A Method for the Construction of Robust Class Hierarchies

Arne Frick* Rainer Neumann* Wolf Zimmermann*

18th June 1996

Abstract

We introduce a method for the construction of robust class hierarchies. The base of this construction is a rigorous definition of inheritance. There are four possible inheritance relationships. At least two of them turned out to be important in practice. According to this for the first time, conditions for the correct use of polymorphism are given. We show how inheritance hierarchies can be constructed in a systematic and rigorous way.

1 Introduction

The paradigm of object-oriented programming has provided a large number of methods for the design and the implementation of applications, most of which do not provide a solid and formal basis for correctness proofs. Though the methods [1, 5] help to develop a system, there is no support to assert or prove certain properties. One reason for this is, that basic concepts like inheritance, polymorphism and genericity are not determined in a formal manner. This makes sense, if one is saling design tools, because this way some kind of language independence is easier to reach. But even formal methods like OBJECT-Z also fail because of this lack of formal definitions. We therefore have to recognize, that after more than one decade of object-oriented modeling basic concepts are not really understood or described in a mathematical way. What is missing, is a method for the correct development of object-oriented systems.

Although the requirements needed to provide correct systems or class libraries were shown in [2], there is still need for a method for the systematic construction of such systems. The task now is to construct a method where a correct development process leads to a correct system. MEYERs *Design by Contract* [4] is one step towards this goal. Linkage between objects is described by pre–/postconditions and invariants. But still the use of inheritance, polymorphism or genericity may lead to incorrect systems – the global system correctness may be destroyed by locally correct classes.

This paper now describes, how object-oriented systems may be designed so that violations of the global system correctness will be detected at compilation time. This method is mentioned not to replace but to support existing methods by given formal descriptions of basic concepts.

The remainder of this paper is organized as follows: Section 2 defines basic notions and repeats important definitions given in [2]. The principles of the construction of robust class hierarchies are discussed in section 3, which results in a concrete method. Section 4 summarizes the results and gives an outlook on future work. The appendix gives an example application of the method describing the constructing of a hierarchy of geometrical classes.

The introduced concepts are not language dependent, but examples are given in a SATHER[1] like notation.

2 Basics

We assume classes to be completely specified using invariants as well as pre– and postconditions. Furthermore we assume attributes to be encapsulated, which means that they are invisible at the external interface of

*University of Karlsruhe, Vincenz-Prießnitz-Str. 3, Postfach 69 80, 76128 Karlsruhe.
EMail: {frick,rneumann,zimmer}@ipd.info.uni-karlsruhe.de

[1] Sather is an imperative object-oriented programming languages. Information can be obtained from URL: http://i44www.info.uni-karlsruhe.de/sather.

a class. The set of methods of a class A can be divided into a set of *kernel* methods and a set of *derived* methods. Kernel methods of A are *abstract* methods, which are not implemented within class A, but must be implemented in concrete subclasses. Derived methods are methods which can be implemented within the (abstract) class A. [2] showed, that derived methods are inefficient in many cases, so that there is often a need to re-implement them in some subclasses. The reason for that is, that abstract classes often cannot take advantage of implementation details. Invariants usually describe the possible states of objects. They may also contain parts of the functional behavior of kernel methods. Pre– and postconditions are used to specify the functional behavior of a single method.

2.1 Correctness of Object-Oriented Programs

A class A is said to be *locally correct* iff

(i) after creation of an instance of class A, the invariant Inv_A holds and

(ii) after calling a method m on an instance of class A, the invariant Inv_A and the postcondition $Post_{m,A}$ of m hold, provided that Inv_A and the precondition $Pre_{m,A}$ were satisfied when calling m.

Any instance of class A must satisfy Inv_A at any time except possibly during execution of a method of A. A class library is called *correct*, if each class within the library is locally correct.

The techniques for the construction of reliable and reusable class libraries described in [2] are based on this notion of correctness. The next section will describe the different inheritance relationships which have been analyzed within that paper, and we will point out their importance for the construction of robust libraries and applications.

2.2 Inheritance Relationships

On the basis on class specifications using pre– and postconditions there are only four different ways to define inheritance relationships between classes with regard to a future polymorphic use (see 1). These are *conformance*, *specialization*, *covariance* and *contravariance*. [2] showed that proving correctness can only take advantage of *conformance* and *specialization*. Those two inheritance relationships can be formalized as follows.

