

Subjectivity

A central goal of domain modeling is to create a design for a *family* of systems

- not all systems of a domain have the same interface

How is it possible to generate software systems with *customized* interfaces from components that have *standardized* interfaces?

In this lecture, we explain:

- how this apparent contradiction is resolved by a fundamental concept of software design called *subjectivity*
- OO implementations of subjectivity
- GenVoca implementations of subjectivity

Domain Modeling

Domain modeling identifies fundamental objects that arise in many or all systems of a domain

Consider the domain of textbook applications

- textbook objects are fundamental
- obvious attributes: **author**, **title**, **content**, **publisher**
- appropriate for finding textbooks on **author**, **title**, etc.
- not appropriate for warehouse (stock and volume info)
- not appropriate for manufacturing (materials info)

*data and operations encapsulated by an object
will vary from application to application
(i.e., from system to system) in a domain*

Subjectivity

Principle of subjectivity (Ossher and Harrison 1992)

- when modeling software domains, objects don't have single interfaces, but are described by a *family of related interfaces*
- appropriate interface is application-dependent (*subjective*)

see Ossher & Harrison *OOPSLA* papers '92, '93, '95

Subjectivity clearly relevant to software reuse, domain models, and (of course) generators

Relevance to Software Reuse

A software module is like a photograph:

- no perspective captures all aspects of an object
- perspective hides or skews different aspects
- analogously, a software module encodes a particular “view” or “perspective” of an object relative to the needs of a particular application
- *software for an object written for one application might not be reusable in another, even if both applications deal with the same conceptual object*

reason: applications have different “perspectives”

Relevance to Domain Modeling

Different domain models use one of two different (partial) solutions to model families of interfaces:

- *multiple inheritance* expresses combinatorial number of interfaces in a domain

doesn't capture combinatorial # of implementations

factored libraries Biggerstaff ICSR 1994

subjectivity Ossher and Harrison OOPSLA '92, '93, '95

- *ignore the problem*: define systems and components with “standardized” interfaces

different systems and components are indistinguishable except for performance- and feature-related metrics

effective solution for addressing combinatorial numbers of implementations, but fails on interface variations

ex: OO frameworks, abstract factory design patterns, and (it would seem) GenVoca... so how to generalize???

Myth of Standardized Interfaces

GenVoca components are composable because they export and import standardized interfaces

- subjectivity says no single interface will suffice
- what does it mean for an interface to be “standardized”?
- how are “core” operations identified/included?
- how are “non-core” operations identified/excluded?

Ans: if GenVoca generators produce high-performance software, then *NO* operation can be excluded

The real story: GenVoca components don't export cast-in-concrete interfaces

- components encapsulate domain-specific feature
- export non-core, component-specific operations
- destroys pretense of immutable realm interfaces
- *interface of GenVoca-generated system changes with addition or removal of a component*

Example: P2 `size_of` layer

`size_of` maintains count of elements in a container

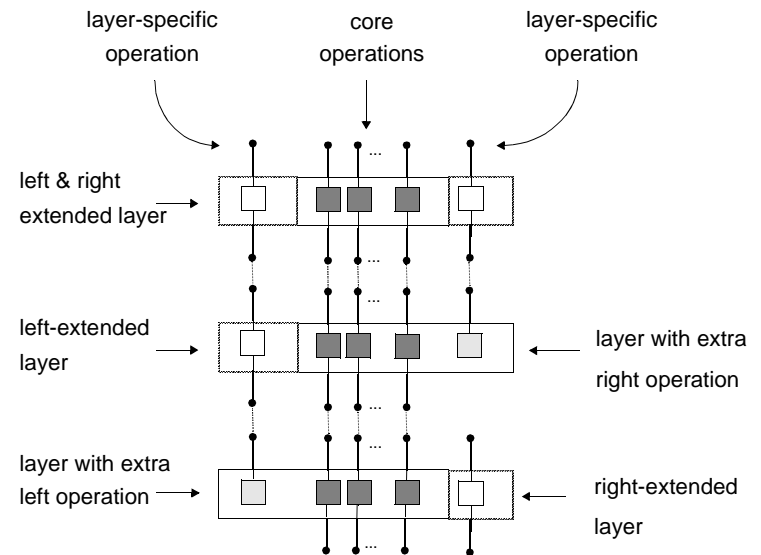
- exports `read_size()` operation to read count variable
- when `size_of` used in type equation, `read_size()` added to container interface

```
// interface with size_of component
class container
{
    container ();      // constructor
    ~container ();    // destructor
    ...
    int read_size (); // added by size_of
}
```

- when `size_of` removed from type equation, `read_size()` removed from container interface

What is going on??

