Fondamenti della Programmazione: Metodi Evoluti

Prof. Enrico Nardelli

Lezione 7: Creazione e Void Safety

Identifiers, entities, variables

An identifier is a name chosen by the programmer to represent certain program elements

It may denote :

- A class, e.g. *ACROBAT*
- A feature, e.g. *count*
- A run time value, such as an object or object reference, e.g. *mario*

An identifier that denotes a run-time value is called an entity, or a variable if it can change its value

During execution an entity may become **attached** to an object

ACROBAT

```
class ACROBAT

feature
        clap (n: INTEGER)
                        -- Clap `n' times and forward to copycat.
            do
                        -- "Clap n times"

                        buddy.clap(n)

            end

        … … … … … … …


        buddy : COPYCAT
                        -- the copycat of this acrobat
end
```

Reference to an object

States of a reference

During execution, a reference is either:

- Attached to a certain object
- Void

- To denote a void reference: use the reserved word **Void**
- To find out if *x* is void, use the condition

  x = **Void**

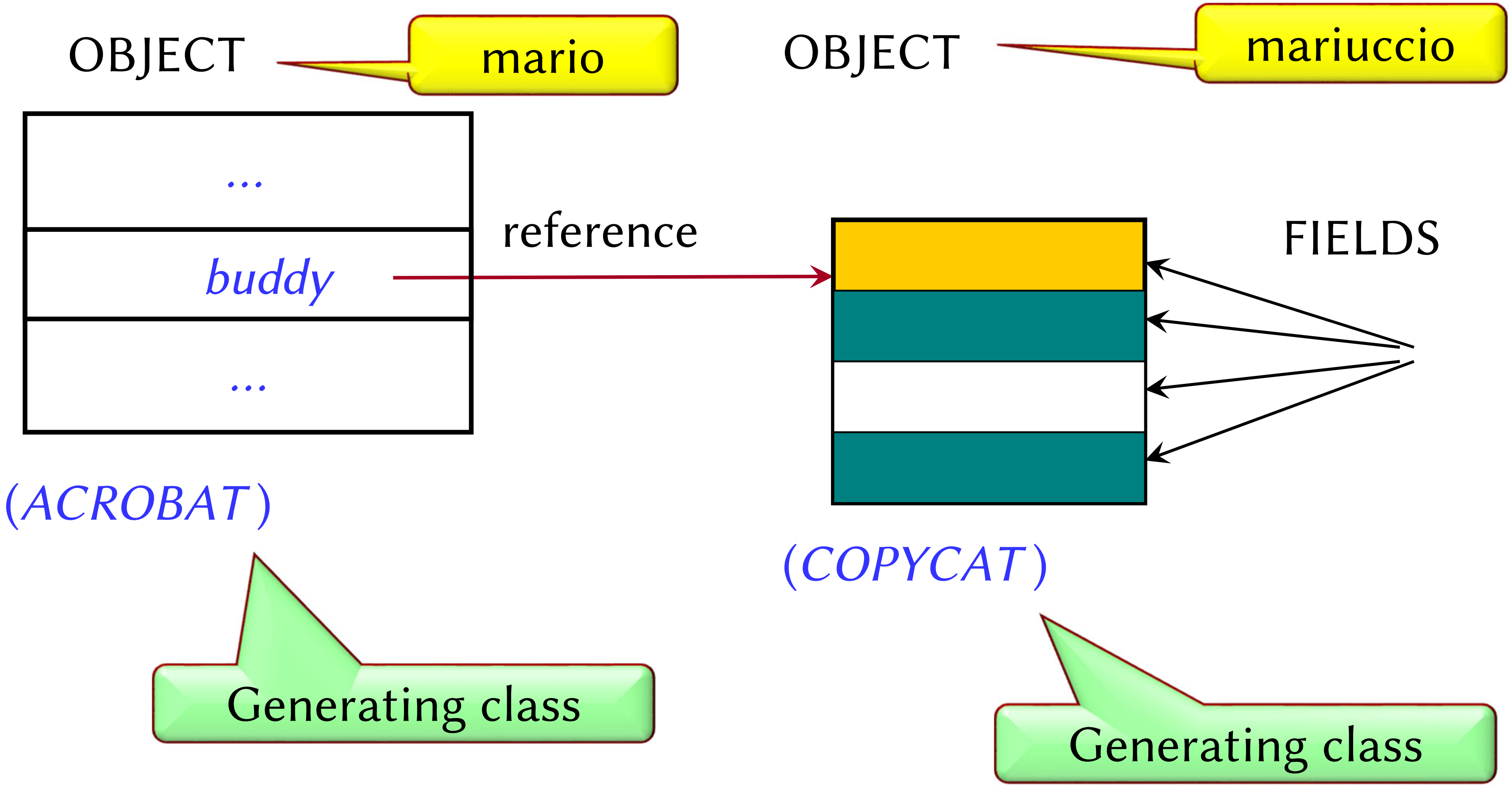
- Inverse condition (*x* is attached to an object):

  x /= **Void**

Entity attached to an object

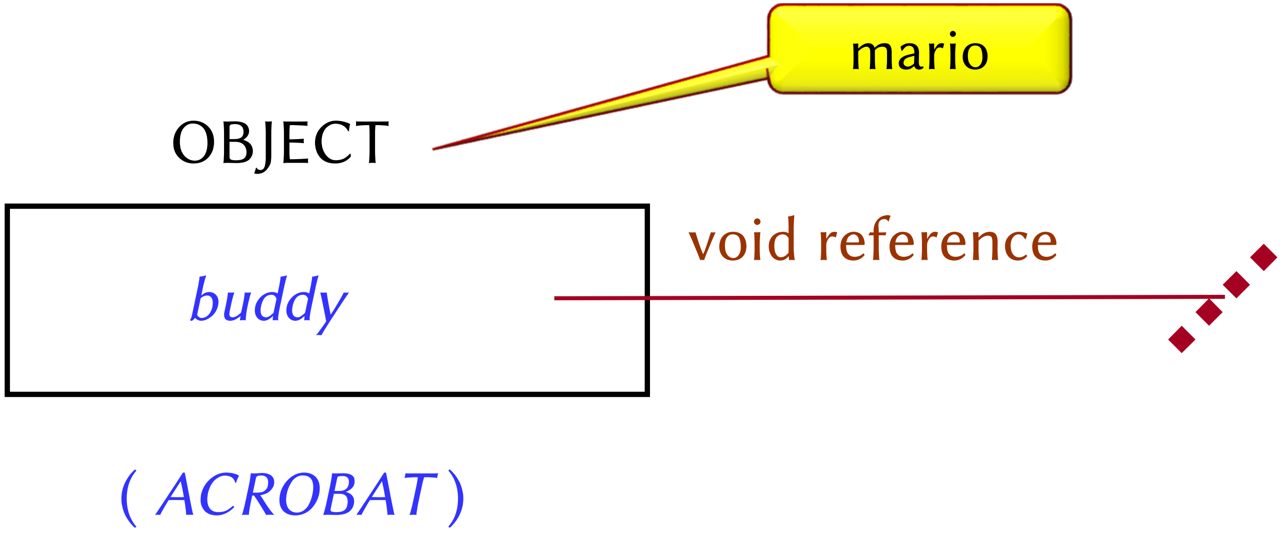
In the program: an entity, such as *mario*