Given two classes A and B. We say B is *conform* to A iff

(i) $Inv_B \Rightarrow Inv_A$

(ii) For each method m of A: $Pre_{m,A} \Rightarrow Pre_{m,B}$ and $Post_{m,B} \Rightarrow Post_{m,A}$.

Class B is said to be *more specific* than class A (B is a Specialization of A) iff

(i) $Inv_B \Rightarrow Inv_A$

(ii) For each method m of A: $Pre_{m,B} \Rightarrow Pre_{m,A}$ and $Post_{m,A} \Rightarrow Post_{m,B}$.

The polymorphic use of classes allows, that the concrete class to which an instance belongs is only known at run time. It is easy to see that correctness of such a system can only be guaranteed, if variables of a polymorph type A can only be assigned instances of conform subclasses of A. The use of specializations in a polymorphic sense may lead to correctness violations and therefore to applications which are not robust.

2.3 Generic Classes

A parameterized class $A(T_1, \ldots, T_n)$ is called *generic class*. Applications using generic classes must instantiate the *generic parameters* with concrete classes $U_1, \ldots, U_n$. This can be done by writing $A(U_1, \ldots, U_n)$, which means that T_i is instantiated with class U_i[2]. Each generic instance $A(U_1, \ldots, U_n)$ is a class. Instances for the generic parameters can be restricted using *type bounds*. In this case we write $A(T_1 < S_1, \ldots, T_k < S_k)$. Any class T_i which functions as generic instantiation to parameter T_k must then be conform to S_i in order to be valid.

[2] Generic parameters can be generic instantiations themselves.

$$\begin{tikzcd}
Inv_A & Pre_{m,A} \arrow[r] \arrow[d] & Post_{m,A} \\
Inv_B \arrow[u] & Pre_{m,B} \arrow[r] & Post_{m,B} \arrow[u]
\end{tikzcd}$$

a. conform

$$\begin{tikzcd}
Inv_A & Pre_{m,A} \arrow[r] & Post_{m,A} \arrow[d] \\
Inv_B \arrow[u] & Pre_{m,B} \arrow[r] \arrow[u] & Post_{m,B}
\end{tikzcd}$$

b. more specific

$$\begin{tikzcd}
Inv_A & Pre_{m,A} \arrow[r] & Post_{m,A} \\
Inv_B \arrow[u] & Pre_{m,B} \arrow[r] \arrow[u] & Post_{m,B} \arrow[u]
\end{tikzcd}$$

c. covariant

$$\begin{tikzcd}
Inv_A & Pre_{m,A} \arrow[r] \arrow[d] & Post_{m,A} \arrow[d] \\
Inv_B \arrow[u] & Pre_{m,B} \arrow[r] & Post_{m,B}
\end{tikzcd}$$

d. contravariant

Figure 1: Possible inheritance relationships. The arrows symbolize logical implications.

3 Construction of Robust Class Hierarchies

This section will discuss the conditions necessary for the use of conformance and specialization between two classes within a hierarchy. First we extend the results of [2] by analyzing inheritance relationships between generic classes. In 3.2 and 3.3 criteria for the construction of specialization or conformance hierarchies are pointed out. The main result of this section is, that restrictions of generic arguments with type bounds lead to more specific generic classes, whereas adding one or more generic parameter leads to a conform subclass restricting the set of possible instances. Finally for the first time we derive a method for the systematic construction of class hierarchies.

3.1 Inheritance Relationships between Generic Classes

If a generic class B inherits from another generic class A, then A is instantiated by B, but still may use generic parameters of class B. Such an inheritance is *valid*, iff each valid instantiation of the parameters of B is a valid instantiation of the parameters of A. Then A is again a generic class within the generic parameters of B.

To transfer the notions of specialization and conformance to generic classes we inherit from a class $A(T_1 < S_1, \dots, T_k < S_k)$ whichs parameters are a subset of the parameters of the generic class B. We say that the generic class $B(T_1 < S'_1, \dots, T_l < S'_l)$ *conforms to*(*is more specific than*) generic class $A(T_1 < S_1, \dots, T_k < S_k)$, if (i) each valid instance of B is a valid instance of A, and (ii) each valid instance of B is a conform subclass (specialization) of the complying instance of A.

3.2 Specialization Hierarchies

Though specializations do not allow polymorphic use, they do make the reuse of existing code possible.

