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:
  
  \[
  \begin{aligned}
  \text{functional emanciation} \\
  \text{functional enrichment}
  \end{aligned}
  \]
  from general- to special-purpose

  \[
  \left\{ \begin{array}{c}
  \text{implementation} \\
  \text{application}
  \end{array} \right. 
  \]

• only the 2\textsuperscript{nd} case is in one line with the goals of family-based software designs
Functional Emaciation

- 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

```
class Foo {
public:
  Foo (){
    virtual int foo ();
  }
};

class Bar {
public:
  Bar (){
    virtual int bar ();
  }
};

class FooBar : public Foo, public Bar {
  int foo ();
  int bar ();
public:
  FooBar ();
};
```

---

Virtual-Function Tables

```
__vt_Foo:
  .long 0
  .long 0
  .long foo__3Foo

__vt_Bar:
  .long 0
  .long 0
  .long bar__3Bar

__vt_GFooBar: ...
  .long foo__6GFooBar
__vt_GFooBar.Bar: ...
  .long __thunk_A_bar__6GFooBar
__thunk_A_bar__6GFooBar: ...
  jmp bar__6GFooBar
```
Late Binding Revisited

Constructors

```assembly
__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
```

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

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

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 code

- 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.
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

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 Foo : public IFop {  
public:  
    virtual int foo () = 0;  
};

class Bar : public IFop {  
public:  
    virtual int bar () = 0;  
};

class FooBar : public Foo, public Bar {  
public:  
    virtual int foobar () = 0;  
};
```

```cpp
class AFop : public IFop, public Fop {  
    int fop () { return Fop::fop(); }  
};

class AFoo : public IFoo, public Foo {  
    int fop () { return Foo::fop(); }  
};

class ABar : public IFoo, public Bar {  
    int fop () { return Bar::fop(); }  
};
```

```cpp
class AFooBar : public IFooBar, public FooBar {  
    int fop () { return FooBar::fop(); }  
    int fop(int) { return FooBar::fop(); }  
    int foo () { return FooBar::foo(); }  
    int bar () { return FooBar::bar(); }  
};
```
Beware of the Design!

Adaptor Patterns *(very overhead-prone)*

\[
\text{C++} \quad 2.91 \rightarrow \text{x86}
\]

---

```assembly
// _vt_7aFooBar.4iBar:
  .1cmdg =4
  .1cmdg 0
  .1cmdg __thunk.4_fop__7aFooBar
  .1cmdg __thunk.4_bar__7aFooBar

// _vt_7aFooBar:
  .1cmdg 0
  .1cmdg 0
  .1cmdg fop__7aFooBar
  .1cmdg foo__7aFooBar
  .1cmdg foobar__7aFooBar
  .1cmdg fop__7aFooBar

... movl $._vt_4iFoo,(%eax)
  movl $._vt_4iBar.4(%eax)
  movl $._vt_7iFooBar,(%eax)
  movl $._vt_7aFooBar.4iBar.4(%eax)
  movl $._vt_7aFooBar,(%eax)
  ...

// fooBar__7aFooBar:
  movl 4(%esp),%eax
testl %eax,%eax
  jne .L34
  xorl %eax,%eax
  jmp .L35
  p2align 4,7
  .L34:
  addl $8,%eax
  .L35:
  pushl %eax
call fooBar__6FooBar
  addl $4,%esp
  ret
```

- aFoobar tables
- aFoobar construction
- adaptor/wrapper
Adaptor Patterns *(less overhead-prone)*

C++ $\rightarrow$ x86

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 hierachical, not its implementation” [2]

Program Family Considered Object-Oriented [4]
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


