Concurrent programming is easy!

An overview of today’s SCOOP

Bertrand Meyer

Microsoft Russian summer school
Saint Petersburg, 23 August 2012

Version 2 of slides

What this is about

SCOOP: a model for bringing concurrent programming under control

Not new (goes back to 90s)

- But in recent years has been patiently refined (PhD theses) and implemented (now part of standard EiffelStudio)

- And is receiving a new impetus
This is the work of many people

Eiffel Software: Emmanuel Stapf, Alexander Kogtenkov, Ian King

ETH: Piotr Nienaltowski, Benjamin Morandi, Sebastian Nanz, Scott West

ITMO: Anton Akhi, Alexander Kogtenkov

SEL at ITMO

Software Engineering Laboratory
Лаборатория Программной Инженерии

Создано в июне 2011

Присоединяйтесь к нам!

- Аспиранты и Кандидаты (на полной ставке)
- Временные гранты ("sabbaticals") для исследователей, 2 до 6 месяцев
What this is about

SCOOP: a model for bringing concurrent programming under control

Plan:
1. The trailer
2. A bit of context
3. The model: successive restrictions
4. The SCOOP type system
5. Open problems and current work
6. Some other stuff! (Teaching, verification...)
SCOOP background

Simple Concurrent Object-Oriented Programming

First version described in CACM article (1993) and chapter 32 of Object-Oriented Software Construction, 2nd edition, 1997

Prototype implementation at ETH (2005-2010)
Recent production implementation at Eiffel Software, part of EiffelStudio; watch for forthcoming 7.1

Recent descriptions: Piotr Nienaltowski’s 2007 ETH PhD; Morandi, Nanz, Meyer (2011)

The issue that SCOOP tackles

Can we bring concurrent programming to the same level of abstraction and convenience as sequential programming?
Dijkstra 68

... [O]ur intellectual powers are rather geared to master static relations and our powers to visualize processes evolving in time are relatively poorly developed.

For that reason we should do ... our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Concurrent programming today

Listing 4.34: Tanenbaum's solution

```python
1  def get_fork(i):
2      mutex.wait()
3      state[i] = 'hungry'
4      test(i)
5      mutex.signal()
6      sem[i].wait()
7
8  def put_fork(i):
9      mutex.wait()
10     state[i] = 'thinking'
11     test(right(i))
12     test(left(i))
13     mutex.signal()
14
15  def test(i):
16      if state[i] == 'hungry' and
17      state(left(i)) != 'eating' and
18      state(right(i)) != 'eating':
19        state[i] = 'eating'
20      sem[i].signal()
```

Allen Downey: The Little Green Book of Semaphores, greenteapress.com/semaphores/
Then and now

**Sequential programming:**
- Used to be messy
- Still hard but key improvements:
  - Structured programming
  - Data abstraction & object technology
  - Design by Contract
  - Genericity, multiple inheritance
  - Architectural techniques

**Concurrent programming:**
- Used to be messy
- **Still messy**
- Example: threading models in most popular approaches
- Development level: sixties/seventies
- Only understandable through operational reasoning

Previous advances in programming

<table>
<thead>
<tr>
<th>Feature</th>
<th>“Structured programming”</th>
<th>“Object technology”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use higher-level abstractions</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Helps avoid bugs</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Transfers tasks to implementation</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Lets you do stuff you couldn’t before</td>
<td>NO</td>
<td>✓</td>
</tr>
<tr>
<td>Removes restrictions</td>
<td>NO</td>
<td>✓</td>
</tr>
<tr>
<td>Adds restrictions</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Has well-understood math basis</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Doesn’t require understanding that basis</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Permits less operational reasoning</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Concurrent programming today

Listing 4.34: Tanenbaum’s solution

```python
1 def get_fork(i):
2     mutex.wait()
3     state[i] = 'hungry'
4     test(i)
5     mutex.signal()
6     sem[i].wait()
7
8 def put_fork(i):
9     mutex.wait()
10    state[i] = 'thinking'
11    test(right(i))
12    test(left(i))
13    mutex.signal()
14
15 def test(i):
16    if state[i] == 'hungry' and
17        state[left(i)] == 'eating' and
18        state[right(i)] == 'eating':
19        state[i] = 'eating'
20        sem[i].signal()
```

Exercise 1 (from Tanenbaum via Downey)

There is a deep canyon somewhere in Kruger National Park and a single rope that spans the canyon. Baboons can cross the canyon by swinging on the rope, but if two baboons going in opposite directions meet in the middle, they will fight and drop to their deaths.

The rope is only strong enough to hold 5 baboons. If there are more, it will break.

Devise a synchronization scheme such that:

- Once a baboon has begun to cross, it is guaranteed to get to the other side without running into a baboon going the other way.
- There are never more than 5 baboons on the rope.
Exercise 2 (from Downey)

Near Redmond, WA, there is a rowboat used by both Linux hackers and Microsoft employees to cross a river. It holds exactly four people; it won't leave with more or fewer. For safety, it is not permissible to put one hacker in the boat with three employees, or one employee with three hackers. Other combinations are safe.

As each thread boards the boat it should call a function `board`. All four threads from each boatload must call `board` before any of the threads from the next boatload do.

After all four threads have call `board`, exactly one of them should call a function `row_boat`, indicating that it takes the oars. It doesn't matter which thread calls it. Don't worry about the direction of travel.

---

Reasoning from APIs

```
transfer (source, target: ACCOUNT; amount: INTEGER)  
    -- Transfer amount from source to target.  
    require  
        source.balance >= amount  
    do  
        source.withdraw (amount)  
        target.deposit     (amount)  
    ensure  
        source.balance = old source.balance - amount  
        target.balance = old target.balance + amount  
    end

invariant
    balance >= 0
```
Reasoning from APIs

\[
\text{transfer (source, target: \hspace{0.5cm} \text{ACCOUNT};} \\
\hspace{0.5cm} \text{amount: INTEGER)} \\
\hspace{1cm} \text{-- Transfer amount from source to target.} \\
\hspace{2cm} \text{require} \\
\hspace{3cm} \text{source.balance} \geq \text{amount} \\
\hspace{2cm} \text{do} \\
\hspace{3cm} \text{source.withdraw (amount)} \\
\hspace{3cm} \text{target.deposit (amount)} \\
\hspace{2cm} \text{ensure} \\
\hspace{3cm} \text{source.balance} = \text{old source.balance} - \text{amount} \\
\hspace{3cm} \text{target.balance} = \text{old target.balance} + \text{amount} \\
\hspace{1cm} \text{end}
\]

