FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Departamento de Informática

Mestrado em Engenharia Informática

Design and Implementation of a Behaviorally Typed Programming System for Web Services

Dissertação de Mestrado

Filipe David Oliveira Militão (26948)

Orientador: Prof. Doutor Luís Caires

Part 1 – Introduction (~10 min)

- Motivation
- What is a Behavioral Type?
- Why do we need Behavioral Types?
- Overview (programmer's perspective)
- Contributions

Motivation

- Increasing *software* complexity
  - requires more sophisticated tools
  - faster feedback on possible errors
  - cut back errors only detectable at runtime

- *Web Services*
  - many standards (WSDL, etc)
  - dynamic combination of services
    - + automatic type compatibility checks
    - - behavior "assumed" compatible

→ ease *Web Services* use/composition
→ statically check concurrent compositions

What is a Behavioral Type?

Imagine a monster with a strange habit of *squashing* cats and then cook them into *pancakes* or *tea* right before going to *sleep*.

What is a Behavioral Type?

Imagine a monster with a strange habit of *squashing* cats and then cook them into *pancakes* or *tea* right before going to *sleep*.

```
Monster Type
    -  squash(Cat)
    -  makePancakes(Cat)
    -  makeTea(Cat)
    -  sleep()
```

What is a Behavioral Type?

Imagine a monster with a strange habit of *squashing* cats and then cook them into *pancakes* or *tea* right before going to *sleep*.

```
Monster Type
    - squash(Cat)
    - makePancakes(Cat)
    - makeTea(Cat)
    - sleep()

Behavior
    1º squash cat
    2º pancakes or tea
    3º sleep
```

What is a Behavioral Type?

Imagine a monster with a strange habit of *squashing* cats and then cook them into *pancakes* or *tea* right before going to *sleep*.

```
Monster Type
    - squash(Cat)
    - makePancakes(Cat)
    - makeTea(Cat)
    - sleep()

Behavior
    1º squash cat
    2º pancakes or tea
    3º sleep
```

Behavioral Type = Type + Behavior

Why do we need Behavioral Types?

- statically check a program's correct *flow* of calls (ignoring possible *trapped errors*)
- **benefits**: avoids less obvious errors such as opened file/sockets not being safely closed after use (could lead to possible loss of data)
- Behavioral checking includes:
  - verifying termination in the use of a behavior (correct resource discard)
  - checking branches, loops and exceptions in a flexible way
  - deciding if/when a behavioral type can be replaced by another behavior

Overview – Programmer's perspective (I)
(requirements)

Don't Panic Airlines wants to create a simple Web Service for its customers and requires:

- all clients must be authenticated (logged in)
- it's possible to choose a *special package*, although some might be sold out
- in the case of booking a *simple flight* there's an additional option of also booking a *return flight*
- it should also be possible to *list all available flights*
- “at most, only one purchase per log in / session”

Overview – Programmer's perspective (II)

(initial approach to the problem)

```
class DPA {

   login(string username, string password) { ... }
   logout(){ ... }

   specialPackage(string type) throws SoldOut { ... }

   bookDestination(string dest){ ... }
   bookReturnFlight(){ ... }

   printAllAvailableFlights(){ ... }

}
```

Overview – Programmer's perspective (II)

(identifying behavioral and “free” methods)

only available on specific situations

```
class DPA {

    login(string username, string password) { ... }
    logout(){ ... }

    specialPackage(string type) throws SoldOut { ... }

    bookDestination(string dest){ ... }
    bookReturnFlight(){ ... }

    printAllAvailableFlights(){ ... }

}
```

can be called freely

Overview – Programmer's perspective (III)

In order to restrict the use of those methods, we define a specific usage protocol to be applied to anyone using the class.

This protocol is only related to the method's name, not their return type or arguments.

Overview – Programmer's perspective (IV)

(sequence protocol)

```
login ; logout
```

Overview – Programmer's perspective (IV)

(adding external choices to the protocol)

```
login ;
  ( bookDestination ; bookReturnFlight? )
                  +
            specialPackage
                  +
                 stop
; logout
```

Overview – Programmer's perspective (IV)

(adding internal choice [*SoldOut exception*] and recursion point [*choose*])

```
login ;
&choose(
   ( bookDestination ; bookReturnFlight? )
                    +
     specialPackage[SoldOut: choose ]
                    +
                  stop
)
; logout
```

Overview – Programmer's perspective (V)

(the complete class definition *with* a behavioral protocol)

usage protocol for class DPA

```
class DPA {

  usage login ;
    &choose(
      ( bookDestination ; bookReturnFlight? )
      + specialPackage[SoldOut: choose ]
      + stop
    ) ; logout

  login(string username, string password) { ... }
  logout(){ ... }

  specialPackage(string type) throws SoldOut { ... }

  bookDestination(string dest){ ... }
  bookReturnFlight(){ ... }

  printAllAvailableFlights(){ ... }
}
```

Overview – Programmer's perspective (VI)

(using the behavioral class DPA - sequential part of the protocol)

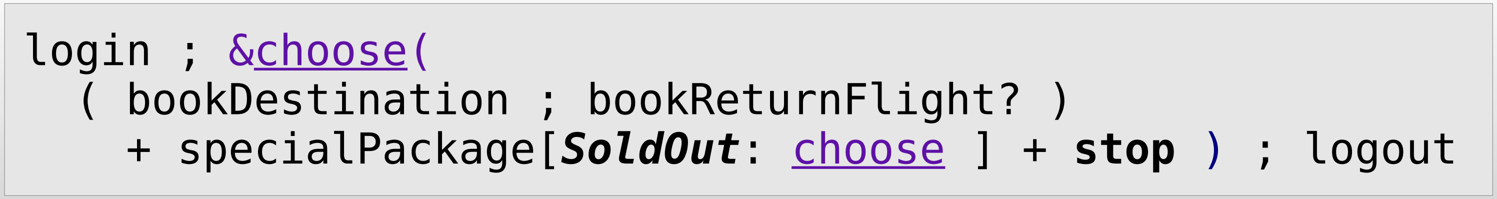
```
requestFlight(DPA s){
   s.login(“usr”,”pwd”);
   //...
   s.logout();
}
```

```
login ; &choose(
  ( bookDestination ; bookReturnFlight? )
    + specialPackage[SoldOut: choose ] + stop ) ; logout
```

Overview – Programmer's perspective (VI)

(using 2 of the 3 possible external choices)

```
requestFlight(DPA s){

    s.login(“usr”,”pwd”);
    s.printAllAvailableFlights();

    if( ? ){
        //choice 1
    }
    else{
        //choice 2
    };

    s.logout();

}
```



```
login ; &choose(
  ( bookDestination ; bookReturnFlight? )
    + specialPackage[SoldOut: choose ] + stop ) ; logout
```

Overview – Programmer's perspective (VI)

(mixing internal and external choices)

```
requestFlight(DPA s){

   s.login(“usr”,”pwd”);
   s.printAllAvailableFlights();

   if( ? ){
      s.bookDestination(“Lisbon”);
      if( ? ){ s.bookReturnFlight(); }
   }
   else{
      try{
         s.specialPackage(“around the world 80”);
      }catch(SoldOut out){
         //never mind then...
      }
   };

   s.logout();
}
```

```
login ; &choose(
  ( bookDestination ; bookReturnFlight? )
    + specialPackage[SoldOut: choose ] + stop ) ; logout
```

Contributions

- Design of the programming language **yak**
- Design and formalization of a behavioral type system
- Implementation of a fully functional proof-of-concept prototype

Contributions

- Design of the programming language **yak**
  - simple (minimalistic)
  - Java “inspired” (similar syntax)
  - apply main features of the type system
- Design and formalization of a behavioral type system
- Implementation of a fully functional proof-of-concept prototype

Contributions

- Design of the programming language **yak**

- Design and formalization of a behavioral type system
  - behavioral termination
  - behavioral ownership
  - branching
  - loops
  - exceptions (new approach in behavioral types)
  - ...

- Implementation of a fully functional proof-of-concept prototype

Contributions

- Design of the programming language **yak**
- Design and formalization of a behavioral type system
- Implementation of a fully functional proof-of-concept prototype
  - language parser
  - interpreter
  - run-time system (WS using HTTP+XML)
  - type checker (based on DFA manipulation)
  - examples
  - available for download

Part 2 - How it works (~7 min)

- Protocol
- Program's Structure
- Type System

Protocol (I)

- Describes sequences of (allowed) behavioral calls
- Any protocol may include:
  - method's names
  - ***exceptions types***
  - recursion labels

- empty behavior: **`stop`** (behavior of basic types)
- operators:

`a + b`	choice
`a ; b`	sequence
`a*`	repetition
`&label(a;stop+label)`	(limited) recursion
`a[Error: b];c`	exceptions

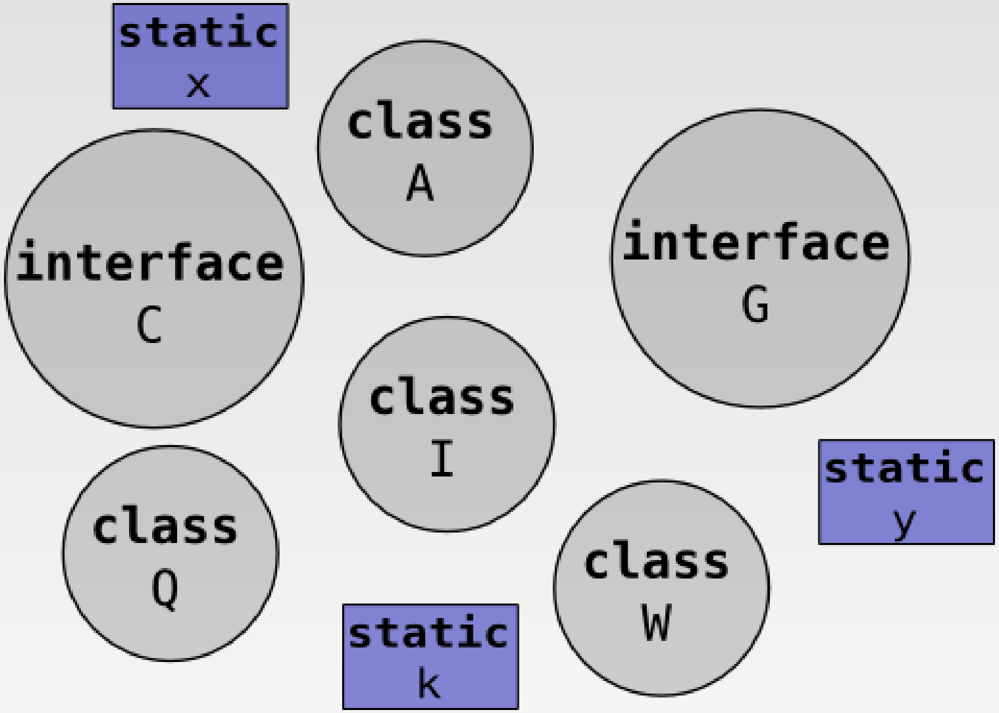
Protocol (II)

- Can express more complex behaviors like “repeat on error”:

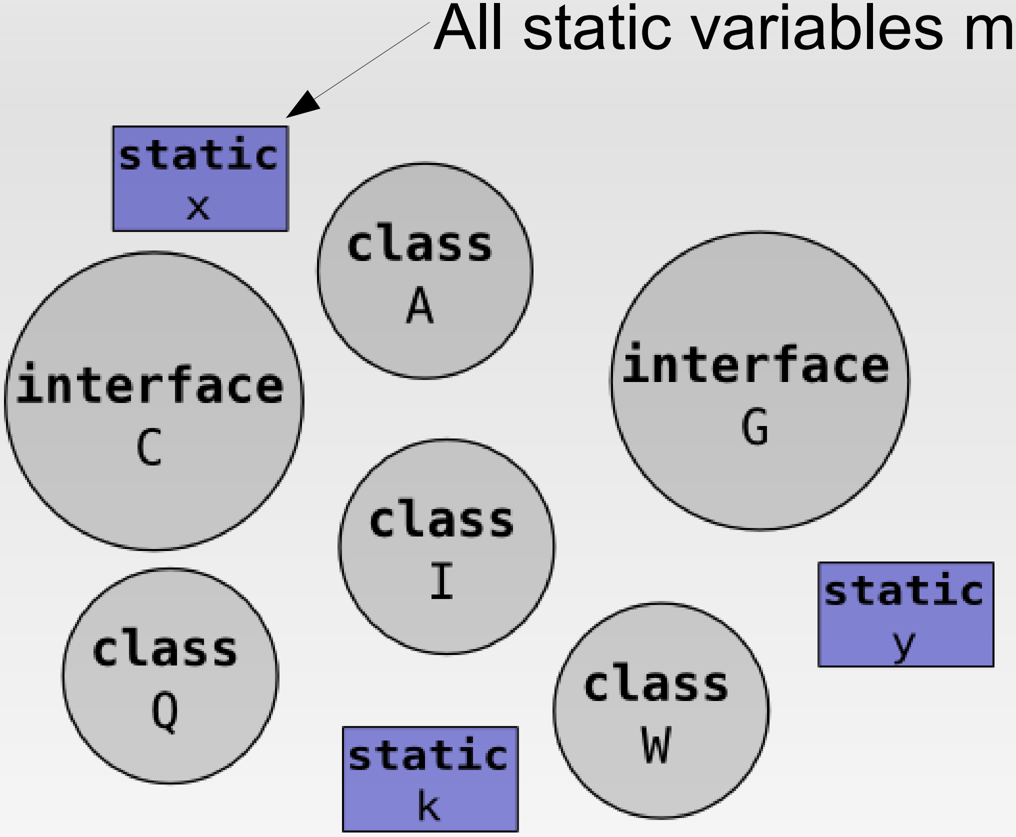
```
&start( hello[NoReply: start];goodbye )
```

- + operator → “external” choice
  - The programmer may choose freely any of the given options
- exceptions → “internal” choice
  - The internal logic of the class decides to change the allowed protocol and “announces” the change as an exception
- Internally, the protocol is converted to a *Deterministic Finite Automaton (DFA)*

Program's Structure



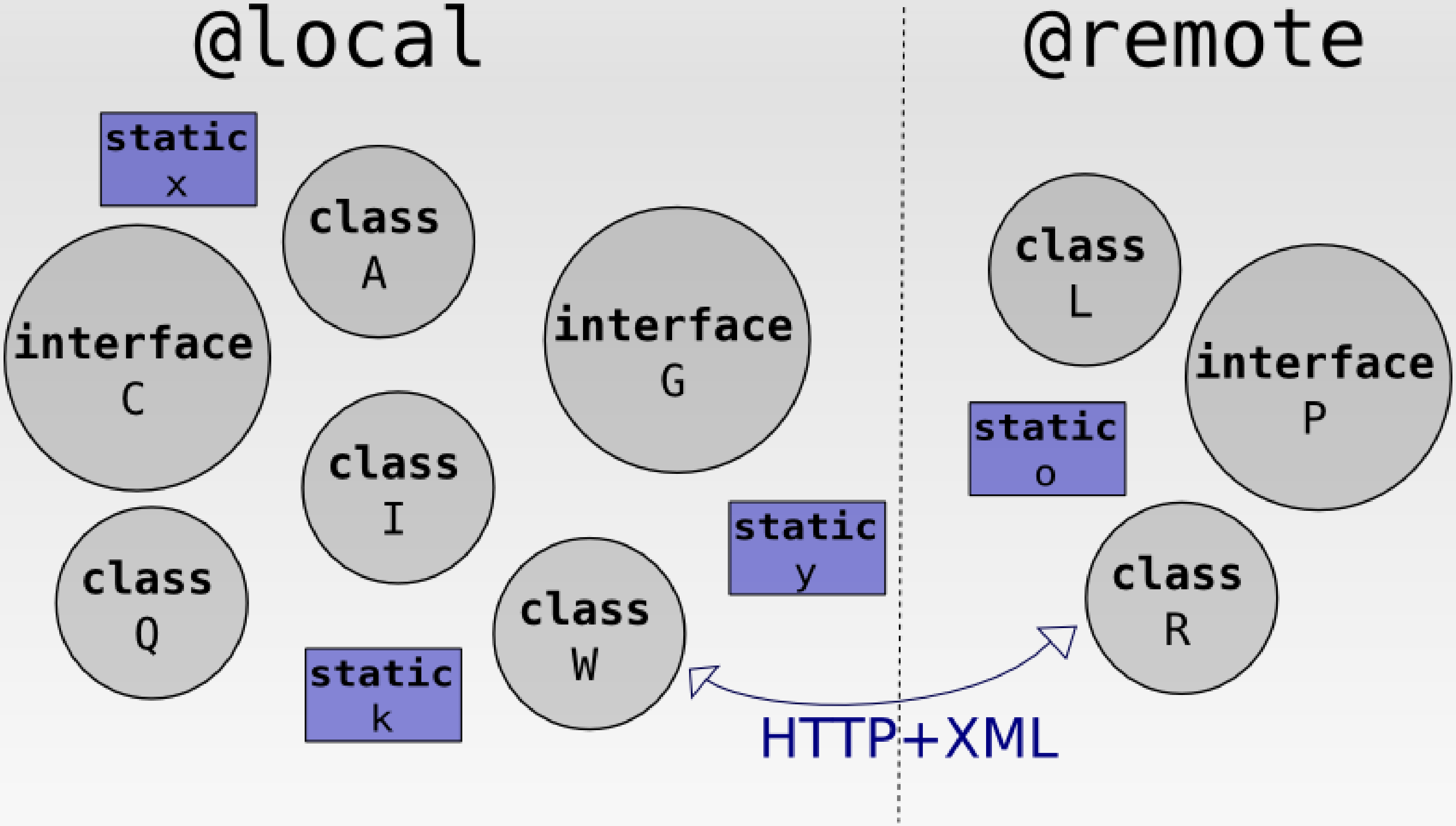
Program's Structure



All static variables must be **#stop**

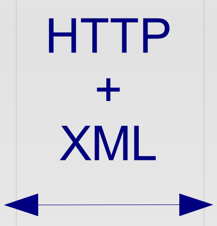
Note: basic values are all **stop** (`boolean#`**`stop`**, etc)

Program's Structure - Distribution



Program's Structure – Distribution Example

```
//client

interface Hello @“localhost:8180”

class RemoteHello @“localhost:8180”

class Main{

 main(){
    Hello newer = new RemoteHello();
    Lib.println( newer.say() );
 }

}
```



HTTP
+
XML

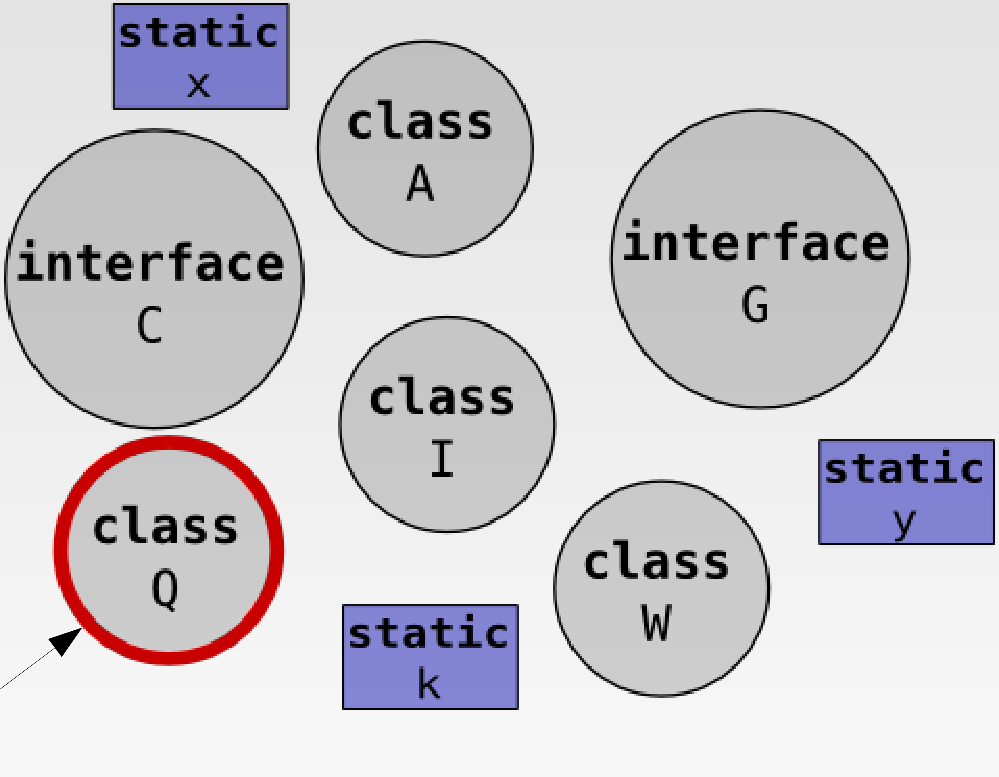
```
//server @localhost:8180


interface Hello{
    string say();
}

class RemoteHello{
    string say(){
        return “I'm remote”;
    }
}
```

REST inspired URL format:

```
(protocol)://(ip:port)/yak/Type/Instance#/Method
```

