# The Inheritance of State Models

Donald G. Firesmith

Donald G. Firesmith is a senior member of the technical staff at Knowledge Systems Corporation. He may be contacted at dfiresmith@ksccary.com or 73664.3513@compuserve.com.

## TYPES OF INHERITANCE

When it comes to ensuring the consistency of state models across inheritance structures, the most important types of inheritance are specialization inheritance and implementation inheritance. *Specialization inheritance* is any inheritance that implements the a-kind-of relationship so that the child is a specialization of its more general parent(s). *Implementation inheritance* is any inheritance whereby a new implementation is incrementally defined in terms of existing implementations. Thus, specialization inheritance is used to capture taxonomies, whereas implementation inheritance is used to permit code reuse. Specialization inheritance ensures the subtyping (i.e., the conformance of interfaces whereby the protocol of the child is a superset of the protocol of the parent) and polymorphic substitutability (i.e., an instance of the child can be used anywhere that an instance of the parent can be used). Implementation inheritance usually does not imply any semantic relationship between child and parent, any conformity of their protocols, or polymorphic substitutability.

Because of these limitations, implementation inheritance has a relatively bad reputation in the object community and usually should be replaced with either aggregation or delegation. Because the state model of the child need not be related to the state model of the parent when implementation inheritance is used, most of the following guidelines assume specialization inheritance.

## THE GUIDELINES

G-01. The state model of a child need not have anything to do with the state models of its parents.

> **Rationale:** Implementation inheritance may have been used in such a manner that the child is not semantically related to the parent. A reductio ad absurdum existence proof of this guideline would be that private inheritance* could be used in such a manner that all inherited operations are overridden with null operations.

G-02. The state model of each specialization must conform to the state models of its generalizations.

> **Rationale:** Because any instance of the specialization is indirectly an instance of the associated generalizations, its behavior must conform to the state machines of the generalization. This guideline thus recognizes the polymorphic substitution of specializations for generalizations. This guideline can be interpreted in terms of the principles of design by contract in light of the following object-oriented definitions: (1) a *state* is an equivalence class of property values that determines the qualitative behavior of the object, (2) a state *transition* is therefore a change in property values that results in a change of equivalence class, (3) a *trigger* (i.e., the immediate cause of a state transition) is an operation that modifies the value of one or more state properties so that the equivalence class changes, and (4) a *guard* on a transition is a precondition on a trigger operation. The remaining guidelines define what is meant by the word *conformance* in this guideline.

G-03. Each specialization inherits the state models (if any) of their generalizations.

> **Rationale:** The specialization inherits from its generalizations the state properties and the operations that both change these properties and depend on the values of these properties.

G-04. The state model of each specialization may add state machines to any state models inherited from its generalizations.

> **Rationale:** The specialization may add additional state properties and operations that depend on the values of these properties. These state properties and operations may implement new state machines as well as modify inherited state machines.

G-05. The state model of each specialization must be either the union of the state models of its generalizations or a consistent

*In private inheritance, all inherited features become private regardless of their visibility in the parent. The use of private inheritance supports information hiding and maintainability of inheritance structures because it allows developers to change ancestors without impacting on their descendants. However, it must be used with great care because it may prevent the child from conforming to its parents by hiding their visible features, thus violating subtyping, specialization inheritance, and polymorphic substitutability.

specialization that extends the state models of its generalizations.

**Rationale:** The specialization inherits from its generalizations the state properties and operations that modify the values of theses properties. The specialization may also add additional state properties and operations that modify the values of theses properties.

G-06. Each substate must be a specialization of its superstates (i.e., if the object is in a substate, it is also indirectly in all superstates of the substate).

**Rationale:** States are defined as equivalence classes of property values that produce the same overall behavior. The equivalence class of a substate is a subset of the intersection of the equivalence classes of its superstates.

**Example:** The "functioning" of an on-off device class may have the substates "On" and "Off," both of which represent valid ways of functioning (Fig. 1).

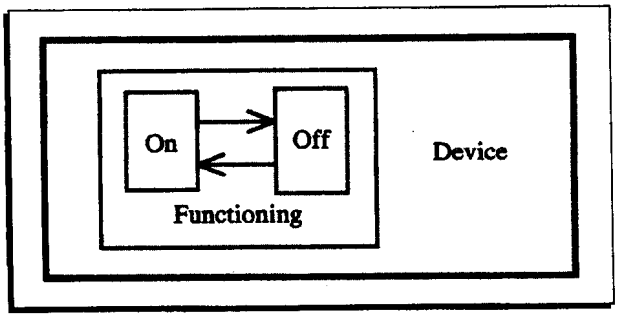


Figure 1.

G-07. The state model of each specialization may add states and substates to those inherited from the state models of its generalizations and delete states inherited from the state models of its generalizations.

**Rationale:** A specialization may add states and substates by adding additional state properties. A specialization may delete states by restricting the range of state properties.

