Inheritance (Chapter 7)

Prof. Dr. Wolfgang Pree

Department of Computer Science
University of Salzburg
cs.uni-salzburg.at

UNIVERSITÄT SALZBURG

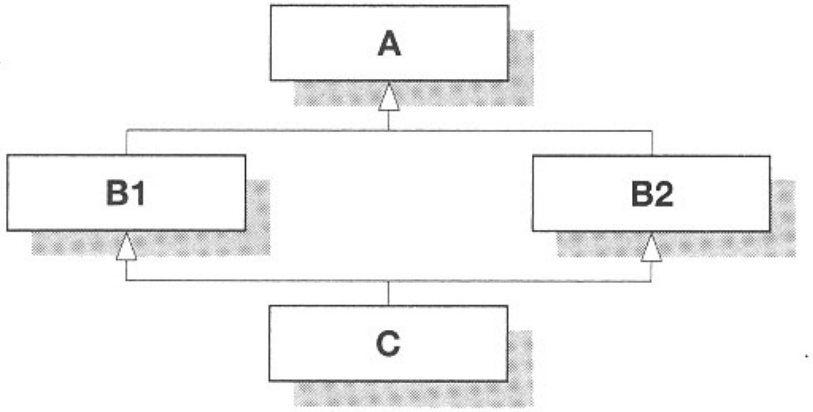
Inheritance – the soup of the day?

- Inheritance combines three aspects: inheritance of implementation, inheritance of interfaces, and establishment of substitutability (substitutability is not required – although recommended).
- Inheritance:
  - subclassing – that is, inheritance of implementation fragments/code usually called implementation inheritance;
  - subtyping – that is, inheritance of contract fragments/interfaces, usually called interface inheritance; and
  - Promise of substituatability.

Multiple inheritance (1)

- In principle, there is no reason for a class to have only one superclass.
- Two principal reasons for supporting multiple inheritance
  - to allow interfaces from different sources to be merged, aiming for simultaneous substitutability with objects defined in mutually unaware contexts.
  - to merge implementations from different sources.
- Multiple interface inheritance does not introduce any major technical problems beyond those already introduced by single interface inheritance.

Multiple inheritance (2)



- Approaches to the semantics of multiple inheritance usually stay close to the implementation strategy chosen.
- Java supports multiple interface inheritance but is limited to single implementation inheritance. The diamond inheritance problem is thus elegantly solved by avoidance without giving up on the possibility of inheriting multiple interfaces.

Mixins

- Multiple inheritance can be used in a particular style called mixin inheritance. A class inherits interfaces from one superclass and implementations from several superclasses, each focusing on distinct parts of the inherited interface.

```
interface Window {
  // whole window:
  void drawWindow ();
  // window borders:
  void drawBorders ();
  void handleMouse (Event ev);
  // window contents:
  void drawContents ();
  Rect getVisibleSection ();
  Rect getContentsBox ();
  void scrollTo (Rect newVisible);
}
```

Mixins (2)

```
abstract class StdWindowShell implements Window
  void drawWindow () {
    drawBorders; drawContents;
  }
}
abstract class MotifWindowBorders implements Window {
  void drawBorders () {
    ... // draw title bar
    Rect visible = getVisibleSection();
    Rect total = getContentsBox();
    ... // draw scroll bars (compute thumbs using visible and total boxes)
  }
  void handleMouse (Event ev) {
    ...
    if (scroll bar thumb moved) {
      ... // compute new visible region
      scrollTo(newVisible);
    }
    ...
  }
}
abstract class TextWindowContents implements Window {
  void drawContents () { ... }
  Rect getVisibleSection () { ... }
  Rect getContentsBox () { ... }
  void scrollTo (Rect newVisible) { ... }
}
```

The fragile base class problem

- The question is whether or not a class can evolve – appear in new releases without breaking independently developed subclasses.
- Potentially tight dependency of independently developed subclasses on their base class is called the fragile base class problem.

The syntactic fragile base class problem

- The syntactic fragile base class problem (syntactic FBC) is about binary compatibility of compiled classes with new binary releases of superclasses.

- A class should not need recompilation just because purely “syntactic” changes to its superclasses’ interfaces have occurred or because new releases of superclasses have been installed.

The semantic fragile base class problem

- The essence of the semantic FBC problem is how can a subclass remain valid in the presence of different versions and evolution of the implementation.
- FBC problems:
  - a compiled class should remain stable in the presence of certain transformations of the inherited interfaces and implementations.
  - the separate treatment and the infrastructural complexity required to solve the syntactic facet alone is costly and must be fully justified.