In memory, during execution: an object



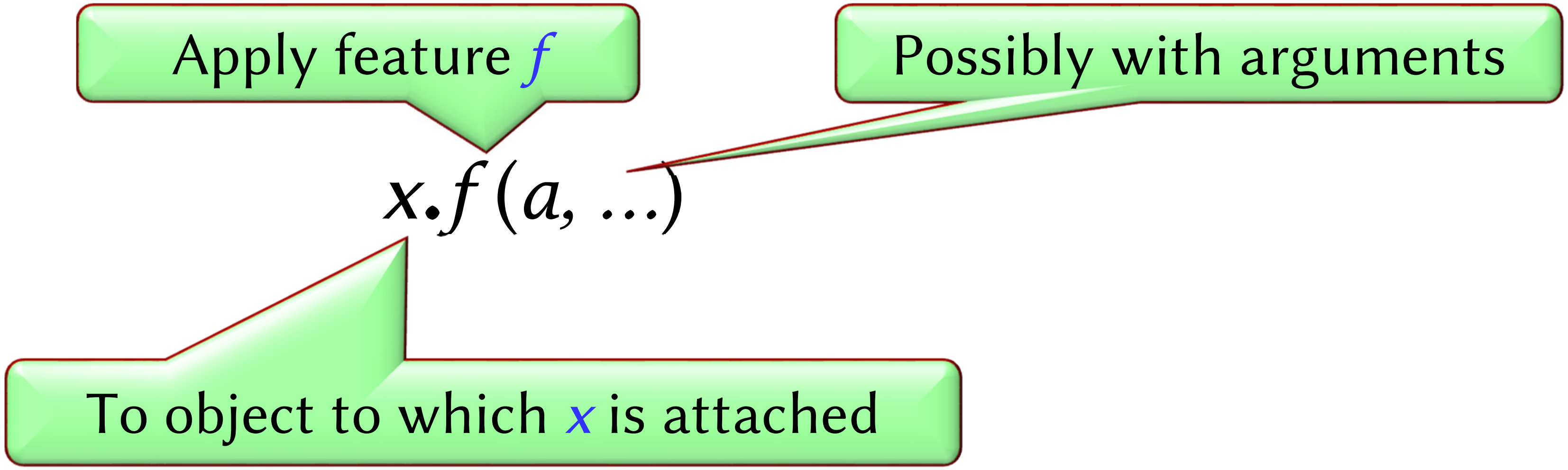
By default

Initially, *buddy* is not attached to any object: its value is a **void** reference



The trouble with void references

The basic mechanism of computation is feature call



Apply feature *f*

Possibly with arguments

$x.f(a, \ldots)$

To object to which *x* is attached

Since references may be void, then *x* might be attached to no object

The call is erroneous in such cases

Example: call on void target

```
class ACROBAT

feature
        clap (n: INTEGER)
                        -- Clap `n' times and forward to copycat.
            do
                        -- "Clap n times"

                        buddy.clap(n)
            end

        … … … … … … …

        buddy : COPYCAT
                        -- the copycat of this acrobat
end
```

If *buddy* is **Void** this is a Void reference

Exceptions

They are abnormal events during execution. For example:

- "Void call": *buddy.clap*
  where *buddy* is void
- Attempt to compute *a / b* where *b* has value 0

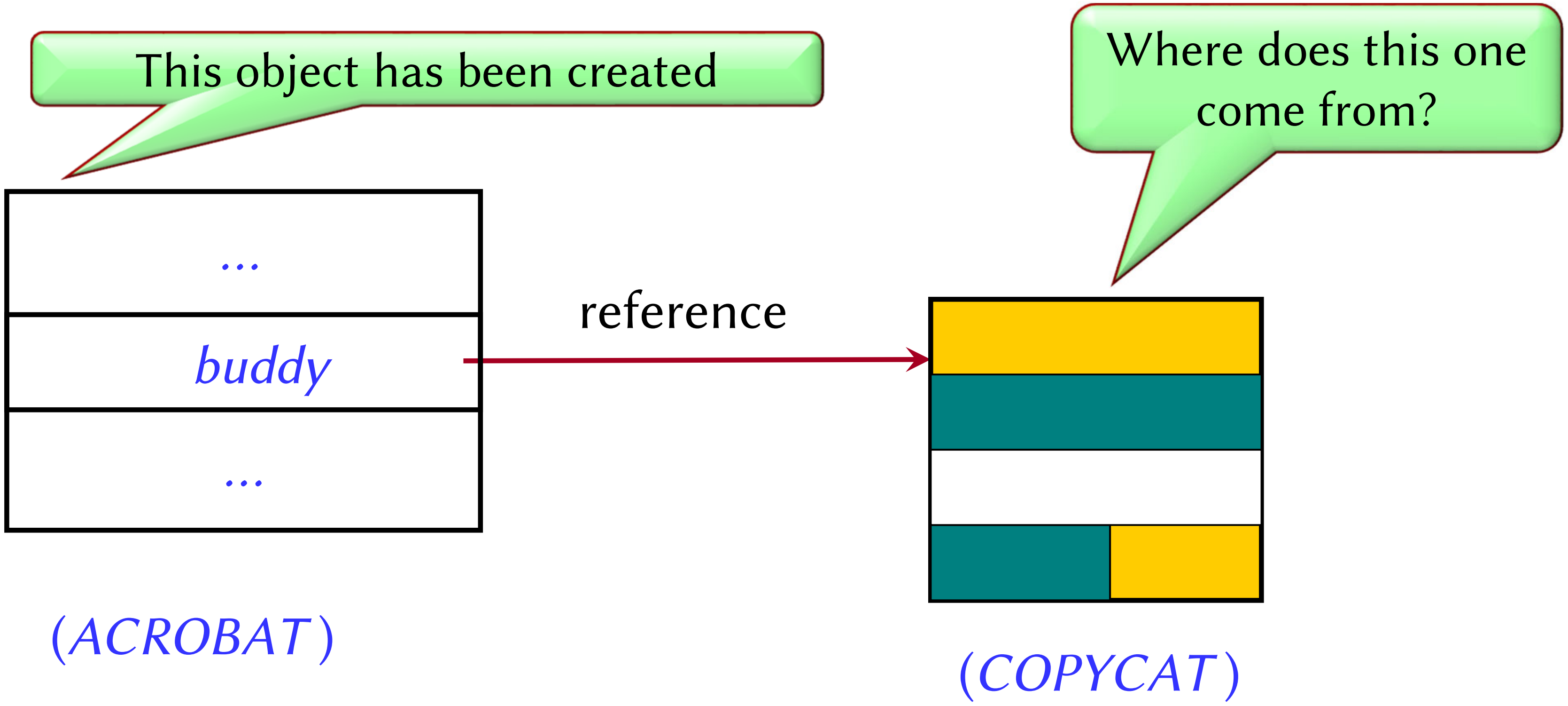
A failure will happen unless the program has code to recover from the exception ("**rescue**" clause in Eiffel, "**catch**" in Java)

