EECS3311 – Software Design

Inheritance and Polymorphism
OOSC2 chapters 14 and parts of 15-17

Abstraction

- **Abstraction.** Omitting or hiding low-level details with a simpler, higher-level idea. Adopting an abstract model/specification which may be used to reason about the implementation, without the complexity of unnecessary detail.
- **Modularity.** Dividing a system into components or modules, each of which can be designed, implemented, tested, reasoned about, and reused separately from the rest of the system.
- **Encapsulation.** Building walls around a module (a hard shell or capsule) so that the module is responsible for its own internal behavior, and bugs in other parts of the system can't damage its integrity.
- **Information hiding.** Hiding a design decision (e.g. details of a module's implementation) from the rest of the system, so that that decision can be changed later without changing the rest of the system. You hide what might later need to change.
- **Separation of concerns.** Making a feature (or "concern") the responsibility of a single module, rather than spreading it across multiple modules.

Abstract Data Types

- The key idea of data abstraction is that a type is characterized by the operations you can perform on it.
- A number is something you can add and multiply; a string is something you can concatenate and take substrings of; a boolean is something you can negate, and so on.
- What makes abstract types new and different is the focus on operations: **the user of the type does not need to worry about how its values are actually stored**, in the same way that a programmer can ignore how the compiler actually stores integers. All that matters is the operations.

ADTs - recall STACK[G]

- The operations of an abstract type are classified as follows:
  - **Creators** create new objects of a type T.
  - **Commands** create new objects from old objects of the type T.
  - **Queries** take objects of the abstract type T and return objects of a different type.
- We summarize these distinctions schematically like this:
- creator: $a \rightarrow T$
- command: $T, a \rightarrow T$
- query: $T, a \rightarrow z$
- Axioms and Theorems

INTEGER ADT

- **creators**: the numeric literals 0, 1, 2, …
- **commands**/operations:
  arithmetic operators +, -, ×, ÷
- **queries**: comparison operators =, /=, <, >

- We do not need to know how the computer stores, represents or implements the integers
- Designing an abstract type involves choosing good operations and determining how they should behave from their properties alone free of implementation details
- It's better to have **a few, simple operations** that can be combined in powerful ways, rather than lots of complex operations.

Module as an ASM

contracts

modules

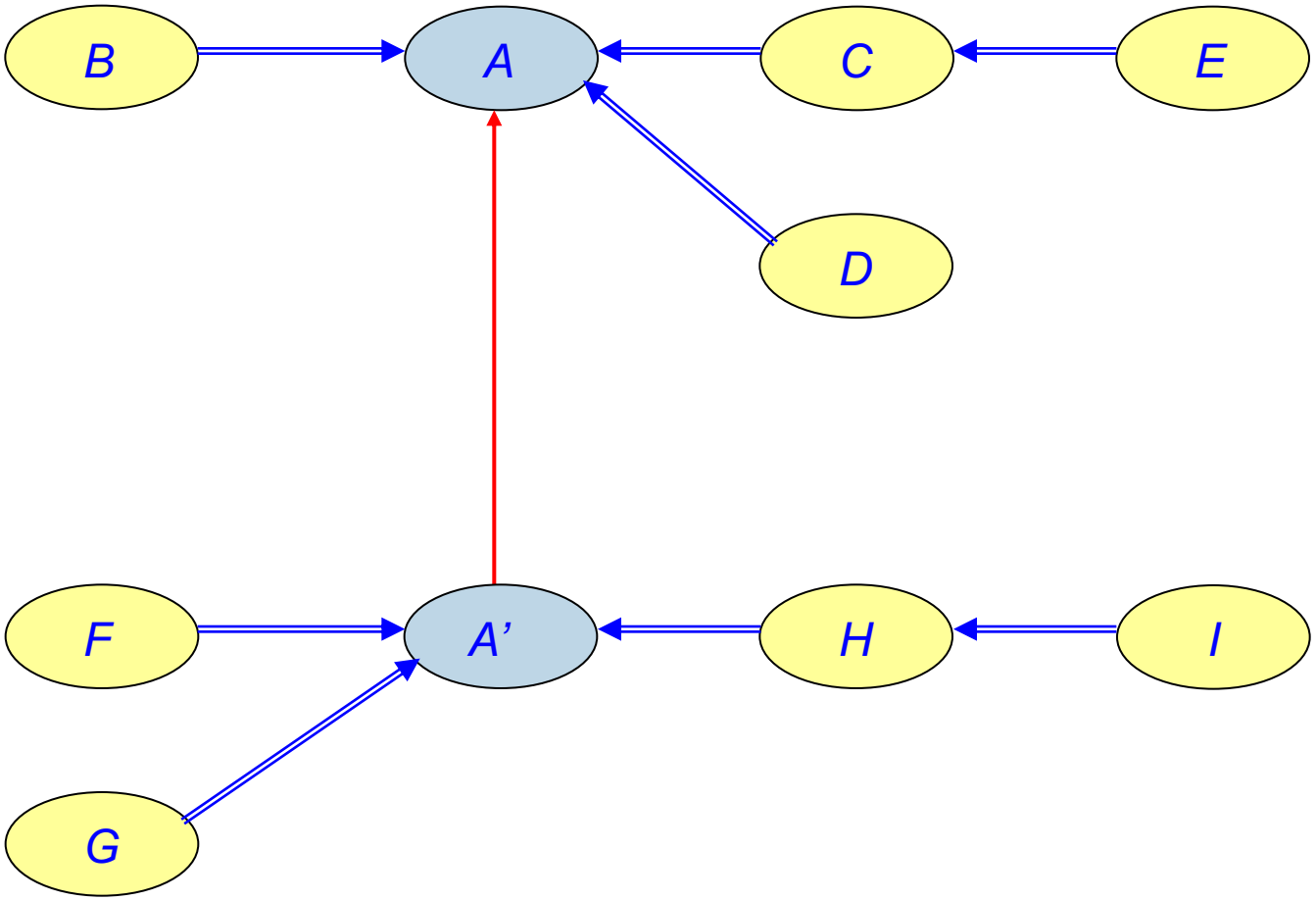
1. *Objects with state.* Parnas thought of a module as an abstract machine with an explicit state. By operating on a module we cause the internal state of the module to change. We use this idea when we put a coin in a drinks machine and press a button to move the machine into a new state, which hopefully results in the output of our chosen drink. The procedural style of programming that includes object-oriented programming is based on the idea of an explicit state, At the level of specification, some approaches define an explicit notion of state for a system and specify a change in state in terms of pre-conditions and post-conditions.

types

2. *Abstract data types with values.* Liskov's concept of data abstraction has been developed at a more abstract level within the logic and algebraic approaches to specification. Operations on abstract data types are specified by functions that return values. The relationships between the operations describe the meaning, or the semantics, of the abstract data type and there is no explicit notion of state. This view is mirrored by the declarative style of programming.

Axioms

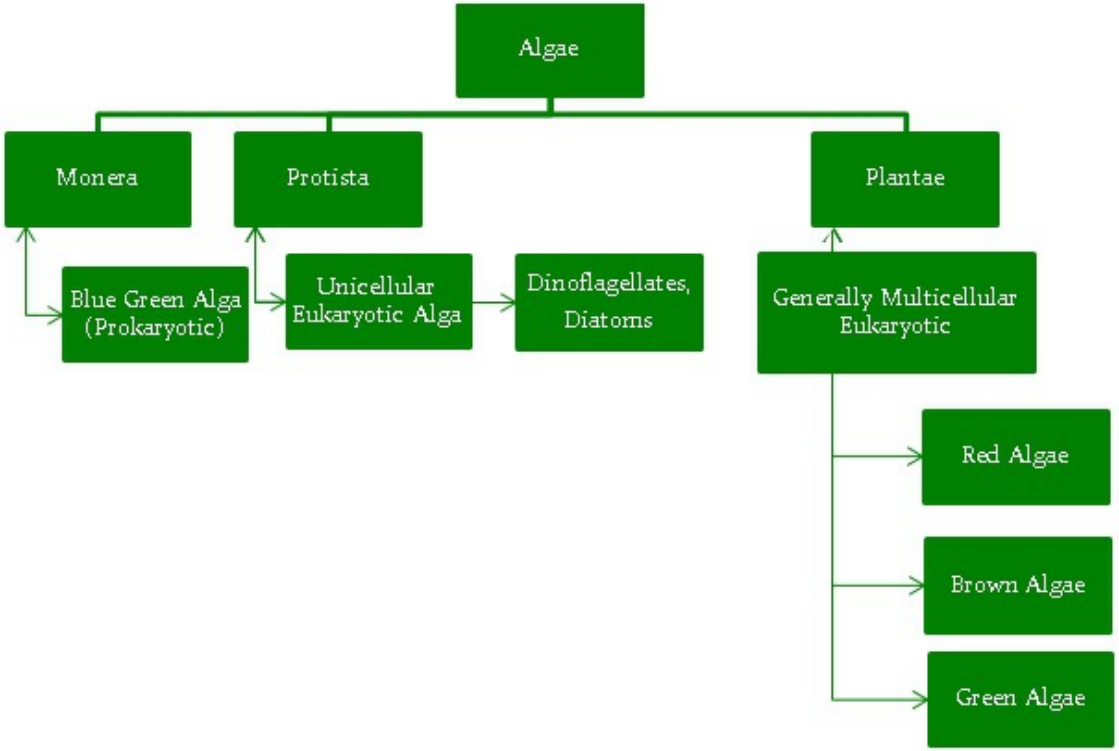
The Open-Closed Principle

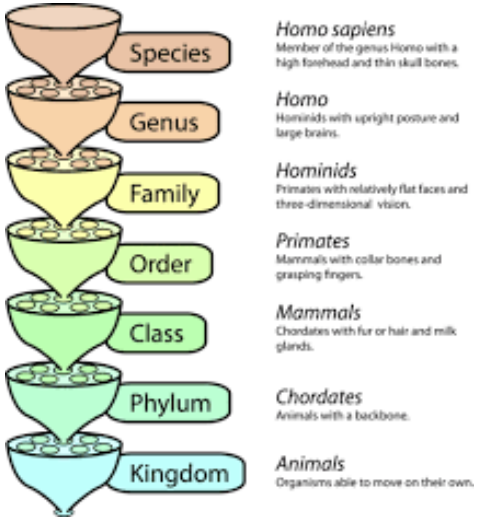


Open-Closed Principle

- Modules should be both Open and Closed
- It is **open** if it is available for extension (**Volatile?)**
- It is **closed** if it's API is fixed i.e. clients can rely on well-defined interface that does not change (**Stable?)**
  - E.g. an implementation may change
- It's a contradiction!
  - Resolved via inheritance!
  - Inheritance is not just for classification (taxonomy) but also for re-use

Linnaeus Taxonomy





What is inheritance?

- Principle: Describe a new class not from scratch but as extension or specialization of one existing class — or several in the case of MULTIPLE inheritance.
  - From the module viewpoint: if *B* inherits from *A*, all the services of *A* are potentially available in *B* (possibly with a different implementation).
  - From the type viewpoint: inheritance is the ‘‘is-a’’ relation. If *B* inherits from *A*, whenever an instance of *A* is required, an instance of *B* will be acceptable.

https://en.wikipedia.org/wiki/Liskov_substitution_principle

Liskov substitution principle

From Wikipedia, the free encyclopedia

"Substitutability" redirects here. For the economic principle, see Substitute good.

Substitutability is a principle in object-oriented programming stating that, in a computer program, if S is a subtype of T, then objects of type T may be *replaced* with objects of type S (i.e. an object of type T may be *substituted* with any object of a subtype S) without altering any of the desirable properties of T (correctness, task performed, etc.). More formally, the **Liskov substitution principle** (**LSP**) is a particular definition

SOLID

Principles
Single Responsibility
Open/closed
Liskov substitution
Interface segregation
Dependency inversion

V • T • E

In addition to the signature requirements, the subtype must meet a number of behavioral conditions. These are detailed in a terminology resembling that of design by contract methodology, leading to some restrictions on how contracts can interact with inheritance:

- Preconditions cannot be strengthened in a subtype.
- Postconditions cannot be weakened in a subtype.
- Invariants of the supertype must be preserved in a subtype.

In the same paper, Liskov and Wing detailed their notion of behavioral subtyping in an extension of Hoare logic, which bears a certain resemblance to Bertrand Meyer's design by contract in that it considers the interaction of subtyping with preconditions, postconditions and invariants.

Single responsibility principle

From Wikipedia, the free encyclopedia

The **single responsibility principle** is a computer programming principle that states that every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class. All its services should be narrowly aligned with that responsibility. Robert C. Martin expresses the principle as, "A class should have only one reason to change."[1]

Single responsibility principle

From Wikipedia, the free encyclopedia

The **single responsibility principle** is a computer programming principle that states that every module or class should ha
functiona
should b
should b
Martin ex
one reas

Example [edit]

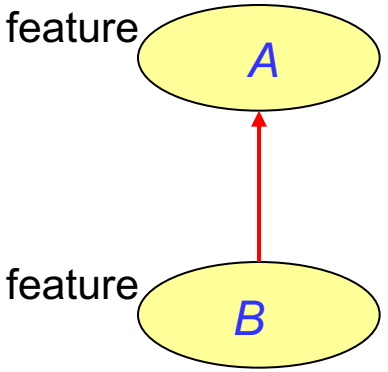
Martin defines a responsibility as a *reason to change*, and concludes that a class or module should have one, and only one, reason to be changed (i.e. rewritten). As an example, consider a module that compiles and prints a report. Imagine such a module can be changed for two reasons. First, the content of the report could change. Second, the format of the report could change. These two things change for very different causes; one substantive, and one cosmetic. The single responsibility principle says that these two aspects of the problem are really two separate responsibilities, and should therefore be in separate classes or modules. It would be a bad design to couple two things that change for different reasons at different times.

Violation of SRP?

```
class SESSION feature
        start-session
                do
                -- HTTP session cookies
                end
        activate-user-account
                do
                --  query the DB and toggle some flag in a column
                end
        generate_randomID ...

        edit_account(a: ACCOUNT)
                do
                --   issue some SQL UPDATE query to update an user account
                end

        login
                do
                --    perform authentication logic
                end

        check_access_rights
                do
                --    read cookies, perform authorization
                end
end
```

Terminology

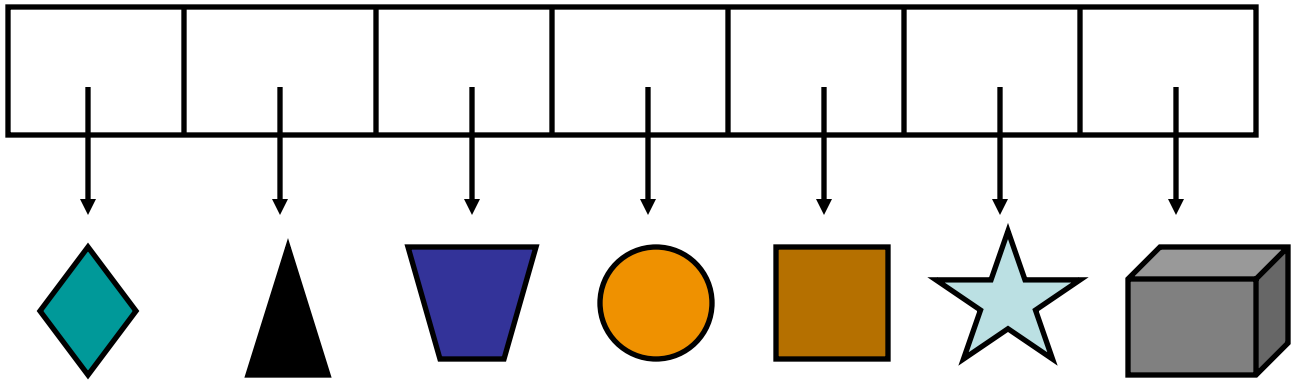
- Parent, Heir
- Ancestor, Descendant
- The ancestors of *B* are *B* itself and the ancestors of its parents.
- Proper ancestor, Proper descendant
- Direct instance, Instance
- The instances of *A* are the direct instances of its descendants.
- (Other terminology: subclass, superclass, base class)



Polymorphism & Dynamic Binding

Design Problem

- Suppose we had an array of figures and want to rotate all the figures in it.
- Each figure has its own rotation method



The traditional architecture

```
figures: ARRAY[ANY]                    -- "Object[] figures" in Java

rotate_all_figures_in_array_by ( d : DEGREE)
    require d > 0
    local
        f: ANY; i: INTEGER
    do
        from   i := figures.lower
        until i = figures.upper
        loop
                f := figures[i]
                rotate(f, d)   -- what happens with an unknown f?
                i := i + 1
        end
    end
```

Traditional

```
rotate (f: ANY; d: DEGREE)
        do
                if ''f is a CIRCLE'' then …
                        …
                elseif ''f is a POLYGON'' then
                        …
                end
        end
```

- and similarly for all other routines such as *display* etc.!
- What happens when we add a new figure type?
  - We must update all routines *rotate*, *display* etc.
  - We also get a runtime error if not handled in the code

Single Choice Principle

- What happens when we add a new figure type?
  - We must update all routines *rotate*, *display* etc.