Inheritance – more knots than meet the eye

- Implementation inheritance is usually combined with selective overriding of inherited methods – some of the inherited methods being replaced by new implementations.
- The separation of models and views, as used in the previous examples, can be an overkill in cases where complexity is low enough and where exactly one view per model is required. Simple controls fall into this category.
- Invocation of an overridden method from within a class is similar to invoking a callback from within a library. Call recursion freely spans class and subclass in both directions.
- In a class, every method can potentially call any other method. In combination with subclasses and overriding any method can become a callback operation from any other method.

Approaches to disciplined inheritance (1)
The specialization interface

- A subclass can interfere with the implementation of its superclasses in a way that breaks the superclasses.
- The special interface between a class and its subclasses is called the specialization interface of that class. Distinguishing between the client and the specialization interface is important for approaches supporting implementation inheritance.
- The specialization interface of a C++ or Java class is the combination of the public and the protected interface. The client interface consists of only the public (non-protected) interface.

Approaches to disciplined inheritance (2) Typing the specialization interface

- Overriding methods needs to be done carefully to ensure correct interactions between a class and its subclasses.
- Type system approach
  - to declare statically which other methods of the same class a given method might depend on.
  - where dependencies form cycles, all the methods in a cycle together form a group.
- If a method needs to call another method, it either has to be a member of the called method's group or of a higher layer's group.

Typing the specialization interface (2)

```
specialization interface Text {
   state caretRep
   state textRep
   abstract posToXCoord
   abstract posToYCoord
   abstract posFromCoord
   concrete caretPos   { caretRep }
   concrete setCaret   { caretRep }
   concrete max   { textRep }
   concrete length   { textRep }
   concrete read   { textRep }
   concrete write   { textRep, caretPos, setCaret }
   concrete delete   { textRep, caretPos, setCaret }
   concrete type   { write, caretPos, setCaret }
   concrete rubout   { delete, caretPos, setCaret }
}
```

Behavioral specification of the specialization interface

- Most semantic problems specific to implementation inheritance are related to problems of re-entrance caused by self-recursion.
- The key requirement to be satisfied by any disciplined use of implementation inheritance would be the preservation of modular reasoning. It should be possible to establish properties of a class formally without a need to inspect any of the subclasses. Modular reasoning is importance for extensible systems, in which the set of subclasses of a given class is open.
- Layered systems are well suited to modular reasoning.
- Attempts at tight semantic control over classes and implementation inheritance naturally lead to a model that is much closer to object composition than to traditional class composition, that is, implementation inheritance.

Reuse and cooperation contracts (1)

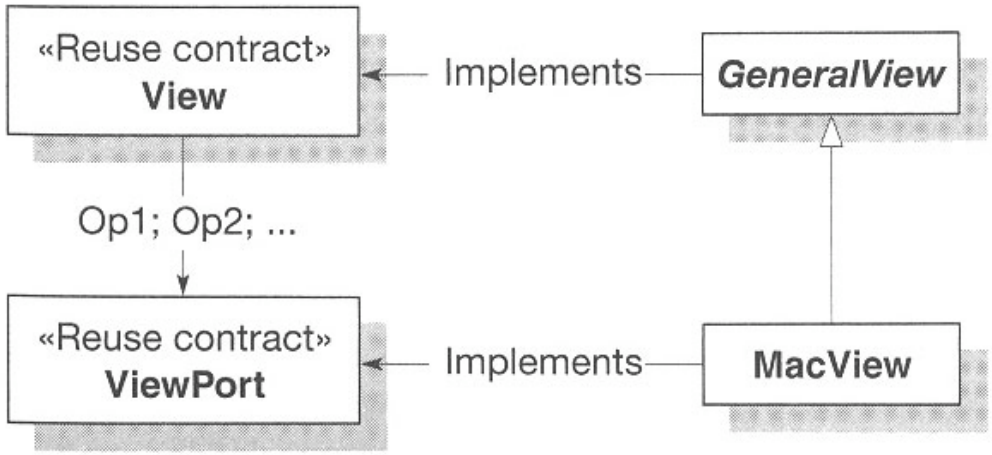
- Annotated interface are called a reuse contract with the intention that classes are reusable assets.
- Reuse contracts: two parties
  - classes
  - and their subclasses
- A reuse contract is just a specialization interface specification with less information.

```
reuse contract Text {
  abstract
    posToXCoord
    posToYCoord
    posFromCoord
  concrete
    caretPos
    setCaret
    max
    length
    read
    write   { caretPos, setCaret }
    delete   { caretPos, setCaret }
    type   { write, caretPos, setCaret }
    rubout   { delete, caretPos, setCaret }
}
```