Every exception has a **type**, appearing in EiffelStudio run-time error messages, e.g.

- Feature call on void reference (i.e. void call)
- Arithmetic underflow

Initial state of a reference

In an instance of *ACROBAT*, may we assume that *buddy* is attached to an instance of *COPYCAT*?



Why do we need to create objects?

Couldn't we assume that a declaration

buddy: COPYCAT

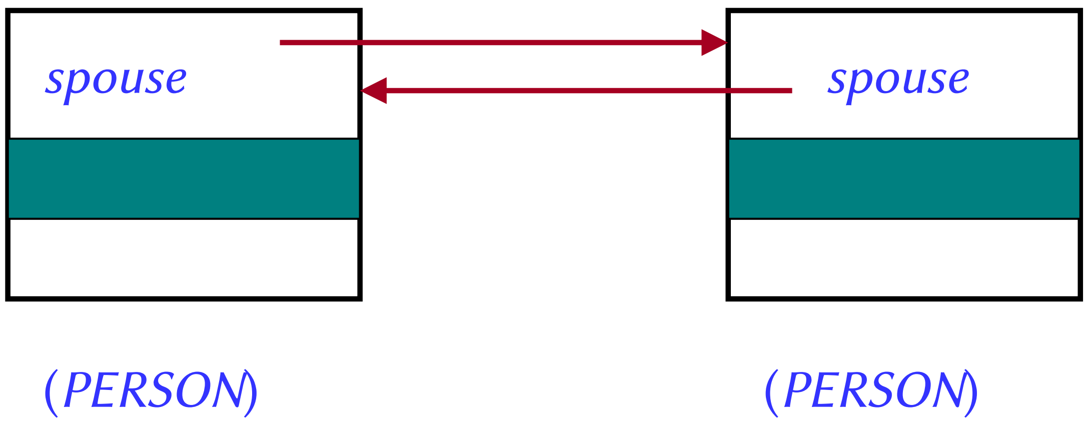
creates an instance of *COPYCAT* and attaches it to *buddy*?

(Answer in a little while...)

Is Void necessary?

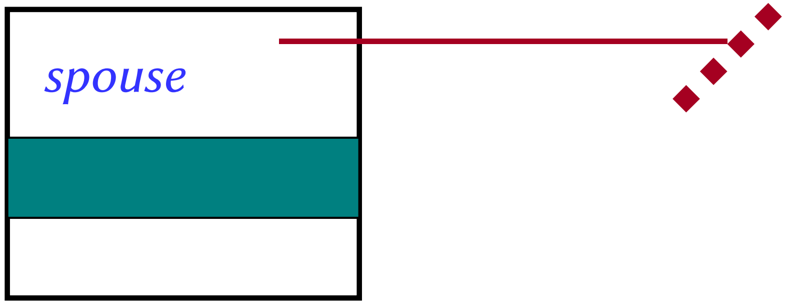
Void references are useful

Consider a representation for married persons:



Void references are useful

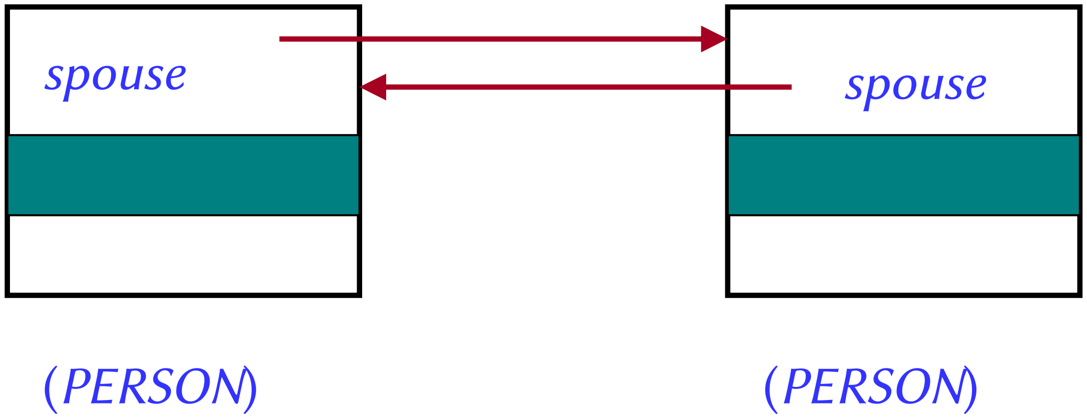
We need a **Void** reference to represent an unmarried person:



(PERSON)

Void references are useful

Even when representing only married persons…

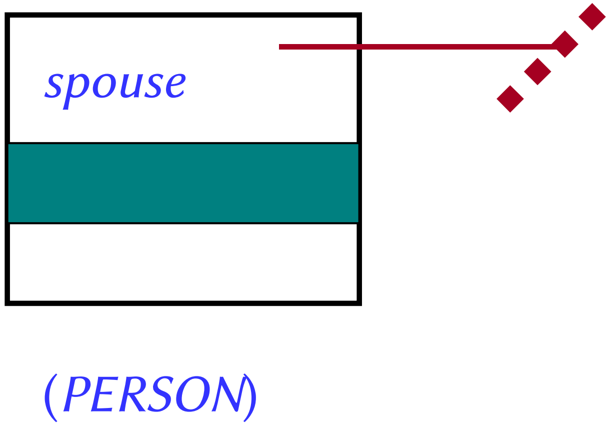


… we shouldn’t create an object for *spouse* every time we create an instance of *PERSON*

(why?)

