O
O S E
E

# Object Orientation and Program Family

Operating-System Engineering

## 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:

$$\left.\begin{array}{l}\textit{functional emaciation}\\ \textit{functional enrichment}\end{array}\right\}\text{from general- to special-purpose}\left\{\begin{array}{l}\text{implementation}\\ \text{application}\end{array}\right.$$

- only the $2^{nd}$ 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*
  - <u>interface inheritance</u> 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



## Late Binding Revisited — Virtual-Function Tables

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

```
__vt_3Foo:
  .long 0
  .long 0
  .long foo__3Foo
```

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

```
__vt_3Bar:
  .long 0
  .long 0
  .long bar__3Bar
```

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

```
__vt_6Foobar: ...
  .long foo__6Foobar

__vt_6Foobar.3Bar: ...
  .long __thunk_4_bar__6Foobar

__thunk_4_bar__6Foobar: ...
  jmp   bar__6Foobar
```

## 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
```



## 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[1]
- 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 <u>all</u> referenced units to the load module before runtime

[1] That 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**

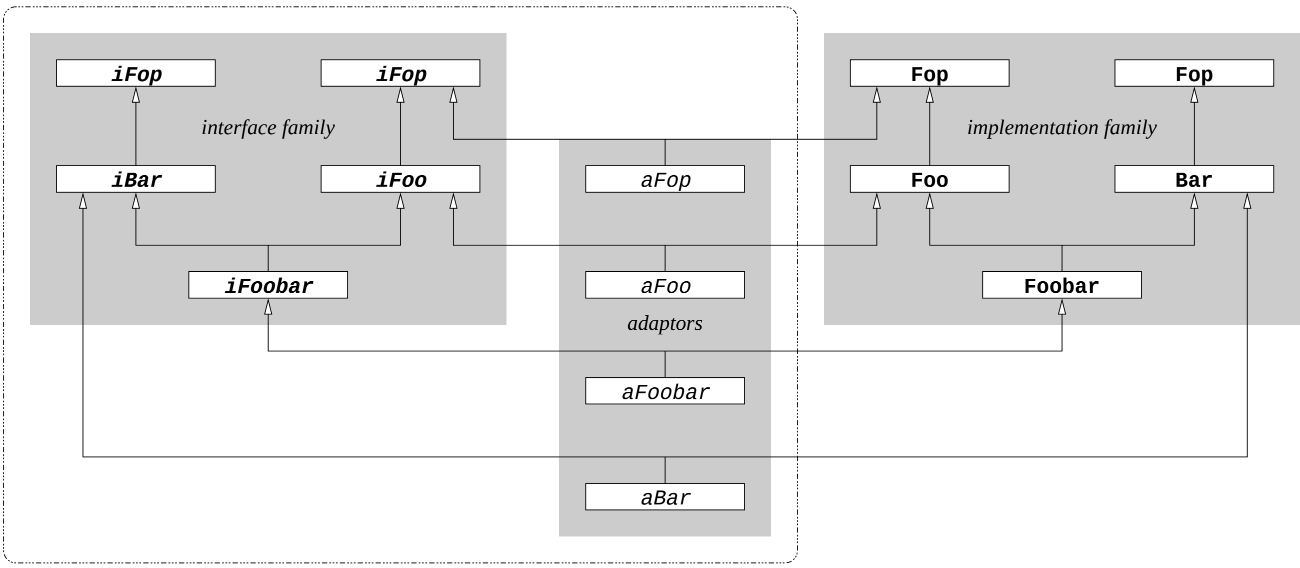
  - this is in contradiction to the concept of family-based software design[3]

- 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

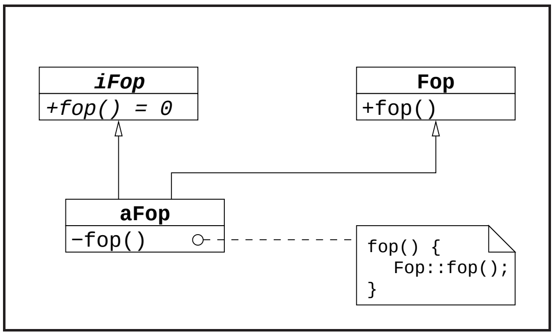


## Non-Functional Aspect of Interface Inheritance



# 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

```
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 fop () { return Foo::fop(); }
  int foo () { return Foo::foo(); }
};
```

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

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

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