Reuse and cooperation contracts (2)

- Real innovation of the reuse contract – a set of modification operators:
  - concretization (its inverse is abstraction) – replace abstract methods by concrete methods (replace concrete methods by abstract methods);
  - Extension (its inverse is cancellation) – add new methods that depend on new or existing methods (remove methods without leaving methods that would depend on the removed methods);
  - Refinement (its inverse is coarsening) – override methods, introducing new dependencies to possibly new methods (removing dependencies from methods, possibly removing methods that now are no longer being depended on).

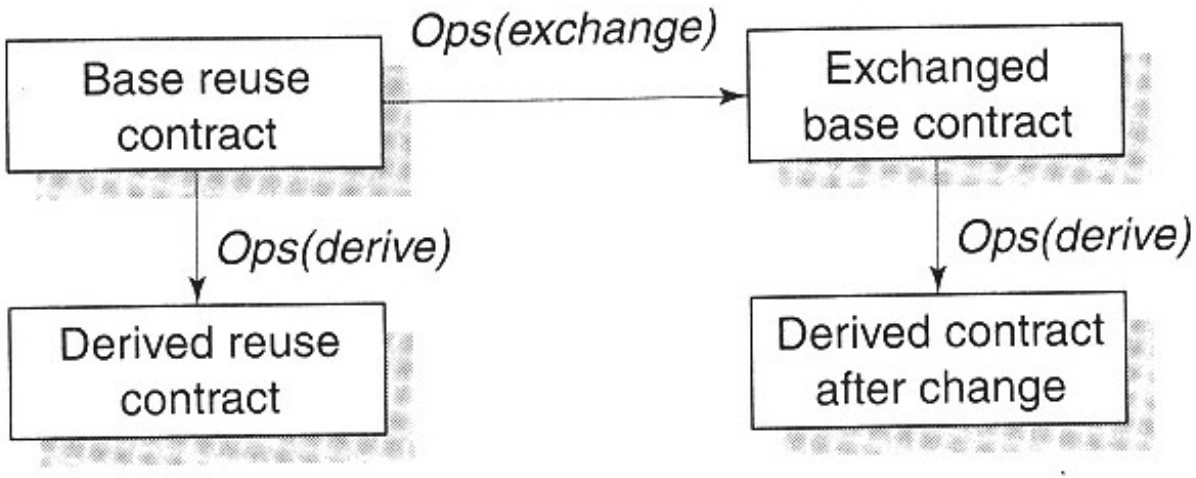
Reuse and cooperation contracts (3)



- Cooperation contracts that build on the ideas of reuse contracts and also those of specialization interfaces.
- Fully formal approach, cooperation contracts are closed at the metalevel through a metaobject protocol of the implementation environment.

Representation invariants and method refinements

- Stephen Edwards demonstrated that the overriding of individual methods in a method group is permissible if the subclass maintains the representation invariant of the group variables.



UNIVERSITÄT SALZBURG

Disciplined inheritance to avoid fragile base class problems

- Avoid the semantic fragile base class problem.
- Construction of subclasses based on a superclass specification such that the subclasses remain valid if the superclass implementation is changed (while still meeting its specification).
- Focus on eliminating the introduction of new cyclic method dependencies between the superclass and its subclasses when moving from the original to a new version of the superclass implementation.
- Superclass instance variables be private; subclasses do not introduce new instance variables. They enabled a rigorous formalization and formal proof that is in itself an interesting contribution.

Creating correct subclasses without seeing superclass code (1)

- Inverse problem of semantic fragile base class.
- The FBC problem assumes existing but unavailable subclasses that need to be preserved in the face of requested base class changes.
- Three parts class specification, that ensure that a subclass can be safely created without requiring access to the source code of the base clas. These first two parts split the class specification into a public and a protected part. The protected part reveals information such as invariants over protected variables and conditions on protected methods. The third part is unusual in that it is based on an automatic analysis of the initial source code of the base class.

Creating correct subclasses without seeing superclass code (2)

- Class library implementer should ensure that for each class in the library:
  - Methods should not directly access any instance variables from unrelated classes;
  - Methods should not call the public methods of an object while temporary side-effects have not been restored in that object; and;
  - Overriding methods should refine the method being overridden.

