Inheritance, substitution, subclasses and subtypes

Beatrice Åkerblom
beatrice@dsv.su.se
Modularity

❖ Modular Decomposability

~ A software construction method satisfies Modular Decomposability if it helps in the task of decomposing a software problem into a small number of less complex subproblems, connected by a simple structure, and independent enough to allow further work to proceed separately on each item.

❖ Modular Composability

~ A SC method satisfies Modular Composability if it favours the production of software elements which may be freely combined with each other to produce new systems, possibly in an environment different from the one in which they were initially developed.
Modularity, cont’d.

❖ **Modular Understandability**
- A SC method favours Modular Understandability if it helps produce software in which a human reader can understand each module without having to know the others, or, at worst, by having to examine only a few of the others.

❖ **Modular Continuity**
- A SC method satisfies Modular Continuity if, in the resulting software architectures, a small change in the problem specification will trigger a change of just one module, or a small number of modules.
Modularity, cont’d.

♦ Modular Protection

~ A SC method satisfies Modular Protection if, in the resulting architectures, the effect of an abnormal condition occurring at run-time in a module will remain confined to that module, or at worst will only propagate to a few neighbouring modules.
Classes aren’t Enough

❖ Classes provide a good modular decomposition technique.

❖ They possess many of the qualities expected of reusable software components:
  ❖ they are homogenous, coherent modules
  ❖ their interface may be clearly separated from their implementation according to information hiding
  ❖ they may be precisely specified

❖ But more is needed to fully achieve the goals of reusability and extedibility
Classes aren’t Enough, cont’d.

* If we use classes to combine the module and the type aspects in one construct, it is tempting to combine the reusability mechanisms of both to give the reusability possibilities for classes.

  ~ The possibility for a module to directly rely on entities defined in another module.

  ~ The concept of subtype, whereby a new type may be defined by adding new properties to an existing type.
What is Inheritance?

- Inheritance gives us the possibility to create something that is partly or totally the same as something else
  - Child classes as extension of an already existing class definition
  - Child class as a specialisation of an already existing class definition
  - Enables subtypes to produced using an already existing supertype
Different Views Imply Different Semantics.

- Model classification hierarchies in an application domain.
  - Subclasses define subtypes or specialisation of their superclasses.
  - The domain itself provides the rules.
- Incidental inheritance -- occurs when when a superclass possess a property which is inherited by a subclass even though it's irrelevant for the subclass.
- Implementation inheritance -- a way of implementing objects by sharing.
  - Unfortunately it can be extremely difficult to determine when a subclass should be defined.
  - Sharing specifications or code.
Inheritance

- Child classes are always an *extension* of the properties associated with the parent class
- The child class has all the properties of the parent class and other properties as well
- A child class is more specialised
Reasons to use inheritance

❖ Code reuse:
   ~ a child class can inherit behaviour from a parent class -- there’s no need to rewrite the code

❖ Concept reuse:
   ~ a child class overrides behaviour defined in the parent -- no code is shared but the child and parent share the definition of the method
Forms of Inheritance

- Inheritance for
  - specialisation (subtyping) -- the new class is a specialised form if the parent class
  - specification -- to guarantee that classes maintain a certain interface
  - extension -- adding totally new abilities to the child class
  - limitation -- the behaviour of the child class is more limited than the behaviour of the parent class (violates the principle of substitution)
  - variance -- when two or more classes have similar implementations, but no relationships between the abstract concepts exist
  - combination -- multiple inheritance
Benefits of Inheritance

❖ Easier to get a picture of the system model
❖ Possibility to create subtypes
❖ Makes code reuse easier
❖ Enables polymorphism
❖ Enables controlling reuse
Disadvantages of Inheritance

- Inheritance can destroy existing systems
- Inheritance doesn’t work the way “it should”
- Unwanted, automatic inheritance can be annoying and memory consuming
- Multiple inheritance can lead to serious problems
- Separation of code into different files makes reading it more difficult
- Slows down execution speed
- Increases program size
- Increases program complexity
Subclass, Subtype and Substitution

In a statically typed, object-oriented language:

❖ Instances of a child class must possess all data members associated with parent class
❖ Instances of a child class must implement all functionality defined for parent class

The instance of the child class can mimic behaviour of the parent class and should be \textit{indistinguishable} from an instance of the parent class if substituted in a similar situation
The Notion of Contract

- When defining a subtype one must obey the laws of contract
  - If an object of the subtype is to be used as if it were one of the supertype it cannot be more demanding and must at least provide the same result quality.