Example 1 The hierarchy of geometric classes given in the appendix is a specialization hierarchy. It is easy to see, that

- some methods first make sense in specialized subclasses. The method area that computes the area of a geometric Object makes sense only for polygons or other *closed* geometric Objects.

- specializations allow more efficient implementations. The classes TRIANGLE and SQUARE allow very efficient implementations of the area method.

We assume a (possible generic) class A with invariant Inv_A and a predicate P defined on instances of A is given and also a class B with at least the same methods as class A but a stronger invariant $Inv_B \equiv Inv_A \wedge \mathcal{P}$.

If we now look at a class as the set of its instances, then B is a subset of A. It is obvious that $Inv_B \Rightarrow Inv_A$. B can now be in any of the inheritance relationships described above. We need to look at the pre– and postconditions of the methods within a concrete context.

Let $m(x_1 : R_1, \ldots, x_q : R_q) : R$ be such a method. Assumed that $m(t_1, \ldots, t_q)$ does not modify the state of the object. Now let o be an instance of B, which fulfills $Pre_{m,A}(t_1, \ldots, t_q)$. After calling $o.m(t_1, \ldots, t_q)$, $Post_{m,A}$ is valid, for o is also an instance of A. Therefore, we can treat $Pre_{m,A} \equiv Pre_{m,B}$ and $Post_{m,A} \equiv Post_{m,B}$ as equivalent without changing the functionality of m.

Now assume that $o.m(t_1, \ldots, t_q)$ modifies the state of $o \in A$. Therefore $Pre_{m,A}$ holds. Let o also be within B. As the following example shows, Inv_B must not hold after execution of $o.m(t_1, \ldots, t_q)$ o, even if $Pre_{m,A} \wedge Inv_B$ hold before calling the method:

Example 2 some (plane) geometric data structures, e.g. the classes POLYGON and TRIANGLE. What kind of inheritance relationship exists between those two classes if any? Assumed that POLYGON provides a method for the insertion of an additional point into the set of defining points. Now TRIANGLE cannot be a conform subclass of POLYGON – the invariant of TRIANGLE holds iff the defining set of points consists of exactly three points; insertion of an additional point is not allowed. The following implications hold:

$$Inv_{\mathsf{TRIANGLE}} \Rightarrow Inv_{\mathsf{POLYGON}} \wedge Pre_{\mathsf{insert\_point,TRIANGLE}} \Rightarrow Pre_{\mathsf{insert\_point,Polygon}},$$

This shows, that class TRIANGLE is a specialization of POLYGON. In order to prove the correctness one has to show that after consistently replacing every usage of class POLYGON by TRIANGLE, correctness can still be proven [2] using the (stronger) precondition $Pre_{\mathsf{insert_point,TRIANGLE}}$ and the (weaker) postcondition $Post_{\mathsf{insert_point,POLYGON}}$.

$Pre_{m,B}$ must be defined so that after calling $o.m(t_1, \ldots, t_q)$ Inv_B holds. Usually only $Pre_{m,B} \Rightarrow Pre_{m,A}$ holds. Therefore B can only be either covariant to or more specific than A. On the other hand, we know $o \in A$. Therefore $Post_{m,A}$ holds and we have $Post_{m,B} \Leftrightarrow Post_{m,A}$.

Theorem 1 *Let A be a (generic) class with an invariant Inv_A and let P be a predicate defined over the instances of A. Given a class B which contains at least the same methods as A, where instances of B fulfill $Inv_A \wedge P$. Then B is more specific than and covariant to A.* □

Remark 2 This theorem is the basis of covariant inheritance mechanisms, e.g. such as those in EIFFEL. ■

Theorem 1 is the basis for a systematic construction of specialization and/or covariance hierarchies.

Given a (generic) class A and predicates $P_1, \ldots, P_m$ on the instances of this class. We call the set of all conjunctions

$$H(P_1, \ldots, P_m) = \left\{ \bigwedge_{i \in I} P_i : I \subseteq \{1, \ldots, m\} \right\}$$

