homework 41-04

---

insert in 6.42.3 at the end in lieu of the comment:

As methods are inherited from multiple chains of ancestors, the determination of which methods implementations exist and are being called, becomes increasingly more difficult for the programmer. Understanding which methods and data components apply to a given (sub)class becomes exceedingly difficult if these methods or components are inherited homographs (i.e., data components with identical names or methods with identical signatures). Different languages have different rules to resolve the resulting ambiguities. Misunderstandings lead to inadvertent coding errors. The complexity increases even more when multiple inheritance is used to model „has-a"-relationships (see also << reference to BLP, Liskov>>): methods never intended to be applicable to instances of a subclass are inherited nevertheless. For example, an instance of class aircraftCarrier may be „turn"ed merely because it obtained its propulsion screw by a „has-a"-inheritance with „turn" being an obviously meaningful method for the class of propulsionScrew. Meanwhile the user has a quite different expectation of what it means to turn an aircraft carrier. The complications increase if the carrier inherits twice from the class propulsionScrew because it has two propulsion screws.

Finally, if ambiguities in method or component namings are resolved by preference rules, changes in the execution of methods can be introduced by adding yet another unrelated but homographic method or data declaration anywhere is the hierarchies of ancestor classes during maintenance of the code. Malicious implementations can thus be added with each release of an object-oriented library and affect the behavior of previously verified code. (see also << reference to BJL, name spaces>>)

The mechanism of failure for these additional dangers caused by multiple inheritance is the inadvertent use of the wrong data components or methods. Knowledge of such incorrect use might be exploitable, as instances of the affected (sub)class may be corrupted by inappropriate operations.

maybe insert in 6.42.5: << they are already in 6.43.5>>

- Prohibit the use of visible inheritance for "has-a" relationships.
- Use components of the respective class for "has-a"-relationships.

Patrice and I have an additional AI on 6.44. (redispatching). Patrice had ideas about solutions, so should do first draft.

AI 41-12 should be closed. Was done 2 meetings ago.

AI 41-05 should be closed. Was done before the last meeting.

AI 41-03 6.37 in a separate document.