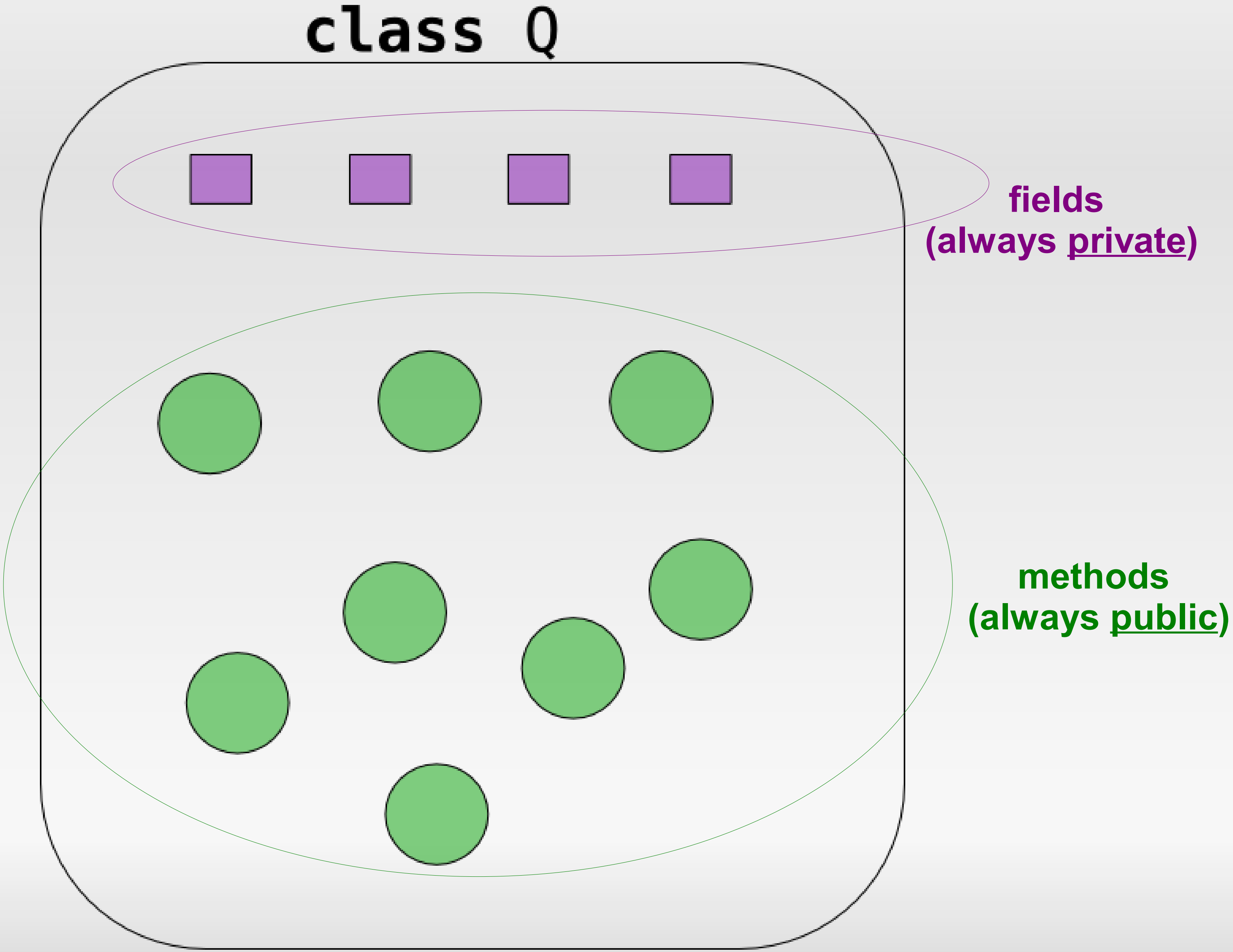
type interface: `http://localhost:8180/yak/RemoteHello`
constructor: `http://localhost:8180/yak/RemoteHello//RemoteHello`
instance: `http://localhost:8180/yak/RemoteHello/1`
method invocation: `http://localhost:8180/yak/RemoteHello/1/say`

Program's Structure

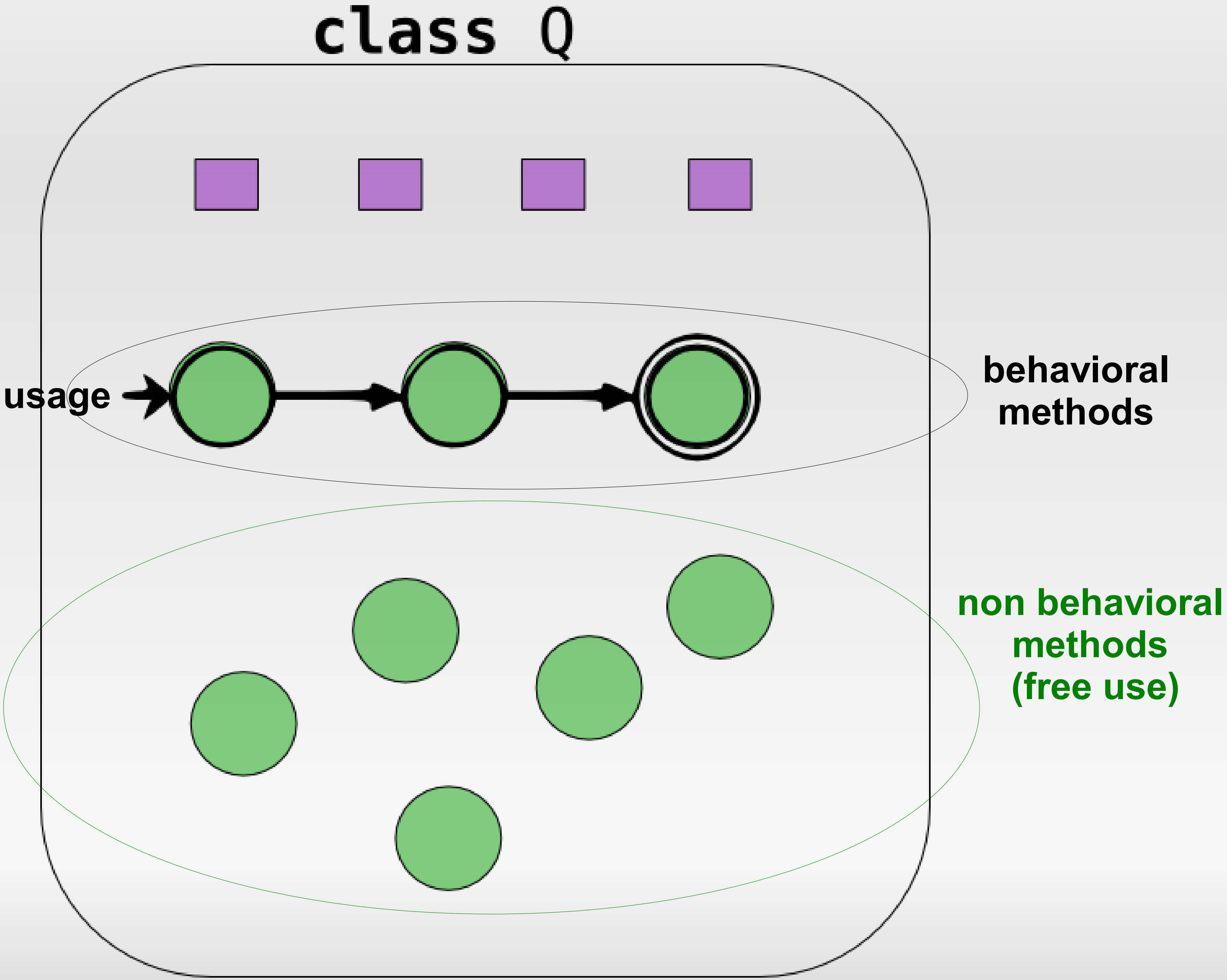


Zooming on a single class

Program's Structure – Class internals (I)



Program's Structure – Class internals (II)



Example – File interface

```
interface File{
```

usage protocol

```
    usage &start( (
       ( openRead ; read* ) +
       ( openWrite; write* ) +
       ( openReadWrite; (read+write)* )
       ; close
       )[ openRead, openWrite, openReadWrite
                -> FileNotFound: stop+(changeFile;start) |
          read, write
                -> IOException: close ] )
```

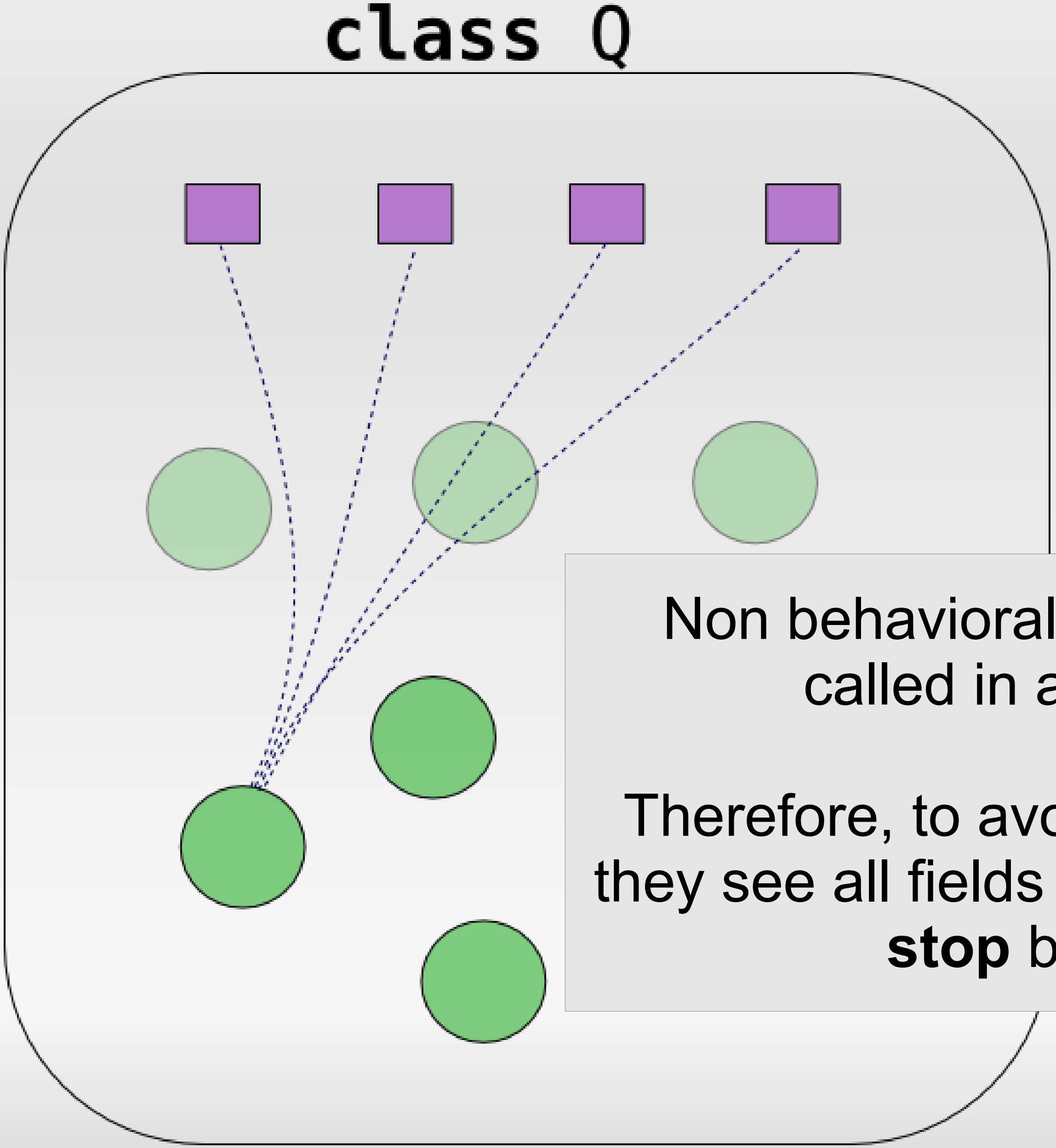
behavioral methods

```
   changeFile(string name);

   openRead() throws FileNotFound;
   openWrite() throws FileNotFound;
   openReadWrite() throws FileNotFound;

   string read() throws IOException;
   write(string content) throws IOException;
   write(string content, integer offset) throws IOException;

    close();
```

