CSE 3302 Notes 9: Data Abstraction

(Last updated 11/17/15 2:21 PM)

References: Gabbrielli: 9

9.1. Abstract Data Types

Programmer-defined types should be handled like built-in types

Primary characteristics:

1. Type name
2. Underlying representation (concrete type)
3. Set of operation names
4. Implementation for each operation
   (`pop` returns a reference to the modified stack, `top` returns last-in without mutating)
5. Opaque interface separating 1. and 3. from 2. and 4.

Stack of integers example:

1. Figure 9.1, no opaque interface
2. Figure 9.2, adds 5. in form of `signature`

9.2. Information Hiding

Based on opaque interface the *principle of representation independence* (p. 271) holds:

> Two correct implementations of (a single specification) of an ADT are operationally indistinguishable by the clients of the types.

Constructor, transformers/operators (Liskov/Wing call these mutators), observers

9.3. Modules

Module: One or more ADTs encapsulated together

Import: Accessing a module for purposes of instantiating

Figures 9.5 and 9.6 (p. 273-274)

9.3. SOME ITEMS FROM W.R. COOK PAPER

Figures 2 and 3, integer sets as ML ADT for specification only

Figure 4, integer set using sorted lists as representation

Disadvantage of many ADT mechanisms: only one implementation per ADT name

Notable Exceptions:

C header files and separately compiled implementations (e.g. makefiles as glue)
C++ (pointer to implementation) pimpl idiom - to avoid recompilation
`https://en.wikipedia.org/wiki/Opaque_pointer`
Java interfaces and generics
C++ templates

Sections 2.4 and 2.5

Existential type = ADT name and operations (Figure 6), but where implementations may differ

Using different implementations for one ADT gives different *incompatible* types, e.g. mixing two recursive implementations

Transition to Notes 10:

From P. Wegner, "Dimensions of object-based language design", OOPSLA '87
`http://dl.acm.org.ezproxy.uta.edu/citation.cfm?doid=38765.38823`

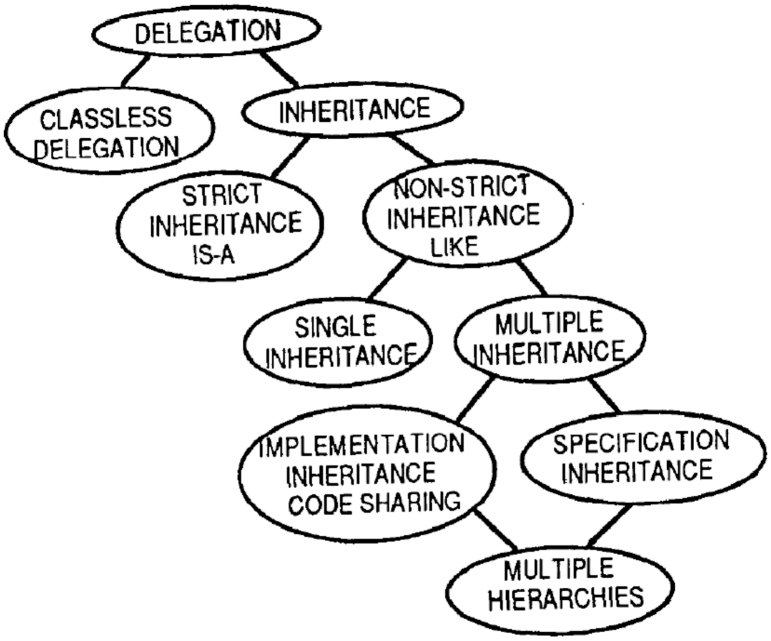

Figure 2. Design Alternatives for Inheritance