

# Implementing Layered Designs with Mixin Layers

Yannis Smaragdakis

(joint work with Don Batory)

## Objects as components?

- Modular decomposition is an ideal in software design and implementation
  - separation of concerns is the only way to fight complexity
  - smaller, intellectually manageable pieces
- Objects are not the right unit of modularity: they are rarely self-sufficient
  - not meaningful in isolation,
  - only in suites of cooperating objects
- Need to examine collections of collaborating objects as a unit
- Easy to do at the design level (e.g., design patterns)
- Support lacking at the implementation level (no explicit programming language constructs)

- In this talk: we present a general idea for large-grain components and object-oriented implementation techniques
- The inspiration for components comes from *collaboration-based* designs
- Implementation is based on *mixins*

## Collaboration-Based Designs

- Objects cooperate to perform tasks
- In collaboration-based design, an application is decomposed into
  - objects
  - *collaborations*: sets of objects and protocols for their interaction
- Each class plays a *role* in a collaboration
- Each collaboration consists of roles for different classes
- Collaborations encode dependencies among classes
  - as closely as possible
  - few outside dependencies remain
- Hence, collaborations are reusable
  - “plug and play” components

## Example

### Object Classes

		<i>Object OA</i>	<i>Object OB</i>	<i>Object OC</i>
Collaborations	Collaboration <i>c1</i>	RoleA1	RoleB1	RoleC1
	Collaboration <i>c2</i>	RoleA2	RoleB2	
	Collaboration <i>c3</i>		RoleB3	RoleC3
	Collaboration <i>c4</i>	RoleA4	RoleB4	RoleC4

- The intersections of classes and collaborations represent roles
- Collaborations are independent:
  - Ideally, they should be reusable
  - All dependencies are intra-collaboration (no outside dependencies)

## Mixins

In object-oriented programming, superclasses are declared at subclass definition time

```
class Foo : public Bar { ... };
```

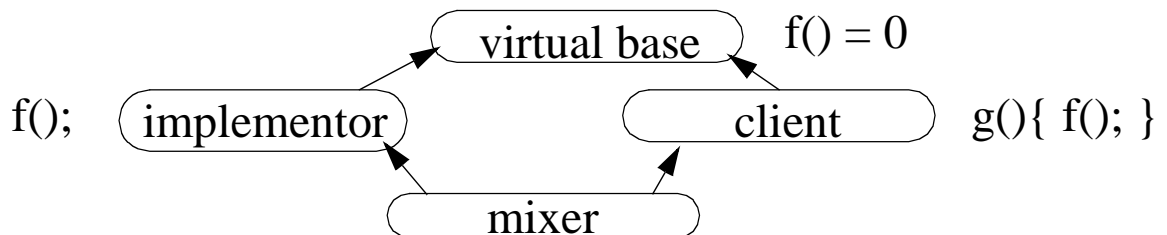
Mixins (or *mixin classes*, or *abstract subclasses*) are like classes without specific superclasses

For instance, in C++:

```
template <class Super>  
class Mixin : public Super { ... };
```

Mixins are well supported in object-oriented languages:

- originated in CLOS (superclass determined by linearization)
- several implementations in Smalltalk
- two ways to express mixins in C++
  - multiple inheritance in “diamond” pattern



- parameterized inheritance (superclass is a template parameter)
- experimental implementations on Java
- supported by other OO languages (e.g., Beta)

Our examples in the C++ templates mixin idiom

## Implementing Collaborations

- Collaboration based designs can be implemented using standard application frameworks
- There are, however, many benefits in implementing collaborations using mixins
  - flexibility (parent classes not fixed)
  - efficient (no virtual methods required — we can order things differently)
  - multiple refinements allowed in the same composition (no naming conflicts, fixed “topmost” class)
- VanHilst and Notkin used C++ mixins to express roles

## Example

```
template<class Super, class OA,  
        class OC>  
class B4 : public Super {  
    ... // Implementation using OA, OC  
};
```

- Roles parameterized by their superclass and any other classes they depend on
- Classes formed by composing mixins

Several problems:

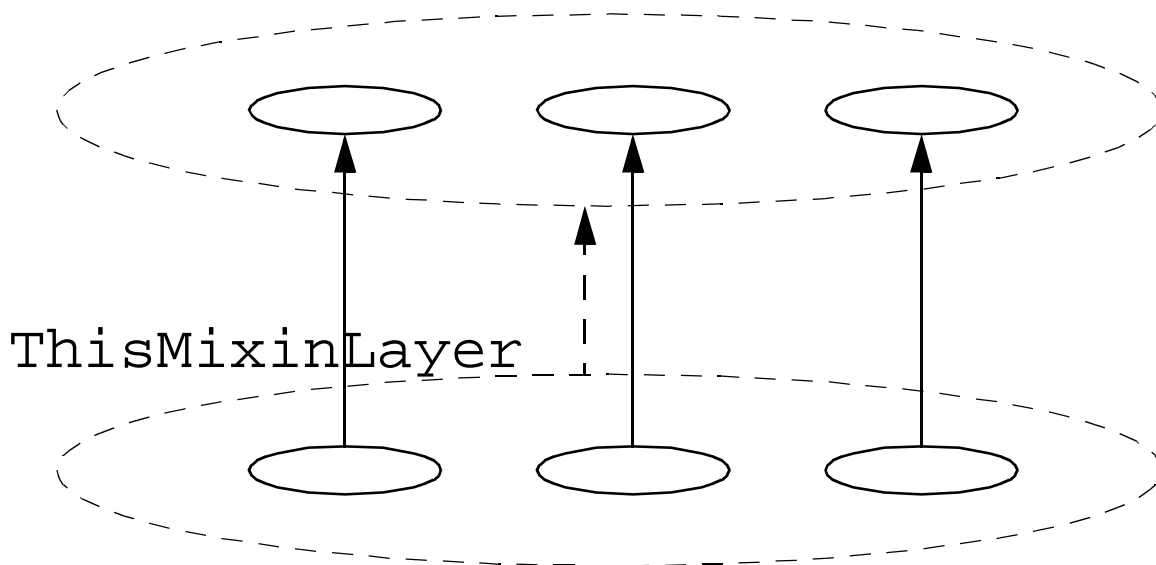
- Redundancy
- Large complexity of parameterizations
  - potentially exponential in the number of collaborations
- Because each role needs to be parameterized by all others that it references. These in turn need to be parameterized by others, etc...
- Solution in the form of *mixin layers*.

## Mixin Layers

### Mixin layers

- are mixins (*outer*) that encapsulate other mixins (*inner*)
- identifying property: the parameter (superclass) of the outer mixin encapsulates all parameters (superclasses) of inner mixins

### SuperMixinLayer



This general form is hard to express in existing programming languages

A more constrained form is straightforward:

- Inner classes are *constrained mixins*: the name of the superclass is known (the superclass can still vary since it can be encapsulated in different outer classes)

Class encapsulation can be implemented in many ways. E.g., in C++, using nested classes:

```
template <class SuperML>
class ThisML : public SuperML {
public:
    class Inner1 :
        public SuperML::Inner1 { ... };
    class Inner2 :
        public SuperML::Inner2 { ... };
    class Inner3 :
        public SuperML::Inner3 { ... };
};
```

Principles translate in other OO languages  
(different flavors)

E.g., in CLOS, using reflection:

```
(defclass first-dummy() (...))
(defclass second-dummy () (...))
(defclass third-dummy () (...))
...
(defclass collab-this () ())
; Encapsulate class as method
(defmethod first-role
  ((self collab-this))
  (cons (find-class `first-dummy)
        (call-next-method)))
(defmethod second-role
  ((self collab-this))
  (cons (find-class `second-dummy)
        (call-next-method)))
...
```

We create a list of classes by multiply inheriting.  
We can then create an inheritance hierarchy  
programmatically.

Mixin layers composed linearly (like simple mixins):

```
Layer1 < Layer2 < Layer3 > > >::Inner1
```

Mixin layers are ideal for implementing collaborations:

```
template <class SuperCol>
class ThisCol : public SuperCol {
public:
    class Role1 :
        public SuperCol::Role1 {...};
    class Role2 :
        public SuperCol::Role2 {...};
    class Role3 :
        public SuperCol::Role3 {...};
};
```

