25

Useful techniques

Examples of object-oriented design given in preceding chapters have illustrated a number of distinctive techniques. Although we are not done yet with our review of methodological issues — we must still explore style rules, O-O analysis concepts, teaching methods, and the software process — it is time to pause briefly to recapitulate some of the principal O-O techniques that we have learned.

This will be the tersest chapter of all: it just enumerates fruitful ideas, followed in a few cases by keywords meant to remind you of some of the examples in which we first encountered the ideas.

25.1 DESIGN PHILOSOPHY

General development scheme

Bottom-up development: build a solid basis, then apply it to specific cases.

Chapter 28 discusses seamlessness and reversibility.

Seamlessness: apply consistent techniques and tools to analysis, design, implementation, maintenance.

Reversibility: let the functional specification benefit from the lessons of implementation.

Generalization: from specialized classes, derive reusable ones. Abstraction, factoring out commonalities.

The structure of systems

Systems are made of classes only.

The development style is bottom-up. Whenever possible, start from what you have.

Try to make classes as general as possible from the start.

Try to make classes as autonomous as possible.

Two inter-class relations: client (with “reference client” and “expanded client” variants); inheritance. Roughly correspond to “has” and “is”.

Use multi-layer architectures to separate abstract interface from implementations for various platforms: Vision, WEL/PEL/MEL.

System evolution

Design for change and reuse.

When improving a design, use obsolete features and classes to facilitate the transition.

25.2 CLASSES

Class structure

Every class should correspond to a well-defined data abstraction.

Shopping List approach: if a feature is potentially useful, and fits in with the data abstraction of the class, put it in.

Facility classes: group related facilities (e.g. a set of constants).

Active data structures (object as abstract machine).

Key decision is what features to make secret and what to export.

Use selective exports for a group of intimately connected classes: *LINKED_LIST*, *LINKABLE*.

Re-engineer non-O-O software by encapsulating abstractions into classes (cf. *Math* library).

Class documentation

Put as much information as possible in the class itself.

Write header comments carefully and consistently; they will be part of the official interface.

Indexing clauses.

Designing feature interfaces

Command-Query Separation principle: a function should not produce any abstract side effect (concrete side effects are OK).

Use only operands as arguments.

Set status, then execute operation.

For each status-setting command, provide a status-returning query.

For argumentless queries, there should be no externally visible difference between an attribute implementation and a function implementation.

Let an object change representation silently as a result of requested operations (example of complex number class).

Cursor structures (*LIST*, *LINKED_LIST* and many others).

Using assertions

The precondition binds the client, the postcondition binds the supplier.

Make precondition strong enough to enable the routine to do its job well — but not stronger.

Two kinds of invariant clause: some clauses come from the underlying abstract data type; others (*representation invariant*) describe consistency properties of the implementation. Use implementation invariant to express and improve your understanding of the relationship between the different constituents of the class, attributes in particular.

For an argumentless query, include abstract properties in the invariant (even if, for a function, the property also appears in the postcondition).

Redeclarations can weaken the precondition to make the routine more tolerant.

To achieve the effect of strengthening the precondition, use an abstract precondition (based on a boolean function) in the original.

Even with no strengthening need, abstract preconditions are preferable.

Any precondition must be enforceable and testable by clients.

Do not overconstrain postcondition, to enable strengthening in a descendant (for example you may want to use one-way **implies** rather than equality).

Dealing with special cases

A priori checking: before operation, check if you can apply it.

A posteriori checking: try operation, then query an attribute (or higher-level function encapsulation) to find out whether it worked.

When everything else fails: use exception handling.

Organized failure: if a **rescue** executes to the end, do not forget to restore the invariant. The caller will get an exception too.

Retrying: try another algorithm, or (the strategy of hope) the same one again. Record what happened through an attribute or local entity; local entities are initialized only when the call starts, not after a **retry**.

25.3 INHERITANCE TECHNIQUES

Redeclaration

Redefining a routine to use a more specific algorithm, for more efficiency: *perimeter* in *POLYGON*, *RECTANGLE*, *SQUARE*.

Redefining a routine into an attribute: *balance* in *ACCOUNT*.

Effecting a feature that was deferred in the parent.

Joining two or more features through effecting (all but one inherited as deferred; the effective one takes over). Undefine some effective ones if needed.

Redefining two or more effective features together.

Accessing parent version in a redefinition: *precursor*.

Redeclarations preserve semantics (rules on assertions).

Deferred classes

Deferred classes capture high-level categories.

Deferred classes also serve as an analysis and design tool, to describe abstractions without commitment to an implementation.

Behavior classes: capture general behavior. Effective routines call deferred ones. Class will be partially deferred, partially implemented (covers partial choice of ADT implementation).

Polymorphism

Polymorphic data structures: through inheritance and genericity, combine right amount of similitude and variation.

Handles: describe a variable-type component through a polymorphic attribute.

Dynamic binding: avoid explicit discrimination.

Dynamic binding on a polymorphic data structure: apply to each element of a structure an operation that the element will apply in its own appropriate way.

For the point of single choice, pre-compute a data structure with one object of each possible type (as in the undoing pattern).

Forms of inheritance

Make sure all uses of inheritance belong to one of the categories in the taxonomy.

Inheritance for subtyping.

Inheritance for module extension.

Marriage of convenience: implement abstraction through concrete structure.

Restriction inheritance: add constraint.

Inheriting general-purpose mechanisms from facility classes.

Functional Variation inheritance: “organized hacking”, Open-Closed principle.

Type Variation inheritance: covariance.