the *conjunctive closure* of $P_1, \ldots, P_m$ with $\bigwedge_{i \in \emptyset} P_i = true$. Further given two predicates $Q_1, Q_2 \in H(P_1, \ldots, P_m)$ with $Q_1 \Rightarrow Q_2$. Let Q_1_A (Q_2_A) be the class of instances of A for which Q_1 (Q_2) holds. We now know that Q_1_A and Q_2_A are specializations of A. We also know that Q_1_A is also a specialization of Q_2_A. The implication relationship between the two predicates obviously induces a specialization relationship between the corresponding classes. As already shown in 2, specialization relationships are transitive. Therefore we can define a specialization hierarchy using the implications $H(P_1, \ldots, P_m)$. It may be that two elements $Q_1, Q_2 \in H(P_1, \ldots, P_m)$ are *equivalent*, which means that $Q_1 \Leftrightarrow Q_2$. In this case we do not need to introduce two classes that are specializations of each other, because we know from section 2, that invariants of the classes and pre– and postconditions of the methods are equivalent. Therefore we look at equivalence classes of $H(P_1,\ldots,P_m)$. If $\overline{Q}$ is the equivalence class of $Q \in H(P_1,\ldots,P_m)$ or a representative of this class, then $\overline{H}(P_1,\ldots,P_m) := \{\overline{Q} : Q \in H(P_1,\ldots,P_m)\}$ defines a partial order relationship $\Rightarrow$. Knowing that $\overline{P_1 \wedge \cdots \wedge P_m} \Rightarrow \overline{Q}$ and $\overline{Q} \rightarrow \mathit{true}$ for every $Q \in H(P_1,\ldots,P_m)$, it is obvious to see that $(\overline{H}(P_1,\ldots,P_m), \Rightarrow, \overline{P_1 \wedge \cdots \wedge P_m}, \mathit{true})$ is a lattice. We can now say that Q_1_A is a specialization and covariant subclass of Q_2_A, iff $Q_1 \Rightarrow Q_2$ and $\overline{Q_1} \neq \overline{Q_2}$.

Theorem 3 *Let A be a (generic) class and $P_1,\ldots,P_m$ predicates over A. Then the lattice induces*

$$(\overline{H}(P_1,\ldots,P_m), \Rightarrow, \overline{P_1 \wedge \cdots \wedge P_m}, \mathit{true})$$

a specialization and covariance hierarchy

$$(\mathcal{A}, \textit{more specific than/covariant to}, \bot, \top)$$

with $\mathcal{A} = \{Q\_A : \overline{Q} \in \overline{H}(P_1,\ldots,P_m)\}$, $\bot = P_1\_\ldots\_P_m\_A$ and $\top = A$. □

We now can derive a design technique for class hierarchies from this theorem:

1. Given a class A, that defines properties of instances needed for a concrete application or which must be provided by a library. These properties are defined using predicates $P_1,\ldots,P_m$.
2. Compute the conjunctive closure $H(P_1,\ldots,P_k)$ and the corresponding equivalence classes $\overline{H}(P_1,\ldots,P_m)$.
3. For each equivalence class $\overline{Q}$ create a class Q_A, where the instances hold the invariant $\mathit{Inv}_A \wedge Q$.
4. Compute the graph using the the detected classes as vertices and the implication relationships as edges. Do not insert transitive edges.
5. If $Q_1 \Rightarrow Q_2$ is contained within this graph, define class Q_1_A as a specialization of class Q_2_A. Inherit the methods of Q_2_A and replace the preconditions of the modifying methods so that the stronger invariant $\mathit{Inv}_A \wedge \mathit{Inv}_{Q_2}$ always holds. Whenever a precondition had to be modified, the corresponding method needs to be reimplemented.

3.3 Hierarchies of Conform Subclasses

We now look at hierarchies of conform subclasses. The non-generic case is clearly discussed in 2.2 and [2]. What remains is a discussion of the generic case. Therefore let $A(T_1,\ldots,T_k)$ be a generic class, where the allowed instances of the generic parameters of A hold for a predicate P_1. Instead of giving new requirements for the instances of A (see 3.2), we analyze the inheritance relationship for the case that B only defines new restrictions on the generic parameters. Let $B(T_1,\ldots,T_k)$ be a generic class, where the allowed instances of the generic parameters of B hold the predicate P_2. We further assume that $P_2 \Rightarrow P_1$. In particular B behaves like A within the set of valid instances. This means that each method $m \in B$ preserves Inv_A and guarantees the postcondition $\mathit{Post}_{m,A}$ provided that $\mathit{Pre}_{m,A}$ holds when calling the method. As already shown this can only be the case if each valid instantiation of the generic class B conforms to its corresponding instantiation of class A. This leads to the following

Theorem 4 *Let $A(T_1,\ldots,T_k)$ be a generic class and $P_1(T_1,\ldots,T_k)$ a predicate, that reduces the set of valid generic parameters. Let $B(T_1,\ldots,T_k)$ be a generic class, where its instantiations are also instantiations of class $A(T_1,\ldots,T_k)$, but hold the stronger predicate $P_2(T_1,\ldots,T_k) \Rightarrow P_1(T_1,\ldots,T_k)$. Then B conforms to A.* □

Additional restrictions of the generic parameters may allow the introduction of new methods, which can make use of the additional type information (see example 3). Furthermore this information often allows more efficient implementations of certain methods.

