Object Orientation and Program Family

Object Orientation vs. Program Family

- At first sight it seems as if program families are by-product of object orientation
  - Inheritance is a measure to extend, refine, and specialize a set of classes
    * Thus, to reuse interfaces and/or implementations
  - To “extend”, “refine”, and “specialize” are key issues of program families
- But note that object orientation may be employed in quite different ways:
  - Functional emaciation
  - Functional enrichment
  - Only the 2nd case is in one line with the goals of family-based software designs
- Customization of a “default implementation” can be achieved using late binding
  - Interface inheritance enables specialization transparently to clients
    * Problem-aware implementations can be added to a problem-unaware one
    * Less efficient implementations can be replaced by more efficient ones
  - But this does not automatically cause the “replaced” functions to disappear
- Late binding is not for free and may entail a certain amount of overhead
  - In terms of: (1) Waste of main memory and (2) Loss of execution performance
- The problem comes with virtual-function tables and object construction

Functional Emaciation

Late Binding Revisited

Virtual-Function Tables
Late Binding Revisited

Constructors

__6Foobar:
pushl %ebx
movl 8(%esp),%ebx
pushl %ebx
call __3Foo
leal 4(%ebx),%eax
pushl %eax
call __3Bar
movl $__vt_6Foobar.3Bar,4(%ebx)
movl $__vt_6Foobar,(%ebx)
addl $8,%esp
movl %ebx,%eax
popl %ebx
ret

__3Foo:
movl 4(%esp),%eax
movl $__vt_3Foo,(%eax)
ret

__3Bar:
movl 4(%esp),%eax
movl $__vt_3Bar,(%eax)
ret

Object Orientation Considered Harmful?

- an explosion of program size may be the outcome of the sketched problem
  - at runtime unused but, at generation time, referenced units are present

  Less demanding users will be forced to pay for the resources consumed by the unneeded features
  - this is in contradiction to the concept of family-based software design[3]

- interface inheritance is a typical case of a non-functional requirement
  - in a family-based design it needs to be modeled as a separate feature
  - this modeling can be implemented in an object-oriented manner

- object orientation becomes efficient by a supplementing family-based design

Late Binding Revisited

Object Construction

- the starting point of all evils is object construction at runtime
  - constructors contain code sequences which reference virtual-function tables
  - virtual-function tables contain references to program code¹

- the construction of an object happens from base class to derived class
  - constructors associate the object with a virtual-function table
  - an association made at base-class level may be overwritten at derived levels
  - yet do the overwritten bindings remain existent in terms of program code

- the (static) binder adds all referenced units to the load module before runtime

¹That is, the tables contain references to redefined methods and/or thunks referencing redefined methods.

Non-Functional Aspect of Interface Inheritance
Adaptor Pattern

- Interface and implementation can be patched up using the *adaptor pattern* [1]
  - "convert the interface of a class into another interface clients expect"

- Clients are interfaced by an *abstract class*
  - Made of "pure virtual functions"

- A *wrapper* uses multiple inheritance
  - Specializing the abstract class
  - Reusing the implementation class

- Manual implementation is (mostly) straightforward—and a case of automation

C++ Adaptor Patterns

```cpp
class iFop {
public:
  virtual int fop () = 0;
};

class aFop : public iFop, public Fop {
  int fop () { return Fop::fop(); }
};

class iFoo : public iFop {
public:
  virtual int foo () = 0;
};

class aFoo : public iFoo, public Foo {
  int foo () { return Foo::foo(); }
};

class iBar : public iFop {
public:
  virtual int bar () = 0;
};

class aBar : public iBar, public Bar {
  int bar () { return Bar::bar(); }
};

class iFoobar : public iFop, public iBar {
public:
  virtual int foobar () = 0;
};

class aFoobar : public iFoobar, public iBar {
  int fop () { return iFop::fop(); }
  int foo () { return iFoo::foo(); }
  int bar () { return iBar::bar(); }
  int foobar () { return iFoobar::foobar(); }
};
```

Beware of the Design!

Adaptor Patterns (very overhead-prone)

C++ → x86

```
...movl $__vt_4iFoo,(%eax)
movl $__vt_4iBar,4(%eax)
movl $__vt_7iFoobar,(%eax)
movl $__vt_7aFoobar.4iBar,4(%eax)
movl $__vt_7aFoobar,(%eax)
...
_vt_7aFoobar.4iBar:
  long -4
  long 0
  long __thunk_4_fop__7aFoobar
  long __thunk_4_bar__7aFoobar

_vt_7aFoobar:
  long 0
  long 0
  long __vt_4iFoo,(%eax)
movl $__vt_4iBar,4(%eax)
movl $__vt_7iFoobar,(%eax)
movl $__vt_7aFoobar.4iBar,4(%eax)
movl $__vt_7aFoobar,(%eax)
...

pushl %eax
call foobar_6Foobar
addl $8,%eax
L34:
addl $4,%espret
ret
```

aFoobar tables

aFoobar construction

Adaptor/Wrapper
Adaptor Patterns (less overhead-prone)

Patterns Considered Harmful?

- care must be taken about the consequences a pattern might have
  - sometimes a pattern implementation requires late binding
  - some other time late binding may be left up to the programmer
  - next time late-binding overhead is unacceptable due to the compiler

- design patterns define a trade-off of maintenance and performance
  - software maintenance is improved, development times can be reduced
  - all at the expense of performance, as many patterns imply late binding

- nothing is for free—but system designers must be aware of the effective costs

Patterns as Aspects of Design

- the design decision for late binding is to be postponed as far as possible
  - exploit late binding only when it becomes a functional requirement
  - leave it off from the (hand-made) implementation otherwise

- non-functional and functional features of a design must never be mixed up
  - design patterns are different from implementation patterns
  - the former may be streamlined and the latter may be added automatically

- design patterns must not always have counterparts in the implementation
  - “it is the system design which is hierarchical, not its implementation” [2]
Summary

- extensible and/or contractible system-software design should be family-based
  - start from a minimal subset of system functions
  - perform incremental machine design by stepwise functional enrichment
  - functional enrichment goes hand in hand with minimal system extensions
- object orientation supports an efficient implementation of family-based designs
  - encapsulate the minimal subset of system functions by base classes
  - exploit inheritance to achieve functional enrichment, not emaciation
  - encapsulate the minimal system extensions by derived classes
- encapsulate “componentized branches” of the family using abstract classes

Bibliography