Such systems are also sensitive to subtle naming errors. In a dynamically searched type [class] hierarchy there is a large pool of available procedure names, increasing the chances that a mistyped procedure call will invoke the wrong procedure [Ponder 94, 36].

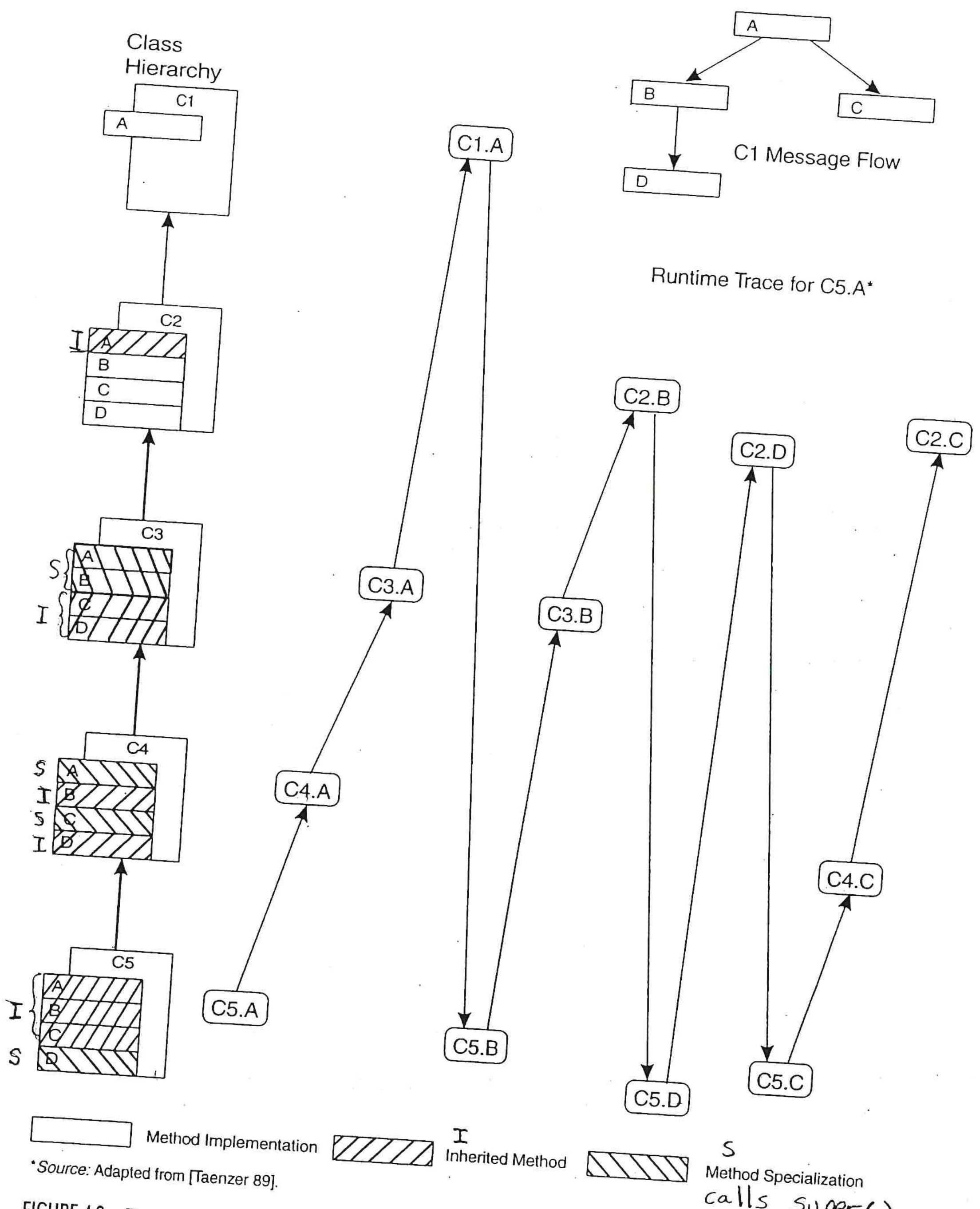
The Yo-Yo Problem

An analysis of reusability suggests many fault hazards due to the message binding mechanisms employed in Objective-C [Taenzer+89]. As classes grow deeper and application systems become wider, the likelihood of misusing a dynamically bound method increases.