Example 3 Given a class SET(T) which implements the corresponding datastructure.

```
class ORDERED(T) is
  isLessThan(s:T):Boolean is
    -- Pre: T has an order relation '<'
    return self<s;
  end;
end;
class ORDERED_SET(T<Ordered(T)) is
  subtype of Set(T);
  ⋮
  minimum:T is
    min:=choose;  -- random selection of an element
    loop temp:=elts!; if temp<min then min:=temp; end; end;
  end;
  ⋮
end;
```

- The maximum or minimum of the set can be computed if a total order on the type of elements exists. This may be achieved by restricting the generic parameter T:
- The complement of a set can only be computed if the universe of the underlying element is finite.
- Very efficient implementations of sets can be made if bit vectors may be used. This is only possible if the underlying type has a finite (or better tiny) universe, e.g. characters.

We define the conjunctive closure $H(P_1, \ldots, P_m)$ and the equivalence classes $\overline{H}(P_1, \ldots, P_m)$ as an analogy to the previous section. Those equivalence classes again a lattice again with the top element *true*, which means that there are no restrictions. The equivalence class $Q\_A(T_1, \ldots, T_k)$ which only allows instantiations of $(T_1, \ldots, T_k)$ that fullfill Q is shown to conform to class $A(T_1, \ldots, T_k)$. In detail $Q\_A(T_1, \ldots, T_k)$ conforms to each class $R\_A(T_1, \ldots, T_k)$ that only allows instantiations by $(T_1, \ldots, T_k)$ that fullfill R, if $Q \Rightarrow R$ holds.

Theorem 5 *Let $A(T_1, \ldots, T_k)$ be a generic class and $P_1, \ldots, P_m$ predicates over the instantiations of $(T_1, \ldots, T_k)$. Then the lattice $(\overline{H}(P_1, \ldots, P_m), \Rightarrow, \overline{P_1 \wedge \cdots \wedge P_m}, true)$ induces a lattice $(\mathcal{A}, conform\ to, \perp, \top)$ of conforming subclasses with*

$$\mathcal{A} = \{Q\_A : \overline{Q} \in \overline{H}(P_1, \ldots, P_m)\},$$

where $\perp = P_1\_\ldots\_P_m\_A$ und $\top = A$. □

This theorem gives a design technique which is an analogy to the one given in 3.2.

4 Conclusion

In an earlier paper [2], we showed that there are different kinds of inheritance relationships between classes only one of which can be said to be robust within polymorphic use: Only conform subclasses are allowed to be used polymorphically without introducing new errors. On the other hand it was pointed out that specialization is at least as important in practice. This produces several questions: How can we distinguish between the inheritance relationships? How can we decide which relationship to use? How can class hierarchies be created?

This paper provides the answers to these questions. We developed techniques for the design of class hierarchies. We showed that specialization hierarchies arise out of the definition of predicates over the instances of a given class A. These predicates can be partially ordered using the implication relationship. It was also shown that this partial order induces a specialization hierarchy of the corresponding classes, where the root class of the hierarchy is the initial class A. Theorem 1 shows that specialization hierarchies can be seen as covariant hierarchies. It is still an open question whether these two kinds of inheritance relationships appear seperately. The observation of algebraic datastructures may help to answer this question.

A quite similar procedure led to a lattice of conform subclasses. In that case, we examined generic classes and predicates over the instantiations of their generic parameters. It was shown that every valid instantiation of such a restriction leads to a conform generic subclass which means that all instantiations of the restricted generic class are conform subclasses of the initial generic class (without the additional restrictions). We pointed out the use of such restrictions using a well known example.

The design technique which we developed only deals with the conjunction of predicates. The disjunction of those predicates may lead to new superclasses. This observation provides a large potential for the creation of class hierarchies.

References

[1] G. Booch. *Object-Oriented Design with Applications.* Benjamin/Cummings Publishing Company, Reading, Mass., 1991.