free methods

```
    integer size();
    string name();
}
```

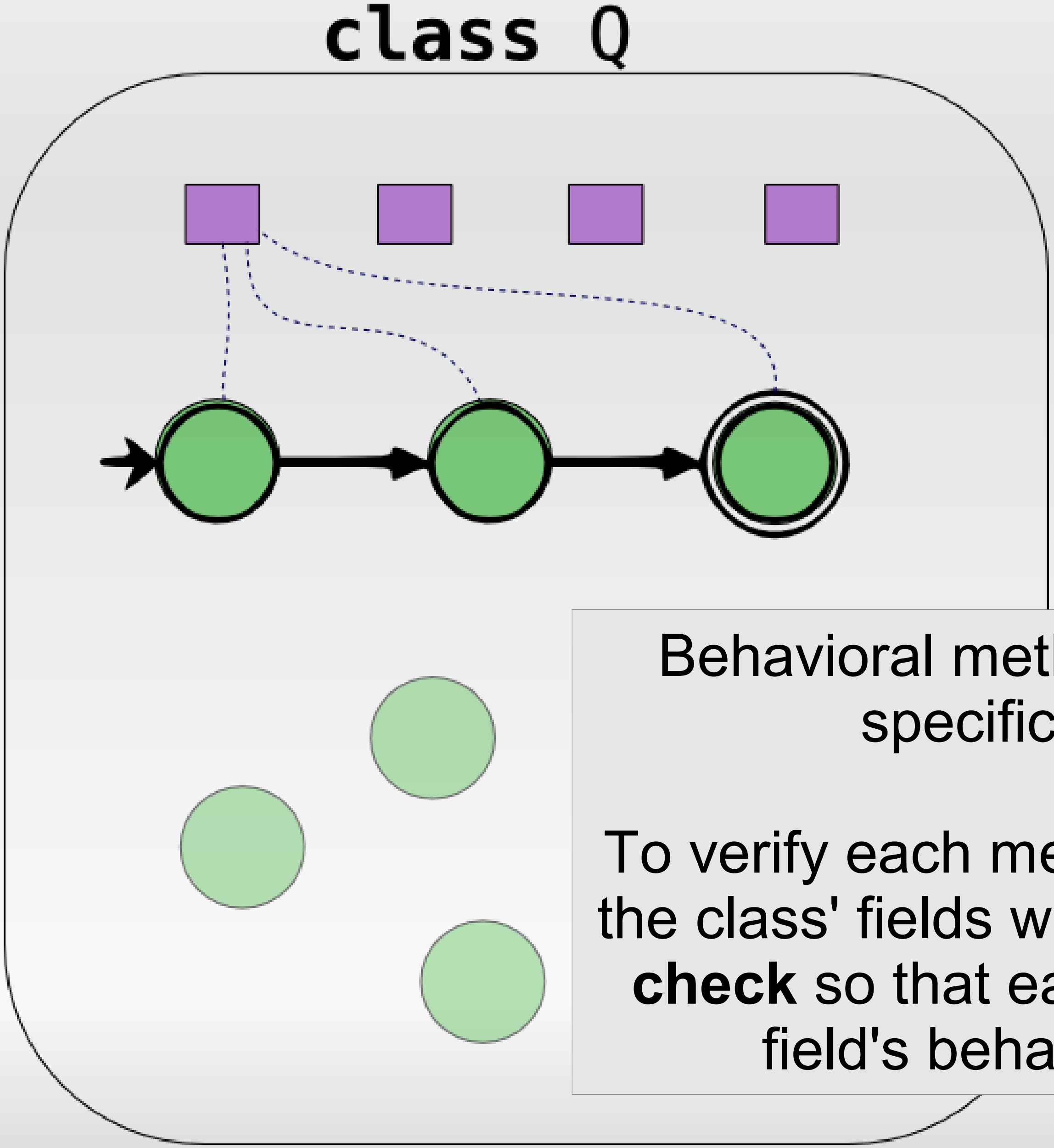
Program's Structure – Fields permissions (I)



Non behavioral methods can be called in any context.

Therefore, to avoid inconsistencies they see all fields as constants with a **stop** behavior.

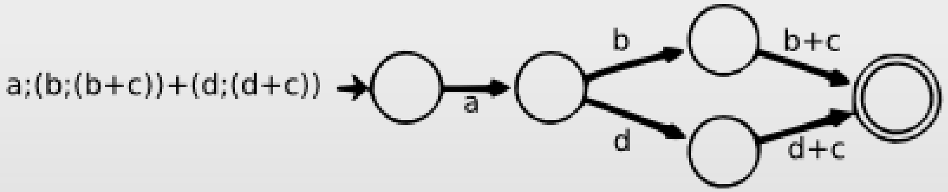
Program's Structure - Fields permissions (II)



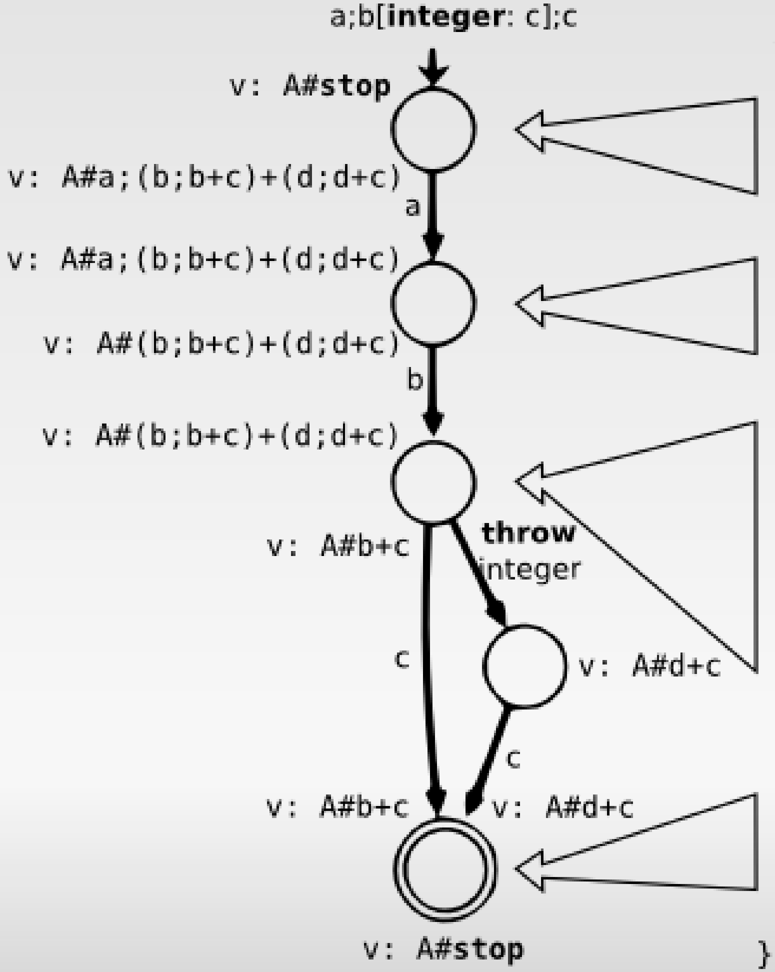
Behavioral methods are called in specific contexts.

To verify each method correctly uses the class' fields we do a **consistency check** so that each method uses a field's behavior correctly.

Consistency check

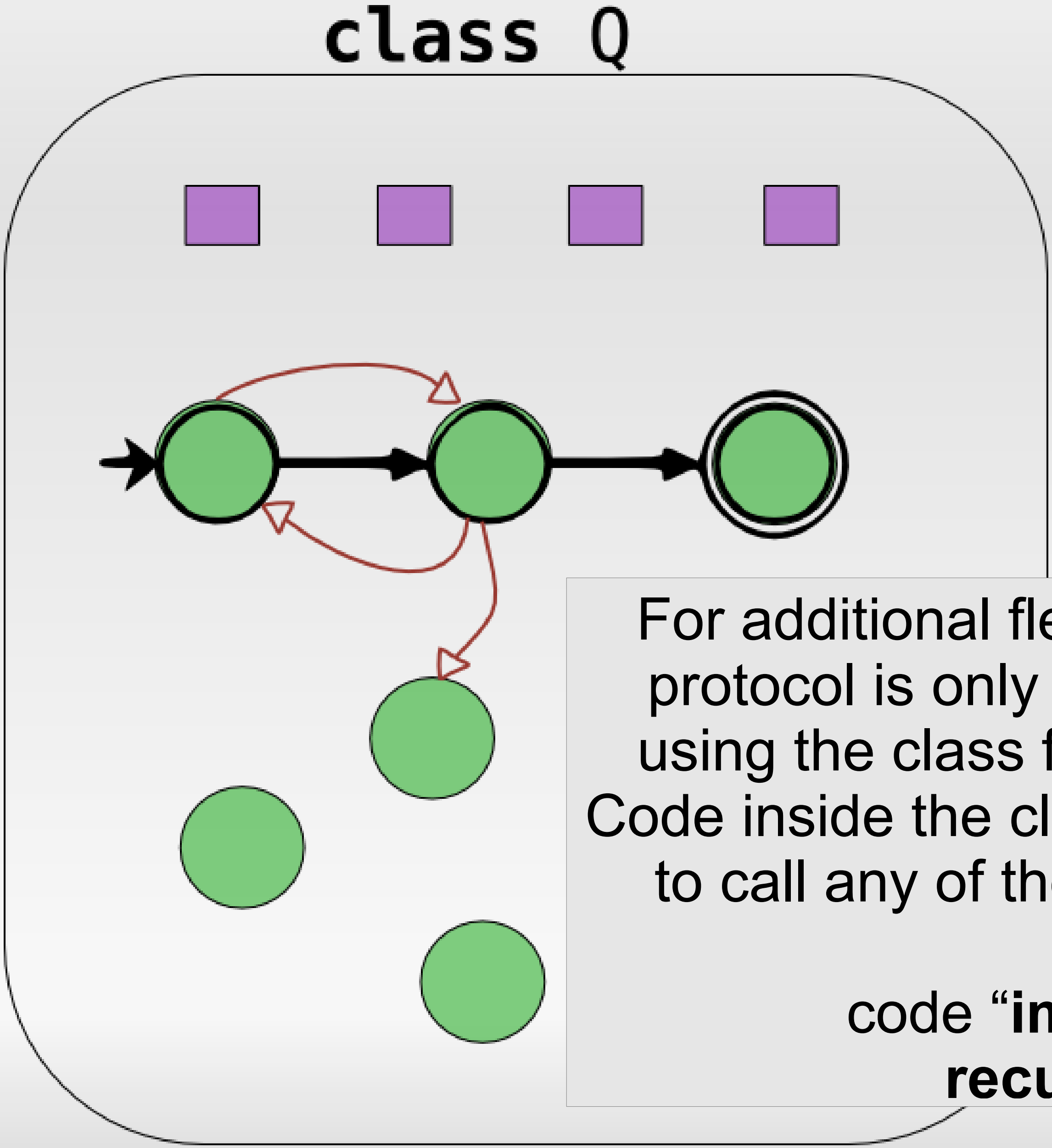


class C - Consistency Check



```
class C{
 usage a;b[integer: c];c

 A#a;(b;(b+c))+(d;(d+c)) v;

 C(){
  v = new A();
 }

 a(){
  v.a();
 }

 b() throw integer{
  if( ? ){
   v.d();
   throw 1337;
  }else{
   v.b();
  }
 }

 c(){
  v.c();
 }

}
```