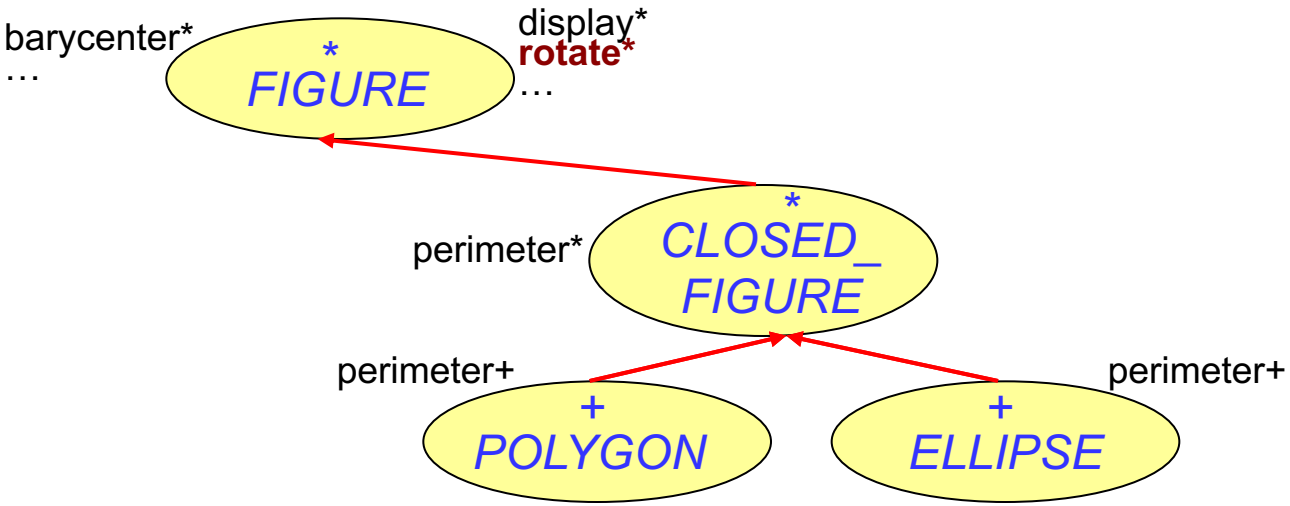
- **Single Choice Principle**
  - Whenever a software system must support a set of alternatives (e.g. variants of a figure in graphic systems), one and only one module in the system should know their exhaustive list
  - Later, if variants are added (e.g. by introducing a class in the figure hierarchy for a new figure), we only have to update a single module – the point of single choice!
  - Consistent with the Open-Close Principle

New solutions required!

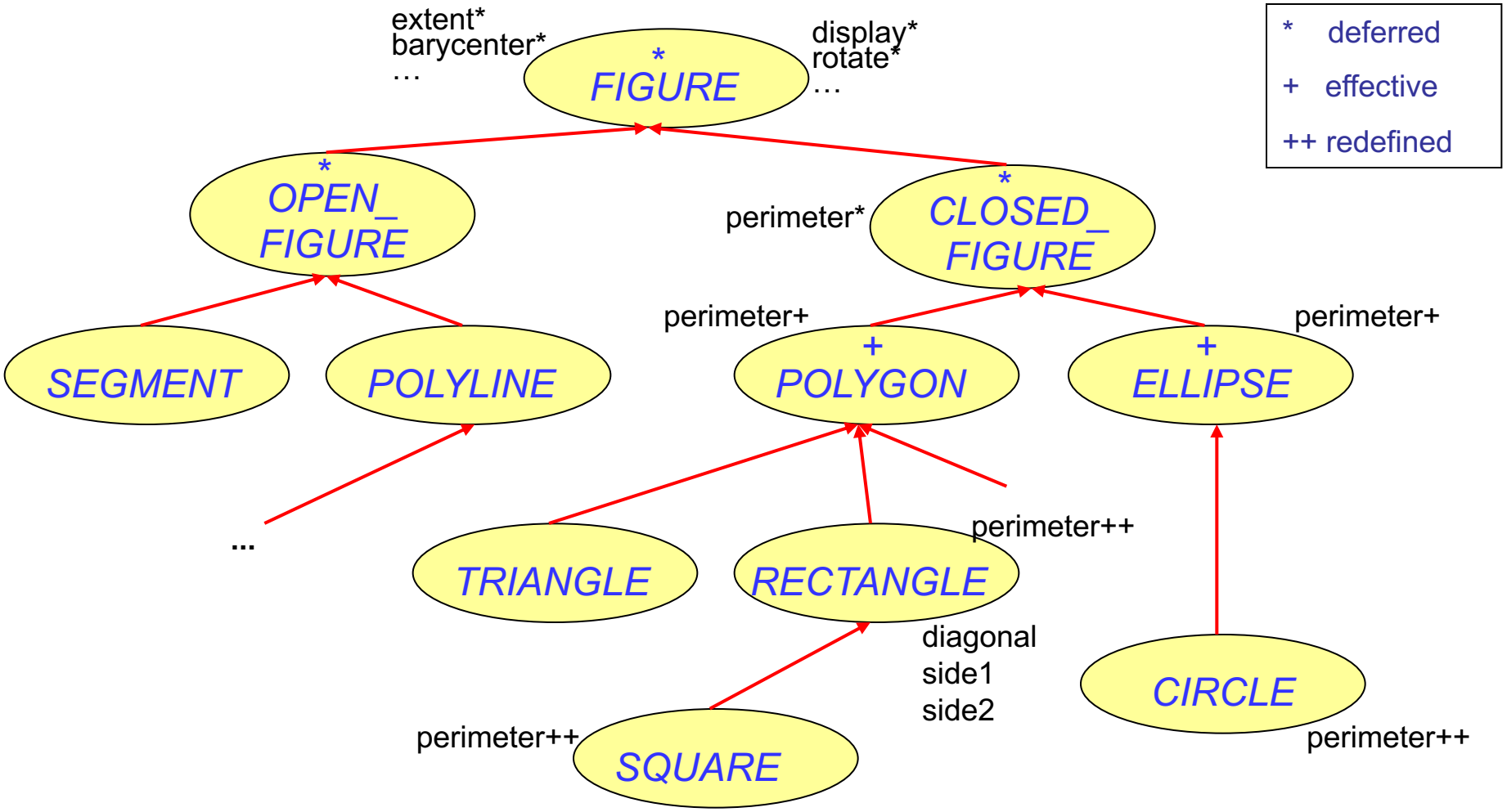
- Want to be able to add new kinds of figures without
  - breaking previous programs
  - without modifying the method that rotates all figures

- Solution
  - polymorphism & dynamic binding

FIGURE polymorphism



Example hierarchy

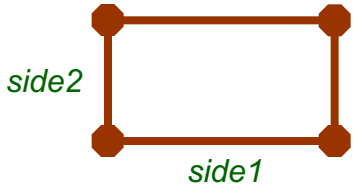


Redefinition

```
class POLYGON inherit CLOSED_FIGURE create
        make
feature
        vertices: ARRAY [POINT]
        count: INTEGER – number of vertices in poygon
        perimeter: REAL
                                -- Perimeter length
                do
                    from ... until ... loop
                                Result := Result +
                                                (vertices[i]) . distance (vertices [i + 1])
                                            ...
                                end
                end
invariant
        vertices.count >= 3
        count = vertices.count
end
```

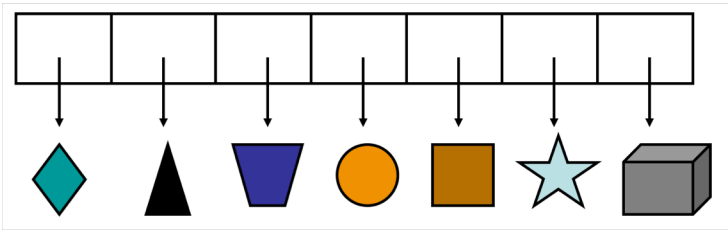
Redefinition (cont'd)

```
class RECTANGLE
inherit
        POLYGON
                redefine
                        perimeter
                end
create
        make
feature
        diagonal, side1, side2: REAL
        perimeter: REAL
                        -- Perimeter length
                do
                        Result := 2 * (side1 + side2)
                end
invariant
        vertices_count = 4
end
```



Solution

```
figures: ARRAY[FIGURE]

rotate_all_figures_in_array_by ( d : DEGREE)
    require d > 0
    local
        f: FIGURE; i: INTEGER                    -- static typing
    do
        from  i := figures.lower
        until i = figures.upper
        loop
                f := figures[i]
                f.rotate(d)                      -- polymorphism
                                                 -- dynamic binding
                i := i + 1
        end
    end
```



Applying the Single Choice Principle

```
f: FIGURE
c: CIRCLE
p: POLYGON
...
create c.make (...)
create p.make (...)
...
if ... then
      f := c
else
      f := p
end
...
f.move (...)              -- dynamic binding selects the right move
f.rotate (...)
f.display (...)
...
```

Polymorphism

- Typing
- Binding

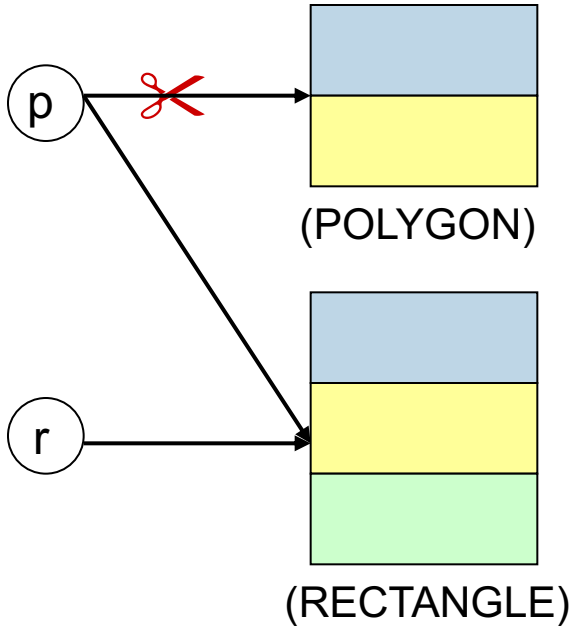
Inheritance and polymorphism

- Assume:

  p: *POLYGON*; *r*: *RECTANGLE*; *t*: *TRIANGLE*;
  x: *REAL*

- Permitted:

  x := *p*.*perimeter*
  x := *r*.*perimeter*
  x := *r*.*diagonal*
  p := *r*



- ~~NOT permitted:~~

  x := *p*.*diagonal* (even just after *p* := *r* !)
  r := *p*

Dynamic binding

- What is the effect of the following (assuming *some_test* true)?

```
if some_test then
        p := r
else
        p := t
end
x := p.perimeter
```

- Redefinition: A class may change an inherited feature, as with *RECTANGLE* redefining perimeter.
- Polymorphism: *p* may have different forms at run-time.
- Dynamic binding: Effect of *p.perimeter* depends on run-time form of *p*.

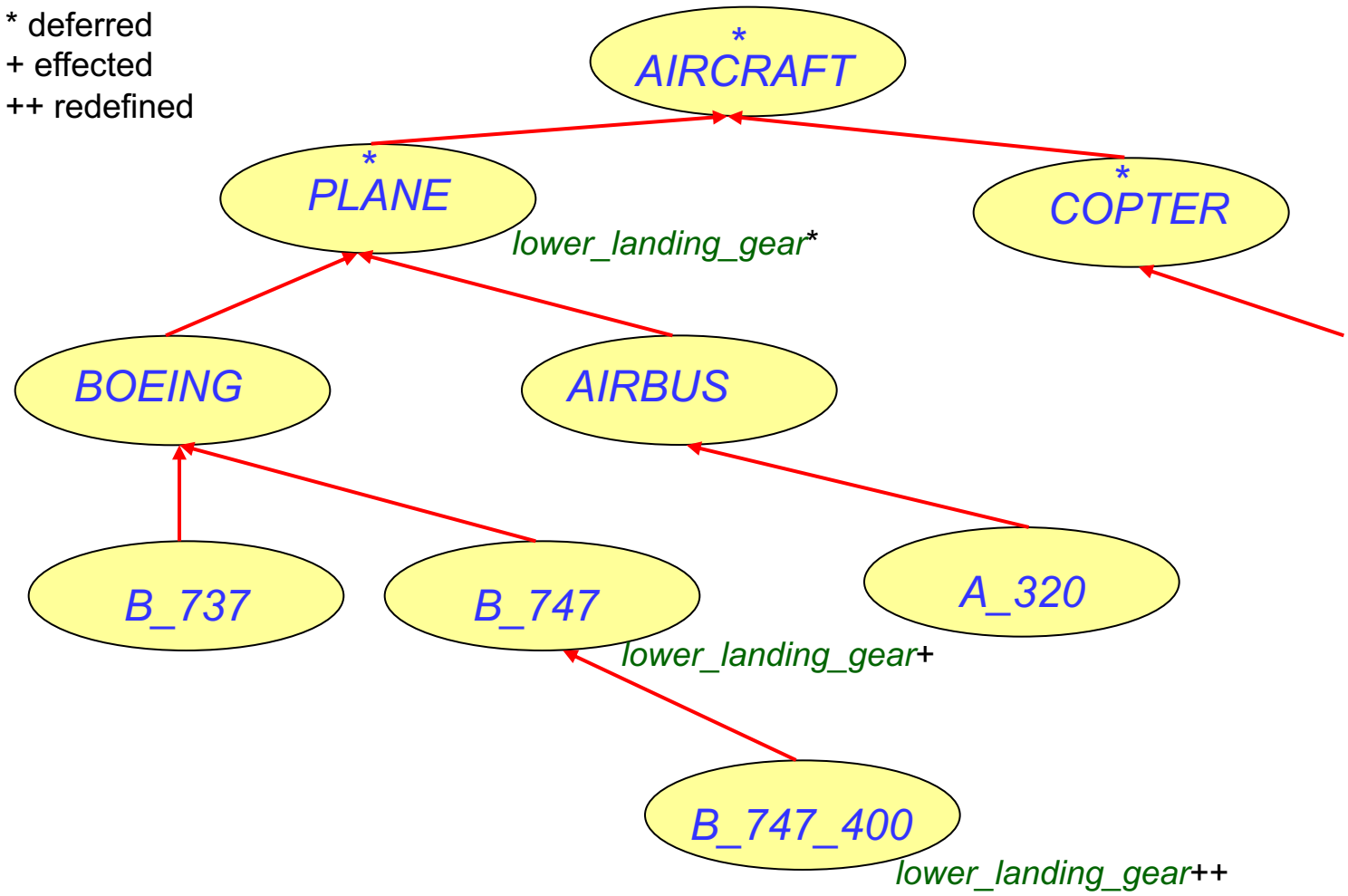
Polymorphism

- Polymorphism means the ability to take several forms
- In OO, it is entities (attributes, arguments, local variables) which can become attached to objects of different types
- P: POLYGON; r:RECTANGLE; t: TRIANGLE
  The following are both valid:
  - p := r
  - p := t
  - Assignments in which the type of the source (RHS) is different than the target (LHS) are called polymorphic assignments
- Since both RECTANGLE and TRIANGLE inherit from POLYGON we say that the type of the source conforms to the type of the target.
  - In the reverse direction the assignment r := p is not valid because p does not conform to r.

Type checking

- When should we check typing and binding?
  - statically (compile time)
  - dynamically (runtime)

Typing – OOSC2 chapter 17



Check at compile time or runtime?

a: PLANE

b: B_747

a := b

a.take_off

…

a.lower_landing_gear

…

(1) when should we check that there is a feature lower_landing_gear (compile time or runtime)?

(2) If there is more than one (polymorphism), when should we determine which one to use?

Check at compile time or runtime?

a: PLANE

b: B_747

a := b

a.take_off

…

a.lower_landing_gear

…

runtime
is much too late
to find out
that you do not have
landing gear

Typing vs. Binding

- Binding should be checked at runtime (dynamically)
  - So that the correct version of lower_landing_gear is invoked
- Typing should be checked at compile time (statically)
  - Runtime is much too late to determine that you do not have lower_landing_gear

Typing vs. binding

- What do we know about the feature to be called?
- Static typing:

  At least one

- Dynamic binding:

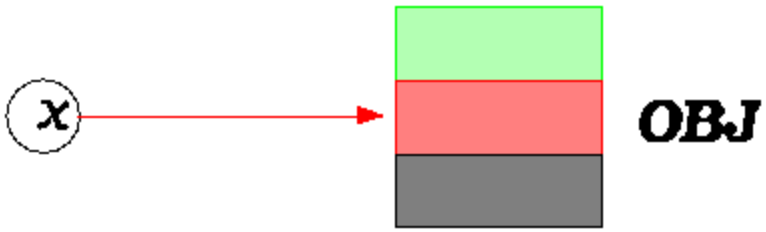
  The right one

- Example:

  my_aircraft.*lower_landing_gear*

The Typing problem

- Basic OO Construct: *x.f (arg)*
- meaning: execute on the object attached to *x* the operation *f*, using the argument *arg*
- The model is simple – at run time, this is what our systems do – calling features on objects, and passing arguments as necessary
- From the simplicity of the model follows the simplicity of the typing problem, whose statement mirrors the structure of the Basic Construct:
- When, and how, do we know that:
  - There is a feature corresponding to *f* and applicable to the object?
  - *arg* is an acceptable argument for that feature?