Creating correct subclasses without seeing superclass code (3)

- For each class that can be subclassed:
  - There should be no mutually recursive methods;
  - Methods should not pass a self-reference as an argument when calling a method of a related class;
  - Methods should not self-call or super-call non-pure public methods;
  - Non-public methods should not call the public methods of a related class; and
  - If a protected concrete instance variable of type T cannot hold all values of type T, then its domain should be described in a protected invariant.

Creating correct subclasses without seeing superclass code (4)

- A subclass implementer should ensure that, for each subclass:
  - Additional side-effects are avoided when overriding non-public methods that modify superclass instance variables;
  - No group of mutually recursive methods is created that involves a method of a superclass;
  - Overriding methods refine the overridden methods;
  - The protected invariant of the subclass implies the protected invariant of the superclass if superclass instance variables have been exposed to the subclass or other classes.

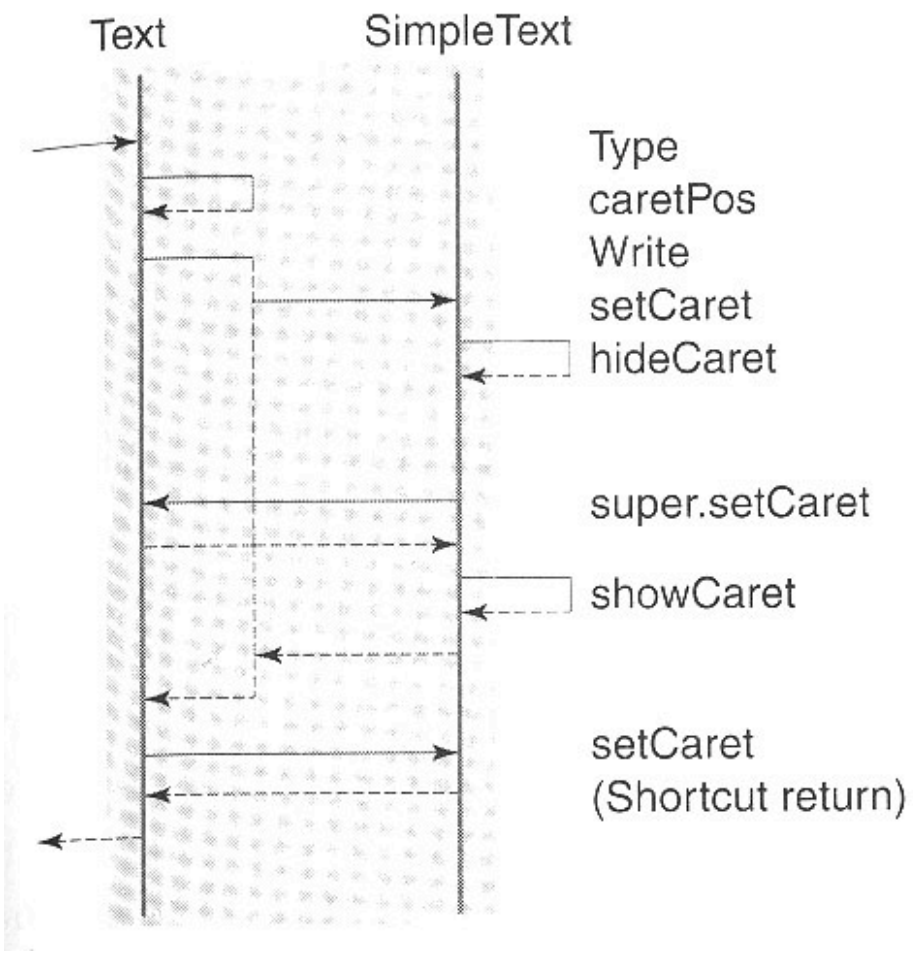
From class to object composition (1)