- The subtype:
  - must provide at least the same methods (name compatibility)
  - methods must accept the same type of arguments and provide the same type of result (interface compatibility)
  - must have a behaviour compatible with that of the supertype (behaviour compatibility)
Different Takes on Overriding Methods

❖ In Java and Smalltalk descendent methods control whether ancestor methods are called.

❖ In Beta, ancestor methods control whether descendent methods are called.

❖ In Eiffel, descendents can cancel or rename ancestor features.
Substitution

- $B <: A$ -- $B$ is a subtype of $A$
- Any expression of type $A$ may also be given type $B$
  - in any situation
  - with no observable effect
- Nominal subtyping or structural subtyping?
- Almost always:
  - reflexive (meaning $A <: A$ for any type $A$)
  - transitive (meaning that if $A <: B$ and $B <: C$ then $A <: C$)
Substitution

❖ Subtype has same attributes and behaviour as supertype

❖ An object can be used wherever an object from a supertype is expected.

❖ The Liskov substitution principle is a definition of subtype introduced by Barbara Liskov and Jeannette Wing in “Family Values: A Behavioral Notion of Subtyping”, 1993.

~ Let \( q(x) \) be a property provable about objects \( x \) of type \( T \). Then \( q(y) \) should be true for objects \( y \) of type \( S \) where \( S \) is a subtype of \( T \).
Substitution in DPLs and STLs

- The importance of the principle of substitution differs between dynamically typed and statically typed languages
  - in statically typed languages objects are (typically) characterised by their class
  - in dynamically typed languages objects are (typically) characterised by their behaviour
Substitution in a STL

* The method below will only operate on arrays of instances of the BaseballPlayer class, (or instances of subclasses of BaseballPlayer)

```java
public int sumOfWages( BaseballPlayer[] bs ) {
    int sum = 0;
    for ( int i=0; i < bs.length; ++i ) {
        sum += bs[ i ].wage( );
    }
    return sum;
}
```
Substitution in a DTL

\* In a dynamically typed language, you can send any message to any object, and the language only cares that the object can accept the message — it doesn't require that the object be a particular type.

```python
def sumOfWages( aList ):
    sum = 0
    for item in aList:
        sum += item.wage()
    return sum
```
Substitutability Paradox

- To create a subclass we simply have to create it as an extension of another existing class

```java
class SubClass extends SuperClass {
    // class definition here
}
```

- This will not result in any guarantees about the behaviour of the new class

- In statically typed class based OOPLs subclasses will be used as subtypes
Substitutability Paradox

❖ A child class inherits all data fields defined in the parent class

❖ A child class must recognise all the behaviours associated with the parent class, either by inheriting them or by overriding with preserved signature

❖ Therefore, an instance of a child class can be used in any situation where an instance of the parent class is expected
Is this a problem?

- To create a subclass that is not a subtype we have to redefine behaviour inherited from the parent class in such a way that it compromises some property expected by the parent class, but without changing method signatures...

  ~ This will probably not happen very often, but when it does it will not remain unnoticed for long
Subtype

- Subtyping is a relation on types such that if \( T' \) is a subtype of \( T \), then any object of type \( T' \) is also an object of type \( T \).
- The term subtype is used to refer to a subclass relationship in which the principle of substitution is maintained.
- A subclass relationship may or may not satisfy this principle of substitution.
- The exact definition of subtyping depends on the definition of type.
Subclasses vs. Subtypes

Palsberg & Schwartzbach, 1992

❖ A subtype S of a type T is a subset of the objects of T, possibly characterised by a type predicate that is more restrictive than P

~ Some languages (e.g. Ada) support explicit subtyping. Example: `subtype NATURAL is INTEGER range 0..INTEGER’MAX;`

~ Captures the “is-a” relation

❖ An object s of type S is an object of type T if S is a subtype of T

❖ A subclass D of a class C is a class that reuses the structural description and implementation of C in some specified way

~ Typically through inheritance, but the semantics of inheritance vary from language to language

~ Does not always result in a sound “is-a” relationship
Subclasses vs. Subtypes

Bruce, 1996

❖ Subclasses support reuse of code inside a class
  ~ Reuse of components

❖ Subtypes support reuse external to a class
  ~ Reuse of the context in which a class is used -- Given a subtype and its supertype, any operation that can be applied to an object of the supertype can be applied to an object of the subtype

❖ Subtyping deals with interfaces, not with implementations
  ~ There need not be any connection between types and classes
  ~ Objects of different classes can belong to the same type

❖ But most OO languages conflate types and classes as well as subtyping and inheritance
Different Kinds of Subtyping

❖ A subtyping mechanism is sound if and only if method implementations are protected by the types of its formal parameters

❖ This protection is used by the language to guarantee certain invariants about the dynamic behaviour of programs, e.g. the absence of certain run-time errors.