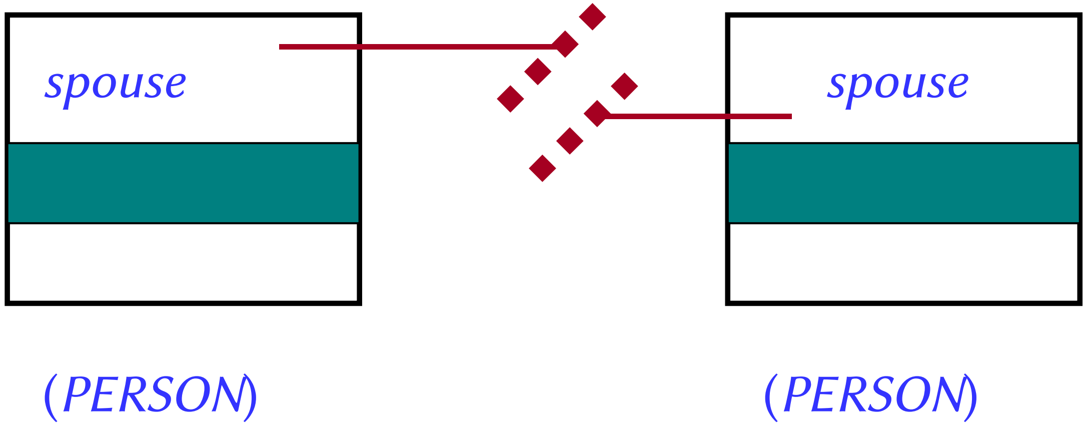
Using void references

Create every *PERSON* object with a void *spouse*



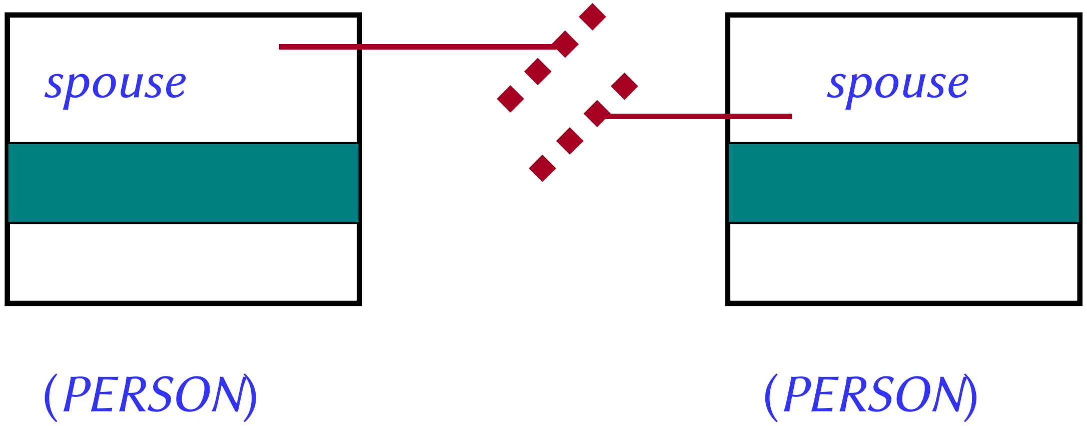
Using void references

Create every *PERSON* object with a void *spouse*



Using void references

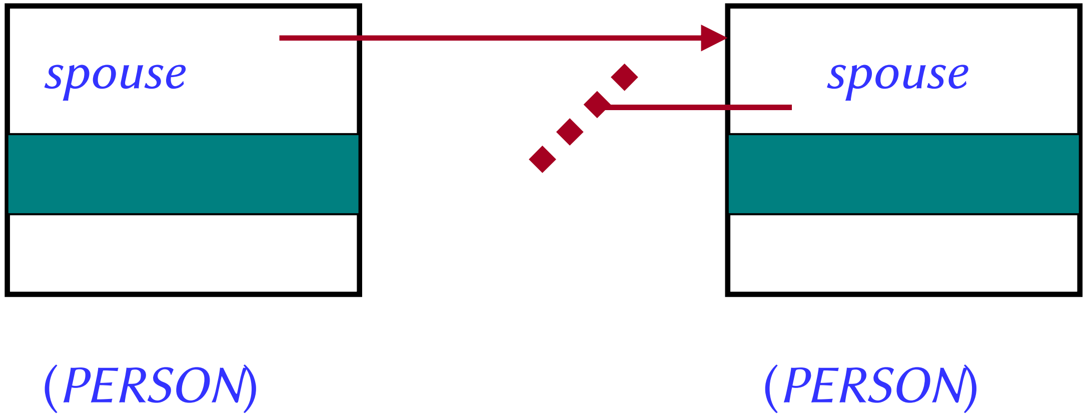
Create every *PERSON* object with a void *spouse*



... then attach the *spouse* references as desired, through appropriate instructions

Using void references

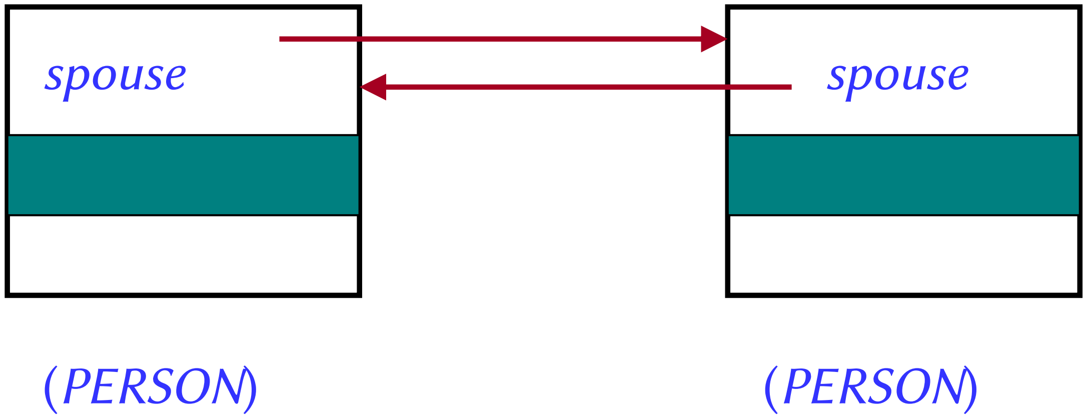
Create every *PERSON* object with a void *spouse* ...