**Example:** The inherited superstate "Functioning" of a device class may have the substates "On" and "Off" in a specialized subclass of devices (Fig. 2). A state property modeled by an integer in the generalization and yielding three states (e.g., negative, 0, positive) may be restricted to only positive integers in the specialization.

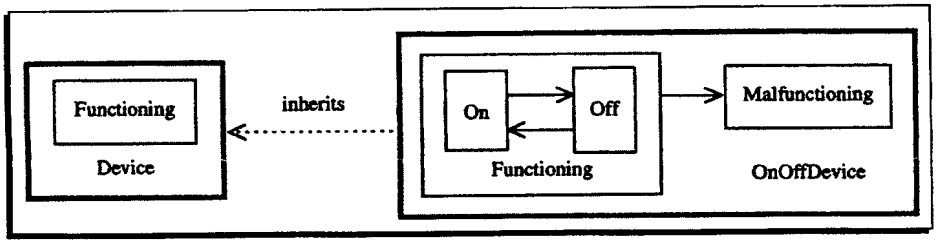


Figure 2.

G-08. The state model of each specialization may add transitions to those inherited from the state models of its generalizations, but should never delete transitions from the state models of its generalizations.

**Rationale:** A specialization can do anything that its generalizations can do, and maybe more. The subclass can add a new operation that performs the transition or override an inherited operation in such a manner that the postcondition of the operation is strengthened to include the transition. Deleting a transition connecting existing states would imply weakening a postcondition, which is not allowed by design by contract for specializations (Fig. 3).

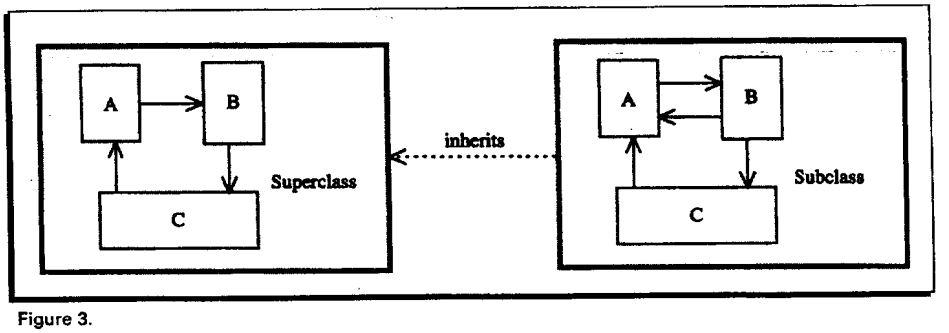


Figure 3.

G-09. The state model of each specialization may add triggers to transitions inherited from the state models of its generalizations, but should never delete triggers from the transitions inherited from the state models of its generalizations.

**Rationale:** Specializations may add new operations that also trigger the same transition to those inherited from its generalizations. The postconditions of inherited operations (which include state transitions) should not be weakened (Fig. 4).

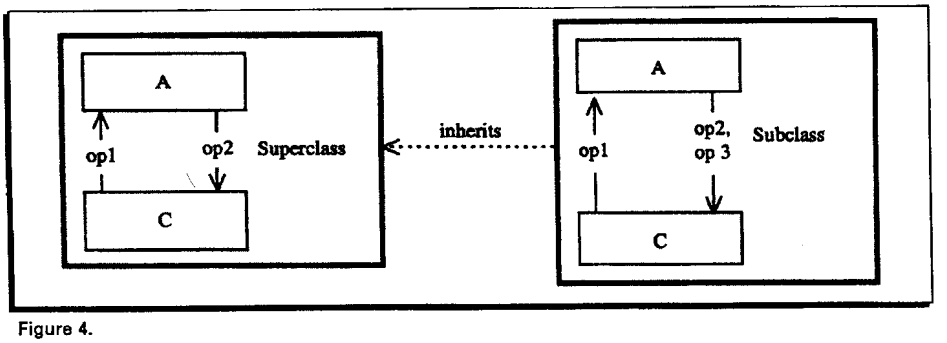


Figure 4.

G-10. An inherited operation that was a trigger in a parent that has *not* been overridden in the child continues to transition the child from the prior state of the parent into the post state of the parent.

**Rationale:** Because the trigger operation has not changed, it performs the same function, including making the same transition in the child as it did in the parent.

G-11. An inherited operation that was a trigger in a generalization that has been overridden in the specialization transitions the specialization from the same or a superstate of the prior state of the generalization to the same or a substate of the post state of the generalization.

**Rationale:** The overridden trigger operation of the specialization must perform the same functional abstraction as the corresponding operation in the generalization or else be a specialization of it. Specialization inheritance cannot weaken preconditions (e.g., prior state of the transition) or strengthen postconditions (e.g., post states of the transition) (Fig. 5).