## Example

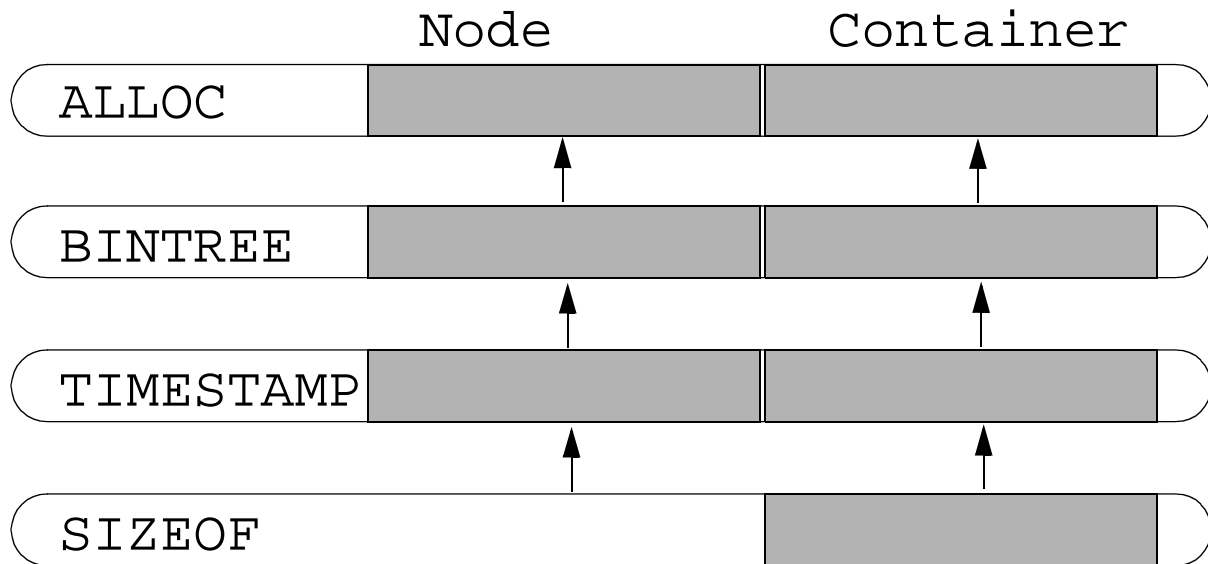
```
template <class Super>
class BINTREE : public Super {
public:
    class Node :
        public Super::Node {...};
    class Container :
        public Super::Container {
            Node *header; ...
        };
};
```

Inner classes are inherited!

```
template <class Super>
class SIZEOF : public Super {
public:
    class Container :
        public Super::Container { ... };
};
```

E.g., `SIZEOF` does not add anything to the `Node` class — it inherits the definition from its superclass

```
sizeof<timestamp<bintree<alloc<int>
> > >
```



Actual classes nested inside mixin layer:

```
typedef
    sizeof < timestamp < bintree < alloc
        < int > > > >
        MyDataStructure;
MyDataStructure::Container Col;
MyDataStructure::Node *Cursor;
```

Mixin layers represent components that are:

- Reusable
- Interchangeable

```
typedef
    BINTREE < ALLOC < int > > Tree1;
```

```
typedef
    SIZEOF < TIMESTAMP < BINTREE < ALLOC
        < int > > > >
    CycleC;
```

A result of the independence of collaborations

- For instance, the BINTREE collaboration/mixin layer encapsulates all the functionality of a binary tree (even though it spans different classes)
- Examples with many more collaborations and roles have been encountered.

## Compared to V&N's mixins:

- No redundancy: every inner mixin can reference all classes “above” it — outer mixin acts as a namespace
- Inner mixins are inherited — outer mixin acts as a subclass
- Linear length of parameterizations

```

class Empty {};
class WS      : public WorkspaceNumber      {};
class WS2     : public WorkspaceCycle       {};
class VGraph  : public VertexAdj<Empty>     {};
class VWork   : public VertexDefaultWork<WS,VGraph> {};
class VNumber : public VertexNumber<WS,VWork>  {};
class V       : public VertexDFT<WS,VNumber>  {};
class VWork2  : public VertexDefaultWork<WS2,V>  {};
class VCycle  : public VertexCycle<WS2,VWork2>  {};
class V2      : public VertexDFT<WS2,VCycle>  {};
class GGraph  : public GraphUndirected<V2>     {};
class GWork   : public GraphDefaultWork<V,WS,GGraph> {};
class Graph   : public GraphDFT<V,WS,GWork>    {};
class GWork2  : public GraphDefaultWork<V2,WS2,Graph> {};
class GCycle  : public GraphCycle<WS2,GWork2>  {};
class Graph2  : public GraphDFT<V2,WS2,GCycle>  {};

class NumberC : public DFT < NUMBER < DEFAULTTW < UGRAPH > > > {};
class CycleC  : public DFT < CYCLE < DEFAULTTW < NumberC > > > {};

```

There is more to this:

- Other elements fit in the mixin layers design
  - design rule checking by adding properties as nested classes and using access control
  - adding member functions to mixin layers (to simulate Ada-like packages)
- Language specifics
  - Primitive static type checking for templates in C++
  - Type checking only after composition
- See also: <http://www.cs.utexas.edu/users/smaragd/research.html>

## In conclusion...

- Large-scale components (encapsulating functionality for multiple classes) are an important unit of modularity
- It is possible to express such components as mixin layers using OO implementation techniques (mixins and class nesting)
- Such implementations have many desirable properties
  - reusability
  - orthogonality of components
- Mixin layers are an interesting programming language concept
  - they can occur in different flavors across different OO languages
  - they offer a practical way to implement collaboration-based designs