Example 1: bank accounts

\[
\text{if acc1.balance} \geq 100 \quad \text{then transfer (acc1, acc2, 100) end}
\]

\[
\text{if acc1.balance} \geq 100 \quad \text{then transfer (acc1, acc3, 100) end}
\]

\[
\text{invariant} \\
\text{balance} \geq 0
\]
The SCOOP solution

```coq
transfer (source, target: ACCOUNT; amount: INTEGER)
    -- Transfer amount from source to target.
    require
        source.balance >= amount
    do
        source.withdraw (amount)
        target.deposit (amount)
    ensure
        source.balance = old source.balance - amount
        target.balance = old target.balance + amount
end
```

Reasoning from API: a queue

```coq
put (buf: QUEUE; v: G)
    -- Add v to buf.
    require
        not buf.is_full
    do
        ...
    ensure
        ...
end
```
Reasoning from APIs

```plaintext
put (buf: QUEUE; v: G)
    -- Insert v into buf.
    require
        not buf.is_full
    do
        ...
    ensure
        ...
end
```

The SCOOP solution

```plaintext
put (buf: QUEUE; v: G)
    -- Add v to buf.
    require
        not buf.is_full
    do
        ...
    ensure
        ...
end
```
Example 2: hexapod robot

Hind legs have force sensors on feet and retraction limit switches

Hexapod locomotion

Alternating protraction and retraction of tripod pairs

- Begin protraction only if partner legs are down
- Depress legs only if partner legs have retracted
- Begin retraction when partner legs are up
Hexapod coordination rules

**R1**: Protraction can start only if partner group on ground
- **R2.1**: Protraction starts on completion of retraction
- **R2.2**: Retraction starts on completion of protraction
**R3**: Retraction can start only when partner group raised
**R4**: Protraction can end only when partner group retracted


Sequential implementation

```java
TripodLeg lead = tripodA;
TripodLeg lag = tripodB;

while (true) {
    lead.Raise();
    lag.Retract();
    lead.Swing();
    lead.Drop();

    TripodLeg temp = lead;
    lead = lag;
    lag = temp;
}
```
Multi-threaded implementation

```java
private object m_protractionLock = new object();

private void ThreadProcWait(object obj)
{
    TripodLeg leg = obj as TripodLeg;
    while (Thread.CurrentThread.ThreadState != ThreadState.Abandoned)
    {
        // Waiting for protraction lock
        lock (m_protractionLock)
        {
            // Waiting for partner leg drop
            leg.Partner.DroppedEvent.WaitOne();
            leg.Raise();
        }
        leg.Swing();

        // Waiting for partner retraction
        leg.Partner.RetractedEvent.WaitOne();
        leg.Drop();

        // Waiting for partner raise
        leg.Partner.RaisedEvent.WaitOne();
        leg.Retraction();
    }
}
```

