

Inheritance

CSE 58x
Winter 2000
Mark Morrissey

Many Faces

- Inheritance is a loaded term
- Implementation Inheritance, Subclassing
- Interface Inheritance, Subtyping
- The Principle of Substitutability
- Subclassing and Subtyping are often *both* implied
 - They are different and should be distinguished
- Substitutability cannot be guaranteed by language or compiler
 - But may be the intent of the presence of inheritance
 - Simula 67 v. Smalltalk

Intent

- Some languages intend for subclassing to yield substitutability - C++ and Java are examples
- Some languages have no such intent - Smalltalk, Eiffel
- Many languages do not distinguish between subtyping and subclassing
- Some languages provide separate inheritance hierarchies for classes and types - Java
- Some languages only provide subtyping - OMG IDL
- Inheritance must be discussed in relation to the intent of the language and the type of inheritance or it is meaningless

Implementation Inheritance

- Implementation inheritance can be problematic
- One of the main uses of inheritance is to extend, modify or replace existing behavior.
- For languages with multiple inheritance, there are many problems, especially when you can inherit the same method from multiple sources - which one do you choose?
- Implementation inheritance must be done carefully, fully understanding the inherited methods, to avoid problems

Multiple Inheritance

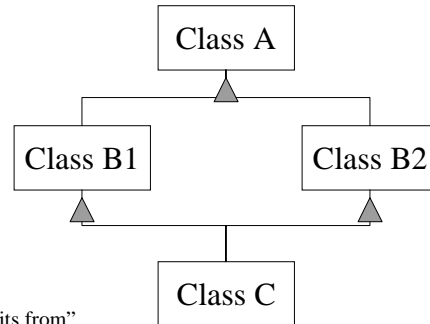
- Covers both interface and implementation inheritance
- Only problematic for implementation inheritance
 - So you identical interface specifications multiple times, what's the harm?
- Driving forces:
 - Merge interfaces from multiple sources
 - Merge implementations from multiple sources
- Merging interfaces is a clean way to establish compatibility with multiple, independent, contexts
 - Maintains extensibility and substitutability independence also

Multiple Inheritance

- Mixing implementations is more problematic
 - When inheritance hierarchies are fully independent and there are no name clashes, the world is good.
 - This is nearly impossible, in practice
 - Semantic problems can result when superclasses are not fully disjoint

Diamond Inheritance Problem

- Two problems are lurking, what are they?



CSE 58x

Building Software Systems with Components

7

State

- Do classes B1 and B2 get their own copy of class A's state?
- If so, then they operate on the state independently
- If not, then changes to the state in B1 is visible in B2 - breaks object encapsulation
- If state is not shared, which state does class C inherit?
 - If random or not defined, then the consistency of Class C is in danger
- Class C cannot itself fix this problem unless it knows and understands the implementation of both B1 and B2

CSE 58x

Building Software Systems with Components

8

Behavior

- The behavior of class A is defined by its method implementations
- If B1 or B2 (or both) override some of A's implementation, which methods should make it to class C
- It is possible that class C wants to pick and choose depending on the situation
- In some languages, when there are multiple choices, the programmer must specify the ordering
 - Assumes detailed knowledge of the implementation
 - Abstraction and encapsulation suffer

Mixins

- Mixins are a powerful type of multiple implementation inheritance if used with discipline
- Class inherits interfaces from one superclass and implementations from several superclasses
- The implementations superclasses are called “mixins” as they are used to mix implementation fragments in to the new class
 - Think “cake mix”

FBC

- The fragile base class problem exists for both single and multiple inheritance
- Fundamentally, how can a class evolve without breaking independently developed subclasses?
- There is more than one problem hiding here

FBC

- According to Microsoft:
 - “The problem is that the “contract” between components in an implementation hierarchy is not clearly defined. When the parent or child component changes its behavior unexpectedly, the behavior of related components may become undefined.”
- According to IBM:
 - “By completely encapsulating the implementation of an object, SOM overcomes what Microsoft refers to as the “fragile base class problem”, i.e., the inability to modify a class without recompiling clients and derived classes dependent on that class.”

FBC

- The first refers to the problem of semantics - maintaining compatibility in the face of implementation changes
- The second refers to the problem of syntax - maintaining binary interface compatibility from release to release

Syntactic FBC

- Concerned with binary compatibility of compiled classes with new binary releases of superclasses
- Syntax changes in the superclass should not necessitate a recompilation in the subclass
- Moving methods within the inheritance hierarchy should be independent
- Adding new methods to a superclass or reordering methods should be independent
- Initializing dispatch tables late (load or invocation time) solves this problem

Semantic FBC

- Concerned with maintaining integrity of the subclass in the presence of different versions and evolution of the implementation of its superclasses
- Occurs when the semantics of an implementation in the superclass changes
- For example, modifying the superclass implementation so that a data item is directly manipulated rather than using accessor functions can break overridden accessor functions in the subclass
- A subclass can interfere with the implementation of its superclass!

Forwarding

- Forward, or object composition, is a powerful alternative to implementation inheritance
- Forwarding suffers from the fact that each object in the composition has its own notion of 'self'
- Forwarded calls cannot be partially implemented and rely on the outer object without arranging for the outer object to pass a reference to the inner object, i.e. there is no 'super'
- Forwarded calls can be determined dynamically, as late as invocation time.

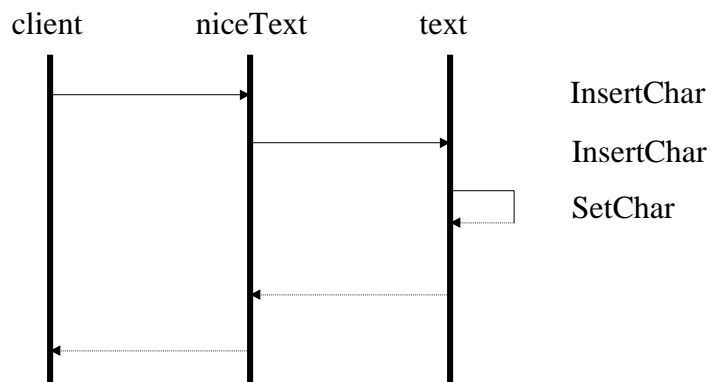
Delegation

- Delegation is not the same as forwarding
- Delegation occurs when the message send is self-recursive
- Subsequent delegated sends are sent back to the original delegator

CSE 58x

Building Software Systems with Components

17



Forwarding

CSE 58x

Building Software Systems with Components

18