... then attach the *spouse* references as desired, through appropriate instructions

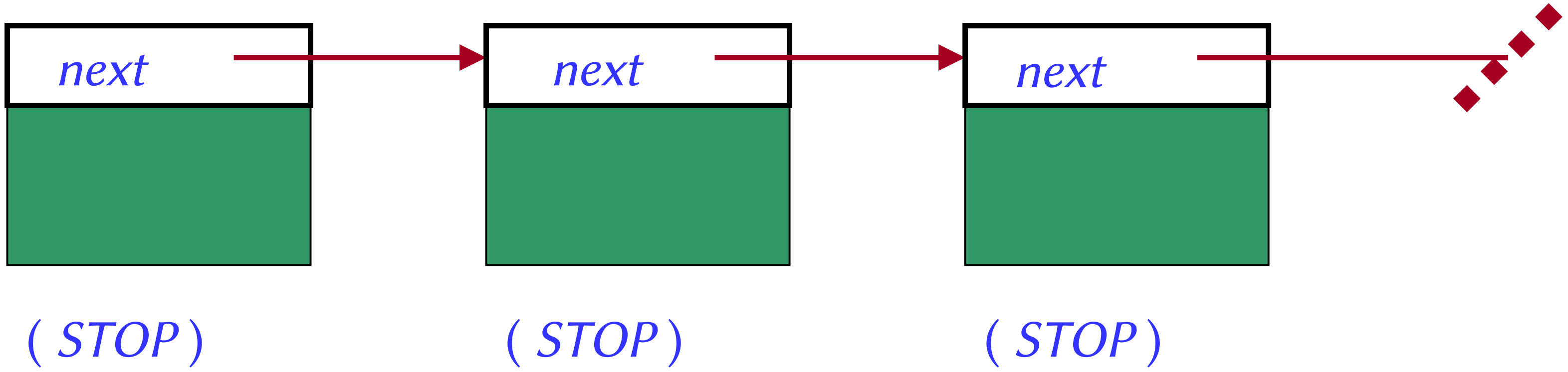
Using void references

Create every *PERSON* object with a void *spouse* ...



... then attach the *spouse* references as desired, through appropriate instructions

References to linked structures



To terminate the list, last *next* reference is void

Object creation

Every entity is declared with a certain type:

mariuccio: *COPYCAT*

A creation instruction

create *mariuccio*

produces, at run time, an object of that type.

To avoid exception

Create and assign referenced object as soon as a referencing object is created:

```
create mario
create mariuccio
mario.pair(mariuccio)
```

Creating referencing object

Creating and assigning referenced object

To be helped not to forget assignment one might also thought to add an invariant to the class:

```
invariant
        buddy_exists: buddy /= Void
```

Try it: what happens?

A better approach: creation procedures

Declare *pair* as a **creation procedure** and merge initialization with creation:

```
create mario.pair (mariuccio)
        -- Same effect as previous two last instructions
```

- Convenience: initialize upon creation
- Correctness: ensure invariant right from the start

Creation procedures are also called **constructors**

Creation principle

If a class has a non-trivial invariant, it **must** list one or more creation procedures, whose purpose is to ensure that every instance, upon execution of a creation instruction, will satisfy the invariant

This allows the author of the class to force proper initialization of all instances that clients will create.

Creation procedures

Even in the absence of a strong invariant, in creation procedures it is useful to combine creation with initialization:

```
class POINT create
        default_create, make_cartesian, make_polar
feature
        ...
end
```

Inherited by all classes, by default does nothing

Valid creation instructions:

```
    create your_point.default_create
    create your_point
    create your_point.make_cartesian (x, y)
    create your_point.make_polar (r, t)
```

Object creation: summary

To create an object:

- If class has no **create** clause, use basic form:

  create *x*

- If the class has a **create** clause listing one or more procedures, you must use

  create *x.make* (...)

  where *make* is one of the creation procedures, and (...) stands for arguments if any.

- A creation procedure is just a regular feature whose name is listed in the **create** clause

- To be able to use also the basic form, the **create** clause must list also *default_create*

- A creation procedure is used to ensure values of just created object's attributes are properly initialized

Correctness of an instruction

For every instruction we must know precisely, in line with the principles of Design by Contract:

- How to use the instruction correctly: its precondition.
- What we are getting in return: the postcondition.

Together, these properties (plus the invariant) define the **correctness** of a language mechanism.

What is the correctness rule for a creation instruction?

Correctness of a creation instruction

Creation Instruction Correctness Rule

Before creation instruction:

1. Precondition of its creation procedure, if any, must hold

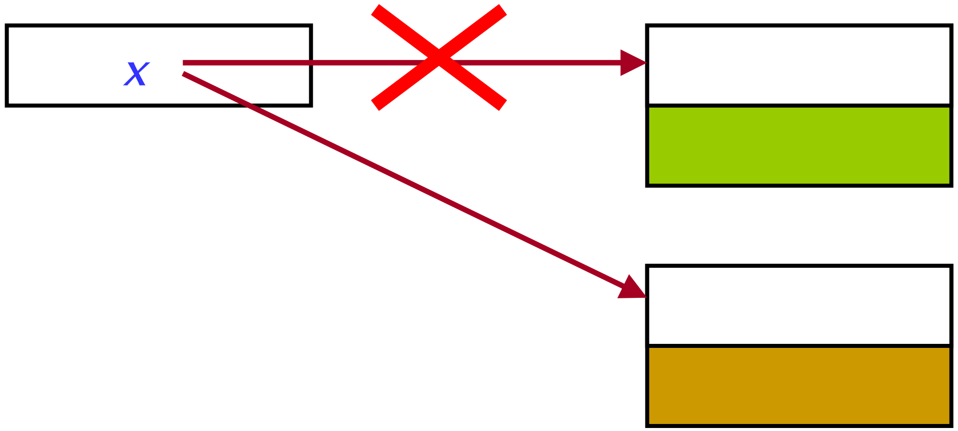
After creation instruction with target *x* of type *C* :