SCOOP version

```SCOOP
begin_protraction (partner, me: separate LEG_GROUP)

require
    me.legs_retracted
    partner.legs_down
    not partner.protraction_pending

do
    tripod.lift
    me.set_protraction_pending

end```

Hexapod coordination rules

**R1:** Protraction can start only if partner group on ground

- **R2.1:** Protraction starts on completion of retraction
- **R2.2:** Retraction starts on completion of protraction

**R3:** Retraction can start only when partner group raised

**R4:** Protraction can end only when partner group retracted


---

Example 3: dining philosophers in SCOOP (1)

class PHILOSOPHER feature
live
do

from getup until over loop
think: eat(left, right)
end

end

eat(l, r: separate FORK)
-- Eat, having grabbed l and r.
do ... end

getup do ... end
over: BOOLEAN
end
The semaphore solution

Example 4: Elevator system

For maximal concurrency, all objects are separate
SCOOP: the context

The end of Moore’s Law as we knew it

Source: Intel
The issue that SCOOP addresses

Can we bring concurrent programming to the same level of abstraction and convenience as sequential programming?

Wrong (in my opinion) assumptions

“Objects are naturally concurrent” (Milner)

- Many attempts, often based on “Active objects” (a self-contradictory notion)
- Lead to artificial issue of “Inheritance anomaly”

“Concurrency is the basic scheme, sequential programming a special case” (many)

- Correct in principle, but in practice we understand sequential best
Four barriers to concurrent computation

- Data races
- Deadlock
- Starvation
- Priority inversion

Data race

<table>
<thead>
<tr>
<th>Time</th>
<th>Svetlana</th>
<th>Agent S</th>
<th>Agent T</th>
<th>Tatiana</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 seat</td>
</tr>
<tr>
<td>2</td>
<td>Seats left?</td>
<td></td>
<td></td>
<td></td>
<td>Yes, 1</td>
</tr>
<tr>
<td>3</td>
<td>Seating left?</td>
<td></td>
<td></td>
<td></td>
<td>Yes, 1</td>
</tr>
<tr>
<td>4</td>
<td>Book it!</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>Book it!</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>Try to book</td>
<td></td>
<td></td>
<td></td>
<td>0 Success!</td>
</tr>
<tr>
<td></td>
<td>Try to book</td>
<td></td>
<td></td>
<td></td>
<td>Failure!</td>
</tr>
</tbody>
</table>
Four barriers to concurrent computation

- Data races
- Deadlock
- Starvation
- Priority inversion

Reminder: the plan

Plan:

1. The trailer
2. A bit of context
3. The model: successive restrictions
4. The SCOOP type system
5. Open problems and current work
6. Some other stuff! (Teaching, verification...)
- 3 -

SCOOP: the design

Where to find my slides


Wait for revised version later today!
What this is about

SCOOP: a model for bringing concurrent programming under control

Plan:

- 1. The trailer
- 2. A bit of context
- 3. The model: successive restrictions
- 4. The SCOOP type system
- 5. Open problems and current work
- 6. Some other stuff! (Teaching, verification...)

The design of SCOOP (and this presentation)

To achieve the preceding goals, SCOOP makes a number of restrictions on the concurrent programming model

This presentation explains and justifies these restrictions one after the other

The goal is not to limit programmers but to enable them to reason about the programs
The design of SCOOP

SCOOP intends to make concurrent programming as predictable as sequential programming

A key criterion is "reasonability" (not a real word!): the programmer’s ability to reason about the execution of programs based only on their text

- As in sequential O-O programming, with contracts etc.

SCOOP is not a complete rework of basic programming schemes, but an incremental addition to the basic O-O scheme: one new keyword

- "Concurrency Made Easy"

Handling concurrency simply

SCOOP narrows down the distinction between sequential & concurrent programming to six properties, studied next:

- (A) Single vs multiple “processors”
- (B) Regions
- (C) Synchronous vs asynchronous calls
- (D) Semantics of argument passing
- (E) Semantics of resynchronization (lazy wait)
- (F) Semantics of preconditions
The starting point (A): processors

To perform a computation is
- To apply certain actions
- To certain objects
- Using certain processors

Sequential: one processor
Concurrent: any number of processors

What makes an application concurrent?

**Processor:**
Thread of control supporting sequential execution of instructions on one or more objects

Can be implemented as:
- Computer CPU
- Process
- Thread
- AppDomain (.NET) ...

The SCOOP model is abstract and does not specify the mapping to such actual computational resources
Object-oriented programming

The key operation is “feature call”

\[ x.f \text{(args)} \]

where \( x \), the \textbf{target} of the call, denotes an object to which the call will apply the feature \( f \)

Which processor is in charge of executing such a call?

\[ \text{(B): Regions} \]

All calls targeting a given object will be executed by a single processor

- The set of objects handled by a given processor is called a \textit{region}
- The processor in charge of an object is its \textit{handler}
Reasoning about objects: sequential

Only \( n \) proofs if \( n \) exported routines!

\[
\begin{align*}
\{ \text{INV and Pre}_r \} & \quad \text{body}_r \quad \{ \text{INV and Post}_r \} \\
\hline
\{ \text{Pre}'_r \} & \ x \cdot r (a) \quad \{ \text{Post}'_r \}
\end{align*}
\]

Priming represents actual-formal argument substitution

The concurrent version of this rule will come later!

In a concurrent context

Only \( n \) proofs if \( n \) exported routines?

No overlapping!

\[
\begin{align*}
\{ \text{INV and Pre}_r \} & \quad \text{body}_r \quad \{ \text{INV and Post}_r \} \\
\hline
\{ \text{Pre}'_r \} & \ x \cdot r (a) \quad \{ \text{Post}'_r \}
\end{align*}
\]

Reasonability
SCOOP restriction: one handler per object

- One processor per object: "handler"
- At most one feature (operation) active on an object at any time

Reasonability

Regions

The notion of handler implies a partitioning of the set of objects:
- The set of objects handled by a given processor is called a region
- Handler rule implies one-to-one correspondence between processors and regions

Diagram:
- Objects
- Region
- Region boundary
- Client
(C) The sequential view: O-O feature calls

\[ x.r(a) \]

\[ \text{Client} \]
\[ \text{previous} \]
\[ x.r(a) \]
\[ \text{next} \]
\[ \text{Processor} \]
\[ \text{Supplier} \]
\[ r(x: A) \]
\[ \text{do} \]
\[ \text{...} \]
\[ \text{end} \]

(C) The concurrent form of call: asynchronous

\[ x.r(a) \]

\[ \text{Client} \]
\[ \text{previous} \]
\[ x.r(a) \]
\[ \text{next} \]
\[ \text{Client's handler} \]
\[ \text{Supplier} \]
\[ r(x: A) \]
\[ \text{do} \]
\[ \text{...} \]
\[ \text{end} \]
\[ \text{Supplier's handler} \]
**The two forms of O-O call**

To wait or not to wait:
- If same processor, synchronous
- If different processor, asynchronous

Difference must be captured by syntax:
- `x: T`
- `x: separate T -- Potentially different processor`

Fundamental semantic rule: a call `x.r(a)`
- Waits (i.e. is synchronous) for non-separate `x`
- Does not wait (is asynchronous) for separate `x`

---

**Why potentially separate?**

`separate` declaration only states that the object *might* be handled by a different processor

- In class `A`: `x: separate B`
- In class `B`: `y: separate A`
- In `A`, what is the type of `x.y`?

In some execution, the value might be a reference to an object in the current region
Call vs application

With asynchrony we must distinguish between feature call and feature application.

The execution

\[ x \cdot r (...) \]

is the call, and (with \( x \) separate) will not wait (the client just logs the call).

The execution of \( r \) happens later and is called the feature application.

Consistency rules: avoiding traitors

\[ \text{nonsep} : T \]
\[ \text{sep} : \text{separate} T \]
\[ \text{nonsep} := \text{sep} \]
\[ \text{nonsep}, p(a) \]

Traitor!

More traitor protection through the type system!

Reasonability
An aside: processor collection

Processors, like objects, can become unreachable!
Object and processor collection is intertwined

\[\text{LO} = s^* (\text{BR} \cup r (\text{LP}))\]

\[\text{LP} = \text{BP} \cup h (\text{LO})\]

Access control policy

Since separate calls are asynchronous there is a real danger of confusion

Consider for example

```java
remote_stack: separate STACK[T]
...
remote_stack.put(a)
... Instructions not affecting the stack...
y := my_stack.item
```

Reasonability
(D) Access control policy

SCOOP requires the target of a separate call to be a **formal argument** of enclosing routine:

```plaintext
put (s: separate STACK[T]; value: T)
   -- Store value into s.
   do
      s.put (value)
   end
```

To use separate object:

```plaintext
my_stack: separate STACK[INTEGER]
create my_stack
put (my_stack, 10)
```

(D) Separate argument rule

The target of a separate call must be an argument of the enclosing routine.

Separate call: `x.f(...) where x is separate`
(D) Holding rule

**A routine call guarantees exclusive access to the handlers (the processors) of all separate arguments**

\[ a\_routine(\text{nonsep}_a, \text{nonsep}_b, \textit{sep}_c, \textit{sep}_d, \textit{sep}_e) \]

Exclusive access to \( \textit{sep}_c, \textit{sep}_d, \textit{sep}_e \) within \( a\_routine \)

An example: from sequential to concurrent

\[
\text{transfer} (\text{source, target: separate ACCOUNT; amount: INTEGER})
\]

-- Transfer amount, if available, from source to target.
\[
do
\]
\[
\text{if } \text{source.balance} \geq \text{amount} \text{ then}
\]
\[
\text{something\_else}
\]
\[
\text{source.withdraw} (\text{amount})
\]
\[
\text{target.deposit} (\text{amount})
\]
\[
\text{end}
\]
\[
\text{end}
\]

Reasonability
Dining philosophers in SCOOP (1)

```coq
class PHILOSOpher feature
  live
    do
      from getup until over loop
        think: eat(left, right)
      end
    end

  eat(l, r: separate FORK)
  -- Eat, having grabbed l and r.
  do ... end

getup do ... end
over: BOOLEAN
end
```

Dining philosophers in SCOOP (2)

```coq
class PHILOSOpher inherit
  REPEATABLE
  rename
    setup as getup
  redefine step end

feature {BUTLER}
  step
    do
      think: eat(left, right)
    end

  eat(l, r: separate FORK)
  -- Eat, having grabbed l and r.
  do ... end
end
```
A library class to describe processes

SCOOP integrates inheritance and other O-O techniques with concurrency, seamlessly and without conflicts ("inheritance anomaly")
No need for built-in notion of active object: it is programmed through a library class such as:

```plaintext
class REPEATABLE feature
    setup do end
    step do end
    over: BOOLEAN
    tear_down do end
    live
        do
            from setup until over loop step end
        tear_down
        end
    end
end
```

Wrong (in my opinion) assumptions

"Objects are naturally concurrent" (Milner)

- Many attempts, often based on "Active objects" (a self-contradictory notion)
- Lead to artificial issue of "Inheritance anomaly"

"Concurrency is the basic scheme, sequential programming a special case" (many)

- Correct in principle, but in practice we understand sequential best
(D) What the hold rule means

Beat enemy number one in concurrent world: atomicity violations

- Data races
- Illegal interleaving of calls

Data races cannot occur in SCOOP

(D) Hold rule (1)

A routine application has exclusive access to all separate arguments

\[ a_{\text{routine}}(\text{nonsep}_a, \text{nonsep}_b, \text{sep}_c, \text{sep}_d, \text{sep}_e) \]

Exclusive access to \[ \text{sep}_c, \text{sep}_d, \text{sep}_e \] within \[ a_{\text{routine}} \]
### Semantics vs implementation

Older SCOOP literature says that feature application “waits” until all the separate arguments’ handlers are available. This is not necessary!

What matters is **exclusive access**: implementation does not have to wait unless semantically necessary. The implementation performs some of these optimizations:

```plaintext
f (a, b, c : separate T)
    do
        something_else
        a.r
        b.s
    end
```

No need to wait for `a` and `b` until here.

No need to wait for `c`!

### Implementation techniques

The literature on transactional memory (Herlihy) criticizes approaches of the form:

- Lock everything first
- Then think

Conceptually, SCOOP is of this kind.

But it can be implemented in a TM-like style.
**Resynchronization: lazy wait**

How do we resynchronize after asynchronous (separate) call? No explicit mechanism!

The client will wait when, and only when, it needs to:

\[
\begin{align*}
&x.f \\
&x.g(a) \\
&y.f \\
&\ldots \\
&\text{value} := x.\text{some\_query}
\end{align*}
\]

Lazy wait (also known as wait by necessity)

---

**Synchrony vs asynchrony revisited**

For a separate target \(x\):

- \(x.\text{command}(\ldots)\) is asynchronous
- \(v := x.\text{query}(\ldots)\) is synchronous
Exercise

If we do want to resynchronize explicitly, what do we do?

(This is sometimes known as a “barrier”)

Contracts

What becomes of contracts, in particular preconditions, in a concurrent context?
(F) Contracts

\[\text{put}(\text{buf}: \text{separate} \text{QUEUE [INTEGER]}; v: \text{INTEGER})\]
\[\text{-- Store v into buf.}\]
\[\text{require} \]
\[\text{not buf.is_full}\]
\[\text{do}\]
\[\text{buf.put}(v)\]
\[\text{ensure} \]
\[\text{not buf.is_empty}\]
\[\text{end}\]

... 
\[\text{put}(\text{my\_very\_own\_buf}, 10)\]

Reminder: reasoning from APIs

\[\text{put}(\text{buf: QUEUE; v: G})\]
\[\text{-- Insert v into buf.}\]
\[\text{require} \]
\[\text{not buf.is_full}\]
\[\text{do}\]
\[\text{...}\]
\[\text{ensure} \]
\[\text{...}\]
\[\text{end}\]
Reminder: reasoning from APIs

```haskell
put (buf: QUEUE; v: G)
  -- Add v to buf.
  require
  not buf.is_full
  do ...
  ensure ...
end
```

Reminder: the SCOOP solution

```haskell
put (buf: QUEUE; v: G)
  -- Add v to buf.
  require
  not buf.is_full
  do ...
  ensure ...
end
```
Reminder: bank accounts

if acc1.balance >= 100 then transfer (acc1, acc2, 100) end
if acc1.balance >= 100 then transfer (acc1, acc3, 100) end

transfer (source, target: ACCOUNT; amount: INTEGER)
-- Transfer amount from source to target.
require
source.balance >= amount
do
source.withdraw (amount)
target.deposit (amount)
ensure
source.balance = old source.balance - amount
target.balance = old target.balance + amount
end

Reminder: the SCOOP solution

transfer (source, target: ACCOUNT; amount: INTEGER)
-- Transfer amount from source to target.
require
source.balance >= amount
do
source.withdraw (amount)
target.deposit (amount)
ensure
source.balance = old source.balance - amount
target.balance = old target.balance + amount
end

invariant balance >= 0
**Contracts**

```plaintext
put (buf: separate QUEUE [INTEGER]; v: INTEGER)
-- Store v into buf.
    require
    not buf.is_full
    do
        buf.put (v)
    ensure
        not buf.is_empty
    end

... put (my_buffer, 10)
```

**Reminder: bank transfer**

```plaintext
transfer (source, target: separate ACCOUNT;
         amount: INTEGER)
-- Transfer amount from source to target.
    require
        source.balance >= amount
    do
        source.withdraw (amount)
        target.deposit (amount)
    ensure
        source.balance = old source.balance - amount
        target.balance = old target.balance + amount
    end
```
**(D)** Hold rule (1)

A routine application has *exclusive access* to all separate arguments

\[ a_{\text{routine}}(\text{nonsep}_a, \text{nonsep}_b, \text{sep}_c, \text{sep}_d, \text{sep}_e) \]

Exclusive access to \( \text{sep}_c, \text{sep}_d, \text{sep}_e \) within \( a_{\text{routine}} \)

---

**(F)** Full hold rule

A routine application has *exclusive access* to all separate arguments, and the guarantee that *separate preconditions* hold before accessing them

*"Separate precondition":*

\[
r(x: \text{separate } T; \ldots)
\]

\[
\text{require}
\]

\[
x.\text{some}_\text{property}(\ldots)
\]

\[
\text{do}
\]

\[
\ldots
\]

Reasonability
Reminder: hexapod robot

Hind legs have force sensors on feet and retraction limit switches

Reminder: hexapod locomotion

Alternating protraction and retraction of tripod pairs

- Begin protraction only if partner legs are down
- Depress legs only if partner legs have retracted
- Begin retraction when partner legs are up
Reminder: hexapod coordination rules

**R1:** Protraction can start only if partner group on ground
- **R2.1:** Protraction starts on completion of retraction
- **R2.2:** Retraction starts on completion of protraction

**R3:** Retraction can start only when partner group raised

**R4:** Protraction can end only when partner group retracted


Reminder: SCOOP hexapod

```c
begin_protraction (partner, me: separate LEG_GROUP)
require
    me.legs_retracted
    partner.legs_down
    not partner.protraction_pending
do
    tripod.lift
    me.set_protraction_pending
end
```
Hexapod coordination rules

**R1**: Protraction can start only if partner group on ground
- **R2.1**: Protraction starts on completion of retraction
- **R2.2**: Retraction starts on completion of protraction

**R3**: Retraction can start only when partner group raised

**R4**: Protraction can end only when partner group retracted


---

Which semantics applies?