Often we get the feeling of riding a yo-yo when we try to understand one of these message trees. This is because in Objective-C and Smalltalk the object *self* remains the same during the execution of a message. Every time a method sends itself a message, the interpretation of that message is evaluated from the standpoint of the original class. . . . This is like a yo-yo going down to the bottom of its string. If the original class does not implement a method for the message, the hierarchy is searched (going up the superclass chain) looking for a class that does implement the message. This is like the yo-yo going back up. *Super* messages also cause evaluation to go back up the hierarchy [Taenzer+89, 33].

Figure 4.2 illustrates this problem using five classes. `C1` is a superclass, `C2` is its subclass, and so on. In this example, the implementation of method `A` uses `B` and `C`, `B` uses `D`. Messages to these methods are bound according to the class hierarchy and the use of `self` and `super`. "The combination of polymorphism and method refinement (methods which use inherited behavior) make it very difficult to understand the behavior of the lower level classes and how they work" [Taenzer+89, 33].

Table 4.2 shows the difference between lexical message structure and a dynamic trace that can result from it. Suppose an object of class `C5` accepts message `A`. `C5's A` is inherited from `C4`, so the search for `A` begins at `C4.A`. `C4.A` is a refinement, so `C3.A` is checked for an implementation. `C3.A` likewise refers to `C1.A`, where the implementation is found and executed. `C1.A` now sends message `B` to itself, causing `B` to be bound to `self` (an object of class `C5`); the search for `B` therefore begins back at `C5`. An implementation of `B` is found in `C3`. `C3.B` sends `B` to `super` (`C2`). `C2.B` executes, sending `D`, which is again bound to `self` (`C5`). The search for `D` continues up to `C2`, where `D` is implemented. `C2.D` executes for the `C5` object, which then sends message `C` to `self` (still `C5`). The search



*Source: Adapted from [Taenzer 89].

FIGURE 4.2 Trace of yo-yo execution.

TABLE 4.2 The Yo-yo Problem

Class	Method A	Method B	Method C	Method D
C1	Implements Sends self to B and C			
C2		Implements Sends self to D	Implements	Implements Sends self to C
C3	Refines Sends super to A	Refines Sends super to B		
C4	Refines Sends super to A		Refines Sends super to A	
C5				Refines Sends super to A

for `C5.C` is resolved at `C4.C`, which sends it to `super` (`C3`). Because `C3` does not implement method `C`, `C2.C` is checked. The implementation `C2.C` is executed on `self` (an object of class `C5`).

Taenzer et al. traced messages in an eight-level class hierarchy and found that a message passed up and down through 58 methods before coming to rest.

4.2.5 Message Sequence and State-related Bugs

The packaging of methods into a class is fundamental to the object-oriented paradigm. As a result messages must be sent in some sequence, raising an issue: which sequences are correct? Which are not? A state-based fault model suggests how state corruption or incorrect behavior can occur. Chapter 7 discusses state models, state faults, and state-based test design in detail. State-based test design patterns are developed in Chapters 10 and 12. State-based testing derives test cases by modeling a class as a state machine. A state model defines allowable message sequences. For example, an `Account` object must have sufficient funds (be in the *open* state) before a `withdrawal` message can be accepted. If the `withdrawal` message causes the balance to become negative, the message causes a transition from the *open* state to the *overdrawn* state. Thus state-based testing focuses on detecting value corruption or sequentially incorrect message responses.

Message Sequence Model

A proposal to generate test cases by an unspecified automatic source code "search" is based on the observation that objects preserve state and typically