Object Orientation and Program Family

- at first sight it seems as if program families are a 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** from general- to special-purpose
- only the 2nd 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

### Late Binding Revisited

```java
class Poo {
    public:
        Poo (i);
        virtual int foo () { return 0; }
    }
}

class Bar {
    public:
        Bar (i);
        virtual int bar () { return 0; }
    }
}

class PooBar : public Poo, public Bar {
    int foo (i);
    int bar (i);
    public:
        PooBar (i);
    }
}
```

### Virtual-Function Tables

```java
class Poo {
    Poo (i);
    long foo () { return 0; }
    long foo2 () { return 0; }
}

class Bar {
    Bar (i);
    long bar () { return 0; }
    long bar2 () { return 0; }
}

class PooBar : public Poo, public Bar {
    int foo (i);
    int bar (i);
    long foo2 () { return 0; }
    long bar2 () { return 0; }
}
```
Late Binding Revisited

Constructors

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 unnecessary 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 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.
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

Beware of the Design!

C++ Adaptor Patterns

```cpp
class lBase { public:
    virtual int lfoo () = 0;
};

class lMo : public lBase { public:
    virtual int lfoo () = 0;
};

class lBar : public lBase { public:
    virtual int lbar () = 0;
};

class lBar : public lBase { public:
    virtual int lbar () = 0;
};
```

Adaptor Patterns (very overhead-prone)

```cpp
C++ → x86
```

aFooBar tables

aFooBar construction

adapter/wrapper
Adaptor Patterns (less overhead proné)

C++ ☞ x86

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]

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
    - 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

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