```haskell
put (buf : separate QUEUE [INTEGER]; i : INTEGER)
require
not buf.is_full
i > 0
do
buf.put(i)
end
```

Wait condition

```haskell
my_buffer : separate QUEUE [INTEGER]
put (my_buffer, 10)
```

Correctness condition
Generalized semantics of preconditions

The different semantics is surprising at first:

- Separate: wait condition
- Non-separate: correctness condition

At a high abstraction level, however, we may consider that

- Wait semantics always applies in principle
- Sequentiality is a special case of concurrency
- Wait semantics boils down to correctness semantics for non-separate preconditions.
  - Smart compiler can detect some cases
  - Other cases detected at run time

What about postconditions?

\[
\text{spawn\_two\_activities (loc1, loc2: separate LOCATION)} \\
\text{do} \\
\quad \text{loc1.do\_job} \\
\quad \text{loc2.do\_job} \\
\text{ensure} \\
\quad \text{loc1.is\_ready} \\
\quad \text{loc2.is\_ready} \\
\text{end}
\]

\[
\text{spawn\_two\_activities (zurich, lausanne)} \\
\text{do\_local\_stuff} \\
\text{get\_result (zurich)}
\]

Should we wait for zurich.is_ready?
Reasoning about objects: sequential

\[
\{ \text{INV and Pre}_r \} \ \text{body}_r \ \{ \text{INV and Post}_r \}
\]

\[
\{ \text{Pre}_r \} \ x.r (a) \ \{ \text{Post}_r \}
\]

Only \( n \) proofs if \( n \) exported routines!

(D) Reminder: access control policy

Since separate calls are asynchronous there is a real danger of confusion

Consider for example

\[
\text{remote_stack}: \text{separate STACK}[T]
\]

\[
\ldots
\]

\[
\text{remote_stack.put}(a)
\]

\[
\ldots \text{Instructions not affecting the stack...}
\]

\[
y := \text{my_stack.item}
\]

Reasonability
Reminder: access control policy

SCOOP requires the target of a separate call to be a formal argument of the enclosing routine:

```SCOOP
put (s: separate STACK[T]; value: T)
-- Store value into s.
do
    s.put(value)
end
```

To use a separate object:
```
my_stack: separate STACK[INTEGER]
create my_stack
put (my_stack, 10)
```

Separate argument rule

The target of a separate call must be an argument of the enclosing routine.

Separate call: `x.f(...)` where `x` is separate
Full hold rule

A routine application has exclusive access to all separate arguments, and the guarantee that separate preconditions hold before accessing them.

"Separate precondition":

\[
\text{r}(x: \text{separate } T; \ldots) \\
\text{require} \\
\quad \text{\texttt{x.some\_property(\ldots)}} \\
\text{do} \\
\quad \ldots \\
\text{...}
\]

Separate argument rule

The target of a separate call must be an argument of the enclosing routine.

Separate call: \texttt{x.f(\ldots)} where \(x\) is separate.
Making it easier?

With \textit{my\_sep : separate } T, you may \textbf{not} use \textit{my\_sep\_do\_something} but have to write a routine:

\begin{verbatim}
wrap\_something (x: separate T)
do x\_do\_something end
\end{verbatim}

and call \textit{wrap\_something (my\_sep)}

Syntax under discussion: accept

\begin{verbatim}
separate my\_sep as x do
  my\_x\_do\_something
end
\end{verbatim}

as a syntactical abbreviation for \textbf{[1]}

Reasoning about objects: SCOOP

\[
\frac{\{INV \land Pre_r (x)\} body_r, \{INV \land Post_r (x)\}}{\{Pre_r (a \text{cont})\} e.r (a) \{Post_r (a \text{cont})\}}
\]

Hoare-style sequential reasoning

Controlled expressions (known statically as part of the type system) are:

- Attached (statically known to be non-void)
- Handled by processor locked in current context
Other aspects

What if a separate call, e.g. in

\[
\begin{aligned}
r(a: \text{separate } T) \\
\text{do} \\
a.f \\
a.g \\
a.h \\
\text{end}
\end{aligned}
\]

causes an exception?

Refresher: Computational Model

- Software system is composed of several processors
- Processors are sequential; concurrency is achieved through their interplay
- Separate variable denotes potentially separate object
- Calls to non-separate objects are synchronous
- Calls to separate objects are asynchronous
Refresher: Synchronization

Mutual exclusion:
- Exclusive access through argument passing
- Routine body is critical section

Condition synchronization:
- Separate preconditions used as wait conditions

Re-synchronisation of client and supplier:
- Wait by necessity

Lock passing: through argument passing

---

**SCOOP: the type system**

From material by Piotr Nienaltowski
A traitor is an entity that
- Statically, is declared as non-separate
- During an execution, can become attached to a separate object.

---

**Traitors here...**

```plaintext
-- in class C (client)
x1: separate T
a: A

r (x: separate T)
do
  a := x.b
end

r (x1)
a.f
```

---

- **Is this call valid?**  
  - **Yes**

- **And this one?**  
  - **No**
Traitors there...

```
-- in class C (client)
x1: separate T
a: A
r (x: separate T)
  do
    x.f (a)
  end
r (x1)
```

```
-- supplier
class T feature
f (b: A)
do
  b.f
end
```

Is this call valid? Yes

And this one?

Consistency rules: first attempt

Original model (Object-Oriented Software Construction, chapter 30) defines four consistency rules that eliminate traitors

Written in English

Easy to understand by programmers

Sound? Complete?
Original consistency rules (3)

Separateness Consistency Rule (1)

If the source of an attachment (assignment or argument passing) is separate, its target must be separate too.

```r
r (buf: separate BUFFER [T]; x: T)
local
  buf1: separate BUFFER [T]
  buf2: BUFFER [T]
  x2: separate T
do
  buf1 := buf -- Valid
  buf2 := buf1 -- Invalid
  r (buf1, x2) -- Invalid
end
```

Separateness Consistency Rule (2)

If an actual argument of a separate call is of a reference type, the corresponding formal argument must be declared as separate.

```-- In class BUFFER [G]:
p (element: separate G)
```

```-- In another class:
store (buf: separate BUFFER [T]; x: T)
do
  buf.put (x)
end
...```
Original consistency rules (3)

Separateness Consistency Rule (3)

If the source of an attachment is the result of a separate call to a query* returning a reference type, the target must be declared as separate.