G-12. The state model of each specialization may weaken (or delete) the guards inherited from the state models of its generalizations, but must never strengthen them or add new guards to inherited transitions.

**Rationale:** Guards correspond to the preconditions of the corresponding trigger operations, and preconditions may be weakened but must never be strengthened. Deleting a guard corresponds to an infinitely weak precondition, but adding a new guard corresponds to strengthening a precondition.

G-13. The state model of each specialization must not transform nonstop states inherited from the state models of its generalizations into stop states, either directly or via new substates, but it may transform an inherited stop state into a nonstop state.

**Rationale:** Stop states correspond to states where the guards (i.e., preconditions) on all exit transitions (i.e., trigger operations) are infinitely strong. A specialization may transform an inherited stop state into a nonstop state by either weakening such guards or by adding new operations that transition the state model out of the state that was previously a stop state. However, a specialization cannot transform a nonstop state into a stop state because this would require infinitely strengthening the corresponding guards, which is not allowed by design by contract.

G-14. The state model of each specialization must not transform visible substates inherited from the state models of its generalizations into hidden substates, but it may transform hidden substates into visible substates.

**Rationale:** The specialization may make an inherited hidden substate visible by adding an operation that transitions the specialization into the formerly hidden substate. However, hiding a formerly visible substate is not allowed because some inherited transition from an external state into the substate would no longer be permitted, which means that either its associated guard (precondition) was strengthened or its associated postcondition was weakened (neither of which is permitted by design by contract).

## CONCLUSION

The proper use of specialization inheritance and its interpretation by the rules of design by contract can be used to ensure that the state model of a child class is consistent with the state models of its ancestors. The preceding guidelines can be used to verify the proper inheritance of state models and can form the basis for automatic checks that can be run by state-modeling tools. ⊠

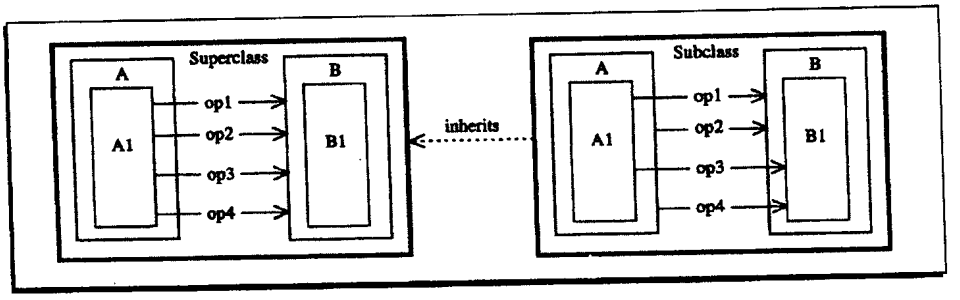


Figure 5.



*continued from page 12*

## CONCLUSION

As MJC says in his comments, the levels of abstraction are increasing into the design aspects of methods and tools. As this trend continues, we approach the translation method set forth in our Star Trek example. Today we elaborate our understanding of a problem through OOA/D to implement a solution. In time we will build compilers or translators to do all the analysis and design work. We are not there yet.

Each methodology represents a transformation. The software development world has changed and our development methods must change with it. While the fundamental metamethod outlined here will remain the same, system engineers and software developers must choose a transformation that makes sense for the level of abstraction supported by their computer. Today, use an O-O methodology to transform your problems into computerized solutions. Tomorrow, as abstraction levels increase beyond the current object-oriented level, our underlying methodology must change. Stay tuned. ⊠

### References


1. Siewiorek, D., C. Bell, and A. Newel. COMPUTER STRUCTURES: PRINCIPLES and EXAMPLES, McGraw-Hill, New York, 1982.
2. Osborne, A., J. Kane, R. Rector, and S. Jacobson. Z80 PROGRAMMING FOR LOGIC DESIGN, Osborne and Associates, Berkeley, CA, 1978.
3. Shah, A., J. Rumbaugh, J. Hamel, and R. Borsari. DSM: An object-relationship modeling language, ACM SIGPLAN, November 11, 1989, pp. 191–202.
4. Rumbaugh, J. and G. Booch. Unified Method V 0.8, Rational Software, Santa Clara, CA, October 1995.
5. Gamma, E., R. Helm, R. Johnson, and J. Vlissides. DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE, Addison-Wesley, Reading, MA, 1995.
6. Pree, W. DESIGN PATTERNS FOR OBJECT-ORIENTED SOFTWARE DEVELOPMENT, ACM Press, 1995.
7. Booch, G. OBJECT-ORIENTED ANALYSIS AND DESIGN WITH APPLICATIONS, 2nd ed., Benjamin/Cummings, Redwood City, CA, 1994.
8. Rumbaugh, J. What is a method?, JOURNAL OF OBJECT-ORIENTED PROGRAMMING 8(6):10–16, 26, 1995.