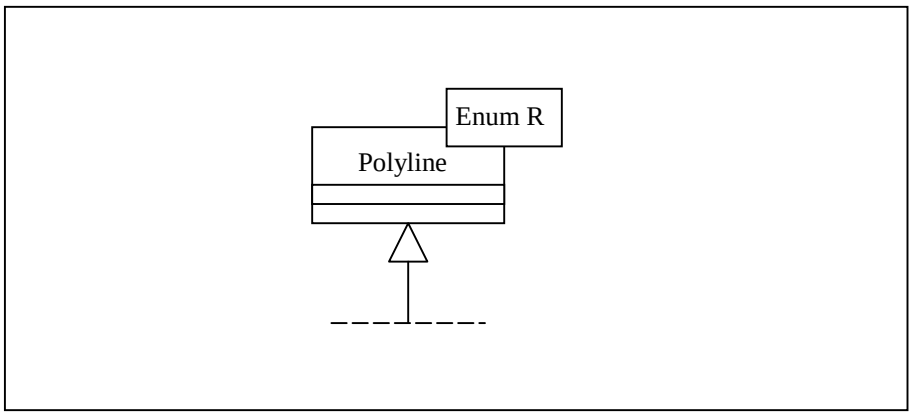

Fig. 4. UML representation of case 4. The Polyline object-class parameter *R* represents different internal representations of the polyline co-ordinate attribute.

In this case the parameter is of an enumerated type, and the number of instantiations is limited. This means that code-explosion is no issue. In addition, there are only a limited number of functions that actually needs separate implementation.

The main advantage of this approach is to support a uniform object-class definition with different optimized implementations:

1. Uniform object-class definition means easier to use for an application programmer.
2. Functions without different implementation means code of higher quality by not having to copy identical code from one object-class to another.
3. Separate implementation in separate function bodies means less erroneous switch-case type of code.

Of cause the same effect could be achieved using normal inheritance. The Polyline object could for instance have several sub-classes, one for each variant. However, the problem is linked to the fact that this type of inheritance is added to the system because implementation reasons. Having in mind that the Polyline object is already a part of a structural inheritance tree, means that this approach leads to mixing of two types of inheritance strategies. This leads to a rather dirty inheritance model. In other

words, parameterization combined with inheritance may be an efficient solution when trying to avoid mixing of structural and implementation inheritance models.

Combining several parameterization models

Cases 1-4 show different possible object-class parameters for geometry object-classes. However, parameters may be combined, meaning that the different parameterization models may also be combined.

The main disadvantage of a combined model is that the number of possible instantiation increases rapidly which in turn may cause severe code explosion. However, in practice, the instantiation of one parameter is often coupled with the instantiation of one specific or a few values of another parameter. This fact will reduce the number of instantiated object-classes tremendously.

For instance, geometry belonging to a certain type of spatial reference system according to case no. 3 will often be linked to a well-defined co-ordinate value-type according to case no. 1. This geometry will furthermore only be available in a limited number of spatial reference system instances according to case no. 2. Finally, the choice of geometry representation according to case no. 4 will be well defined, and often limited to one specific variant.

Parameterized inheritance models for geometry

Parameterized object-classes may inherit, and in principles there are 3 different cases (fig. 5 a-c), dependent on the type of inheritance:

1. Fig. 5a shows an inheritance relation of *one* parameterized object-class inheriting *one* other parameterized object-class.
2. Fig. 5b shows an inheritance relation of *many* object-classes (one for each parameter value) inheriting *one* other object-class.
3. Fig. 5c shows *one* object-class inheriting *many* object-classes (one for each parameter value), i.e. multiple inheritance.

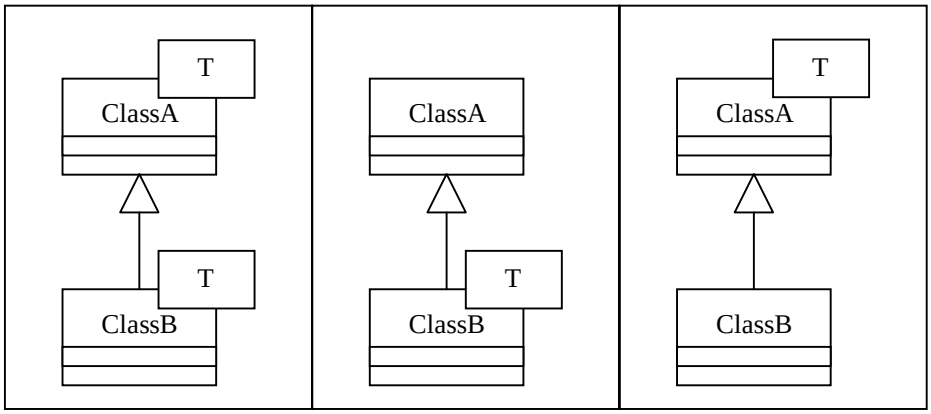

Fig. 5. (a-c) The three inheritance models for parameterized object-classes.