---

-- In class BUFFER [G]:
  item: G

-- In another class:
  consume (buf: separate BUFFER [T])
  local
    element: separate T
  do
    element := buf.item
    ...
  end

(*A query is an attribute or function)

Original consistency rules (4)

Separateness Consistency Rule (4)

If an actual argument or result of a separate call is of an expanded type, its base class may not include, directly or indirectly, any non-separate attribute of a reference type.

---

-- In class BUFFER [G]:
  put (element: G)

  -- G not declared separate

-- In another class:
  store (buf: separate BUFFER [E]; x: E)
  do
    buf.put (x)
    -- E must be "fully expanded"
  end

  ...


A type system for SCOOP

Goal: prevent traitors through static (compile-time) checks

Simplifies, refines and formalizes SCOOP rules

Integrates expanded types and agents with SCOOP

Tool for reasoning about concurrent programs

- May serve as basis for future extensions, e.g. for deadlock prevention schemes

Three components of a type

Notation:

\[ \Gamma 
\vdash 
x :: (\gamma, \alpha, C) \]

1. Attached/detachable: \( \gamma \in \{!, ?\} \)

2. Processor tag \( \alpha \in \{\bullet, T, \bot, <p>, <a\cdot handler>\} \)

3. Ordinary (class) type \( C \)
Examples

\begin{align*}
  u &\colon U \\
  v &\colon \text{separate } V \\
  w &\colon \text{detachable separate } W \\

\text{--- Expanded types are attached and non-separate:} \\
  i &\colon \text{INTEGER} \\
  V &\colon \text{Void} \\
  C &\colon \text{Current} \\
  x &\colon \text{separate } <px> T \\
  y &\colon \text{separate } <px> Y \\
  z &\colon \text{separate } <px> Z
\end{align*}

Informal Subtyping Rules

Conformance on class types like in Eiffel, essentially based on inheritance:

\[ D \preceq \text{Eiffel } C \iff (\gamma, \alpha, D) \preceq (\gamma, \alpha, C) \]

Attached \preceq detachable:

\[ (I, \alpha, C) \preceq (? , \alpha, C) \]

Any processor tag \preceq T:

\[ (\gamma, \alpha, C) \preceq (\gamma, T, C) \]

In particular, non-separate \preceq T:

\[ (\gamma, \circ, C) \preceq (\gamma, T, C) \]

\( \bot \preceq \) any processor tag:

\[ (\gamma, \bot, C) \preceq (\gamma, \alpha, C) \]
Assignment examples

- a: separate T  
  -- a :: (l, T, T)
- b: T  
  -- b :: (l, ●, T)
- c: detachable T  
  -- c :: (? , ●, T)
- f (x, y: separate T) do ... end  
  -- x :: (l, T, T), y :: (l, T, T)
- g (x: T) do ... end  
  -- x :: (l, ●, T)
- h (x: detachable T): ! p T do ... end  
  -- x :: (? , ●, T) : (!, p, T)

Unified rules for call validity

Informally, a variable x may be used as target of a separate feature call in a routine r if and only if:

- x is attached
- The processor that executes r has exclusive access to x's processor.
Feature Call rule

An expression \( exp \) of type \((d, p, C)\) is **controlled** if and only if \( exp \) is attached and satisfies *one* of the following conditions:

- \( exp \) is non-separate, i.e. \( p = \cdot \)
- \( exp \) appears in a routine \( r \) that has an attached formal argument \( a \) with the same handler as \( exp \), i.e. \( p = a \cdot \)handler

A call \( x \cdot f(a) \) appearing in the context of a routine \( r \) in a class \( C \) is valid if and only if *both*:

- \( x \) is controlled
- \( x \)'s base class exports feature \( f \) to \( C \), and the actual arguments conform in number and type to formal arguments of \( f \)

Unqualified explicit processor tags

Unqualified explicit processor tags rely on a processor attribute.

- \( p: \text{PROCESSOR} \) -- Tag declaration
- \( x: \text{separate} <p> T \) -- \( x :: (l, <p>, T) \)
- \( y: \text{separate} <p> Y \) -- \( y :: (l, <p>, Y) \)
- \( z: \text{separate} Z \) -- \( z :: (l, T, Z) \)

Attachment (assume that \( Y \) and \( Z \) are descendants of \( T \))

- \( x := y \) -- *Valid because* \((l, <p>, Y) \preceq (l, <p>, T)\)
- \( y := z \) -- *Invalid because* \((l, T, Z) \not\preceq (l, <p>, Y)\)

Object creation

- \( \text{create } x \) -- *Fresh processor created to handle* \( x \).
- \( \text{create } y \) -- *No new processors created;* \( y \) *is put* -- on \( x \)'s processor.
Qualified explicit processor tags

Declared using “feature” handler on a read-only attached entity (such as a formal argument or current object)

\[ x: \text{separate} \ <y\text{-handler}> T \]
\[ \quad \text{-- } x \text{ is handled by handler of } y \]

Attachment, object creation:
\[ r (\text{list: separate LIST } [T]) \]
\[
\text{local}
\begin{align*}
\quad & s1, s2: \text{separate} <\text{list\text{-}handler}> \text{STRING} \\
\quad & \text{-- } s1, s2 :: (l, <\text{list\text{-}handler}>, \text{STRING})
\end{align*}
\]
\[ \text{do} \]
\[
\begin{align*}
\quad & s1 := \text{list} [1] \\
\quad & s2 := \text{list} [2] \\
\quad & \text{list\text{-}extend (s1 + s2)} \quad \text{-- Valid} \\
\quad & \text{create } s1\text{.make\_empty} \quad \text{-- } s1 \text{ created on list’s processor} \\
\quad & \text{list\text{-}extend (s1)} \quad \text{-- valid}
\end{align*}
\]
\[ \text{end} \]

Processor tags

Processor tags are always \textit{relative} to the current object

For example an entity declared as non-separate is seen as non-separate by the current object. Separate clients, however, should see the entity as separate, because from their point of view it is handled by a different processor.

Type combinators are necessary to calculate the relative type:

\begin{itemize}
\item Formal arguments
\item Result
\end{itemize}
**Result type combinator**

What is the type $T_{\text{result}}$ of a query call $x.f(...)$?

$$T_{\text{result}} = T_{\text{target}} \times T_f$$

$$= (\alpha x, px, TX) \times (\alpha f, pf, TF)$$

$$= (\alpha f, p_r, TF)$$

**Argument type combinator**

What is the expected actual argument type in $x.f(a)$?

$$T_{\text{actual}} = T_{\text{target}} \otimes T_{\text{formal}}$$

$$= (\alpha x, px, TX) \otimes (\alpha f, pf, TF)$$

$$= (\alpha f, p_a, D)$$

$pr$

$pa$
Expanded objects are always attached and non-separate.