The General Paradigm



- all components above are symmetric
- middle and bottom components extend their interface
- composition triggers extension of all components

General Paradigm (Continued)

What's going on??

- in layered systems, lower layers can export operations that only they understand
- these operations are propagated through higher layers through the top of the system
- thus, layers “expand” upon instantiation
- subjectivity explains this phenomena

GenVoca components really don't have single interfaces — their instances can export any one of a family of related interfaces

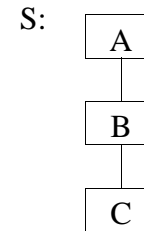
When GenVoca components are composed, their interfaces are automatically adjusted to a standard that is type equation-specific

Standardized interfaces in GenVoca doesn't mean cast-in-concrete; they are indeed subjective

- components with subjective interfaces are “freeze-dried” — they enlarge upon instantiation

Upcoming Topics

Central question:



$S = A[B[C]];$

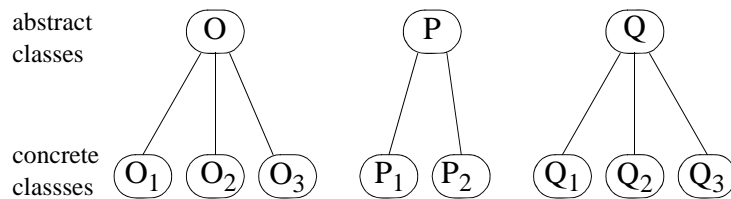
What is relationship of GenVoca components to classes?

Examine OO implementation techniques for GenVoca components with subjective interfaces

- OO frameworks
- Relationship of inheritance to layering, GenVoca
- GenVoca (non-OO) implementations

OO Frameworks

A *framework* is a carefully crafted set of abstract classes with multiple concrete class implementations



Realm interface defined by tuple of abstract classes

- realm \mathbf{R} interface = (O, P, Q)
- suppose realm membership: $\mathbf{R} = \{ \mathbf{x}, \mathbf{y}, \mathbf{z} \}$

GenVoca component defines a tuple of concrete classes:

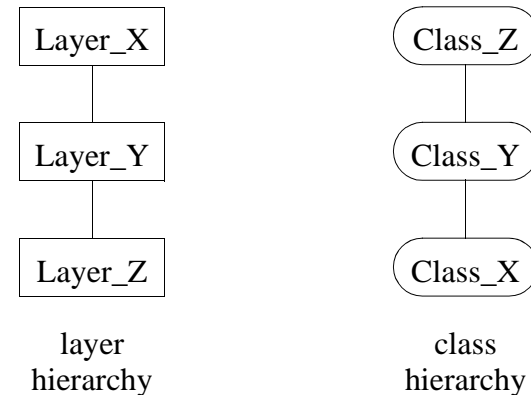
- component $\mathbf{x} = (\mathbf{O}_1, \mathbf{P}_1, \mathbf{Q}_1)$
- component $\mathbf{y} = (\mathbf{O}_2, \mathbf{P}_1, \mathbf{Q}_2)$
- component $\mathbf{z} = (\mathbf{O}_3, \mathbf{P}_2, \mathbf{Q}_3)$

Frameworks are not enough — don't explain subjectivity & operation propagation among components

Key Idea: Inheritance and Layering

An inheritance (subclassing) hierarchy is an “inverted” layer hierarchy if:

- layers are symmetric
- layers encapsulate single class

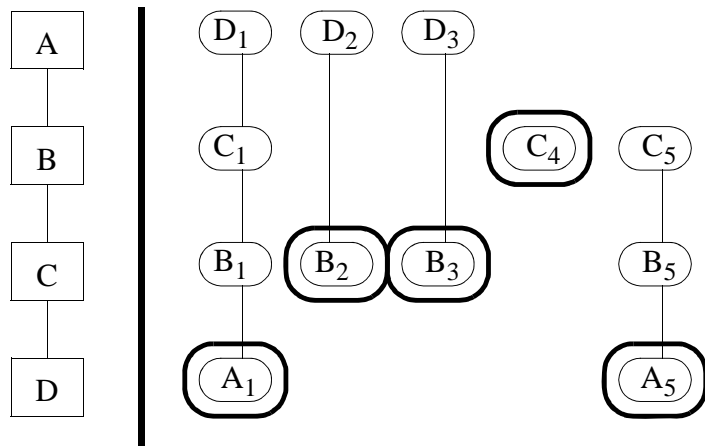


Note:

- *both express sequence of refinements*
- *both propagate operations*

Relationship to GenVoca

GenVoca components encapsulate consistent refinement of multiple classes:



Note:

- layer D encapsulates classes D_1 , D_2 , D_3
- layer C encapsulates classes C_1 , C_4 , C_5
- layer B encapsulates classes B_1 , B_2 , B_3 , B_5
- layer A encapsulates classes A_1 and A_5

Relationship (Cont)

Additional notes:

- *composition of GenVoca components create progressively broader and deeper forests of inheritance hierarchies*
- application classes are terminals of inheritance hierarchies
- non-terminal classes are *intermediate derivations* of application classes

Insights:

- GenVoca components encapsulate stereotypical refinements of classes in a domain
- typical inheritance hierarchies fix (cast-in-concrete) the order in which refinements are applied

GenVoca components allow much greater generality: they permit flexible ordering/reordering of refinements

Static v.s. Dynamic Compositions

Components may be composed

- *statically* — at application generation/compilation time
- *dynamically* — at application run-time

Van Hilst showed for *static* compositional implementations:

- implement components as *mixins* — classes whose superclasses are specified via *parameters*

```
template <class superclass>
class A1 : public superclass { ... }
```

```
template <class superclass>
class A5 : public superclass { ... }
```

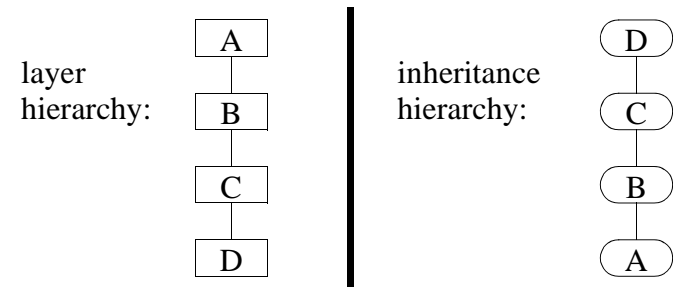
- compositions tend to be ugly because one has to compose individual classes:

```
typedef A1< B1< C1< D1 > > > BoldClass1;
typedef B2< D2 > BoldClass2;
typedef B3< D3 > BoldClass3;
typedef C4 BoldClass4;
typedef A5< B5< C5 > > BoldClass5;
```

Static Compositions (Cont)

Smaragdakis simplified Van Hilst's design by nested mixins and a standardized class naming scheme

First, represent every layer as a mixin:



```
template <class superclass>
class A : public superclass { ... }
```

```
template <class superclass>
class B : public superclass { ... }
```

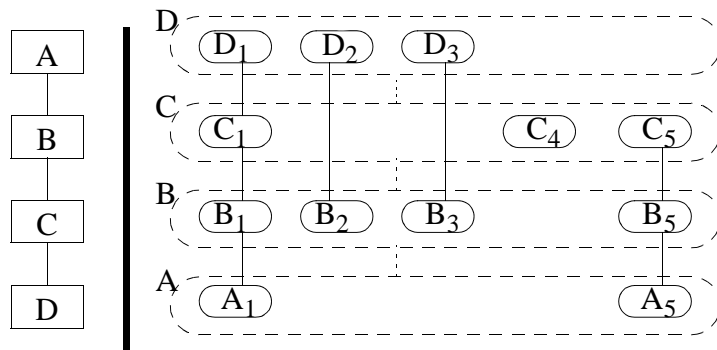
...

- note: **superclass** parameter is the realm parameter of a component!

Static Compositions (Cont)

Second, recognize that the resulting encapsulations deal with nested classes

- class/component **D** encapsulates nested classes **D₁**, **D₂**, **D₃**

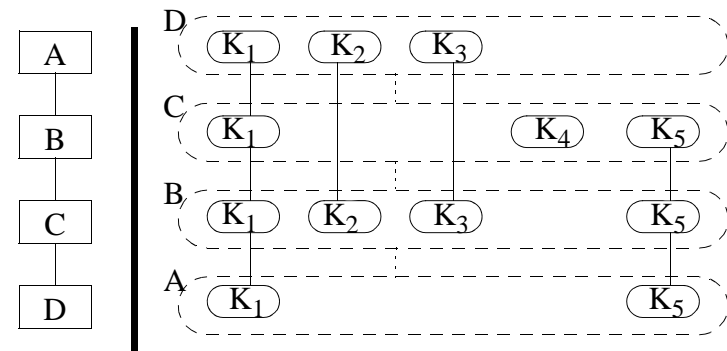


Third, note most nested classes are themselves mixins!