❖ A sound notion of subtyping must guarantee that sufficient protection is provided by the subtypes of the types of the formal parameters.

When a type is: | Subtyping becomes:
class + subclasses | subclassing
name compatibility | more methods
interface | conformance
behaviour | weaker preconditions
| stronger postconditions
When classes are used as types, the predicate properly requires that the object is an instance of a particular class or any of its subclasses.

This definition depends on the particular choice of subclassing mechanism.

When a class C is used as a type, which other classes are type compatible with C?

Increasing expressiveness of type predicative.
Specification Types versus Implementation Types

❖ Should we separate classes and types?
  ~ Should we use specification types or implementation types?

❖ When we base our type system on classes, then we require that all instances of a class must have the same type. This is not the case for specification types, where class relationships need have nothing in common with type relationships.
Specification Types versus Implementation Types

❖ When using implementation types we cannot substitute two different implementations of the same concept for each other (e.g. two different implementations of a stack).

❖ Furthermore, we cannot reflect if a class implements more than one type.
A major problem with specification types—in the form of interfaces—is that conceptually unrelated types may conform to each other.

For example, a type Cowboy with methods draw, move, and shoot conforms to a type Rectangle with methods draw and move.

Behavior types may seem to be an appropriate compromise; however, they may make type checking undecidable and offer several technical challenges.
Contravariance and Covariance

The strict match between signatures of methods in the supertype and its subtypes may be relaxed using the rules of contra and covariance.

**Type** Point
- SetX(Integer)
- SetY(Integer)
- GetX(): Integer
- GetY(): Integer
- MoveXY(Integer, Integer)

**Type** CPoint **SubtypeOf** Point
- SetColour(Integer)
- GetColour(): Integer

**Type** 3DCPoint **SubtypeOf** CPoint
- SetZ(Integer)
- GetZ(): Integer
- MoveXYZ(Integer, Integer, Integer)
Contravariance

Type PHandler
   SelectPoint(CPoint): CPoint

Type AnotherPHandler SubtypeOf PHandler
   SelectPoint(Point): CPoint

* Assuming that we do not require any colour of the argument. Since any CPoint still can be passed as an argument to AnotherPHandler’s SelectPoint, it is a proper subtype of PHandler
Covariance

Type PHandler
   SelectPoint(CPoint): CPoint

Type AnotherPHandler SubtypeOf PHandler
   SelectPoint(Point): 3DCPoint

- SelectPoint can now give a better result. The third coordinate is probably independent of the argument. AnotherPHandler is still a proper subtype of PHandlers since the resulting 3DCPoint can be used anywhere a CPoint was expected.
Contravariance and Covariance

Contravariance and covariance are type relations that are used to ensure compatibility between types.

- Contravariance of argument types requires that the types of the arguments of a method $M$ in a type $T$ must be subtypes of the corresponding arguments of $M$ in any subtype $S$ of $T$.
  - Argument types vary against the variation of the enclosing types.

- Covariance of result types requires that the type of the result of a method $M$ in a type $T$ must be a supertype of the result type of $M$ in any subtype $S$ of $T$.
  - Result types vary with the variation of the enclosing types.
Behavioural Subtyping

- Subtyping “for real”
- Separate implementation hierarchy and type hierarchy?
- Control of behaviour -- control of interface?
- Statically or dynamically?
Behavioural Subtyping

- Behavioural subtyping cannot be deduced without formal semantic specification of behaviour. Without such definitions, subtypes can only be deduced on the basis of external interfaces of a syntactic nature -- operation names and signatures.

- This is very difficult to achieve with available technology (if not impossible), but attempts have been made.

- Most tools that attempt to prove adherence to some behaviour are computationally expensive.

- Require training in logic that most programmers don’t have.
A Behavioural Notion of Subtyping

Liskov & Wing, 1994

- A sound and reasonable way of using classes as types in object-oriented designs
  - A sound “is-a” relation over the classes in a class hierarchy

- Subtype Requirement
  - Let $\phi(x)$ be a property provable about objects $x$ of type $T$.
  - Then $\phi(y)$ should be true for objects $y$ of type $S$ where $S$ is a subtype of $T$.
  - Example: Integers and Naturals

- How to implement this requirement?
Fragile Base Class

- A fundamental architectural problem of object-oriented programming where base classes are considered "fragile" since seemingly safe modifications to a base class, when inherited by the derived classes, may cause the derived classes to malfunction.