Both * and ⊗ preserve expanded types

- \((\gamma, \alpha, C) * (!, \bullet, \text{INTEGER}) = (!, \bullet, \text{INTEGER})\)
- \((\gamma, \alpha, C) \otimes (!, \bullet, \text{BOOLEAN}) = (!, \bullet, \text{BOOLEAN})\)

\[
\begin{align*}
x_1 & : T \quad -- \ x_1 :: (!, \bullet, T) \\
y_1 & : \text{separate } Y \quad -- \ y_1 :: (!, T, Y) \\
y_1 \cdot r (x_1) \quad -- \ (!, \bullet, T) \leq (!, T, Y) \otimes (!, \bullet, T) \\
\end{align*}
\]

-- so call is valid

---

The non-separateness of expanded objects needs to be preserved when such an object crosses processor barriers.

Import operation (implicit): like copy, but clones (recursively) all non-separate attributes.

**Variations**

- **Deep import**: The relative separateness of objects is preserved; copies are placed on the same processors as their originals.
- **Flat import**: The whole object structure is placed on the client’s processor.
- **Independent import**: The relative separateness of objects is preserved but copies are placed on fresh processors.
Recall: Traitors here...

-- in class C (client)
x1: separate T
a: A

-- supplier
x1 :: (!, T, T)
class T
a :: (!, ●, A)

--- A

--- in class C (client)
x1: separate T

--- A

Recall: Traitors there...

-- in class C (client)
x1: separate T
a: A

-- supplier
x1 :: (!, T, T)
class T
a :: (!, ●, A)

--- A

--- A

--- A

--- A

Implicit types

An attached non-writable entity $e$ of type $T_e = (I, \alpha, C)$ also has an implicit type $T_{e, \text{imp}} = (I, e.\text{handler}, C)$.

**Example**

```plaintext
r (x: separate T; y: detachable Y)

local
  z: separate Z

  do ... end

s: STRING = "I am a constant"

u: separate U once ...

```

```
x :: (!, T, T) = (!, x.\text{handler}, T)

y :: (?!, \text{●}, Y) no implicit type because y is detachable

z :: (!, T, Z) no implicit type because z is writable

s :: (!, \text{●}, STRING) = (!, s.\text{handler}, STRING)

u :: (!, T, U) = (!, u.\text{handler}, U)
```

False Traitors

```plaintext
meet_friend (person: separate PERSON)

local
  a_friend: PERSON

  do
    a_friend := person.friend -- invalid assignment.
    visit (a_friend)

end
```

wife

spouse friend

spouse friend

hubby

spouse friend

Urs

spouse friend

wed
Handling false traitors with Object Tests

Use Eiffel object tests with downcasts of processor tags.

An object test succeeds if the run-time type of its source conforms in all of:
- Detachability
- Locality
- Class type to the type of its target.

This allows downcasting a separate entity to a non-separate one, provided that the entity represents a non-separate object at runtime.

```
meet_friend (person: separate PERSON)
  do
    if attached (PERSON) person.friend as a_friend then
      visit (a_friend)
    end
  end
```

Object creation

```
p: PROCESSOR
a: separate X
b: X
c, d: separate <p> X
create a
create b
create c
create d
```

Processor tag
Create fresh processor for a
Place b on current processor
Create fresh processor p for c
Processor p already exists: place d on p
Without lock passing

\[ r \left( x: \text{separate} \ X; y: \text{separate} \ Y \right) \]

\[
\text{local} \\
\text{z: separate ANY} \\
\text{do} \\
x.f \\
x.g \left( y \right) \\
y.f \\
z := x.\text{some\_query} \\
\text{end}
\]

\text{Waits for y to become available}

\text{Deadlock: wait for some\_query to finish}

Lock passing

... \hspace{1cm} P_c \hspace{1cm} P_y \hspace{1cm} P_x

\[ x.f \]

\[ x.g \left( y \right) \]

... \hspace{1cm} y.f

\text{y.f}

\text{g (y: separate Y)} \hspace{1cm} \text{do} \\
\text{y.f} \\
\text{...} \\
\text{...} \\
\text{...} \\
\text{...} \\
\text{end}
Lock passing

- Must not compromise the atomicity guarantees
- Clients must be able to decide to pass or not to pass a lock
- The mechanism should increase the expressiveness of the language, not restrict it
- The solution must be simple and well integrated with other language mechanism

If a call x.f (a1, ..., an) occurs in a routine r where some ai is controlled, the client’s handler (the processor executing r) passes all currently held locks to the handler of x, and waits until f terminates.

When f terminates, the client resumes its computation.

```plaintext
r (x: separate X; y: separate Y)
local
z: separate ANY
do
  x.f
  x.g (y)
  y.f
  z := x. some_query
end
```

Pass locks to g and wait for g to finish
Synchronous

Synchronous
Lock passing combinations

<table>
<thead>
<tr>
<th>Actual</th>
<th>Attached</th>
<th>Detachable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference, controlled</td>
<td>Lock passing</td>
<td>no</td>
</tr>
<tr>
<td>Reference, uncontrolled</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Expanded</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

Lock passing: example

```
class C feature
  x1: X
  z1: separate Z
  c1: separate C
  i: INTEGER
  r (x: separate X; y: separate Y)
do
  x1.f (5)  
  x1.g (x)  
  i := x1.h (Current) 
  x.f (10)  
  x.g (z1)  
  x.g (y)   
  x.m (y)   
i := x.h (c1) 
i := x.h (Current) 
end
p (...) do ... end
end
```

```class X feature
  f (i: INTEGER) do ... end
  g (a: separate ANY) do ... end
  h (c: separate C): INTEGER do c.p (...) end
  m (a: detachable separate ANY) do ... end
end
```
- 5 -

**SCOOP: the future**

### Next steps

**Work in progress**
- Object import
- Exception handling
- Profiling
- Performance improvements

**Open problems**
- Multiple readers
- Deadlock prevention and detection

**Research topics**
- More libraries
- Full axiomatic semantics, integration into verification environment (EVE)
- Object migration
- Distributed computing
- Web services
- Real-time
- Transactions
- SIMD computing
Status

- All of SCOOP as described here implemented
- Part of EiffelStudio since 7.0
- Preprocessor and library available for download
- Numerous examples available for download

se.ethz.ch/research/scoop.html

What can SCOOP do for us?