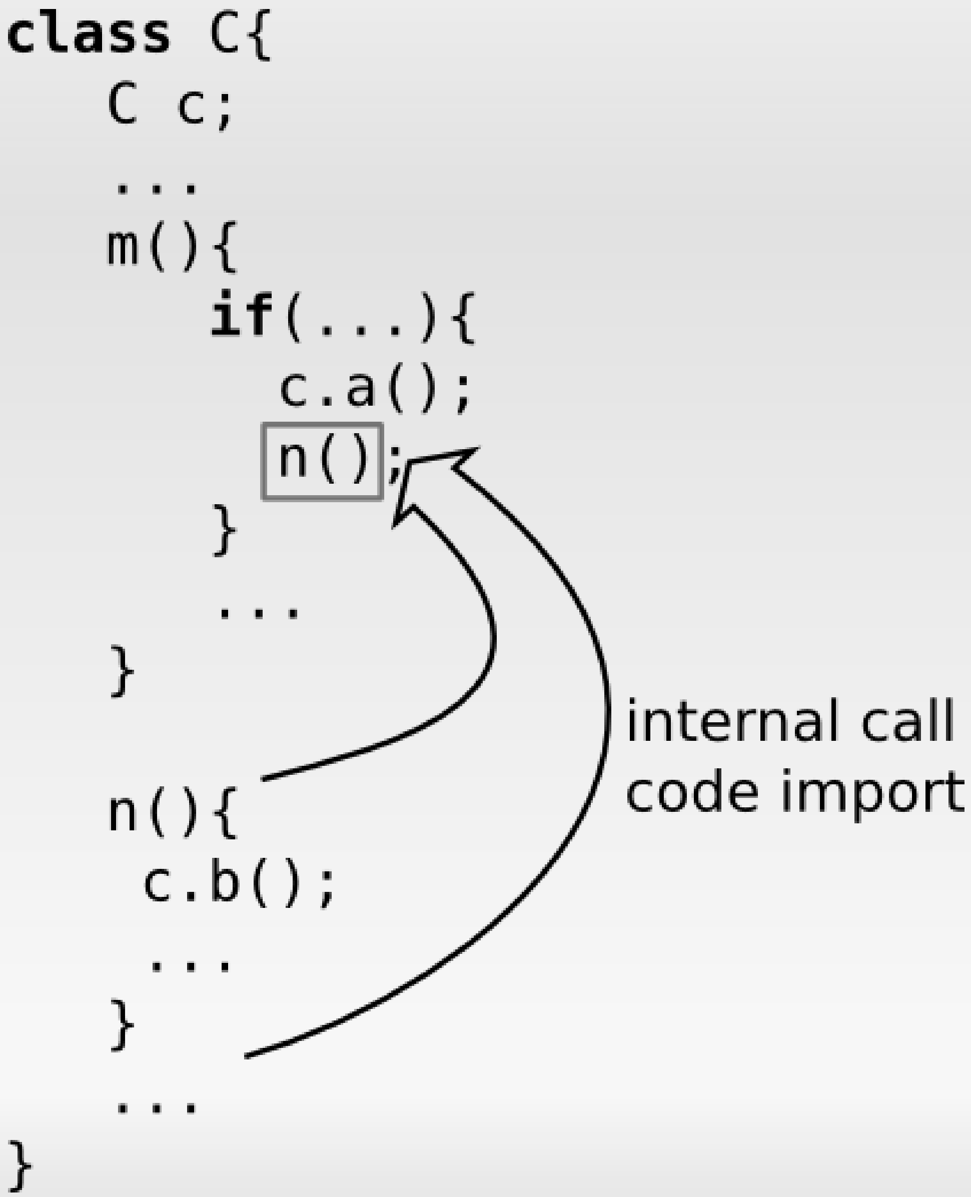
Program's Structure – Internal calls



For additional flexibility, the usage protocol is only related to anyone using the class from the “outside”. Code inside the class' body is allowed to call any of the class' methods.

code “**importing**” **recursion**

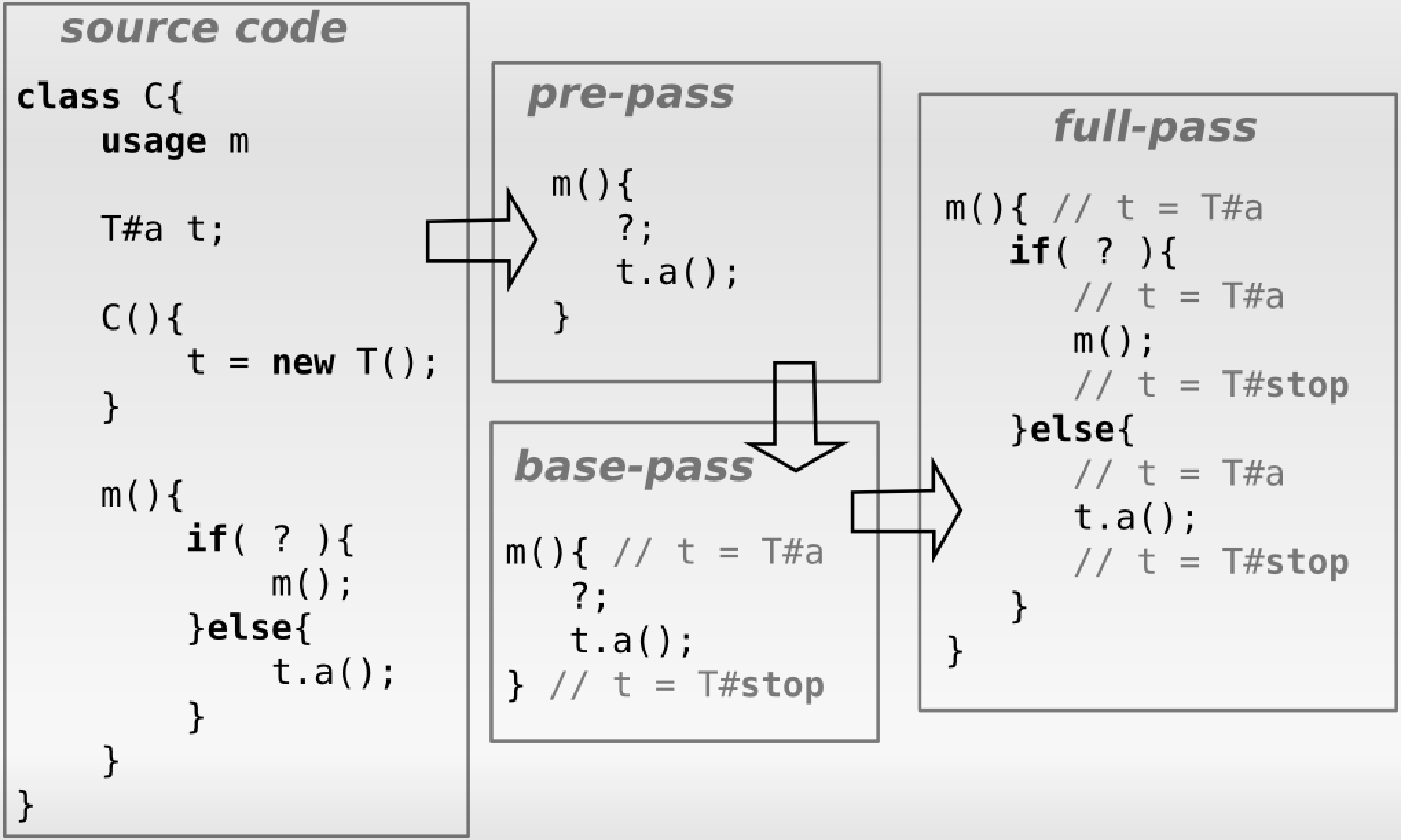
Code Import



```
class C{
    C c;
    ...
    m(){
        if(...){
          c.a();
          n();
        }
        ...
    }

    n(){
     c.b();
     ...
    }
    ...
}
```

internal call
code import

Recursion



source code

```
class C{
    usage m

    T#a t;

    C(){
        t = new T();
    }

    m(){
        if( ? ){
            m();
        }else{
            t.a();
        }
    }
}
```

pre-pass

```
m(){
   ?;
   t.a();
}
```

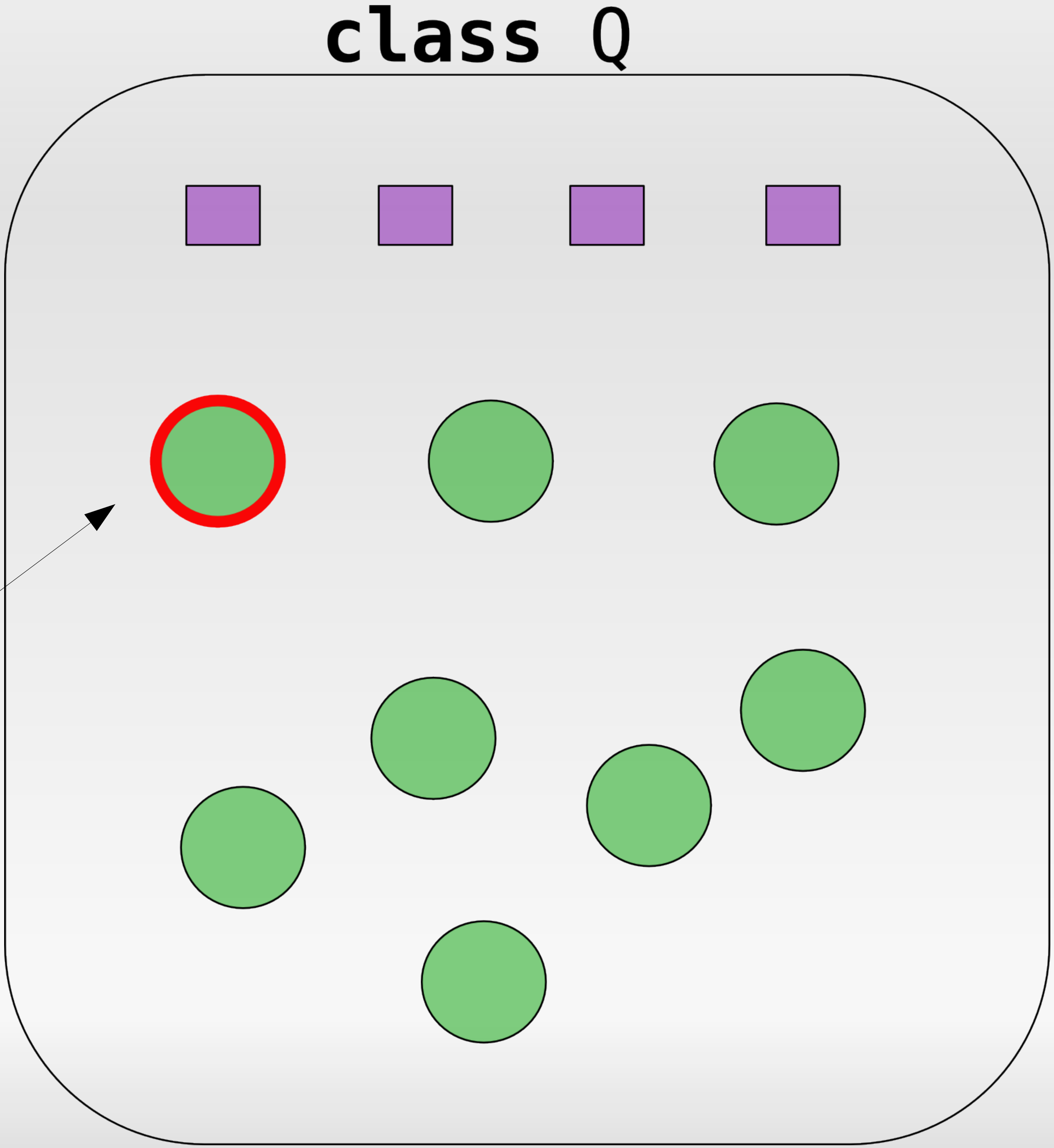
base-pass

```
m(){ // t = T#a
   ?;
   t.a();
} // t = T#stop
```