Definition – statically typed system

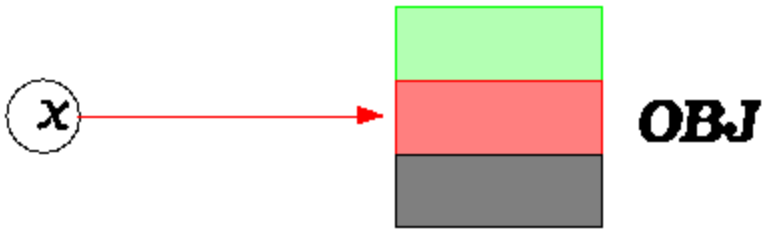
- Type violation: a runtime type violation occurs in the execution of a call *x.f(arg) where x* is attached to an object OBJ if either
  - there is no feature corresponding to *f* and and applicable to OBJ
  - There is a feature but *arg* is not acceptable

- An OO language is statically typed if it is equipped with a set of consistency rules so that a compiler can detect that there will be no runtime type violations merely by inspecting the program text (at compile time)

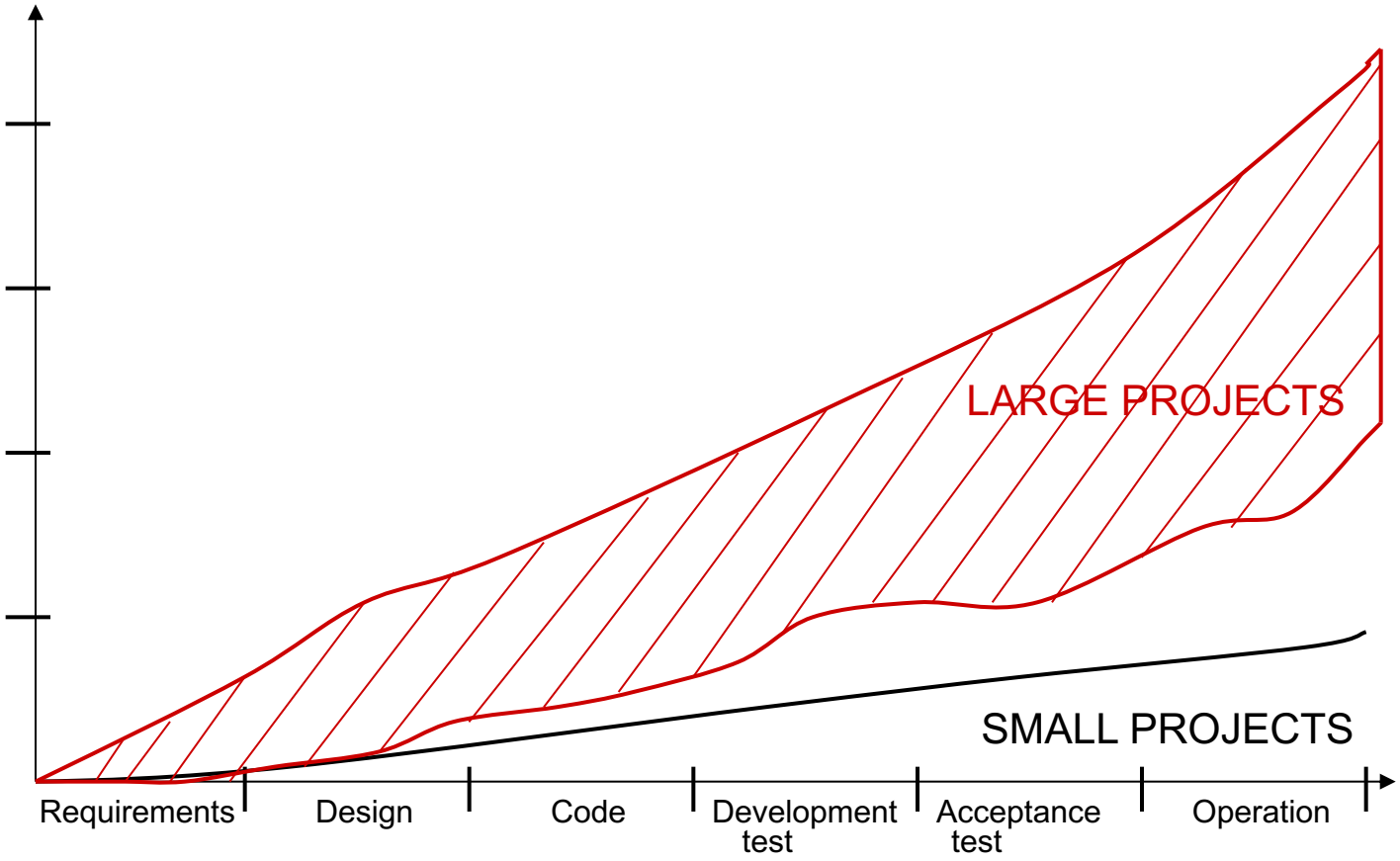
Consistency Rules (partial)

- In class C
  x: D …
  x.f(arg)

  - Declaration Rule: Every entity (e.g. *x*) must be explicitly declared to be of a certain type
  - Call rule: If a class C contains the call *x.f(arg)* there must be feature *f* in class D exported to C
  - Attachment Rule: In an assignment x := y (or argument passing) the base class of y must be a descendant of the base class of x.



(After Barry W. Boehm)



Dynamic Typing: Python

account.py - C:\Python31\account.py

File Edit Format Run Options Windows Help

```
def test():
    a = Account(150)
    a.deposit(75)
    print(a.inquiry())
    a.withdraw(60)

class Account:
    def __init__(self,balance):
        self.balance = balance
    def deposit(self,a):
        self.balance = self.bala
    def inquiry(self):
        return self.balance
```

```
>>> ========== RESTART ==========
>>> import account
>>> account.test()
225
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    account.test()
  File "C:\Python31\account.py", line 5, in test
    a.withdraw(60)
AttributeError: 'Account' object has no attribute 'withdraw'
>>>
```

Part of the program executed correctly!

There is no compile time!

Runtime error!

Dynamic Typing: php

Type Error: Argument "b" is not a number

Type Error: There is no routine "withdra"

(spelling error)

```php
<?php
class Account {
    private $balance;
    function __construct($m) {
        $this->balance = $m;}
    public function deposit($m) {
        $this->balance += $m;}
    public function withdraw($m) {
        $this->balance -= $m;}
    public function inquiry() {
        return $this->balance;}
};
function test() {
    $a = new Account(150); $b = new Account(100);
    $a->deposit(75);
    $a->withdraw($b);
    $a->withdraw(1); echo $a->inquiry() . "\n";
    $a->withdra(1);}
test();
```

```
indigo 309 % /xsys/pkg/php/bin/php bank.php
PHP Notice:  Object of class Account could not be converted to int in
    /cs/home/jonathan/tools/php/bank.php on line 9
223  (This is wrong! Account "b" interpreted as withdraw "1")
PHP Fatal error:  Call to undefined method Account::withdra() in
    /cs/home/jonathan/tools/php/bank.php on line 18
```

Casting breaks the type system

- What programming languages offer genericity that you know of? Java? C++? Other?
- C++ has the template: Set < int > s ;
- Java 5 added genericity
- What is the effect of genericity on
  - compile time
  - size of the generated code
  - execution time
  - execution space
- **Warning**: generics cheap in Eiffel – used to be expensive in C++

Non-generic Java Stack

```
public class Stack {
    private int count;
    private int capacity;
    private int capacityIncrement;
    private Object[] itemArray; // instead of ARRAY[G]

 Stack() {    //constructor  limited to class name
        count= 0;
        capacity= 10;
        capacityIncrement= 5;
        itemArray= new Object[capacity];
    }

    public boolean empty() {
        return (count == 0);
    }
```

Java Stack (cont.)

```
public void push(Object x) {
        if(count == capacity) {
            capacity += capacityIncrement;
            Object[] tempArray= new Object[capacity];
            for(int i=0; i < count; i++)
                tempArray[i]= itemArray[i];
            itemArray= tempArray;
        }
        itemArray[count++]= x;
    }

    public Object pop() {
        if(count == 0)
              return null;
        else return itemArray[--count];
    }
```

Java Stack (cont.)

```
 public Object peek() {
        if(count == 0)
          return null;
        else return itemArray[count-1];
    }
}
```

```
public class Point {
    public int x;
    public int y;
}
```

Java Stack runtime error

```
class testStack {
   public static void main(String[] args) {
     Stack s = new Stack();
     Point upperRight = new Point();
     upperRight.x= 1280;                          // breaks information hiding
     upperRight.y= 1024;                          // cannot guarantee any contract
      s.push("red");
      s.push("green");                             // push some String
      s.push("blue");                              // objects on the stack

      s.push(upperRight);                         // push a POINT object on the stack


      while(!s.empty()) {                         // cast all items from OBJECT
          String color = (String) s.pop();        // to STRING in order to print
          System.out.println(color);

       }}}          // causes a run-time error
                    // ClassCastException: Point at testStack.main(testStack.java:18)
                    // With genericity, it is a compile time error
```

Casting in Java isn't magic, it's you telling the compiler that an Object of type A is actually of more specific type B, and thus gaining access to all the methods on B that you wouldn't have had otherwise. You're not performing any kind of magic or conversion when performing casting, you're essentially telling the compiler "trust me, I know what I'm doing and I can guarantee you that this Object at this line is actually an <Insert cast type here>." For example:

```
Object o = "str";
String str = (String)o;
```

To preserve strong typing in Eiffel, casting is not allowed. Use type safe **attached** query instead

There's two ways this could go wrong. Firstly, if you're casting between two types in completely different inheritance hierarchies then the compiler will know you're being silly and stop you:

```
String o = "str";
Integer str = (Integer)o; //Compilation fails here
```

Secondly, if they're in the same hierarchy but still an invalid cast then a `ClassCastException` will be thrown at runtime:

```
Number o = new Integer(5);
Double n = (Double)o; //ClassCastException thrown here
```

This essentially means that you've violated the compiler's trust. You've told it you can guarantee the object is of a

```
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer canno
t be cast to java.lang.Double
        at HelloWorld.main(HelloWorld.java:11)
```

Why Twitter Switched to Scala (2009)

- Twitter began its life as a Ruby on Rails application, and still uses Ruby on Rails to deliver most user-facing web pages. In this interview, three developers from Twitter—Steve Jenson, system engineer; Alex Payne, API lead; and Robey Pointer, member of the service team—sit down with Bill Venners to discuss Twitter's real-world use of Scala.
- **Bill Venners**: I'm curious, and the Ruby folks will want it spelled out: Can you elaborate on what you felt the Ruby language lacked in the area of reliable, high performance code?
- **Steve Jenson**: ... Another thing we really like about Scala is static typing that's not painful ...

Why Twitter Switched to Scala (2009)

- **Alex Payne**: I'd definitely want to hammer home what Steve said about typing. As our system has grown, a lot of the logic in our Ruby system sort of replicates a type system, either in our unit tests or as validations on models. I think it may just be a property of large systems in dynamic languages, that eventually you end up rewriting your own type system, and you sort of do it badly.
- You're checking for null values all over the place. There's lots of calls to Ruby's kind_of? method, which asks, "Is this a kind of User object? Because that's what we're expecting. If we don't get that, this is going to explode." It is a shame to have to write all that when there is a solution that has existed in the world of programming languages for decades now.

Using Scala/Joda library

This is a DEBACLE!

The joda time constructors include one which takes Object. That means you lose 100% of your type safety unless you luck into a conflict it will notice, because scala thinks "oh, you must have meant to call the Object constructor with a 6-tuple, let me help you."

No scala buddy, I didn't mean that.

scala> import org.joda.time._
import org.joda.time._

scala> def time = new DateTime(2010, 5, 7, 0, 0, 0, 0)
time: org.joda.time.DateTime

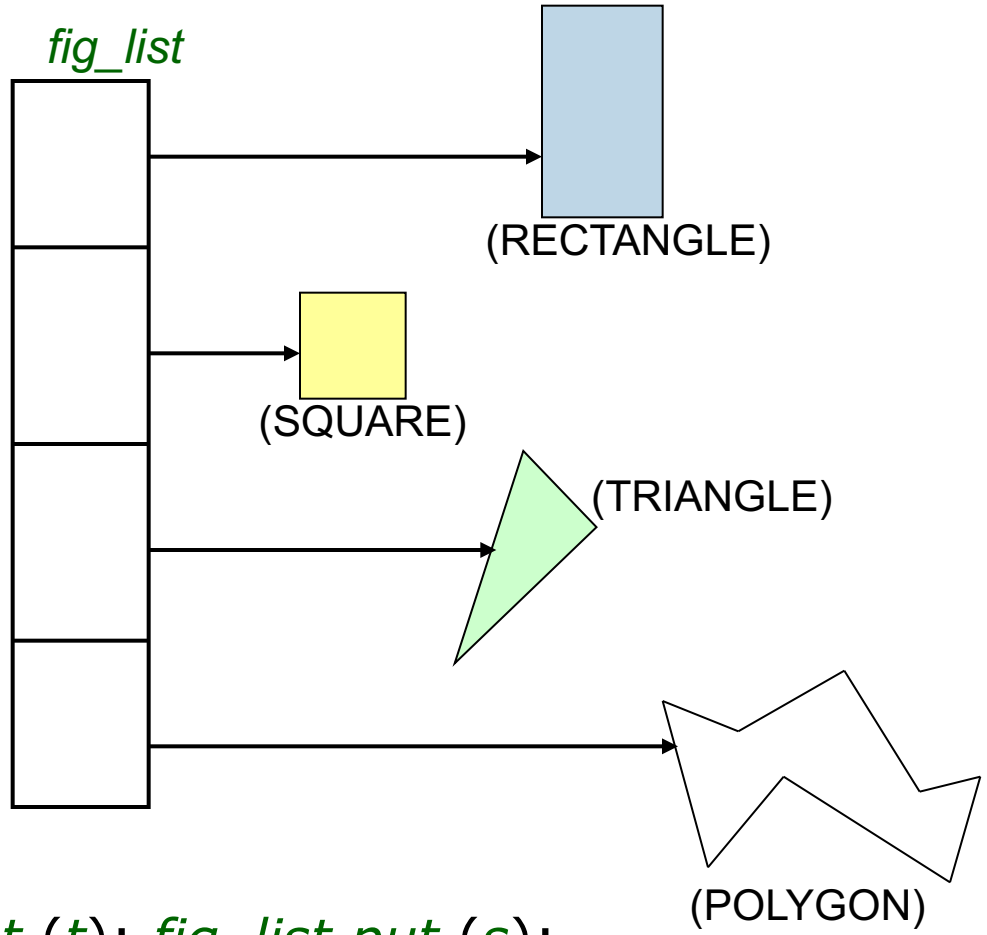
scala> time
res0: org.joda.time.DateTime = 2010-05-07T00:00:00.000-07:00

scala> def time = new DateTime(2010, 5, 7, 0, 0, 0)
time: org.joda.time.DateTime

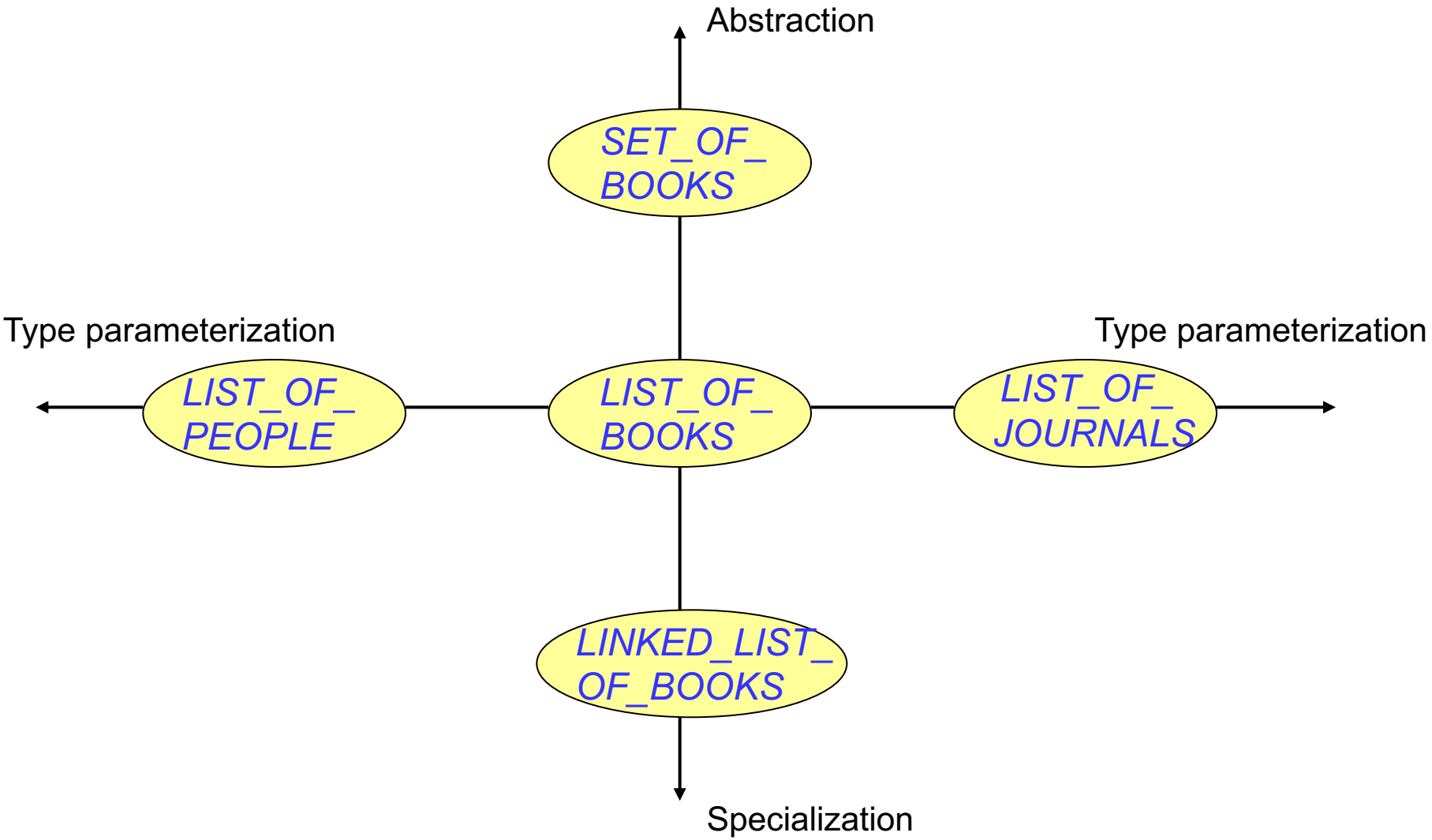
scala> time
java.lang.IllegalArgumentException: No instant converter found for type: **scala**.Tuple6 ....