[2] Arne K. Frick, Walter Zimmer, and Wolf Zimmermann. On the Design of Reliable Libraries. In *Proceedings of TOOLS'95*, p. 13–23. Prentice Hall, 1995. ftp://i44ftp.info.uni-karlsruhe.de/pub/papers/frick/tools95.ps.gz.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Software Components.* Addison-Wesley, 1995.

[4] Bertrand Meyer. Applying "Design by Contract". *IEEE Computer (Special Issue on Inheritance & Classification)*, 25(10):40–52, October 1992.

[5] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modelling and Design.* Prentice Hall, 1991.

A Geometric Objects

We define the points within a two dimensional cartesian coordinate system using floating point arithmetic as the basis for our objects.

Starting at this point we define classes like POINT, LINE, MULTI_LINE, POLYGON, RECTANGLE and TRIANGLE. We can split these classes into *basic classes* such as POINT or LINE and the more complex classes such as MULTI_LINE, POLYGON, RECTANGLE and TRIANGLE.

We leave the basic classes aside and now define predicates starting from MULTI_LINEs. We can distinguish between simple and other MULTI_LINEs. Simple means, that the defining lines do not intersect. Furthermore we can distinguish between closed and open MULTI_LINEs. The closed ones are POLYGONs. Figure 2 shows the partial order given by the implication realtions between those predicates. The next figure (3) shows the resulting specialization hierarchy induced by this partial order.

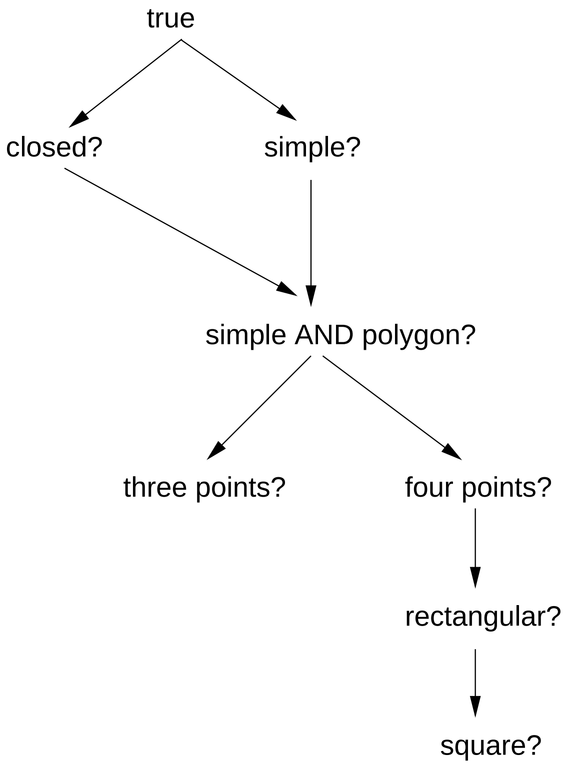

Figure 2: Implication relationships between the properties of geometric objects

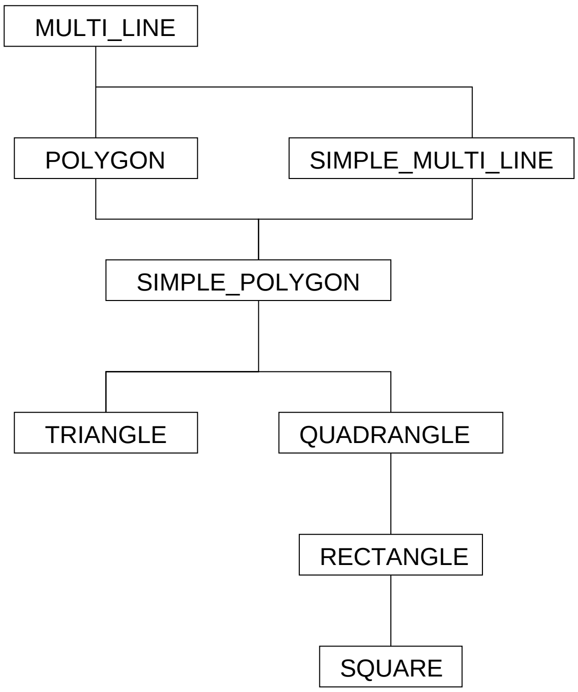

Figure 3: Induced specialization hierarchy