full-pass

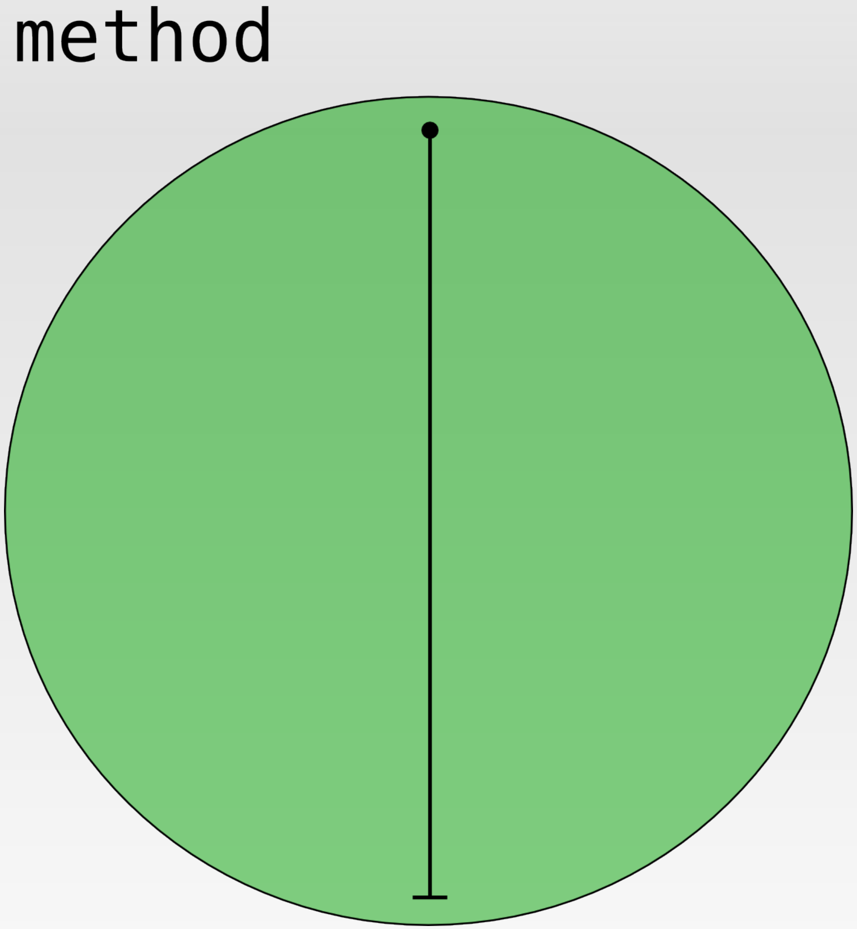
```
m(){ // t = T#a
   if( ? ){
      // t = T#a
      m();
      // t = T#stop
   }else{
      // t = T#a
      t.a();
      // t = T#stop
   }
}
```

Program's Structure



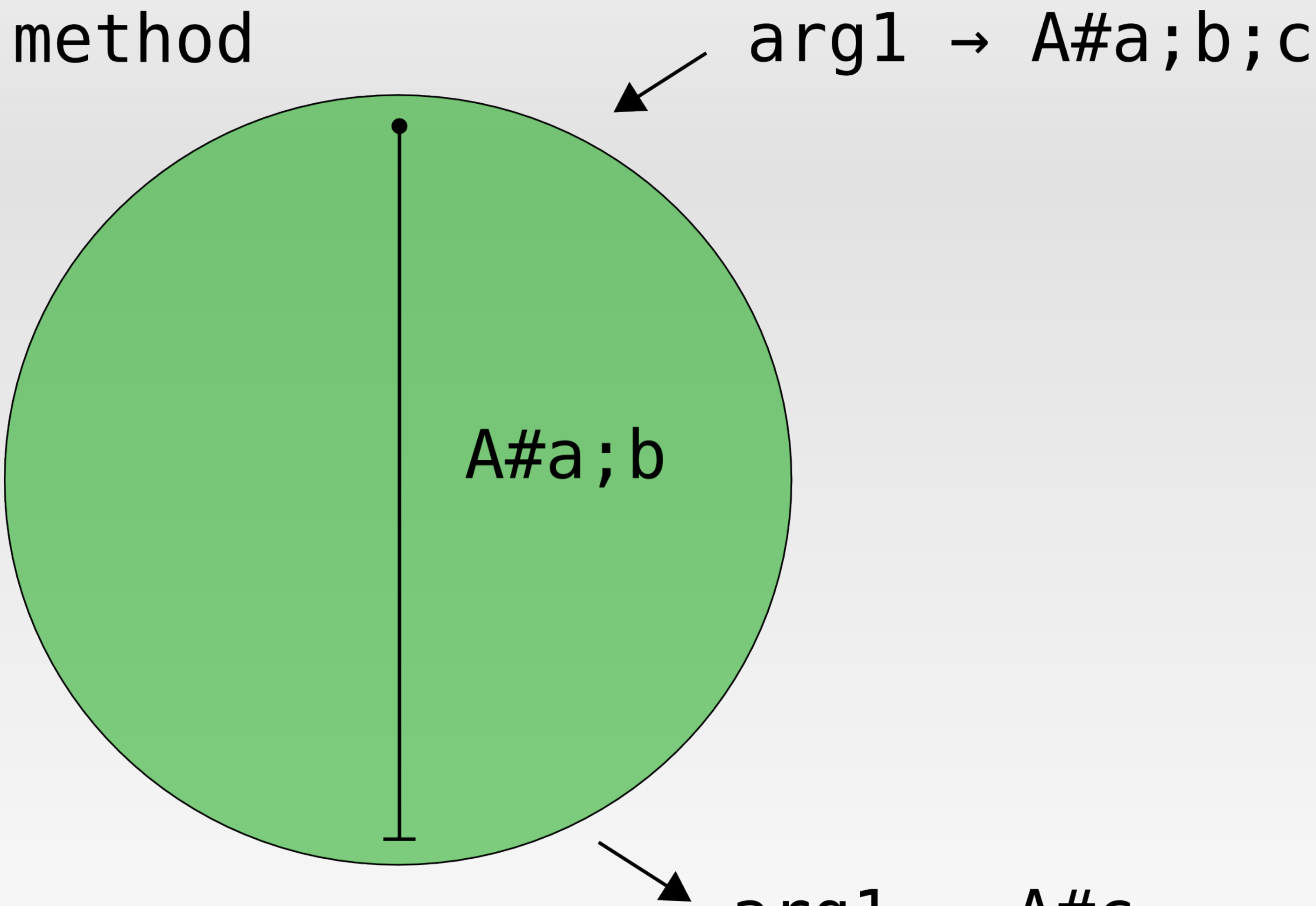
Zooming on a single method

Program's Structure



Any local variable must fulfill its behavior before the method ends.

Program's Structure



method

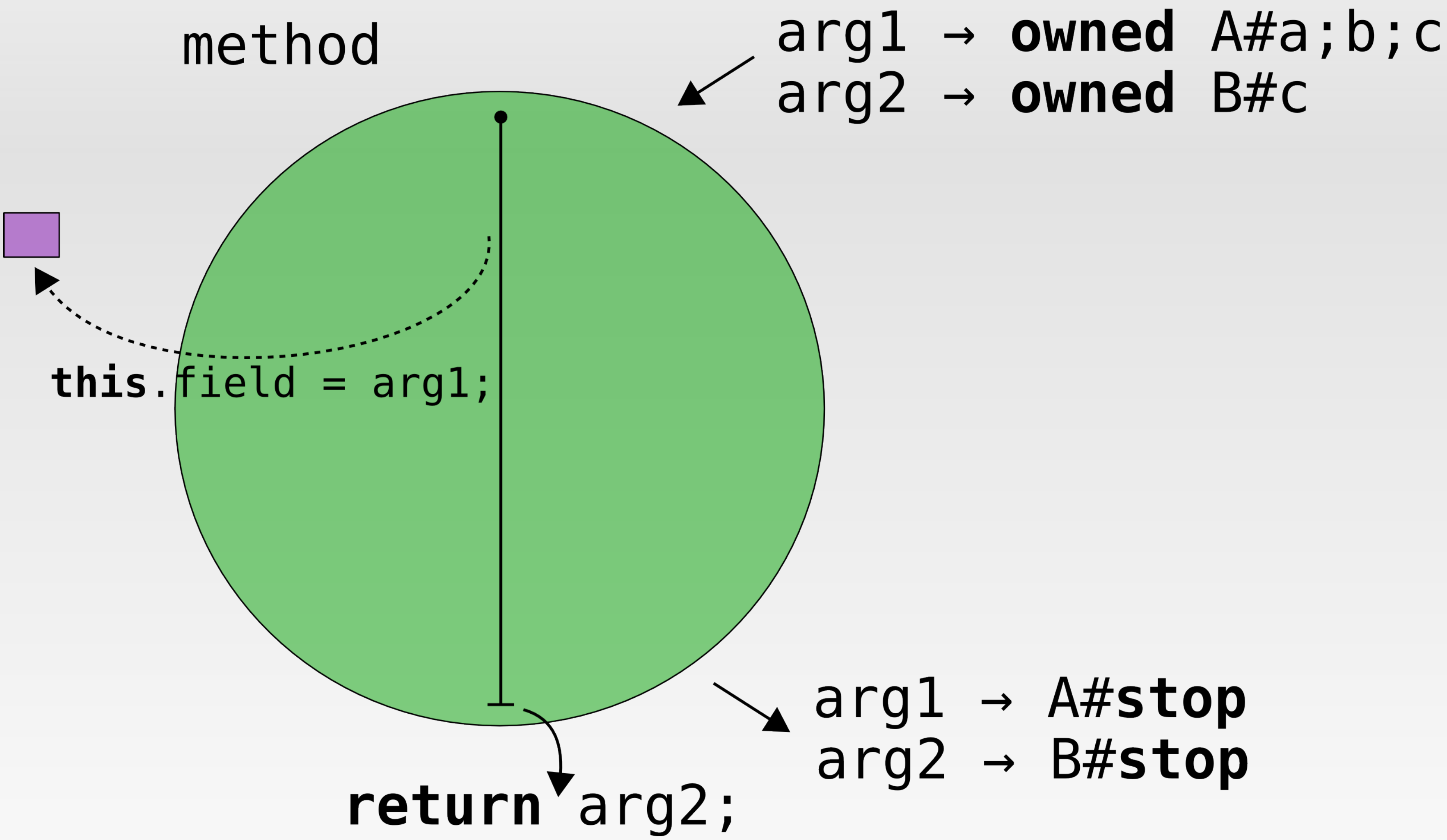
arg1 → A#a;b;c

A#a;b

arg1 → A#c

```
A a = new A(); // a -> A#a;b;c
m(a);          // a -> A#c
a.c();         // a -> A#stop
```