Genericity + Inheritance: Polymorphic data structures

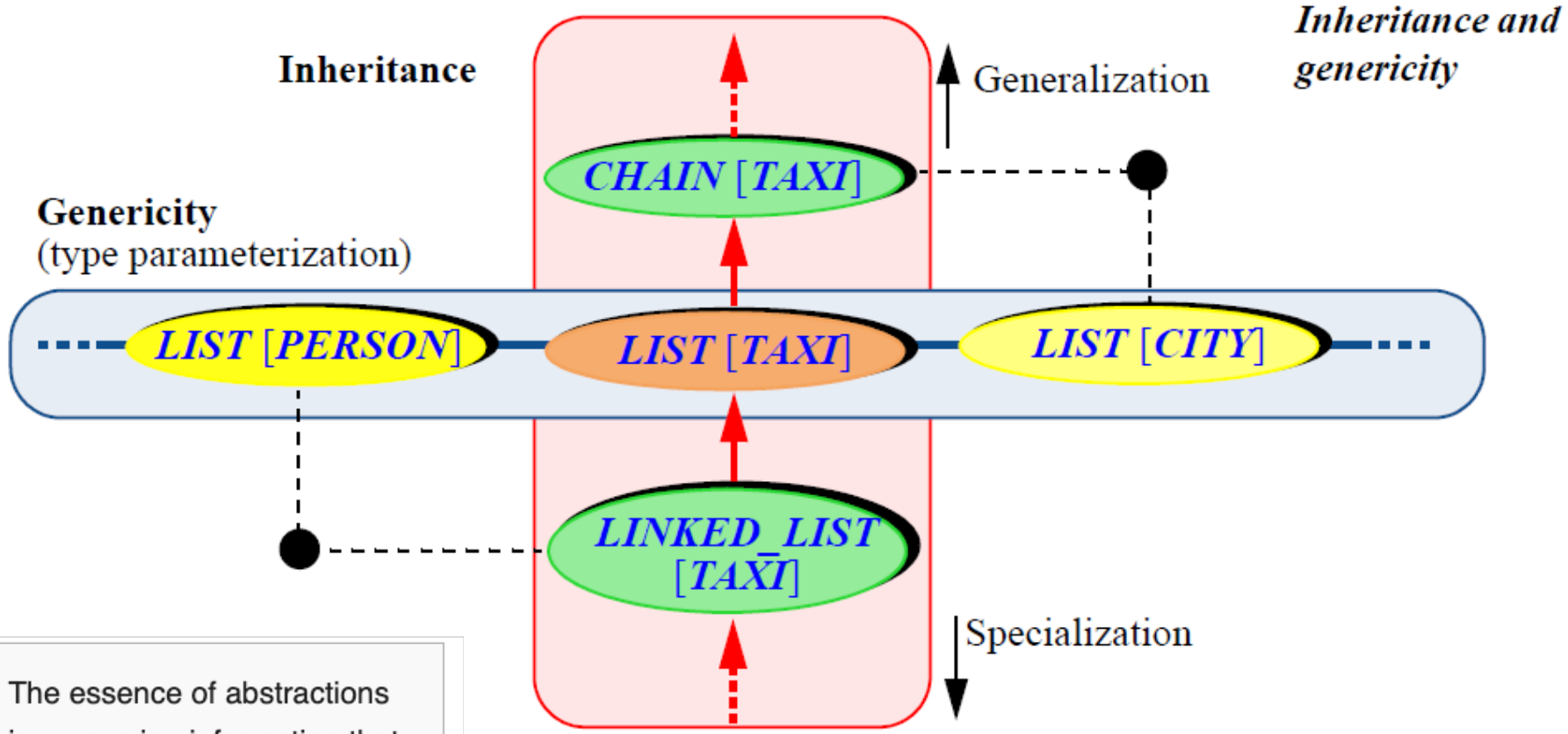
```
class
  LIST [G]
feature
  ...
  item: G ...
  put (x: G) ...
end

fig_list: LIST [FIGURE]
r: RECTANGLE
s: SQUARE
t: TRIANGLE
p: POLYGON
...
 fig_list.put (p); fig_list.put (t); fig_list.put (s);
fig_list.put (r); fig_list[i].display
```



Genericity vs. Inheritance



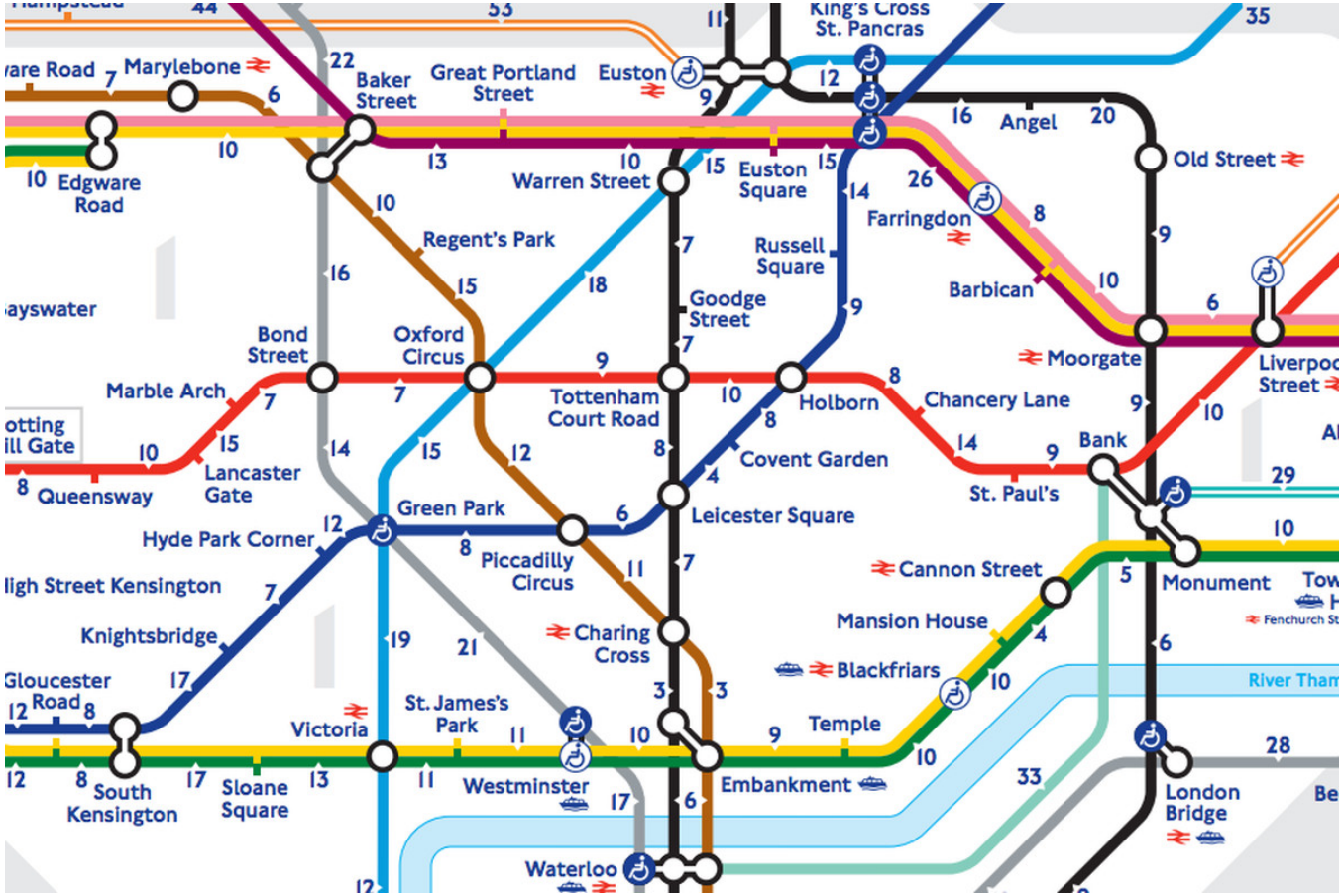
Genericity vs. Inheritance



> The essence of abstractions is preserving information that is relevant in a given context, and forgetting information that is irrelevant in that context.
>
> – John V. Guttag[1]

Abstraction

> The essence of abstractions is preserving information that is relevant in a given context, and forgetting information that is irrelevant in that context.
>
> – John V. Guttag[1]



Inheritance & sub-contracting

Weak vs. Strong Assertions

- Describe each assertion as ***a set of satisfying value***.
  - $x > 3$ has satisfying values $\{4, 5, 6, 7, \ldots\}$ **q** --weak
  - $x > 4$ has satisfying values $\{5, 6, 7, \ldots\}$ **p** -- strong

- An assertion *p* is ***stronger*** than an assertion *q* if *p*'s set of satisfying values is a subset of *q*'s set of satisfying values.
  - Logically speaking, *p* being stronger than *q* (or, *q* being weaker than *p*) means $p \Rightarrow q$.
  - e.g., $x > 4 \Rightarrow x > 3$

- What's the weakest assertion?
- What's the strongest assertion?

[TRUE]
[FALSE]

Theorem:

$$(\forall x \mid : P \Rightarrow Q) \equiv \{x \mid P\} \subseteq \{x \mid Q\}$$

Proof:

$\{x \mid P\} \subseteq \{x \mid Q\}$

$=$ ⟨Def. of Subset $\subseteq$, with v not occurring free in P or Q⟩

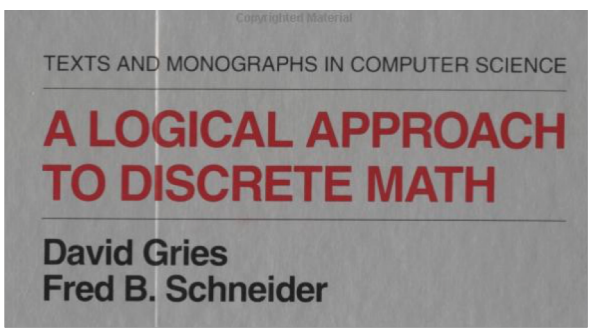
$(\forall v \mid v \in \{x \mid P\} : v \in \{x \mid Q\})$

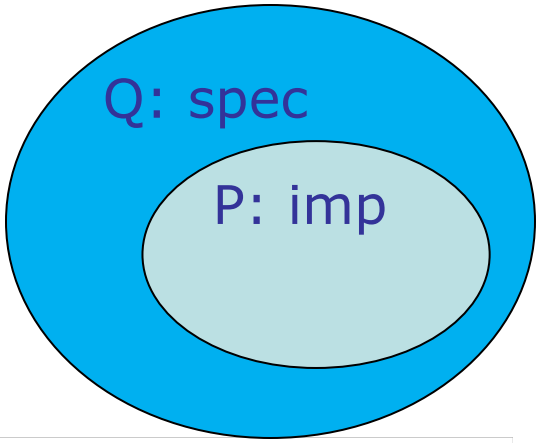
$=$ ⟨$v \in \{x \mid R\} \equiv R[x := v]$, twice⟩

$(\forall v \mid P[x := v] : Q[x := v])$

$=$ ⟨Trading; Dummy renaming⟩

$(\forall x \mid : P[x := v][v := x] \Rightarrow Q[x := v][v := x])$





Weak vs. Strong Assertions

- Describe each assertion as ***a set of satisfying value***.
  - $x > 3$ has satisfying values $\{4, 5, 6, 7, \ldots\}$ **q** --weak
  - $x > 4$ has satisfying values $\{5, 6, 7, \ldots\}$ **p** -- strong

Weak or Strong

- In this discussion "stronger" means "**implies**" and "**weaker**" means "**implied by**".
- More precisely "a is stronger than b" means
  - (a **implies** b) **and not** (a = b), and
- "weaker" is the inverse relation

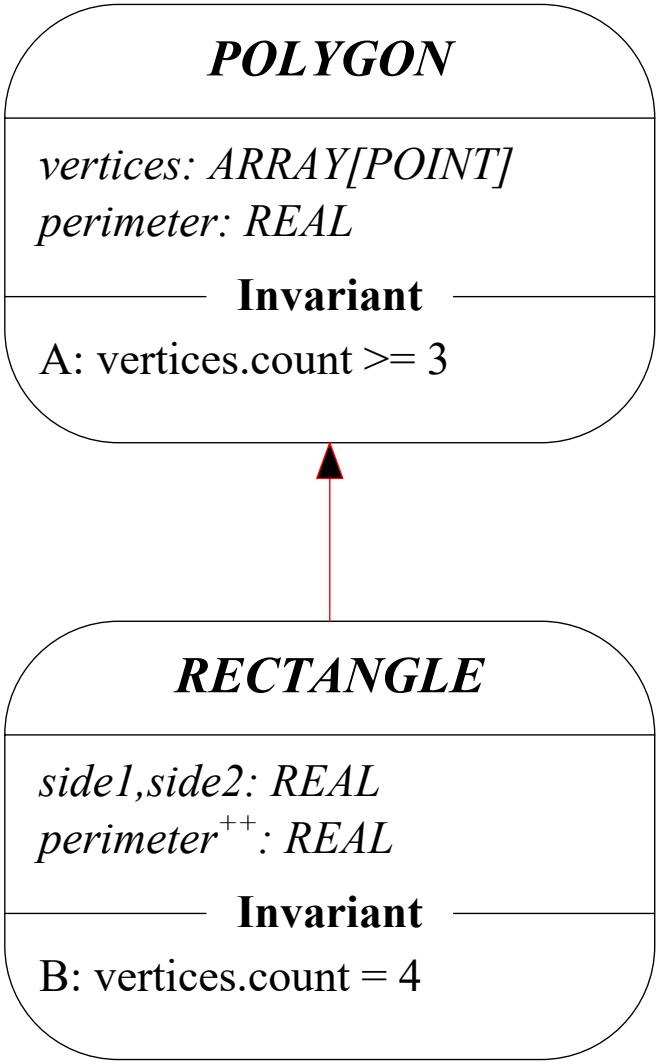
See Section 16.9 Touch of Class

Invariant accumulation

- What is the "flat" contract for RECTANGLE?
- Every class inherits all the invariant clauses of its parents.
- These clauses are conceptually "and"-ed.

$$vertices.count \geq 3 \wedge vertices.count = 4$$

$$\equiv vertices.count = 4$$



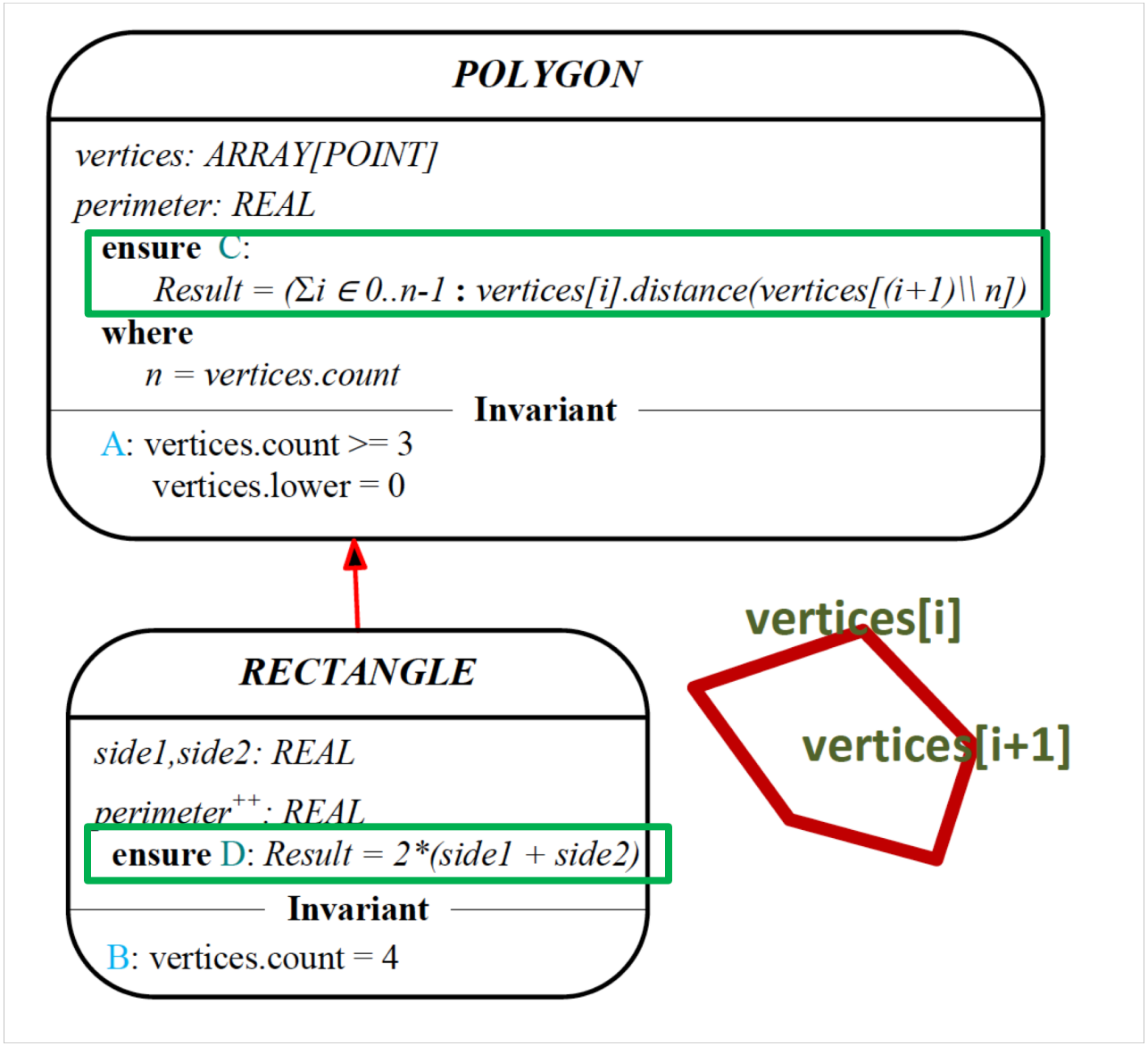
Accumulating invariants

$$vertices.count \geq 5 \wedge vertices.count = 4$$

$$\equiv false$$

- Every routine will fail with an invariant contract violation

Sub-Contracting (postcondition)