2. *x* /= **Void** holds
3. Postcondition of creation procedure holds
4. Object attached to *x* satisfies invariant of *C*

Successive creation instructions

The correctness condition does not *require* *x* to be void before creation:

```
        -- Here x needs not to be void
create x
        -- Here x is certainly not void
create x
        -- Here the object previously attached to x is lost
```



object created by the first **create**

object created by the second **create**

Effect of creation instruction

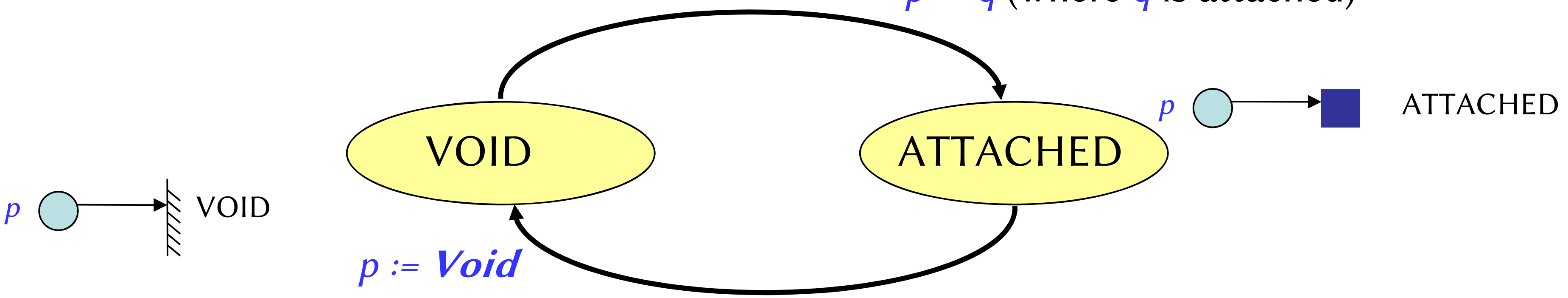
- *x* won't be void after creation instruction (whether or not it was void before)

- If there is a creation procedure, its postcondition will hold for newly created object

- The object will satisfy the class invariant

Objects and references

States of a reference:



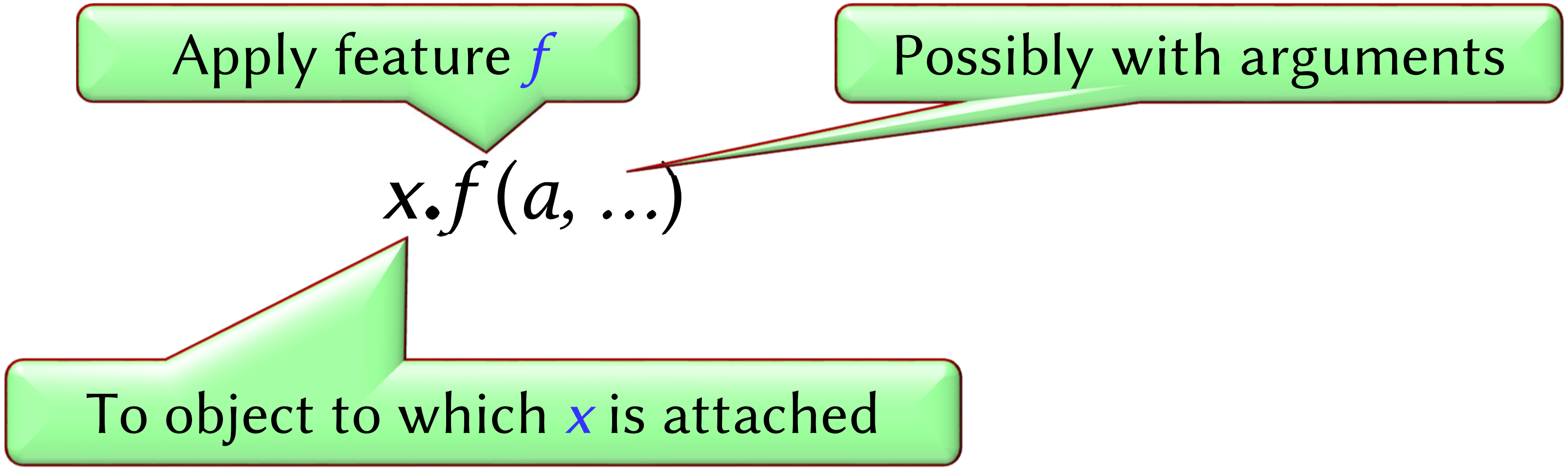
create *p*
OR
p := *q* (where *q* is attached)

p := ***Void***
OR
p := *q* (where *q* is void)

N.B.: No need to **create** *p* to assign *q* to *p* !

The trouble with void references (once again)

The basic mechanism of computation is feature call



Apply feature *f*

Possibly with arguments

$x.f\,(a, \ldots)$

To object to which *x* is attached

Since references may be void, then *x* might be attached to no object

The call is erroneous in such cases

The inventor of null references

I call it my billion-dollar mistake. It was the invention of the null reference in 1965.

At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W).

My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler.

But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement.

This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

By Tony Hoare, 2009

Problems of void-calls

Entities are either

- **Attached**: referencing a valid object
- **Detached**: Void (or null)

Calls on detached entities cause a runtime error

Runtime errors are bad...

How can we prevent this problem?

Solution to void-calls

Statically attached: property (referencing a valid object) that can be determined at compile-time

Dynamically attached: property (referencing a valid object) that can be determined at run-time

If we ensure consistency, that is if we ensure that:

If *f* is statically attached, its possible runtime values are dynamically attached.

then the solution to void calls is:

A call *f.x (...)* is only allowed, if *f* is statically attached.

Void calls: the full story (1)

- In ISO Eiffel, void calls do not happen any more thanks to the notion of **attached** type.
- A type declared in the normal way, say *CITY*, is called an **attached** type and guaranteed to prevent void references.
  - Types representing objects from the application domain usually should be attached and hence exclude void: there is no such thing as a void city.
- A type only allows void references if it is declared with the **detachable** keyword, as in

  s: **detachable** *STOP*

  - Types representing linked data structures generally must support void values.