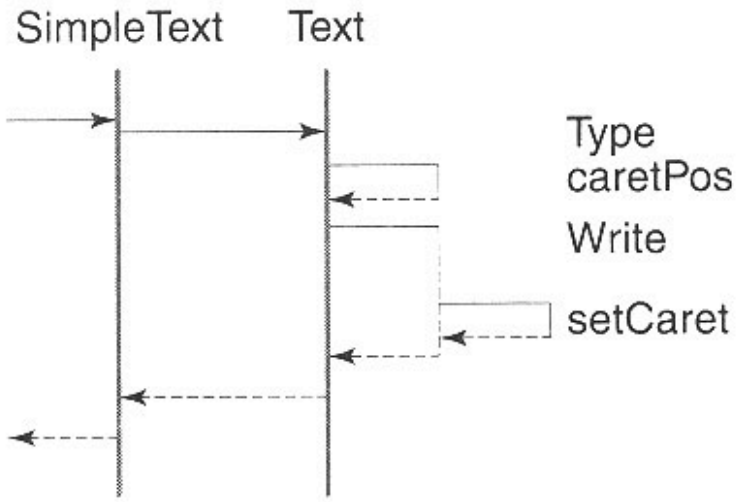
- In an inheritance-based scheme, evolutionary improvement of parent classes automatically improves subclasses.
- Modification of parent classes leads to an automatic modification of subclasses.
- Object composition is a simpler form of composition than implementation inheritance.
- Whenever an object does not have the means to perform some task locally, it can send messages to other objects, asking for support. If the helping object is considered a part of the helped object, this is called object composition. An object is a part of another one if references to it do not leave that other object. The part object is called an inner object because it can be seen as residing inside its owning object, the outer object.

From class to object composition (2)

- Sending a message on from one object to another object is called forwarding.
- The combination of object composition and forwarding comes close to implementation inheritance.
- An outer object does not reimplement the functionality of the inner object when it forwards messages. It reuses the implementation of the inner object.
- If the implementation of the inner object is changed, then this change will “spread” to the outer object.
- The difference between object composition with forwarding and implementation inheritance is called “implicit self-recursion” or “possession of a common self”.
- Object composition is dynamic. An inner object can be picked at the time of creation of an outer object.

From class to object composition (3)



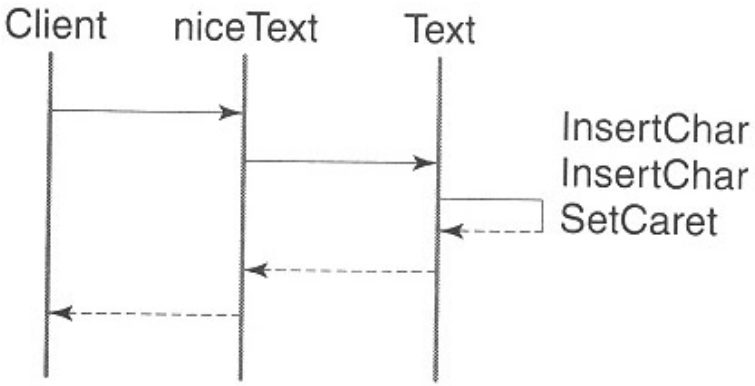


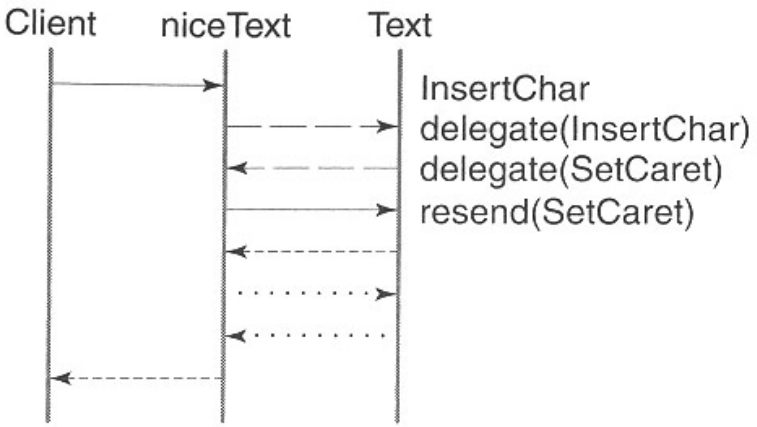
UNIVERSITÄT SALZBURG

Forwarding versus delegation (1)

- Objects composed using object references and forwarding of messages lack the notion of an implicit common “self”.
- If an object was not designed for composition under a common identity, it cannot be used in such a context.
- Another way to close the gap between forwarding-based object composition and implementation inheritance-based class composition. Instead of making inheritance more dynamic, forwarding could be strengthened. The resulting approach to message passing is called delegation.
- Each message-send is classified either as a regular send (forwarding) or a self-recursive one (delegation). Whenever a message is delegated (instead of forwarded), the identity of the first delegator in the current message sequence is remembered. Any subsequently delegated message is dispatched back to the original delegator.

Forwarding versus delegation (2)





- Delegation requires one further concept. Besides being able to send to the current “self” a method implementation should be able to call the base method in the object down the delegation chain.