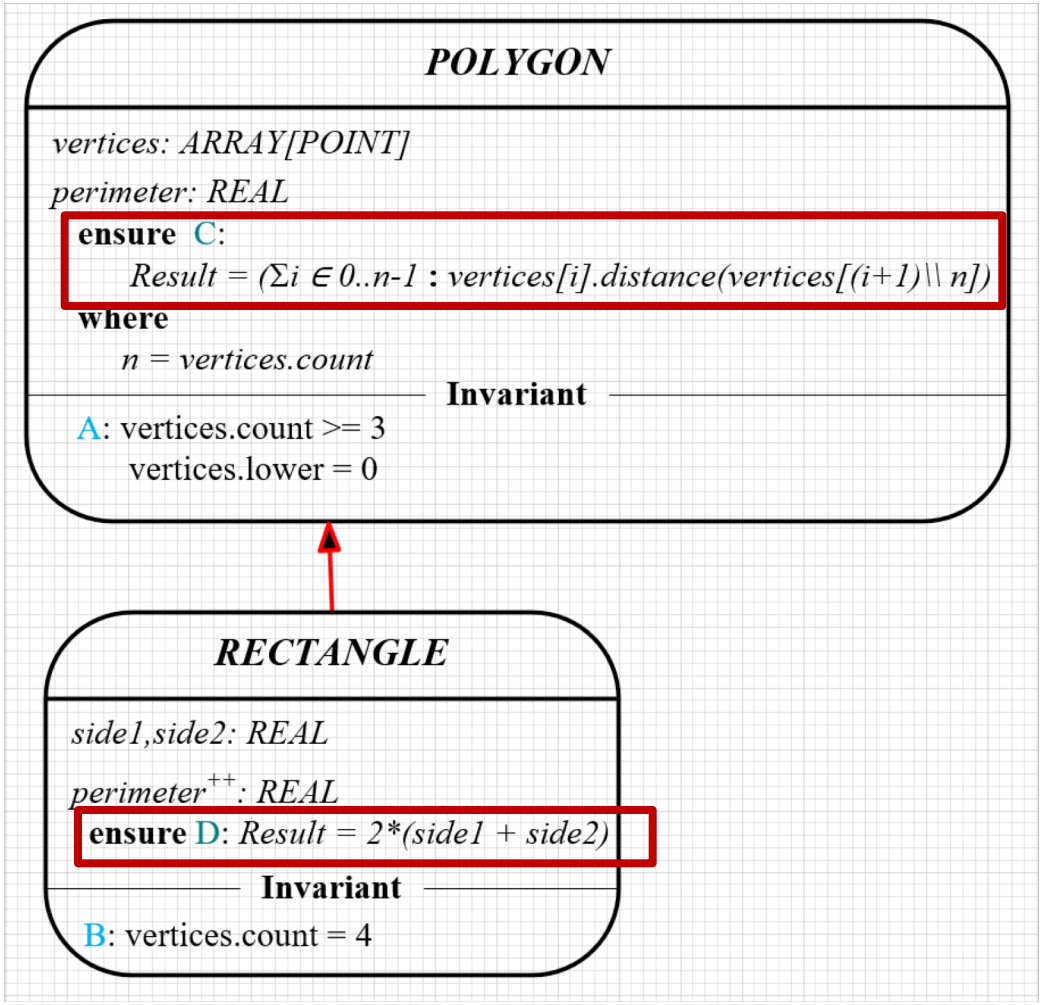


Inheritance and assertions

- What is the relationship between postcondition C in POLYGON and D in RECTANGLE?
- In descendant RECTANGLE
  - Require less
  - Ensure more
    - i.e. $D \Rightarrow C$

Why?

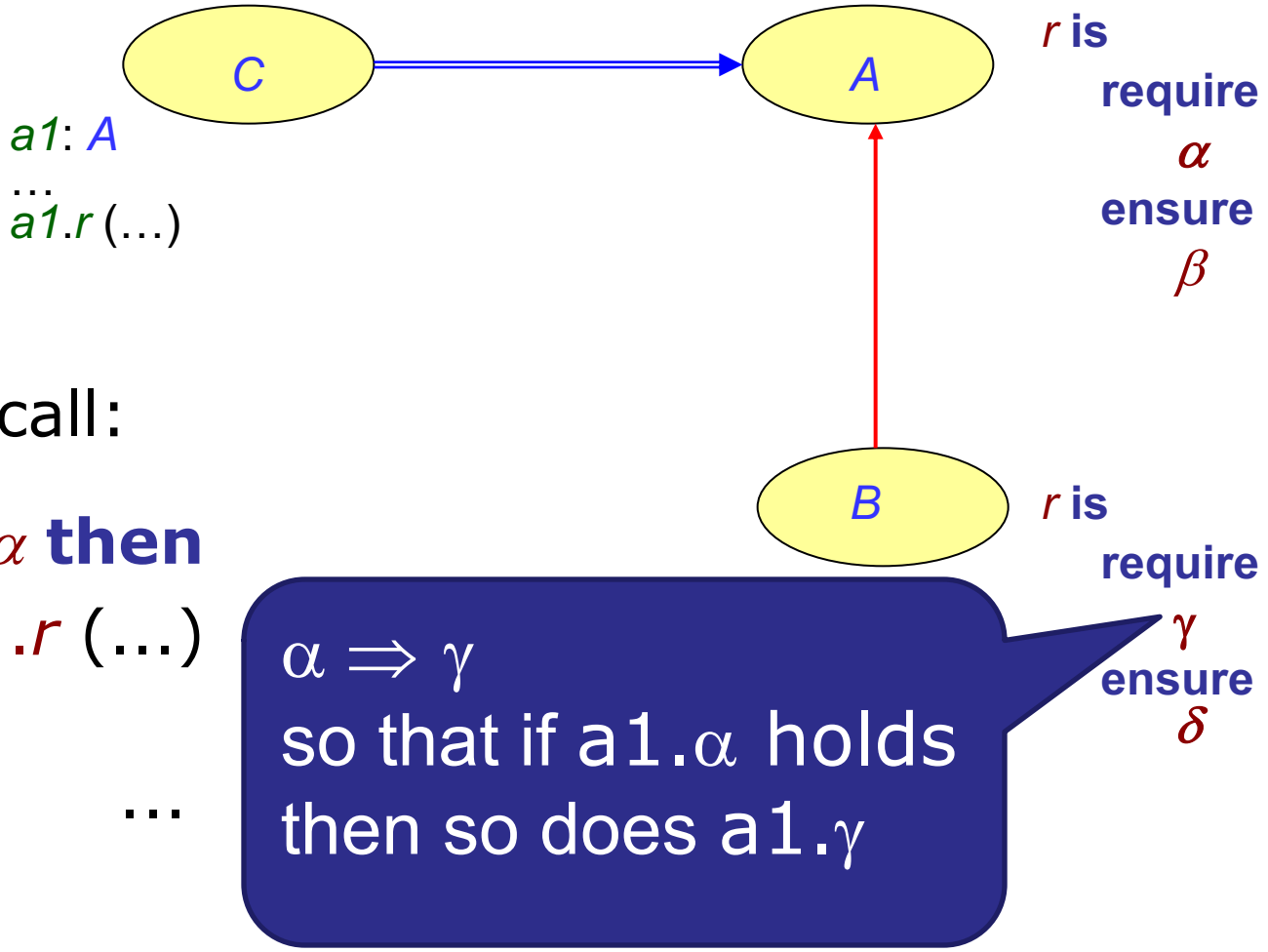
- So that RECTANGLE should not be able to change the meaning of perimeter to area
- Substitutivity



Substitutivity

- Require less / Ensure more constrains an instance of RECTANGLE to act like an instance of POLYGON.
- Hence the instance of RECTANGLE can always be substituted for an instance of POLYGON without changing the meaning of function routine *perimeter*
- Without contracts you could not truly ensure substitutivity

Precondition Subcontracting



Correct call:

if *a1*.α **then**

a1.r (...)

else

...

end

Assertion redeclaration rule

- Redefined version may not have **require** or **ensure**.
- May have nothing (assertions kept by default), or

require else *new_pre*
ensure then *new_post*

- Resulting assertions are:
  - *original_precondition* **or** *new_pre*
  - *original_postcondition* **and** *new_post*

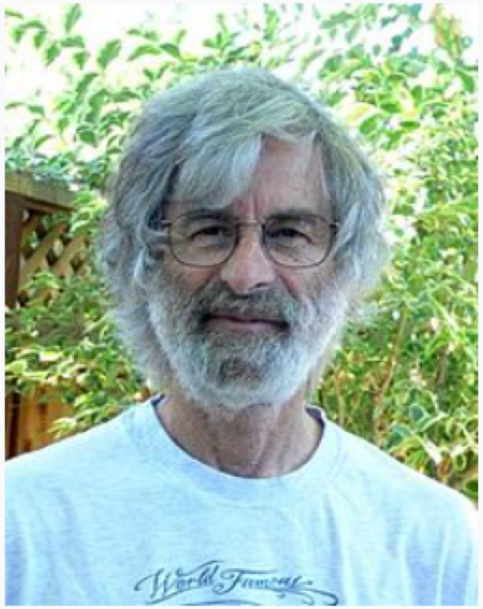
Leslie Lamport

From Wikipedia, the free encyclopedia

Leslie B. Lamport (born February 7, 1941) is an American computer scientist. Lamport is best known for his seminal work in distributed systems and as the initial developer of the document preparation system LaTeX.[2] Leslie Lamport was the winner of the 2013 Turing Award[3] for imposing clear, well-defined coherence on the seemingly chaotic behavior of distributed computing systems, in which several autonomous computers communicate with each other by passing messages. He devised important algorithms and developed formal modeling and verification protocols that improve the quality of real distributed systems. These contributions have resulted in improved correctness, performance, and reliability of computer systems.[4][5][6][7][8]

Contents [hide]

Leslie Lamport



http://www.budiu.info/blog/lamport.html

Specifications

- Architects draw detailed plans before a brick is laid or a nail is hammered. Programmers and software engineers don't. *Can this be why houses seldom collapse and programs often crash?*
- Blueprints help architects ensure that what they are planning to build will work. "Working" means more than not collapsing; it means serving the required purpose. Architects and their clients use blueprints to understand what they are going to build before they start building it.
- But few programmers write even a rough sketch of what their programs will do before they start coding.
- **Specifications**: To designers of complex systems, the need for formal specifications should be as obvious as the need for blueprints of a skyscraper. But few software developers write *specifications* because they have little time to learn how on the job, and they are unlikely to have learned in school. Some graduate schools teach courses on specification languages, but few teach how to use specification in practice. It's hard to draw blueprints for a skyscraper without ever having drawn one for a toolshed.

Why Contracts?

- Specifications (vs. implementations)
- Documents the contract between client and supplier
  - contract view (information hiding)
- Validating the implementation
  - Check that the implementation satisfies the specification
  - Catches many errors
  - Verification via formal methods
- Defines an exception (contract violation)
- Subcontracting (ensures substitutivity)

The meaning of inheritance

- The type perspective:
  - Automatic adaptation of operations to the dynamic form of their targets (extendibility): Representation independence.
  - Factoring out reusable operations: Commonality.
- The module perspective (reusability):
  - Open-closed modules
  - Modularity
  - The Single Choice Principle

People fiercely resist any effort to make them change what they do. Given how bad they are at writing programs, one might naively expect programmers to be eager to try new approaches. But human psychology doesn't work that way, and instead programmers will find any excuse to dismiss an approach that would require them to learn something new. On the other hand, they are quick to embrace the latest fad (extreme programming, templates, etc.) that requires only superficial changes and allows them to continue doing things basically the same as before.

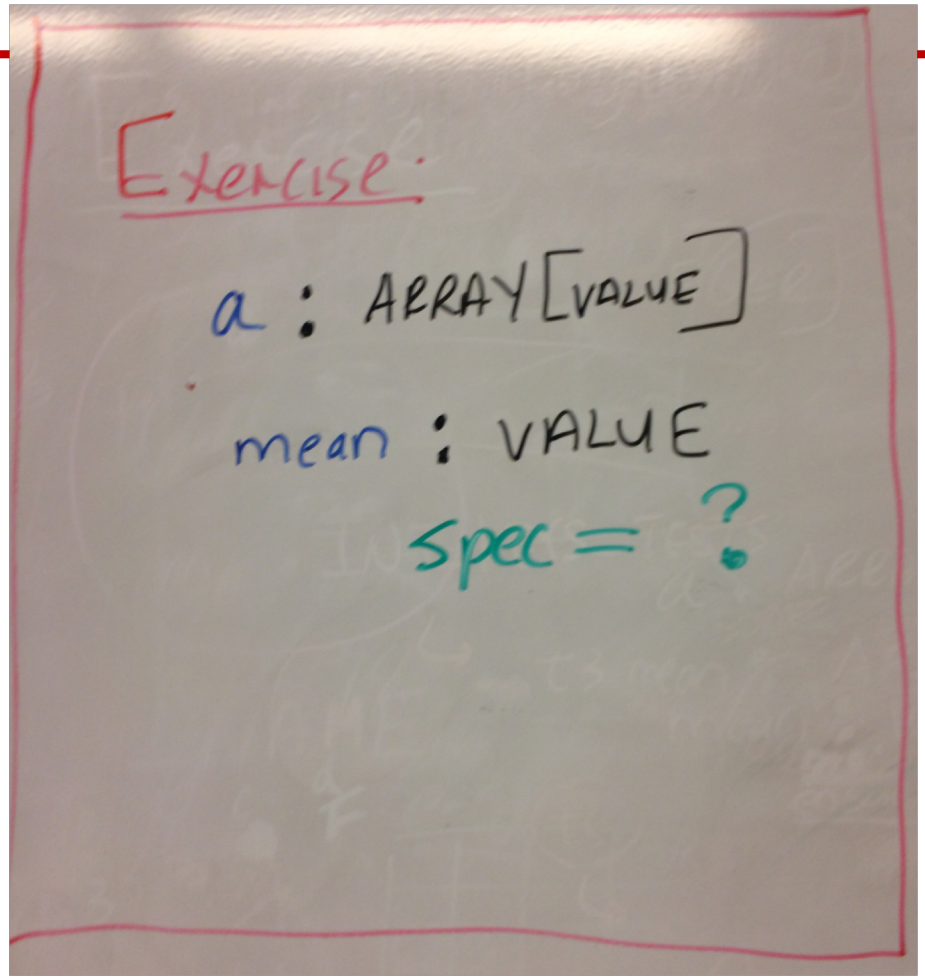
The fundamental idea behind verification is that one should think about what a program is supposed to do before writing it. Thinking is a difficult process that requires a lot of effort. Spending a few hours thinking before writing code can save days of debugging and rewriting.

I can offer only two general pieces of advice on how to think. The first is to write. As the cartoonist Guindon once wrote, "writing is nature's way of showing you how fuzzy your thinking is." Before writing a piece of code, write a description of exactly what that piece of code is supposed to accomplish. This applies whether the piece is an entire program, a procedure, or a few lines of code that are sufficiently non-obvious to require thought. The best place to write such a description is in a comment.