- ex: nested class **C::C1** is a subclass of **D::D1**
(*C::C1's superclass must be specified by parameter if C is to be freely composable with other components*)

Static Compositions (Cont)

Fourth, use standardized naming scheme for nested classes/nested mixins



Fifth, represent static-compositional layer directly:

```
template <class superclass>
class B : public superclass {
    class K1 : public superclass::K1 { ... };
    class K2 : public superclass::K2 { ... };
    class K3 : public superclass::K3 { ... };
    class K5 : public superclass::K5 { ... };
}
```

Static Compositions (Cont)

Sixth, compositions written like type equations:

```
typedef A< B< C< D > > > T;

// corresponds to type equation
// T = A[ B[ C[ D ] ] ] ;
```

What Smaragdakis's result tells us:

- static-compositional components rely on a sophisticated use of parameterization, nested classes, and design standardizations when expressed in OO form
- “type equation” really is a specification that manufactures a type or set of types (in this example, 5 types/classes)

In general, see Van Hilst and Smaragdakis papers for more details.

Dynamic Compositions

Components that are composed *dynamically* at application *run-time* requires different implementation techniques

- remember: one is creating inheritance hierarchies (i.e., changing the superclass of a class) at *run-time*
- presumably one can do this in (some) dynamically-typed languages or reflective programming languages, but at a *big* cost in performance

Interesting to note that the GenVoca generators that dealt with subjectivity have mostly focussed on dynamic compositions!

- Avoca/x-kernel, P2, Ficus, ...
- so how were inheritance hierarchies created dynamically?

Ans: lots of techniques... here's one...

Avoca — Communication Protocols — UArizona

Used fixed set of operations for:

- transmitting and receiving messages
- opening and closing sessions (sockets)
- `control()` like Unix `ioctl()`

encoded arbitrary number of component-specific operations

`control()` takes 2 arguments: identifier of operation, pointer to argument list

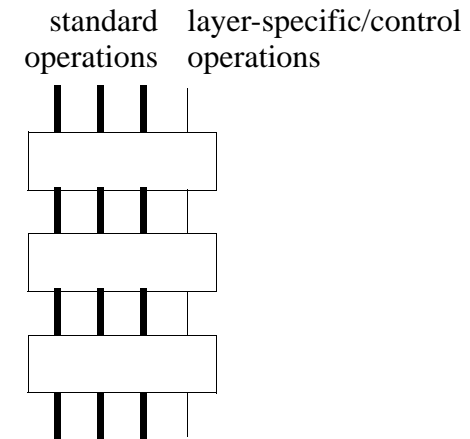
```
void control ( int op_id, void *arg_list )
{
    switch( op_id )
    { case ls_op1: // code for layer specific op1

        case ls_op2: // code for layer specific op2
        ...
        default:      // call layer below
            lower.control( op_id, arg_list );
    }
}
```

Avoca (Cont)

Components (*micro-protocols*) exported and imported syntactically the same interface

- “core” operations called directly
- “layer-specific” operations called through argument marshalling — i.e., via calls to `control(...)`
- components “polled” at run-time to see which component will process a `control` call



Dynamic Compositions (Cont)

Other techniques have less (zero) run-time overhead:

- pre-propagating operations through all components, so that only pure compositions (without inheritance) are needed

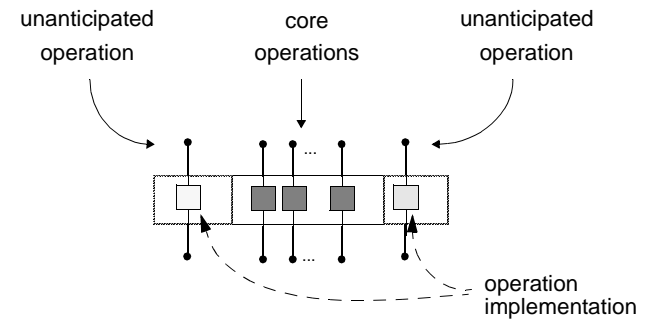
Batory and Geraci paper surveys implementations of dynamic compositions in 4 different GenVoca generators

There is another OO implementation issue ...

- switch gears to consider another (related) topic

Methods for Unanticipated Operations

Question: how are implementations for unanticipated operations supplied on a per layer basis?



- using inheritance, classes typically can't modify operations of superclasses that they don't know about

if the superclass of a class is specified via a parameter, there's potentially lots of superclass operations a class won't know about

- answer: remember that GenVoca components are *refinements*...

First Order Refinements

0th Order refinements:

- *data refinement* — add data members to classes
- *operation refinement* — algorithms for class operations

1st Order refinements:

- utilize metadata (symbol table information) about a class
- ex: what are the operations supported by a class?
- what are their type signatures?
- what are the data members of a class?