Beat enemy number one in concurrent world: atomicity violations
- Data races
- Illegal interleaving of calls

Data races cannot occur in SCOOP
- Why? See computational model ...
SCOOP highlights

- Close connection to O-O modeling
- Natural use of O-O mechanisms such as inheritance
- Built-in guarantee of no data races
- Built-in fairness
- Removes many concerns from programmer
- Supports many different forms of concurrency
- Retains accepted patterns of reasoning about programs
- Simple to learn and use

Reasonability

Exercise 1 (from Tanenbaum via Downey)

There is a deep canyon somewhere in Kruger National Park and a single rope that spans the canyon. Baboons can cross the canyon by swinging on the rope, but if two baboons going in opposite directions meet in the middle, they will fight and drop to their deaths.

The rope is only strong enough to hold 5 baboons. If there are more, it will break.

Devise a synchronization scheme such that:

- Once a baboon has begun to cross, it is guaranteed to get to the other side without running into a baboon going the other way.
- There are never more than 5 baboons on the rope.
Exercise 2 (from Downey)

Near Redmond, WA, there is a rowboat used by both Linux hackers and Microsoft employees to cross a river. It holds exactly four people; it won't leave with more or fewer. For safety, it is not permissible to put one hacker in the boat with three employees, or one employee with three hackers. Other combinations are safe.

As each thread boards the boat it should call a function `board`. All four threads from each boatload must call `board` before any of the threads from the next boatload do.

After all four threads have call `board`, exactly one of them should call a function `row_boat`, indicating that it takes the oars. It doesn't matter which thread calls it.

Don't worry about the direction of travel.

Reminder: the plan

Plan:
- 1. The trailer
- 2. A bit of context
- 3. The model: successive restrictions
- 4. The SCOOP type system
- 5. Open problems and current work
- 6. Some other stuff! (Teaching, verification...)
Teaching programming: concepts or skills?

Skills supporting concepts
### Teaching programming: some critical concepts

| Specification vs implementation, information hiding, abstraction | Notation |
| Change | Syntax vs validity vs semantics |
| Structure | Recursive reasoning |
| Reuse | Classification |
| Function vs data | Complexity & impossibility |
| Complexity | Static vs dynamic |

### Introductory programming teaching

*Teaching first-year programming is a politically sensitive area, as you must contend not only with your students but also with an intimidating second audience — colleagues who teach in subsequent semesters...*

*Academics who teach introductory programming are placed under enormous pressure by colleagues.*

*As surely as farmers complain about the weather, computing academics will complain about students’ programming abilities.*

Raymond Lister: *After the Gold Rush: Toward Sustainable Scholarship in Computing, 10th Conf. on Australasian computing education, 2008*
Some challenges in teaching programming

- Ups and downs of high-tech economy, image of CS
- Offshoring and globalization raise the stakes
- Short-term pressures (e.g. families), IT industry fads
- Widely diverse student motivations, skills, experience

The Facebook generation: 1st-year CS students

Computer experience
- ≥10 yrs: 54%
- 5-9 yrs: 42%
- 2-4 yrs: 4%

Programming experience
- ≥100 classes: 10%
- Some: 42%
- No O-O: 30%
- None: 18%

Averages over 6 years, 2003-2008 (yearly variations small)

Full year-by-year figures: Pedroni, Meyer, Oriol, They know more than we think!, Tech Rep. 631, ETH, 2009
Ways to teach introductory programming

- 1. “Programming in the small”
- 2. Learn APIs
- 3. Teach a programming language: Java, C++, C#
- 4. Functional programming
- 5. Completely formal, don’t touch a computer

Our approach: Outside-In (inverted curriculum)

Concepts or skills?

Skills supporting concepts
Teaching programming: some critical concepts

- Specification vs implementation, information hiding, abstraction
- Change
- Structure
- Recursive reasoning
- Reuse
- Classification
- Complexity & impossibility
- Function vs data
- Algorithmic reasoning
- Scaling up
- Typing
- Complexity
- Static vs dynamic
- Invariant

Levenshtein distance

"Beethoven" to "Beatles"

```
BEETHOVEN
```

```
AL
```

Operation: - - R - D R D - R
Distance: 0 0 1 1 2 3 4 4 5
Levenshtein algorithm

\[
\begin{align*}
\text{across } r: 1 \ldots \text{rows as } i \text{ loop} \\
&\text{across } c: 1 \ldots \text{columns as } j \text{ invariant} \\
&\quad \text{\texttt{\textbf{D}[, ,] := \text{\texttt{D}[, -1, -1]}} if source[i] = target[j] then} \\
&\quad \text{\texttt{D}[i, j] := 1 + min(D[i-1, j], D[i, j-1], D[i-1, j-1]) else} \\
&\quad \text{\texttt{D}[i, j] := D[i-1, j-1] end} \\
&\text{end} \\
\text{Result} := \text{\texttt{D[rows, columns]}}
\end{align*}
\]
Invariant: each $D[i, j]$ is distance from source $[1..i]$ to target $[1..j]$
Outside-in (Inverted Curriculum): intro course

Fully object-oriented from the start, using Eiffel Design by Contract principles from the start

Component based: students use existing software (TRAFFIC library):
- They start out as consumers
- They end up as producers!

“Progressive opening of the black boxes”

TRAFFIC is graphical, multimedia and extendible

(Approach 3: teaching a specific language)

First Java program:

class First {
    public static void main(String args[]) {
        System.out.println("Hello World!");
    }
}
**Principles of the ETH course**

- Reuse software: inspiration, imitation, abstraction
- See lots of software
- Learn to reuse through interfaces and contracts
- Interesting examples from day one
- Combination of principles and practices

Traditional topics too: algorithms, control structures, basic data structures, recursion, syntax & BNF, ...

Advanced topics: closures & lambda-calculus, some design patterns, intro to software engineering...
Distributed software engineering

Today's software development is multipolar
University seldom teach this part!

“Software Engineering for Outsourced and Offshore Development” since 2003, with Peter Kolb

Since 2007: Distributed & Outsourced Software Engineering (DOSE)

The project too is distributed. Currently: ETH, Politecnico di Milano, U. of Nijny Novgorod, Odessa Polytechnic, U. Debrecen, Hanoi University of Technology

The DOSE project

Setup: each group is a collection of teams from different university; usually 2 teams, sometimes 3

Division by functionality, not lifecycle

Results:
- Hard for students
- Initial reactions often negative
- In the end it works out
- The main lesson: interfaces & abstraction

Open to more institutions (mid-Sept to mid-Dec):
http://se.ethz.ch/dose
Verification As a Matter Of Course

Concurrent programming is easy!
An overview of today's SCOOP

Bertrand Meyer
Microsoft Russian summer school
Saint Petersburg, 23 August 2012