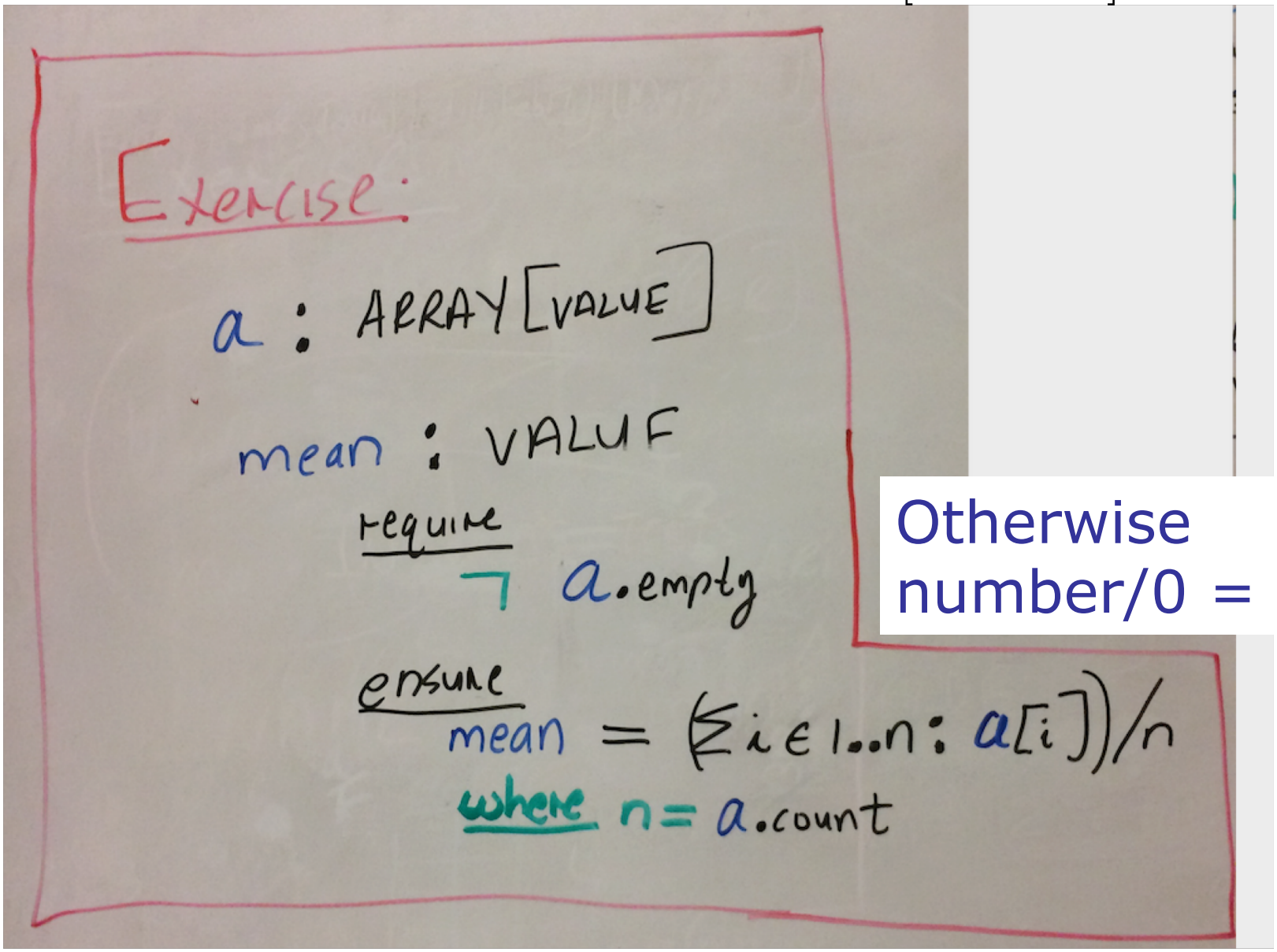
People have come up with lots of reasons for why comments are useless. This is to be expected. Writing is difficult, and people always find excuses to avoid doing difficult tasks. Writing is difficult for two reasons: (i) writing requires thought and thinking is difficult, and (ii) the physical act of putting thoughts into words is difficult. There's not much you can do about (i), but there's a straightforward solution to (ii) — namely, writing. The more you write, the easier the physical process becomes. I recommend that you start practicing with email. Instead of just dashing off an email, write it. Make sure that it expresses exactly what you mean to say, in a way that the reader will be sure to understand it.

```
note
        description: "[
                        Keep track of birthdays of my friends by name,
                        and remind me when it is their birthday
                ]"
class
        BIRTHDAY_BOOK
inherit
        ITERABLE[BIRTHDAY]

feature -- model
        model: FUN[NAME, BIRTHDAY]
                -- maps names of friends to their birthdays
feature
        remind(d: BIRTHDAY): ARRAY[NAME]
                        -- return an array of all names with birthday `d'
                ensure
                        ∀n ∈ Result: n ∈ (model ▷ {d}).domain
                        Result.count = (model ▷ {d}).count
                        -- symbol ▷ is range restriction
                end

        ...
end
```

As you get better at using math to describe things, you may discover that you need to be more precise than mathematicians usually are. When your code for computing the mean of n numbers crashes because it was executed with n = 0, you will realize that the description of the mean above was not precise enough because it didn't define the mean of 0 numbers. At that point, you may want to learn some formal language for writing math. But if you've been doing this diligently, you will have the experience to decide which languages will actually help you solve your problem.



$$mean : ARRAY[VALUE] \nrightarrow VALUE$$



Otherwise number/0 = ∞

Hoare specification of code

{P} prog {Q}

Execution of the code **prog**

- started in a state satisfying **P**
- is guaranteed to terminate in a state satisfying **Q**

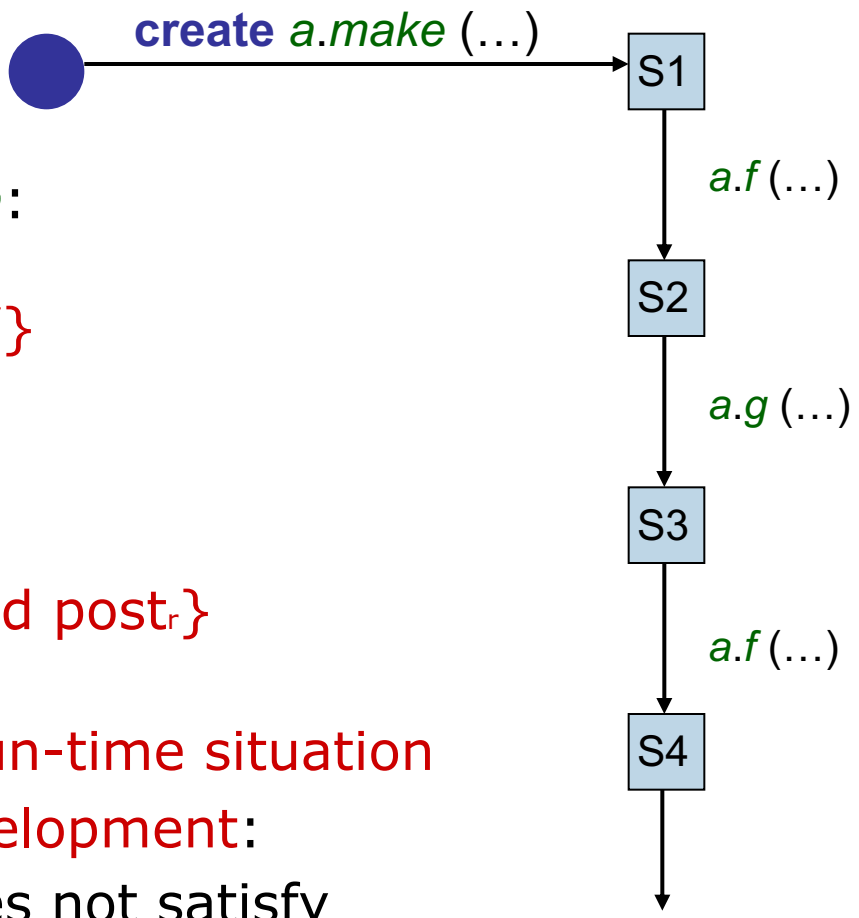
{a:ARRAY[NUMBER]}

sort

{∀i ∈ 1..(a.count-1) : a[i] ≤ a[i+1]

∧ permutation(a, old a)}

The dangers of static binding



- For every creation procedure *cp*:

  $\{pre_{cp}\}\ do_{cp}\ \{post_{cp}\ and\ INV\}$

- For every exported routine *r*:

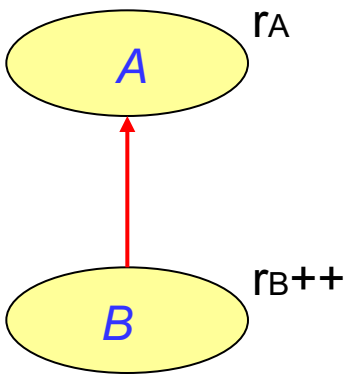
  $\{INV\ and\ pre_{r}\}\ do_{r}\ \{INV\ and\ post_{r}\}$

- The worst possible erroneous run-time situation in object-oriented software development:
  - Producing an object that does not satisfy the invariant of its class.

The dangers of static binding

- $\{INV_A\}\ do_{r_A}\ \{INV_A\}$

- $\{INV_B\}\ do_{r_B}\ \{INV_B\}$



- Consider a call of the form *a1*.*r* where *a1* is polymorphic
  - r_A preserves INV_A and r_B preserves INV_B
  - But r_A has no reason to preserves INV_B
  - So *a1*.*r* (under static binding) executes r_A on objects of type B, thus resulting in an inconsistent object (one that does not satisfy INV_B)

Using original version of redefined feature

```
class BUTTON  inherit
        WINDOW
                redefine
                        display
                end
feature
        display
                do
                        Precursor
                        display_border
                        display_label
                end

        display_label
                do
                        ...
                end
        display_border
                do
                        ...
                end
end
```

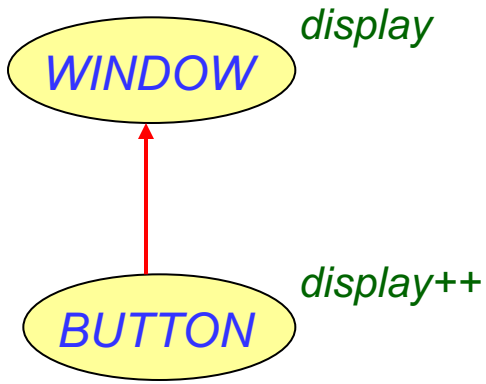
A concrete example

w: *WINDOW*
b: *BUTTON*

create *b*

w := *b*

w.*display*

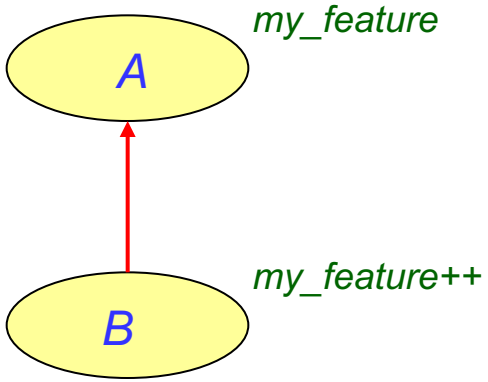


Static binding would execute
{WINDOW}.display
on the button rather than
{BUTTON}.display

Use of Precursor

- Not necessarily the first feature statement.
- May have arguments.

```
class B  inherit
        A
                redefine
                        my_feature
                end
feature
        my_feature (args: SOME_TYPE)
                do
                        -- Something here
                        Precursor (args)
                        -- Something else here
                end

end
```



Assignment Attempt (OOSC2 16.5)

- Suppose we want to find out the longest diagonal (rectangles only) in a list of figures.
- Suppose we were retrieving the list from a network or from a stored file where we could not guarantee the type of the object.

Forcing a type: the problem

```
list: LIST [FIGURE]
x: RECTANGLE
list.store ("FILE_NAME")
```

- Two years later:

```
list := retrieved ("FILE_NAME")
x := list.item          -- [1]
print (x.diagonal)      -- [2]
```

- But
  - If *x* is declared of type *RECTANGLE*, [1] will not compile.
  - If *x* is declared of type *FIGURE*, [2] will not compile.

What not to do!

if

"list.item is of type RECTANGLE"

then

…

elseif

"list.item is of type CIRCLE"

then

etc.

- We could so the above as in *ANY* we have:

  conforms_to (*other*: ANY): BOOLEAN **is**

  -- Is type of current object identical to type of `*other*'?

Assignment Attempt

- x: **detachable** FIGURE
- (and RECTANGLE inherits from FIGURE)

```
if attached {RECTANGLE} x as xr then
   ... xr.diagonal ...
else
   ...
end
```

Calculating largest diagonal in list – Solution

```
somelist: detachable ANY

maximum_diagonal (file_name: STRING): REAL
        -- Max value of diagonals of rectangles in list retrieved from storage or network
        -- -1 if none
    do
        Result := -1
        somelist := retrieved("file_name")

        if attached {LIST[FIGURE]} somelist as list then
                -- file contains an object structure that conforms to LIST[FIGURE]
                from list.start;
                until list.after
                loop
                        if attached {RECTANGLE} list.item as r then
                            Result := Result.max(r.diagonal)
                        end
                        list.forth
                end
        end
    end
```

Object Test via Attached

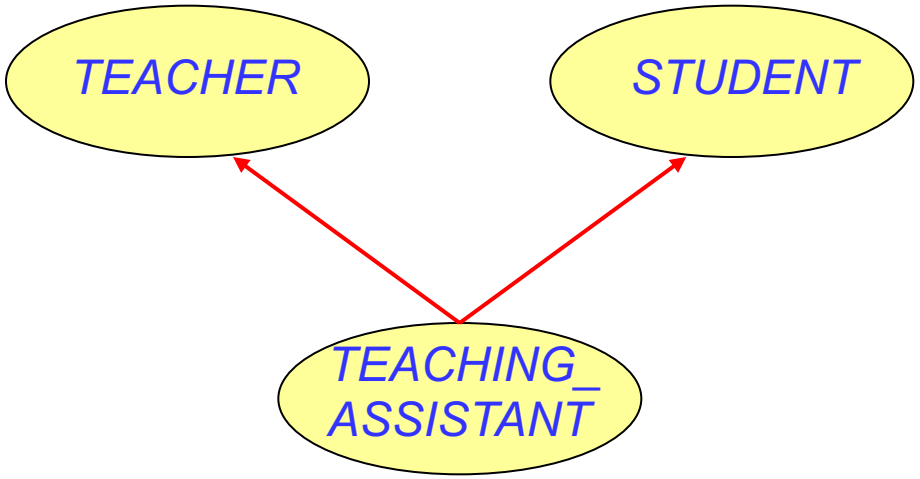
```
x: detachable Tx
y: Tx

x := y

if attached x as xa then
      xa.f (a)
else
      ….
end
```

"**attached** x" is of type BOOLEAN

xa is a local variable of type **attached** Tx

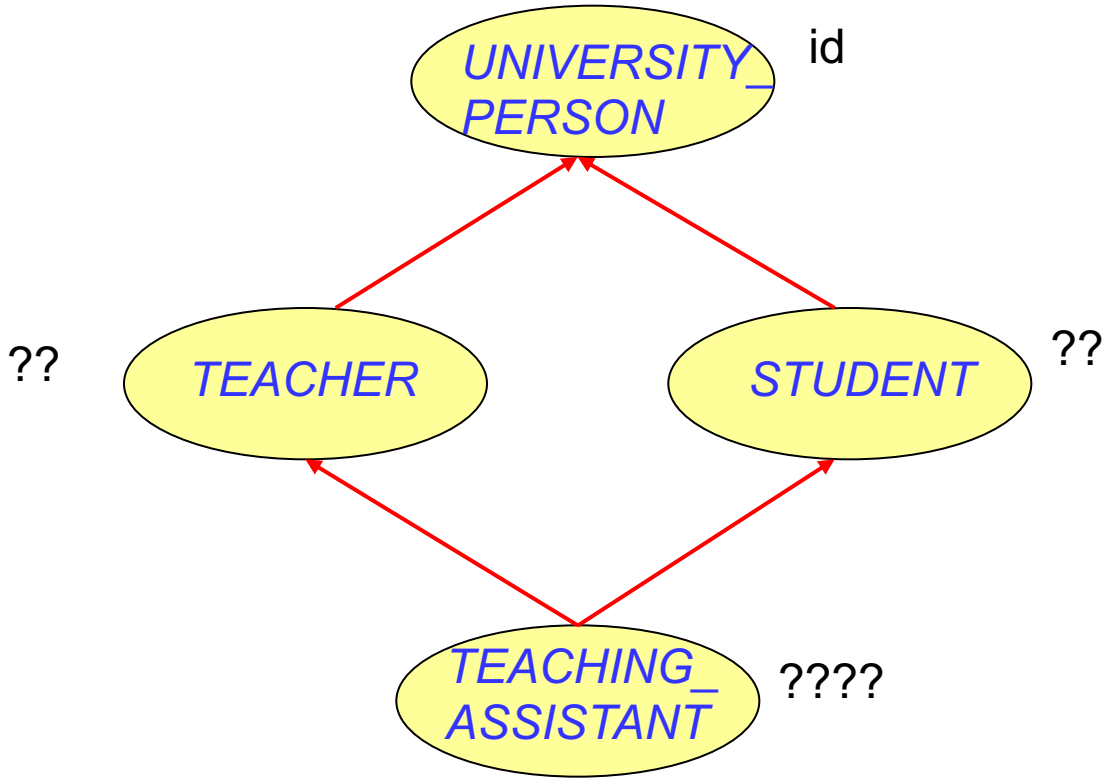
Multiple inheritance

- A class may have two or more parents.
- What not to use as an elementary example: *TEACHING_ASSISTANT* inherits from *TEACHER* and *STUDENT*.