Program's Structure - Ownership



method

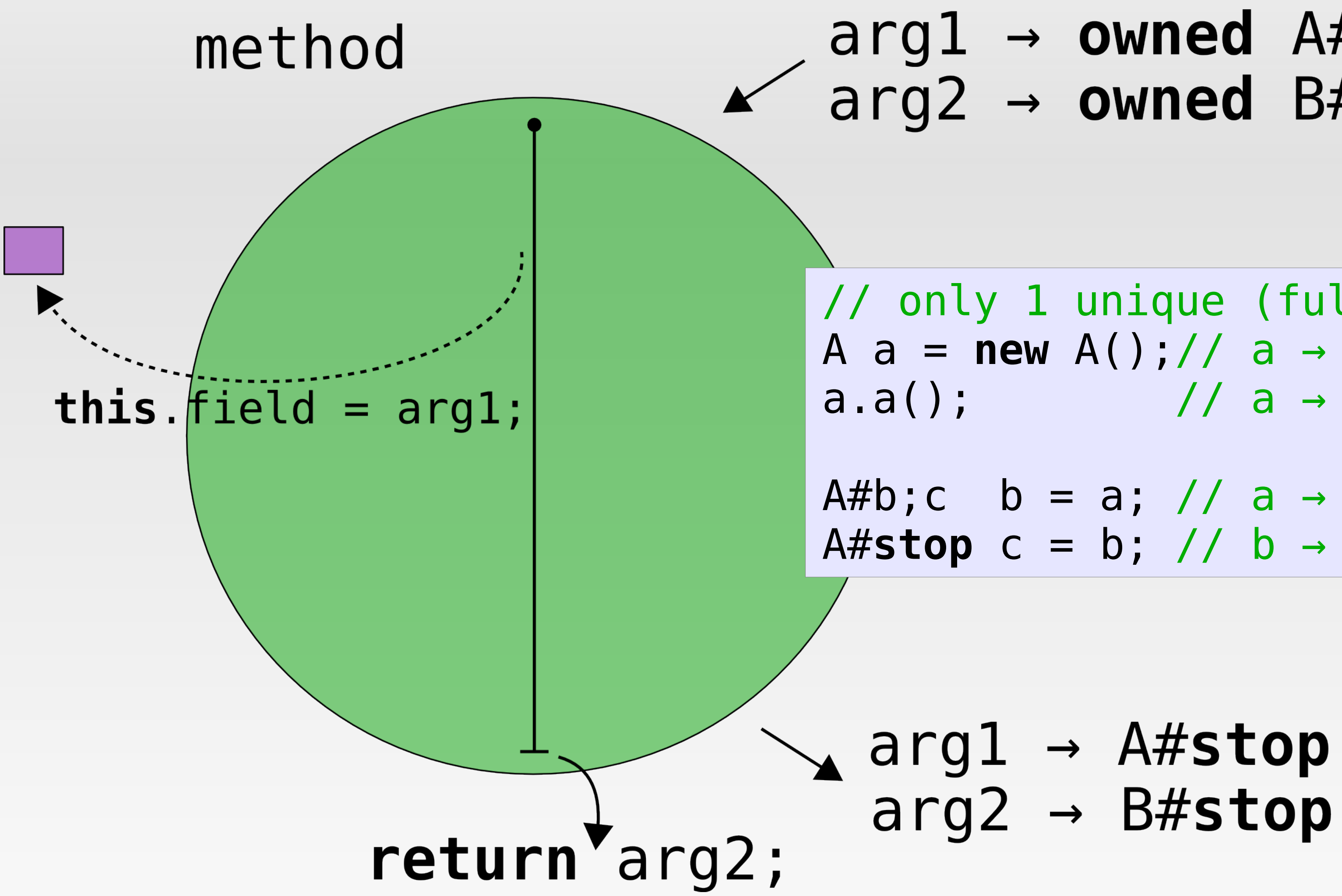
arg1 → **owned** A#a;b;c
arg2 → **owned** B#c

this.field = arg1;

arg1 → A#**stop**
arg2 → B#**stop**

return arg2;

Program's Structure - Ownership



method

arg1 → **owned** A#a;b;c
arg2 → **owned** B#c

this.field = arg1;

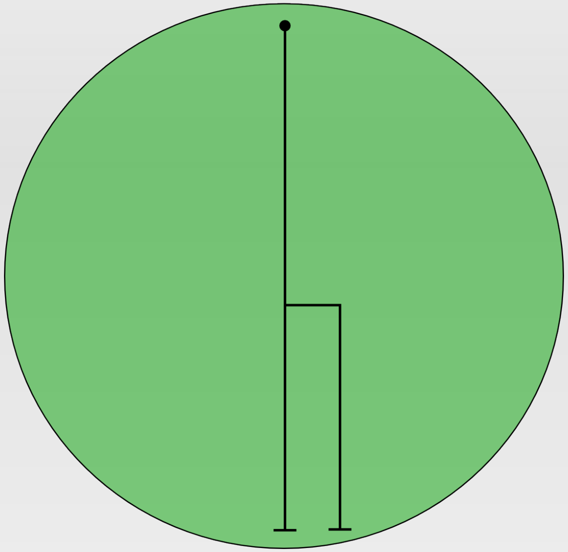
return arg2;

arg1 → A#**stop**
arg2 → B#**stop**

```
// only 1 unique (full) owner
A a = new A();// a → A#a;b;c
a.a();          // a → A#b;c

A#b;c  b = a; // a → A#stop b → A#b;c
A#stop c = b; // b → A#b;c  c → A#stop
```

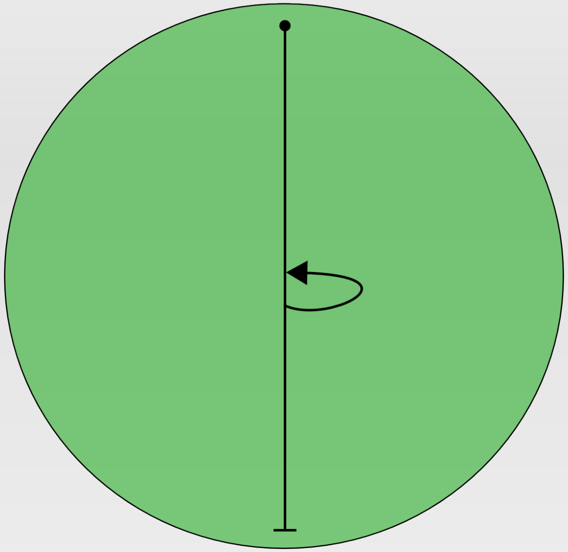
Program's Structure

method



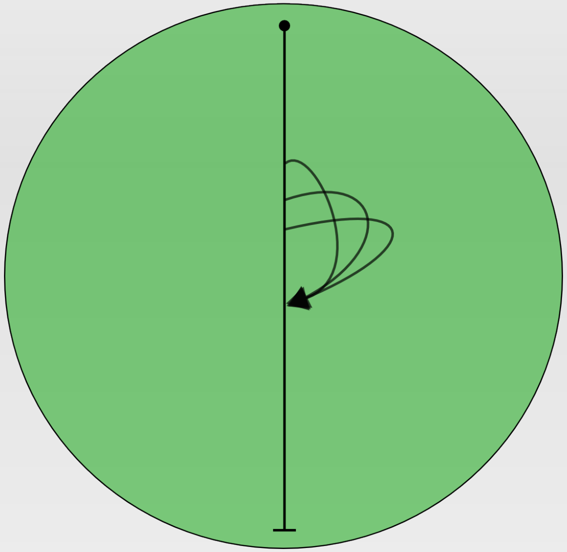
if-else

method



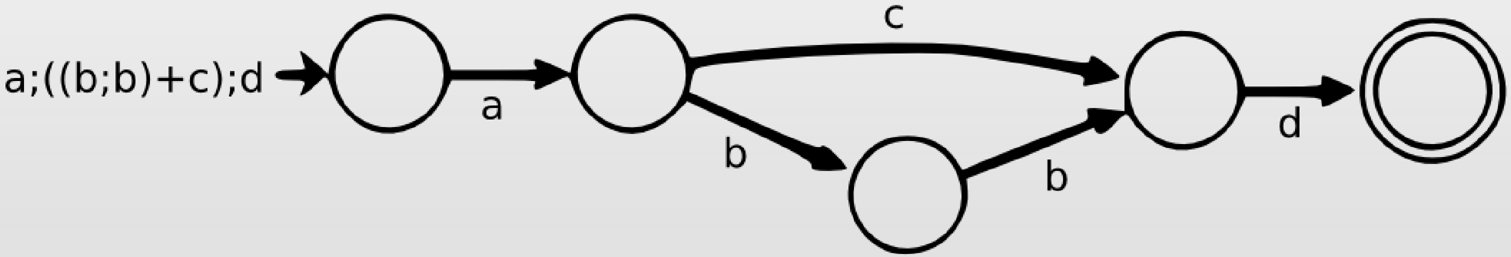
while/repeat

method



exceptions
(try-catch + throw)

Exceptions (try-catch + throw)

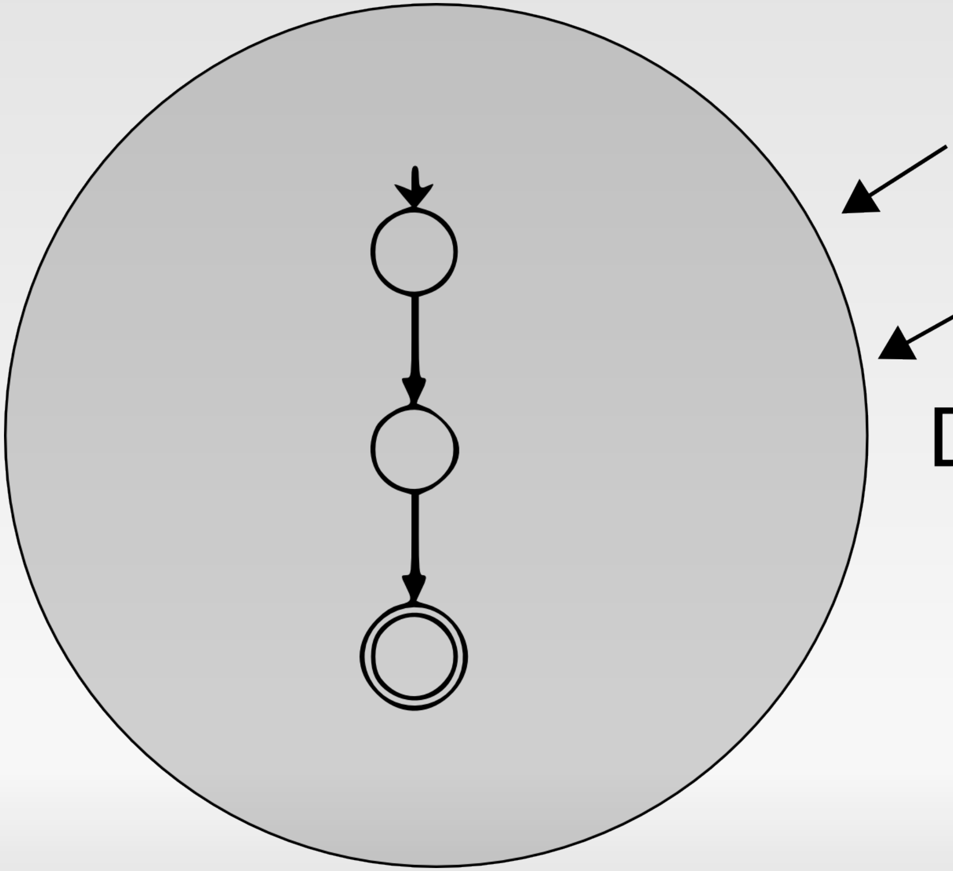


```
                     m(){
                      A#a;(b;b)+c;d v = new A();

                      try{
                       v.a();
                       if( ? ){
                        v.b();
            v: A#b;d    throw 42;
                       };
                       v.c();   v: A#d
                      }
                      catch(integer e){
                       v.b();   v: A#d
                      };

                      v.d();
 v: A#stop           }
```