```
class aFoobar : public iFoobar, public Foobar {
  int fop ()     { return Foobar::Foo::fop(); }
  int fop (int) { return Foobar::Bar::fop(); }
  int foo ()     { return Foobar::foo(); }
  int bar ()     { return Foobar::bar(); }
  int foobar () { return Foobar::foobar(); }
};
```

# Beware of the Design!



## Adaptor Patterns (very overhead-prone)

C++ $\xrightarrow{2.91}$ x86

```
__vt_7aFoobar.4iBar:
  .long -4
  .long 0
  .long __thunk_4_fop__7aFoobar
  .long __thunk_4_bar__7aFoobar
```

```
__vt_7aFoobar:
  .long 0
  .long 0
  .long fop__7aFoobar
  .long foo__7aFoobar
  .long foobar__7aFoobar
  .long fop__7aFoobari
```

aFoobar tables

```
  ...
  movl $__vt_4iFoo,(%eax)
  movl $__vt_4iBar,4(%eax)
  movl $__vt_7iFoobar,(%eax)
  movl $__vt_7aFoobar.4iBar,4(%eax)
  movl $__vt_7aFoobar,(%eax)
  ...
```

aFoobar construction

```
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
```

adaptor/wrapper

## Adaptor Patterns (less overhead-prone)

C++ $\xrightarrow{2.96}$ x86

```
__vt_7aFoobar.4iBar:
  .long -4
  .long __pure_virtual
  .long __thunk_4_fop__7aFoobar
  .long __thunk_4_bar__7aFoobar
```

```
__vt_7aFoobar:
  .long 0
  .long __pure_virtual
  .long fop__7aFoobar
  .long foo__7aFoobar
  .long foobar__7aFoobar
  .long fop__7aFoobari
```

aFoobar tables

```
  ...
  movl $__vt_7aFoobar,(%eax)
  movl $__vt_7aFoobar.4iBar,4(%eax)
  ...
```

aFoobar construction

```
fop__7aFoobar:
  addl $8,4(%esp)
  jmp  fop__3Fop
foo__7aFoobar:
  addl $8,4(%esp)
  jmp  foo__3Foo
bar__7aFoobar:
  addl $9,4(%esp)
  jmp  bar__3Bar
foobar__7aFoobar:
  addl $8,4(%esp)
  jmp  foobar__6Foobar
fop__7aFoobari:
  addl $9,4(%esp)
  jmp  fop__3Fop
```

adaptor/wrapper



## Patterns Considered Harmful?

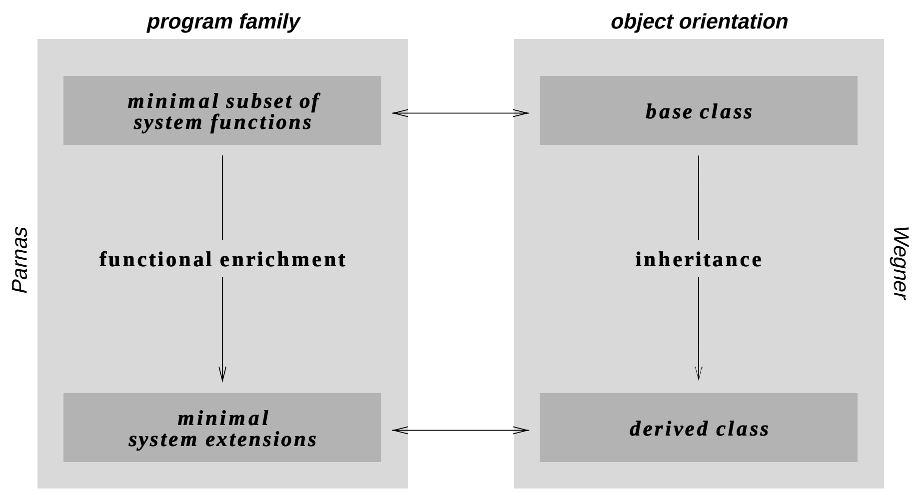
- care must be taken about the consequences a pattern might have
  - sometimes a pattern implementation requires late binding
  - some other time late binding may be left up to the programmer
  - next time late-binding overhead is unacceptable due to the compiler
- design patterns define a trade-off 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 famliy using abstract classes



# Bibliography

[1] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995. ISBN 0-201-63361-2.

[2] A. N. Habermann, L. Flon, and L. Cooprider. Modularization and Hierarchy in a Family of Operating Systems. *Communications of the ACM*, 19(5):266–272, 1976.

[3] D. L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128–138, 1979.

[4] W. Schröder-Preikschat. *The Logical Design of Parallel Operating Systems*. Prentice Hall International, 1994. ISBN 0-13-183369-3.