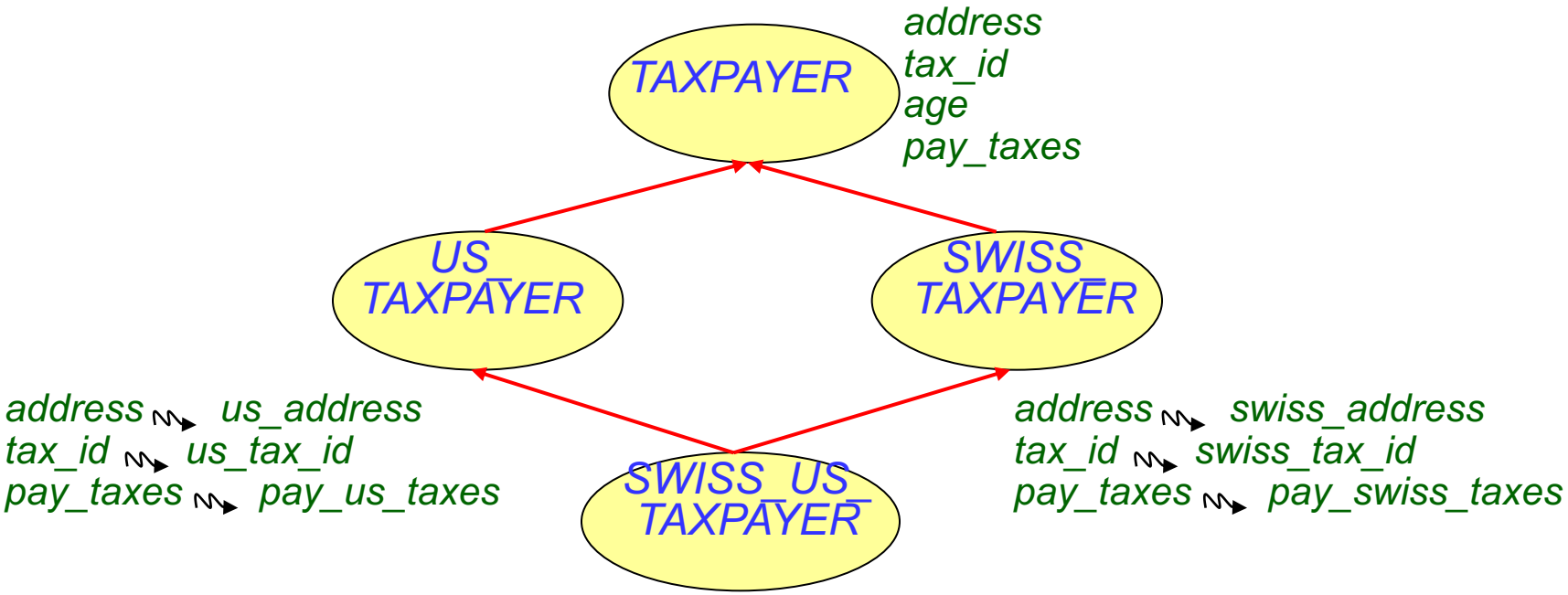
The teaching assistant example

- This is in fact a case of repeated inheritance:



Sharing and replication

- Features such as *age* and *birthday*, not renamed along any of the inheritance paths, will be shared.
- Features such as *tax_id*, *violation_count* inherited under different names, will be replicated.

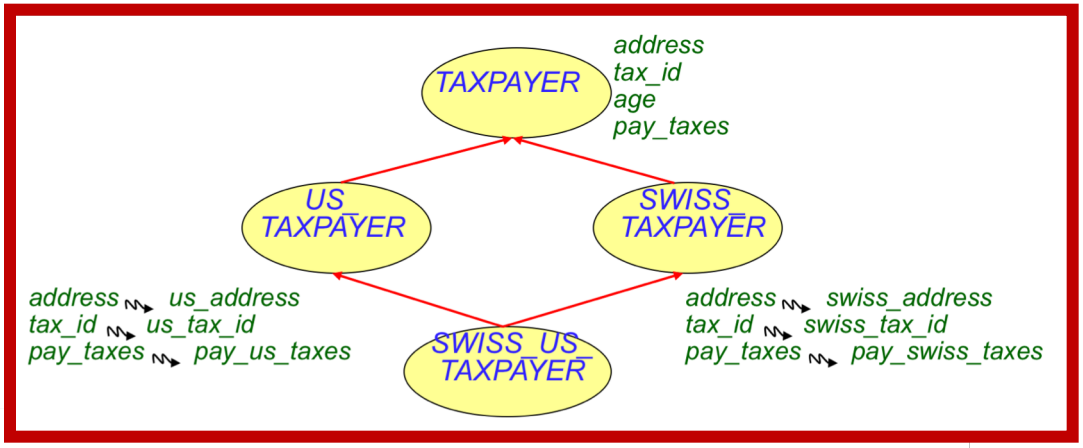


```
class
        US_TAX_PAYER
inherit
        TAX_PAYER

end
```

```
class
    SWISS_US_TAX_PAYER
inherit
    US_TAX_PAYER
        rename
            id as us_id
        select
            us_id
        end
    SWISS_TAX_PAYER
        rename
            id as swiss_id
        end

create
    make
end
```

```
-- FLAT VIEW
class
	SWISS_US_TAX_PAYER
creation
	make
feature {NONE} -- Initialization

	make (a_name: STRING_8; a_id: INTEGER_32)
			-- Initialization for `Current`.
			-- (from TAX_PAYER)
		do
			name := a_name
			us_id := a_id
		end
feature
	swiss_id: INTEGER_32
			-- (from TAX_PAYER)
	us_id: INTEGER_32
			-- (from TAX_PAYER)
			-- This one is selected for
			-- {TAX_PAYER}id
	name: STRING_8
			-- (from TAX_PAYER)
			--
end -- class SWISS_US_TAX_PAYER
```

```
class TAX_PAYER create
    make
feature {NONE} -- Constructor
    make(a_name:STRING; a_id: INTEGER)
        do
            name := a_name
            id := a_id
        end

feature
    name: STRING
    id: INTEGER
end
```

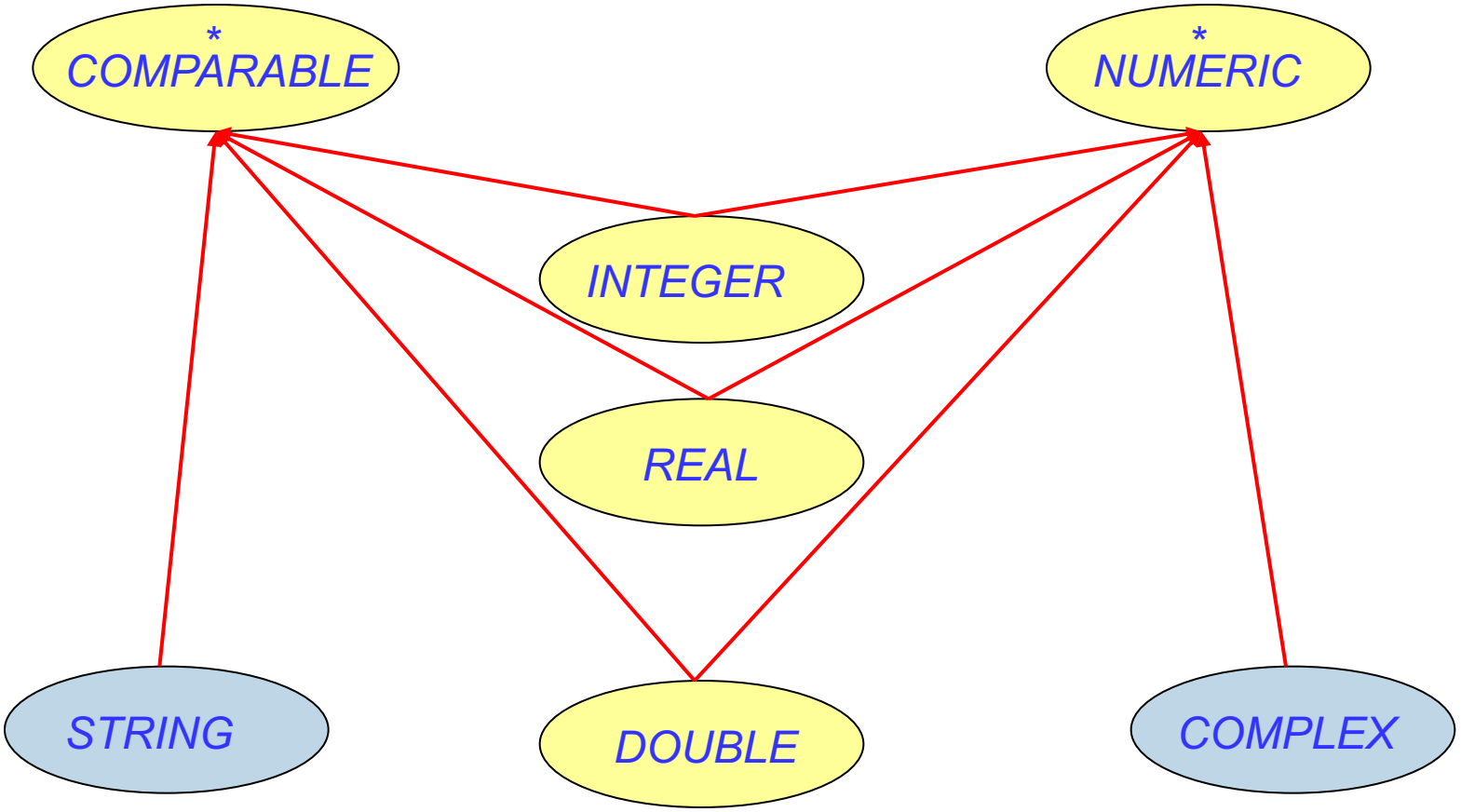



```
t1: BOOLEAN
    local
        t: TAX_PAYER
        st: SWISS_TAX_PAYER
        ut: US_TAX_PAYER
        sut: SWISS_US_TAX_PAYER
    do
        create ut.make ("Washington", 1)
        Result := ut.id = 1
        check Result end
        create st.make ("Zurich", -1)
        Result := st.id = -1
        check Result end
        create sut.make ("Swiss-US", 250)
        Result := sut.us_id = 250 and sut.swiss_id = 0
        check Result end
        t := sut
        Result := t.id = 250
    end
```

Common examples of multiple inheritance

- Combining separate abstractions:
  - Restaurant, train car
  - Calculator, watch
  - Plane, asset

Multiple inheritance: Combining abstractions



Multiple inheritance: Nested windows

- ‘‘Graphical’’ features: *height*, *width*, *change_height*, *change_width*, *xpos*, *ypos*, *move*...
- ‘‘Hierarchical’’ features: *superwindow*, *subwindows*, *change_subwindow*, *add_subwindow*...

class

WINDOW

inherit

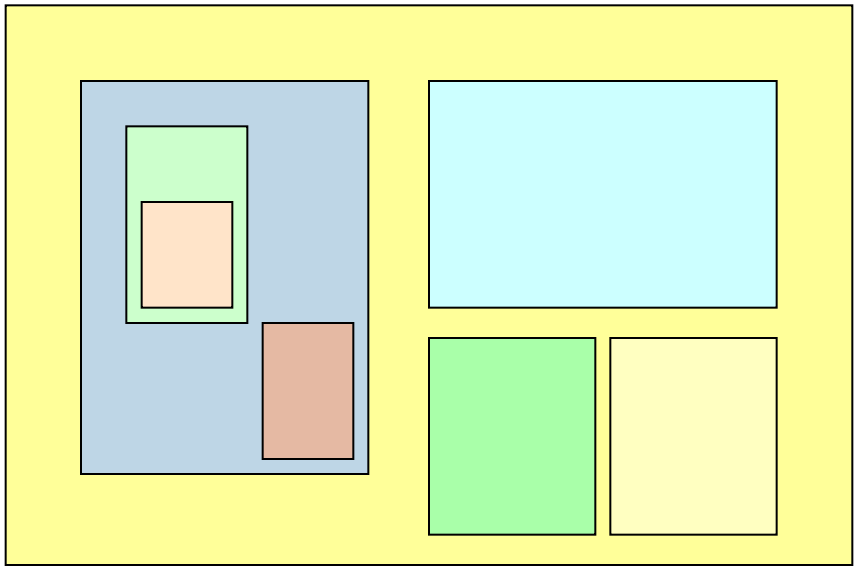
RECTANGLE

TREE [*WINDOW*]

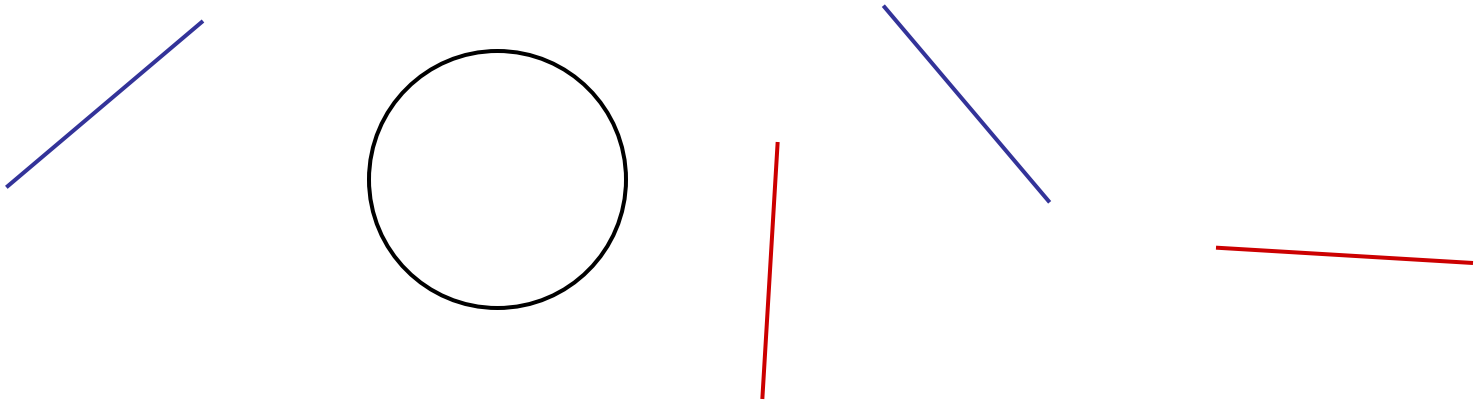
feature

...