- You can't tell whether a base class change is safe simply by examining the base class's methods in isolation, you must look at (and test) all derived classes as well.
  - you must check all code that uses both base-class and derived-class objects too, since this code might also be broken by the new behavior.
  - a simple change to a key base class can render an entire program inoperable
Fragile Base Class -- Example

\[ Bag = \text{class} \]
\[ b : \text{bag of char} \]
\[ \text{init} \triangleq b := \emptyset \]
\[ \text{add(val } x : \text{char}) \triangleq \]
\[ b := b \cup \{x\} \]
\[ \text{addAll(val } bs : \text{bag of char}) \triangleq \]
\[ \text{while } bs \neq \emptyset \text{ do} \]
\[ \text{begin} \]
\[ \text{var } y \mid y \in bs: \]
\[ \text{self.add(y)}; \]
\[ bs := bs - \{y\} \]
\[ \text{end} \]
\[ \text{cardinality(} \text{res } r : \text{int} \text{)} \triangleq \]
\[ r := |b| \]
\[ \text{end} \]

\[ CountingBag = \text{class} \]
\[ n : \text{int} \]
\[ \text{init} \triangleq n := 0; \text{super.init} \]
\[ \text{add(val } x : \text{char}) \triangleq \]
\[ n := n + 1; \text{super.add}(x) \]

\[ Bag' = \text{class} \]
\[ b : \text{bag of char} \]
\[ \text{init} \triangleq b := \emptyset \]
\[ \text{add(val } x : \text{char}) \triangleq b := b \cup \{x\} \]
\[ \text{addAll(val } bs : \text{bag of char}) \triangleq b := b \cup bs \]
\[ \text{cardinality(} \text{res } r : \text{int} \text{)} \triangleq r := |b| \]
\[ \text{end} \]
Fragile Base Class -- Example

Bag
- b: bag of char
- add(val: char)
- addAll(val: bag of char)
- cardinality()

CountingBag
- n: int
- add(val: char)
- cardinality()

Bag
- b: bag of char
- add(val: char)
- addAll(val: bag of char)
- cardinality()
Examples of Errors

* In the paper “A study of The Fragile Base Class Problem” by Mikhajlov and Sekerinski, five aspects of the problem are studied:
  ~ Unanticipated Mutual Recursion
  ~ Unjustified Assumptions in Revision Class
  ~ Unjustified Assumptions in Modifier
  ~ Direct Access to Base Class State
  ~ Unjustified Assumptions of Binding Invariant in Modifier
How can we prevent them?

❖ “No cycles” requirement
❖ “No revision self-calling assumptions” requirement
❖ “No base class down-calling assumptions” requirement
❖ “No direct access to the base class state” requirement

Is that enough?
Do We Really Need Inheritance?

❖ What do we really need to use inheritance to achieve?
❖ What do we really gain?
❖ Is it worth all the problems?
  ~ Inheritance breaks encapsulation
❖ Can we use other solutions instead?
  ~ Many proposals suggest that inheritance should be decomposed into the more basic mechanisms of object composition and message forwarding,
  ~ Delegation?
Inheritance by Aggregation

Eirich & Hauck, 1991

- Mechanisms provided by many existing PLs combine reuse of types and reuse of implementation
- These mechanisms often break encapsulation
- Changes to an inheritance hierarchy affect related implementations
- Solution:
  - Separate type and class
  - Identify visibility of attributes and the binding of attributes to names as basic dimensions of class-based inheritance
  - State requirements to an inheritance mechanism and show how to fulfill them with an approach to inheritance by aggregation
Delegation

- Another way of forwarding messages that the receiver cannot answer itself.
- Used in prototype based languages
- Delegation chains can be altered dynamically

```cpp
class A {
public:
    foo()
    {
        print("Object A doing the job");
    }
}

class B {
public:
    A a;
    foo()
    {
        a.foo();
    }
}
```
Use of delegation

- Delegation — executing a method of some other object but in the context of self
- A lot more powerful than mere forwarding
- Exemplified using resend below
- Delegation can be used to implement inheritance but not vice versa
- Very powerful — delegates are not known statically as in inheritance and can change whenever
Example using io

```plaintext
point := Object clone
point x := 0 // Adding coordinates
point y := 0

point translate := method(p, x = p x; y = y + p y;)

otherPoint := point clone
otherPoint x := 10
point translate( otherPoint )

point3d := point clone
point3d z := 0
point3d translate :=
  method(p, resend; z = z+p z;)
```
References


Lynn Andrea Stein, *Delegation is Inheritance*.

Mikhajlov and Sekerinski, *A Study of the Fragile Base Class Problem*

Luca Cardelli, *Semantics of multiple inheritance*, 1984