GenVoca components encapsulate *both*

- 0th order and
- 1st order refinements

⇒ 1st order refinements provide implementations for unanticipated operations

1st Order Refinement Example

P3 **monitor** refinement/component:

- serializes access to a container by wrapping **wait()** and **signal()** calls around cursor and container operations

- 0th order data refinement:

add semaphore **sem** to **container** class

- 1st order refinement of **container** operations

for each operation **op(...)** of **container** class:

```
container_operation ( ... )
{
    sem.wait();
    lower_container.container_operation (...);
    sem.signal();
}
```

1st Order Refinement (Cont)

- 1st order refinement of `cursor` operations

for each operation `op(...)` of `cursor` class:

```
cursor_operation ( ... )
{
    cont.sem.wait();
    lower_cursor.cursor_operation (...);
    cont.sem.signal();
}

// slightly different than container
// operation wrappers
```

In general, “For each `op(...)` of class `c`” typical of 1st order refinements in GenVoca components...

Implies that we need ability for classes to interrogate the set of methods of its superclass and to define generic actions for “unanticipated” methods

1st Order Refinements = Method Wrappers

CLOS (Common Lisp Object System)

- *before and after* methods (B & A)

```
wait()   ← before method
do-op()  ← primary method
signal() ← after method
```

- B & A defined for inheritance hierarchy of *operations*

```
foo( real )
  ↓
foo( rational )
  ↓
foo( int )
```

- CLOS is reflective, so it could be modified to handle class-based (not operation-based) wrappers

Method Wrappers (Continued)

Closest known match is IBM's SOM

see Forman and Danforth papers OOPSLA 1994-95

- elegant model for defining method wrappers for all operations of a class

classes have explicit meta-classes, where B & A methods are defined

- doesn't solve our problem: SOM propagates wrappers to subclasses (which isn't what we want...)



note: don't want wrappers to be inherited

Conclusions — Lessons Learned

- *tenet of subjectivity: when modeling families of applications, objects do not have single interfaces, but instead have a family of related interfaces*

standardized interfaces \neq cast-in-concrete interfaces;
authors never know “real” component interface;
components have instance-adjustable interfaces

- *composing GenVoca components causes the creation of progressively broader and deeper forests of inheritance hierarchies*

component composition is a compact notation for specifying complex inheritance hierarchies

- *components encapsulate 1st order refinements (method wrappers) to define methods for unanticipated operations*

related to, but not the same as, method wrappers
CLOS and SOM

Further Reading

- D. Batory and B.J. Geraci, "Validating Component Compositions and Subjectivity in GenVoca Generators", *IEEE Transactions on Software Engineering*, February 1997, 67-82.
- D. Batory, B. Lofaso, and Y. Smaragdakis, "JTS: A Tool Suite for Building GenVoca Generators", *ICSR 1998*.
- G. Booch, *Object Solutions: Managing the Object-Oriented Project*, Addison-Wesley, 1995.
- S. Danforth and I. Forman, "Reflections on Metaclass Programming in SOM", *OOPSLA 1994*, 440-452.
- I.R. Forman, S. Danforth, and H. Madduri, "Composition of Before/After Metaclasses in SOM", *OOPSLA 1994*, 427-439.
- P. Graham, *ANSI Common Lisp*, Prentice Hall, 1995.
- W. Harrison and H. Ossher, "Subject-Oriented Programming (A Critique of Pure Objects)", *OOPSLA 1993*, 411-428.
- W. Harrison, H. Ossher, R.B. Smith, and D. Ungar, "Subjectivity in Object-Oriented Systems: Workshop Summary", *Addendum to OOPSLA 1994*.
- G. Kiczales, J. des Rivieres, and D.G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991.
- H. Ossher and W. Harrison, "Combination of Inheritance Hierarchies", *Proc. OOPSLA 1992*, 25-40.
- H. Ossher, et al., "Subject-Oriented Composition Rules", *OOPSLA 1995*, 235-250.
- Y. Smaragdakis and D. Batory, "Implementing Reusable OO Components", *ICSR 1998*.
- Y. Smaragdakis and D. Batory, "Implementing Layered Designs with Mixin Layers", *ECOOP 1998*.

- M. Van Hilst and D. Notkin, "Using Role Components to Implement Collaboration-Based Designs", *OOPSLA 1996*, 359-369.
- M. Van Hilst and D. Notkin, "Using C++ Templates to Implement Role-Based Designs", *JSSST Int. Symposium on Object Technologies for Advanced Software*, Springer-Verlag 1996, 22-37.
- M. Van Hilst and D. Notkin, "Decoupling Change from Design", *SIGSOFT 1996*.