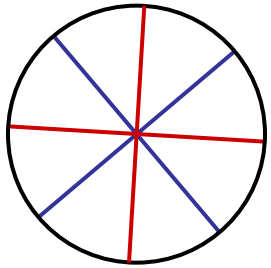
end



Multiple inheritance: Composite figures

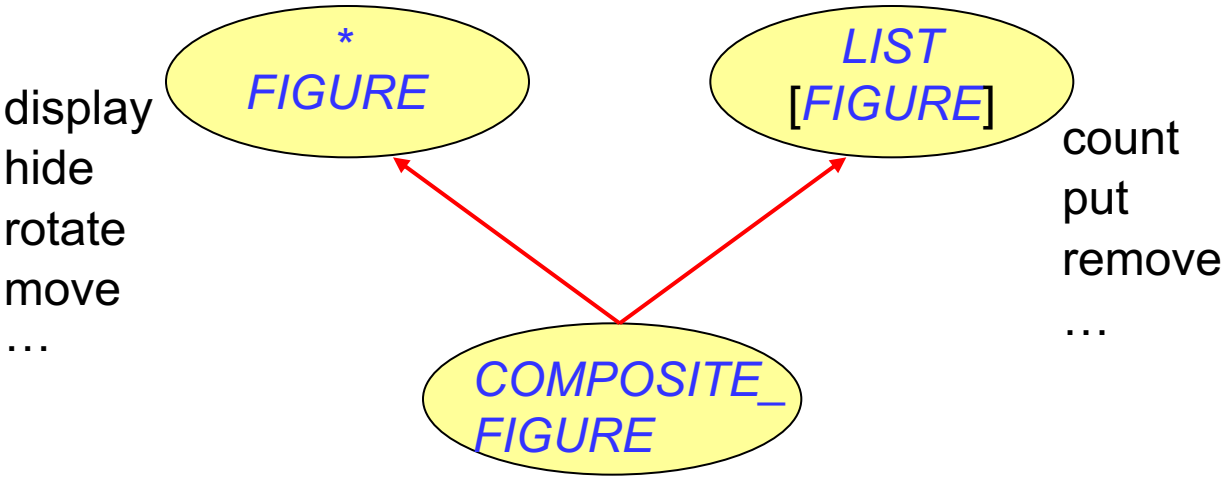


Simple figures

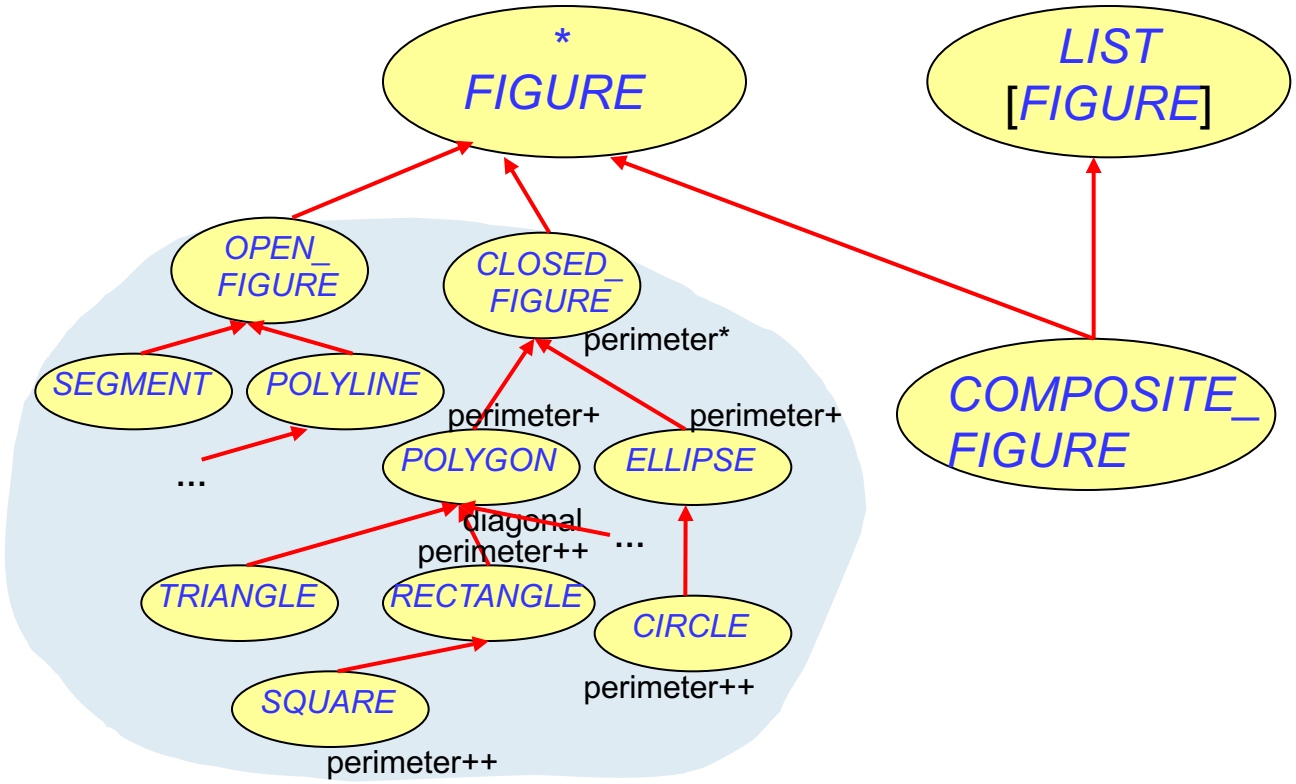


A composite figure

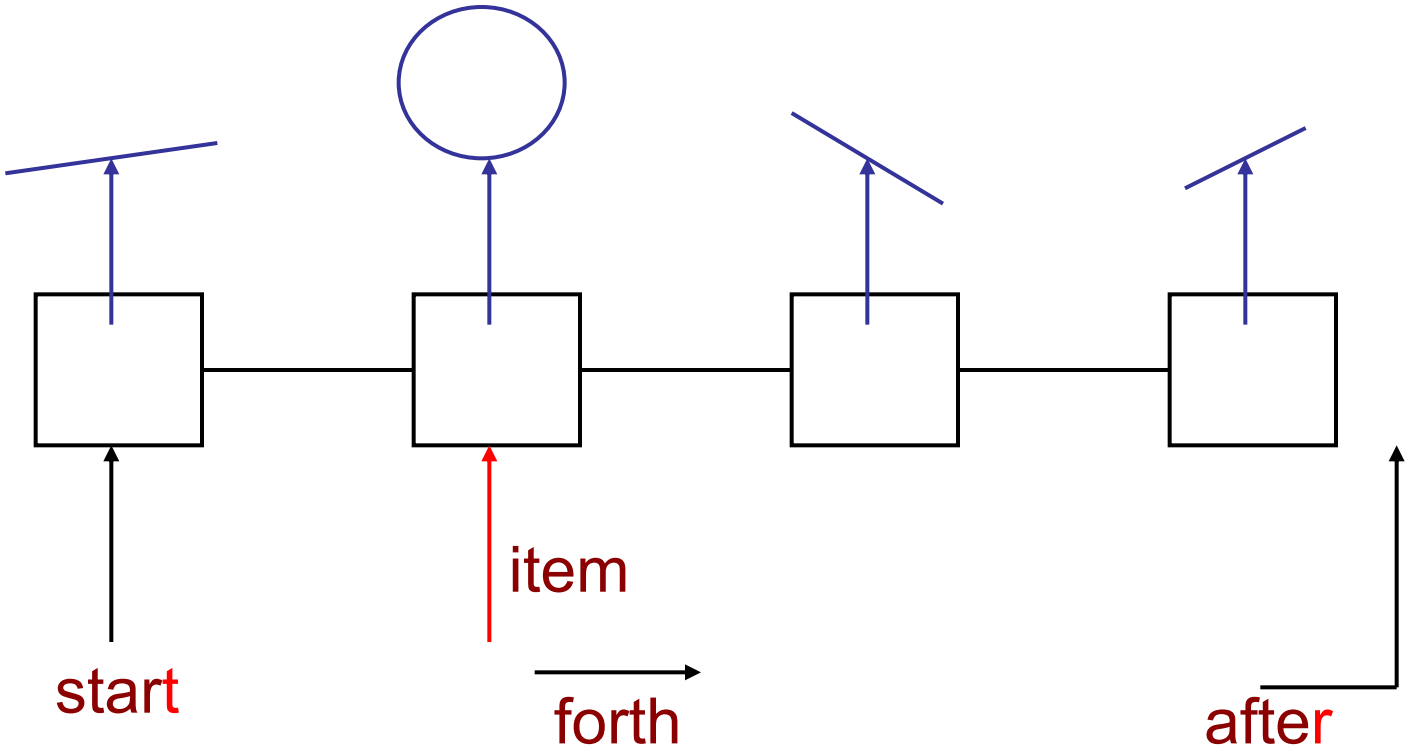
Defining the notion of composite figure



Defining the notion of composite figure through multiple inheritance



A composite figure as a list



Composite figures

```
class COMPOSITE_FIGURE  inherit

    FIGURE
            redefine display, move, rotate, ... end

    LIST [FIGURE]

feature

    display is
                    -- Display each constituent figure in turn.
            do
                    from
                            start
                    until
                            after
                    loop
                            item.display
                            forth
                    end
            end

    ... Similarly for move, rotate etc. ...

end
```

Close to a Template Design pattern

Define the skeleton of an algorithm, deferring some steps (e.g. item.display) to client subclasses.

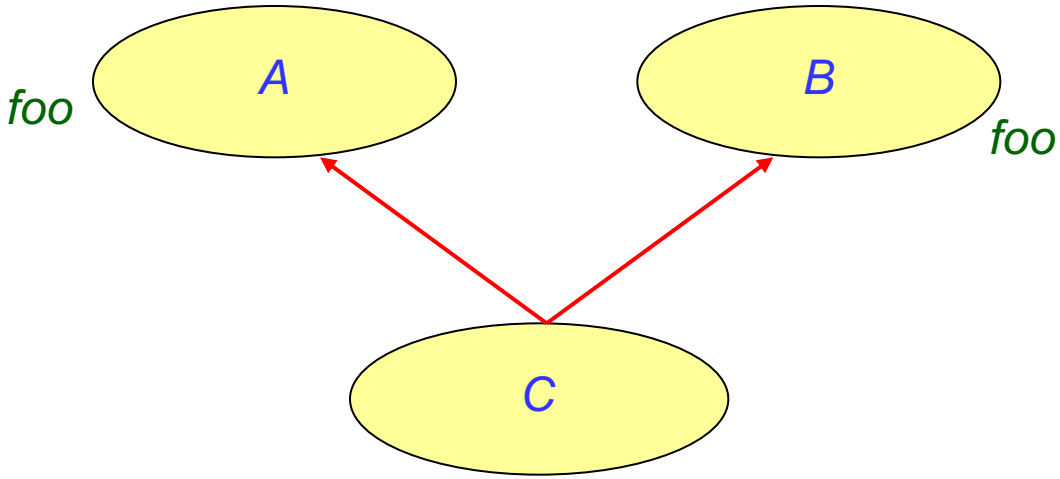
Polymorphism, static typing and dynamic binding

Complex figures

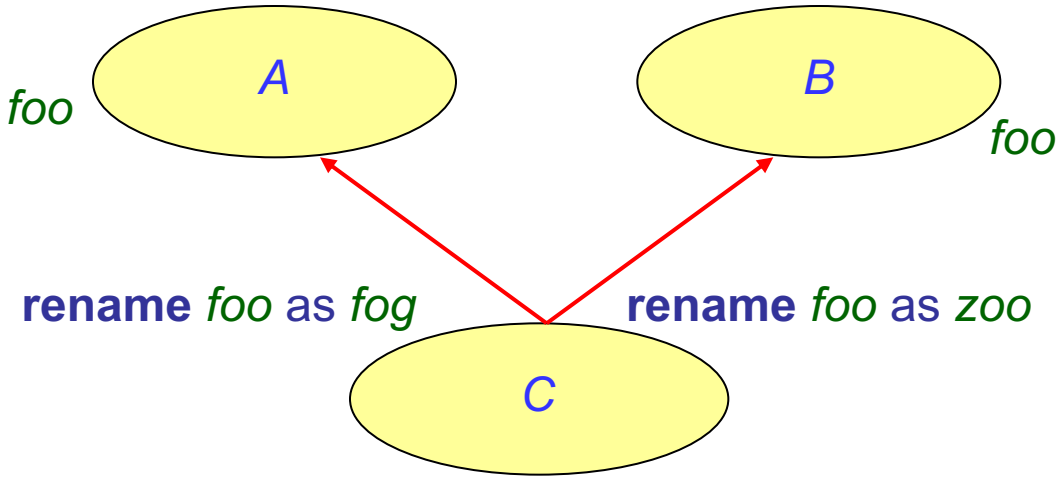
- Note: a simpler form of procedures *display*, *move* etc. can be obtained through the use of iterators.

- Exercise: Use agents for that purpose.

Name clashes under multiple inheritance



Resolving name clashes



Resolving name clashes (cont'd)

class *C*

inherit

A

rename
foo **as** *fog*
end

B

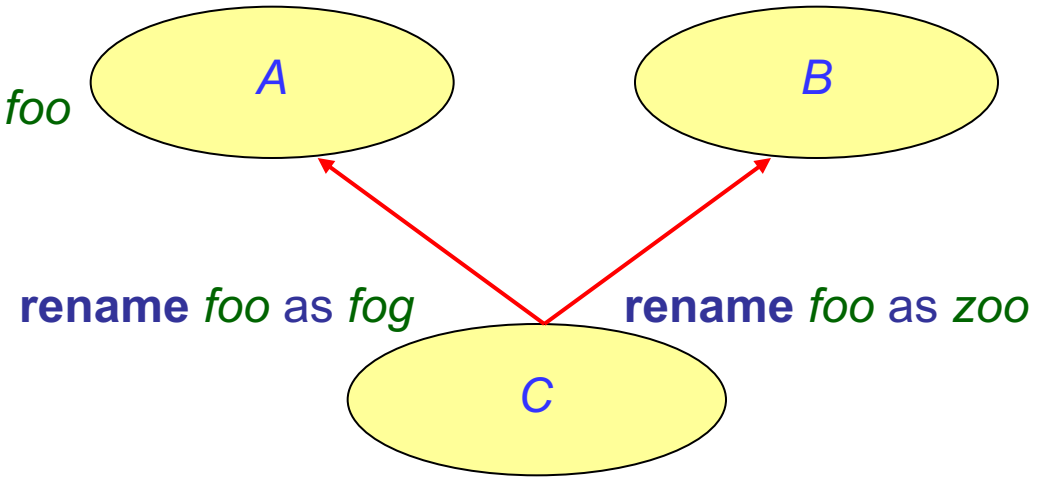
rename
foo **as** *zoo*
end

feature

...

Results of renaming

a1: *A*
b1: *B*
c1: *C*
...
c1.*fog*
c1.*zoo*
a1.*foo*
b1.*foo*



Invalid:

a1.*fog*, *a1*.*zoo*, *b1*.*zoo*, *b1*.*fog*, *c1*.*foo*

Another application of renaming

- Provide locally better adapted terminology.

- ***class*** *TREE* ***feature***
  child: TREE[WINDOW]
  end

- **class** WINDOW **inherit**
  TREE
  feature
  rename
  child **as** subwindow
  end
  end

The marriage of convenience

```
class
    ARRAYED_STACK [G]
inherit
    STACK [G]
    ARRAY [G]
feature
    ...
end

class
    LINKED_STACK [G]
inherit
    STACK [G]
    LINKED_LIST [G]
feature
    ...
end
```

The need for deferred classes

- In the scheme seen earlier:

```
f: FIGURE; c: CIRCLE; p: POLYGON
...
create c.make (...); create p.make (...)
...
if ... then
    f := c
else
    f := p
end
...
f.move (...); f.rotate (...); f.display (...); ...
```

- How do we ensure that a call such as *f.move* (...) is valid even though there is no way to implement a general-purpose feature *move* for class *FIGURE*?

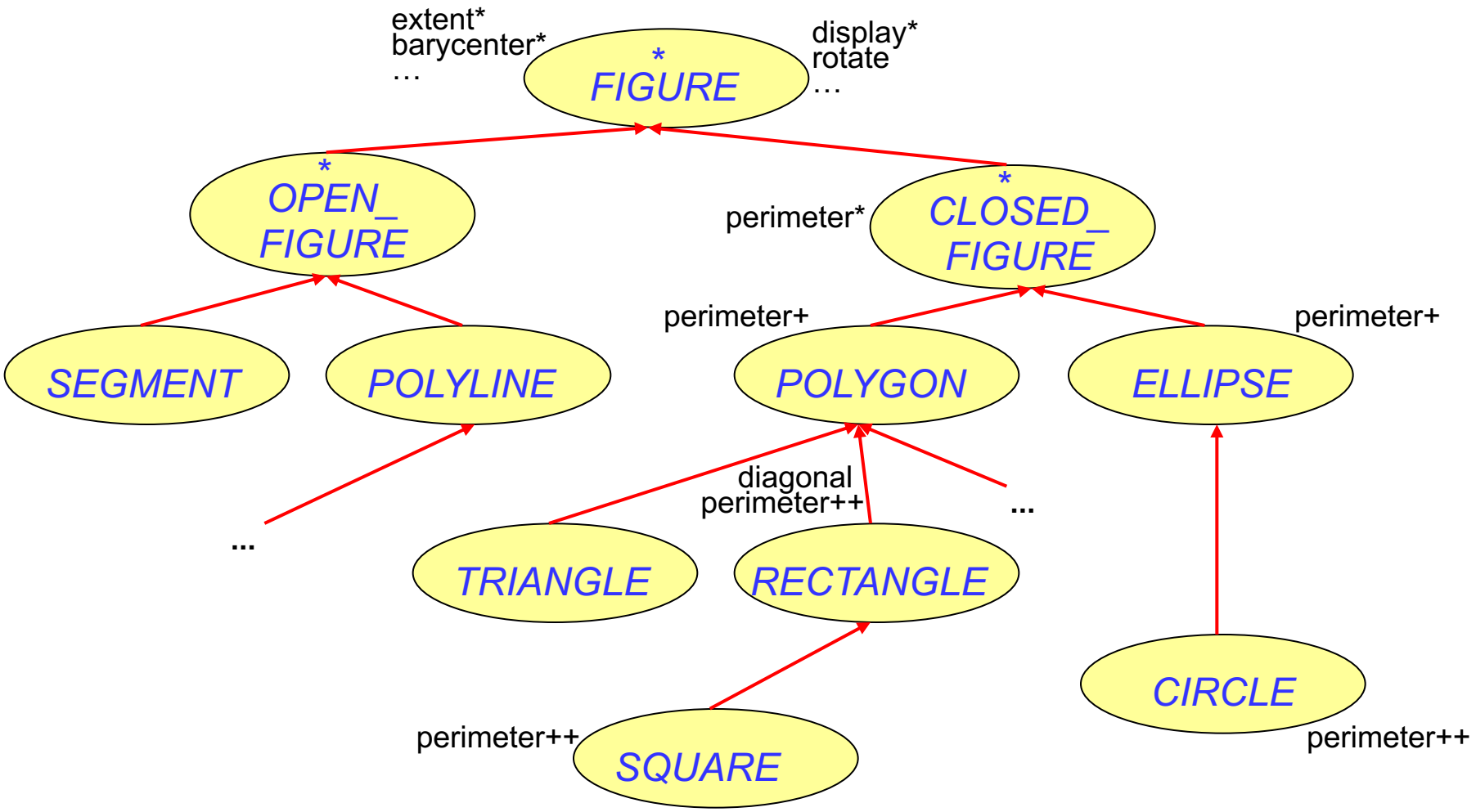
Deferred classes

```
deferred class
        FIGURE

feature

        move (v: VECTOR) is
                deferred
                end

        rotate (a: ANGLE; p: POINT) is
                deferred
                end

        ... display, hide, ...

end
```

Not permitted:

```
        create f ...
```

Example hierarchy



Deferred classes and features

- A feature is either deferred or effective.
- To effect a inherited feature (deferred in the parent) is to make it effective. No need for redefine clause.
- Like a feature, a class is either deferred or effective.
- A class is deferred if it has at least one deferred feature (possibly coming from an ancestor) that it does not effect. It is effective otherwise.
- A deferred class may not be instantiated.
- BUT: A deferred class may have assertions (in particular, a deferred routine may have a precondition and a postcondition, and the class may have a class invariant).

Summary

- **Inheritance**: the importance of polymorphism, static typing and dynamic binding
- **Sub-contracting:** understand OO inheritance (via the Liskov substitutability principle)
- **Specifications**: to think about what a program is supposed to do before writing it
  - Even informal comments
- **True Multiple inheritance**: as a design notation
  - supported in UML for design
  - Open-Closed Principle and other design principles
- **Design**: Thinking above the code level