Void calls: the full story (2)

- Guaranteeing the absence of void calls relies on two complementary techniques:
  - If an entity *x* is of an **attached** type, it must have an associated initialization mechanism (not **Void**) so that before its first use in a call *x.f* (...) it will have been attached to an object.
  - If *x* is of a **detachable** type, any call *x.f* (...) must occur in a context where *x* is guaranteed to be non-void, for example **if** *x* /= **Void then** *x.f* (...) **end**
- The compiler rejects any *x.f* call where *x* could be void in some execution
- In the course we sometime use the old rules
- The compiler will in many cases accept old code
  - When it does reject code, this generally reflects a *real* problem

Attachment for types (1)

Can declare type of entities as attached or detachable

- `att: attached STRING`
- `det: detachable STRING`

Attached types

- Can call features without control: `att.to_upper`
- Can be assign to detachable: `det := att`
- Cannot be set to void: ~~`att := Void`~~

Detachable types

- No feature calls without control: ~~`det.to_upper`~~
- Cannot be assign to attached: ~~`att := det`~~
- Can be set to void: `det := Void`

Attachment for types (2)

Default initial value

- Detachable: `Void`
- Attached: explicit assignment

Initialization rules for attached types

- Attributes: at end of each creation routine
- Locals: before first use
- Compiler uses control-flow analysis

Types without attachment clause

- Default interpretation can be set in project settings
- Default for void-safe projects is **`attached`**

Safe use of detachable types

Certified attachment patterns (CAP)

- For local entities (formal arguments and local entities)
- Code pattern where attachment is guaranteed
- `if x /= Void then x.f end`

  (where x is a local)

Object Test

- Assign result of arbitrary expression to a local
- Boolean value indicating if result is attached
- `if attached x as l then l.f end`

We shall look at them in more detail…

What is a CAP?

The Eiffel standard (2nd edition, June 2006) defines a CAP as follows:

A Certified Attachment Pattern (or CAP) for an expression exp whose type is detachable is an occurrence of exp in one of the following contexts:

1. exp is an Object-Test Local and the occurrence is in its scope.

2. exp is a read-only entity and the occurrence is in the scope of a void test involving exp.

Certified attachment pattern (CAP)

Code patterns where attachment is guaranteed

Basic CAPs for locals and arguments

- Setting value on creation
- Void check with conditional or semi-strict operator

```
capitalize (a_string: detachable STRING)
    do
        if a_string /= Void then
            a_string.to_upper
        end
    end
```

Testing in preconditions, code, postconditions

Does testing in pre-conditions provide a CAP?

```
capitalize (a_string: detachable STRING)
    require
        a_string /= Void
    do
         . . .
        if a_string /= Void then
            a_string.to_upper
        end
    ensure
         attached a_string as s implies s.is_upper
    end
```

contract checking can be disabled at run-time

Static analysis can guarantee this test is executed

Object test (1)

Checking attachment of an expression (and its type)

Assignment to a **read-only** local variable, not declared and only available in one branch

Object test **must** be used for attributes, see why…

```
name: detachable STRING


capitalize_name
    do
        if attached name as n then
            . . . . .
            n.to_upper
        end
    end
```

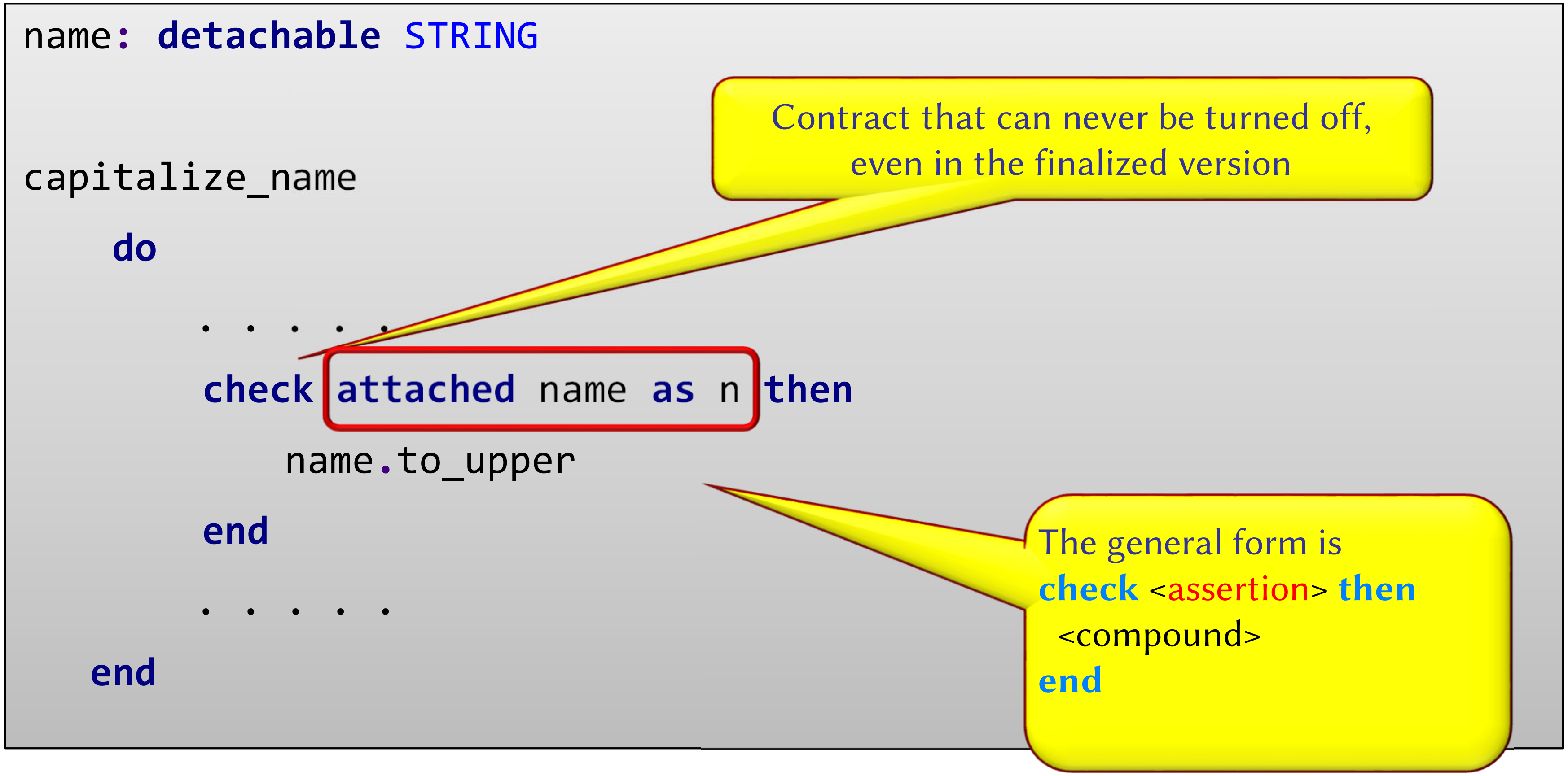
n cannot be reassigned

Object test (2)