The first and second cases are both useful, either stand-alone or mixed while multiple inheritance should be avoided during implementation except when needed for some specific reason.

Geometry is an example of rather complex object-class inheritance structures. Multiple inheritance like the one shown in fig. 5c is of no use because the mechanism that we want to parameterize is often on the lowermost concrete level in the inheritance tree.

Furthermore an attempt to parameterize only the lowermost concrete level in the inheritance tree will often fail because it is vice to push attribute and methods as far up in the inheritance tree as possible. The immediate effect of this fact is that an inheritance structure of the one shown in fig. 5a is needed. Repeating this inheritance model for all geometry object-classes will produce an inheritance tree with a number of instantiated geometry models representing separate geometry object-class models. The uppermost part of such an inheritance tree is shown in fig. 6.

However, for practical implementation reasons, it is often useful to remove the parameters on one or more of the uppermost superclasses of the inheritance tree. For instance, in the case of geometry, it is useful to let the parameterized *Geometry* object-class inherits from a non-parameterized *Geometries* superclass according to the model proposed in fig 6.

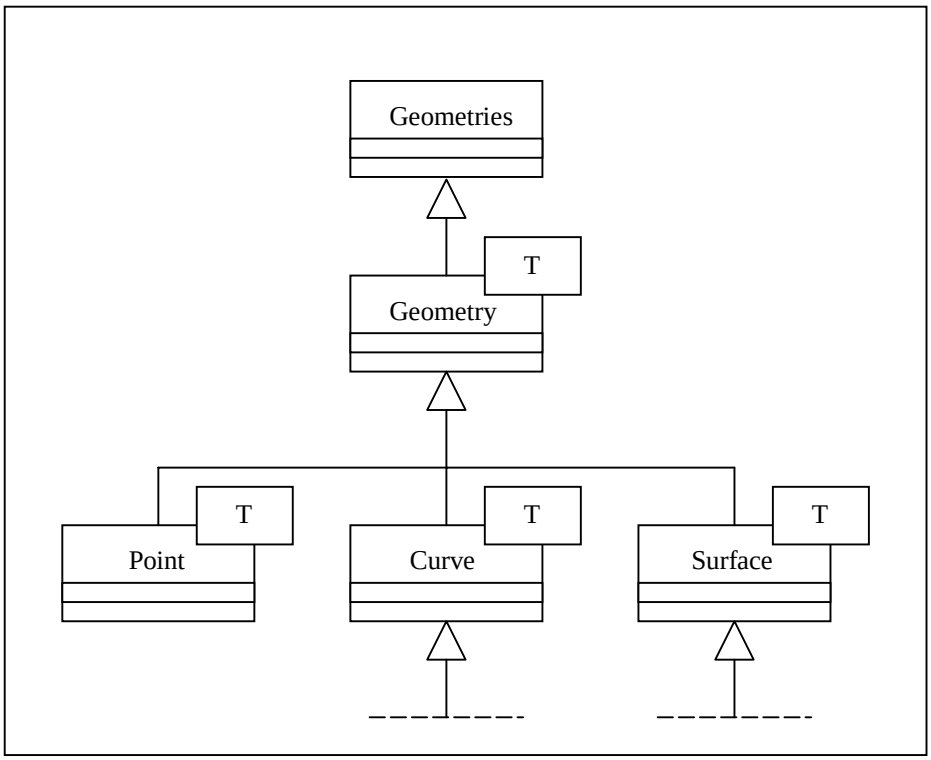

Fig. 6. A proposed inheritance model for geometries

Conclusions

The discussion of the four different proposed cases show advantages and disadvantages of possible parameterization in the implementation of geometry object-classes when used to describe location and extent of geographic features. Some cases of severe disadvantages means that parameterization should be used with prudence. However, when the external conditions are favorable, and when the implementation has been made properly, parameterization is a powerful tool for implementing high quality software.

References

1. The OpenGIS™ Abstract Specification. Topic 1: Feature Geometry. Version 4, *Open GIS Consortium.* 1999.
2. ISO DIS-19107, Geographic information - Spatial schema. *ISO/TC 211, Geographic information/Geomatics*. ISO, 2001.