What to do if Object Test fails? Take appropriate actions in the **else** branch of **if** (if empty nothing is done and the program continues)

A variant of **check** instruction will raise an exception (there is no **else** branch and if the Object Test fails the program stops)

It's **not yet** part of the Eiffel Standard definition



```
name: detachable STRING


capitalize_name

    do

        . . . . .

        check attached name as n then

            name.to_upper

        end

        . . . . .

    end
```

Contract that can never be turned off, even in the finalized version

The general form is
check <assertion> **then**
  <compound>
end

Object test (3)

Must be used also:

in assertions

in class invariants

(in these cases **if** instruction cannot be used)

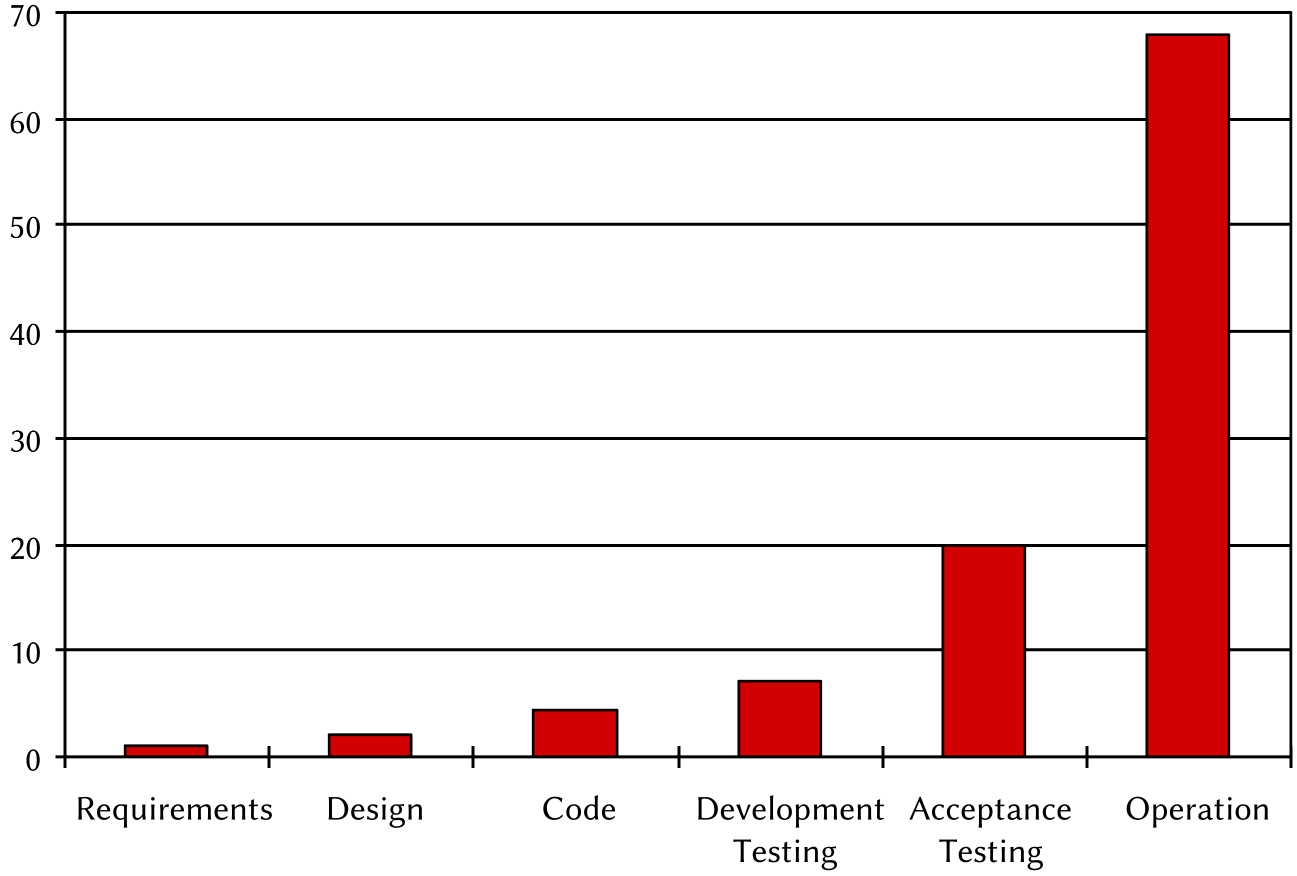
```
name: detachable STRING

capitalize_name
    do

        ...

    ensure
       attached name as n implies n.is_upper
    end
```

Can NOT be used
name /= **Void implies** *name.is_upper*
since it does not bind *name* to a read-only identifier
(the compiler will reject it as non void-safe)

It is worthwhile to discover static errors

Source: Boehm 81

Relative cost to fix a bug



References

Eiffel documentation on void-safety

- http://docs.eiffel.com/book/method/void-safe-programming-eiffel

Avoid a Void: The eradication of null dereferencing

- http://s.eiffel.com/void_safety_paper

Side note on object tests

Object test can also be used to make a type cast

The test is **True**, if object conforms to specified type

Local variable will have specified type

```
name: detachable ANY


capitalize_name
    do
        if attached {STRING} name as l_name then
            l_name.to_upper
        end
    ensure
        attached {STRING} name as n implies n.is_upper
    end
```

Stable attributes

Detachable attributes which are never set to void

They are initially void, but once attached will stay so

```
name: detachable STRING
    note
        option: stable
    attribute
    end

capitalize_name
    do
        if name /= Void then
            name.to_upper
        end
    end
```

Arrays

Arrays can have more storage space then elements

Empty storage space filled with *default* values

What is the default for attached types?

- `a: attached ARRAY [attached STRING]`

```
See Array demo
```