INTERSECTION

Subtyping

Replacing a behavioral type with another, while still obeying behavioral expectations



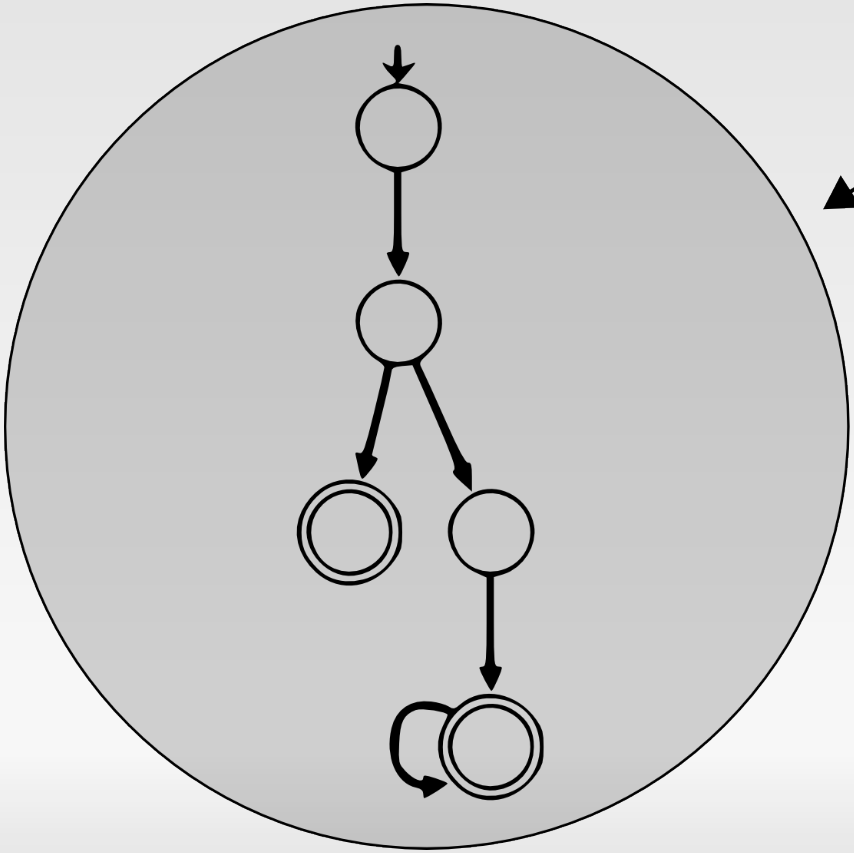
call A

call B

Done.

Subtyping

Replacing a behavioral type with another, while still obeying behavioral expectations



call A

call B

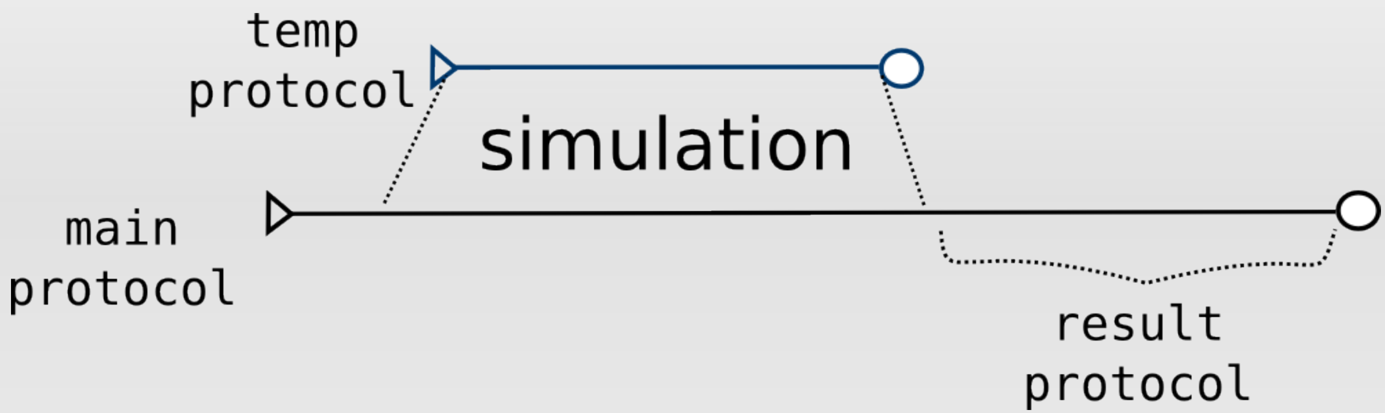
Done.

Example - Order

```
interface Order{
    usage review*;buy?
    /* ... */
}

class TravelOrder{
    usage (packageAlaska+packageArtic)[SoldOut: stop]+
          (flight;hotel); (review*; buy?)
    /* ... */
}

class HotelOrder{
    usage bookGroup+bookPenthouse+bookRoom* ;
        breakfast? ; dinner? ; (review*; buy?)
    /* ... */
}


class User{
    map<Order> orders;
    /* ... */
}
```

Can hold HotelOrders or TravelOrders as long as their only remaining behavior is (review*;buy?)
missing (subtype-wise) behavioral methods will never be called
(can't be used anymore)

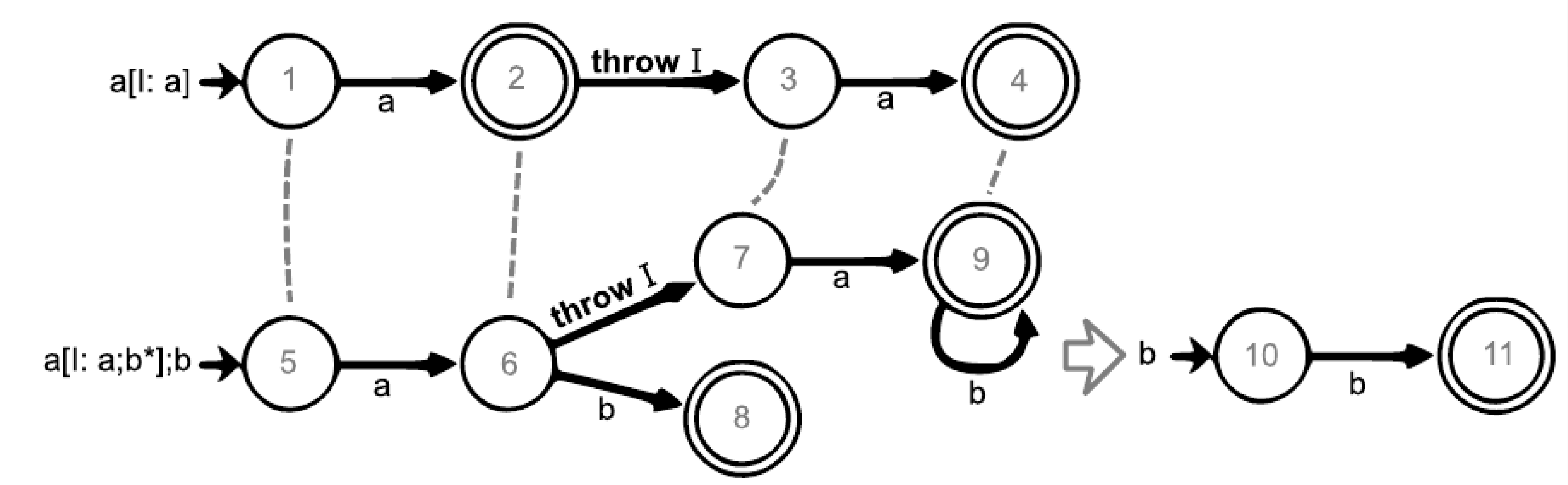
Protocol Simulation (I)



- choices:
  “main” more / “temp” less
  → *Hidden choices in “temp”*

- exceptions:
  “main” less / “temp” more
  → *«Useless» catches*

Protocol Simulation (II)



Type System

- Simplified syntax (no “syntax sugar”)
- Only the core features of the language
- Basic typing judgment:

(typing judgment)

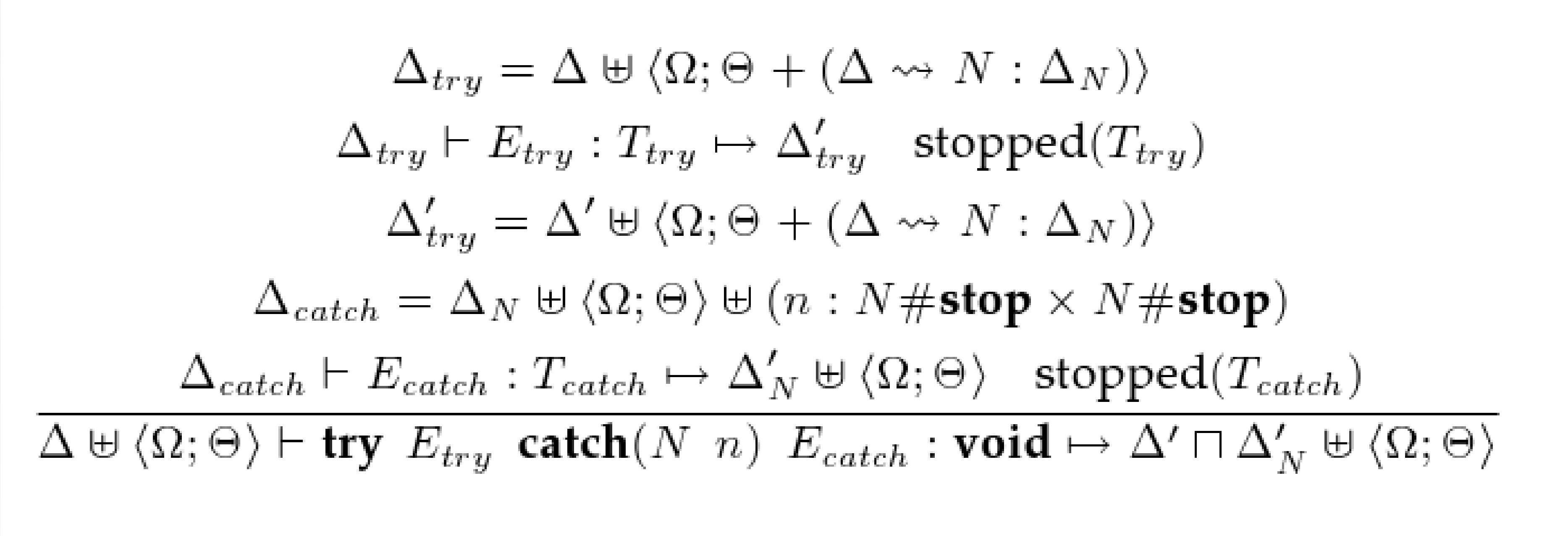
$$\Delta_{before} \vdash E : T_{result} \mapsto \Delta_{after}$$

Type System – if else

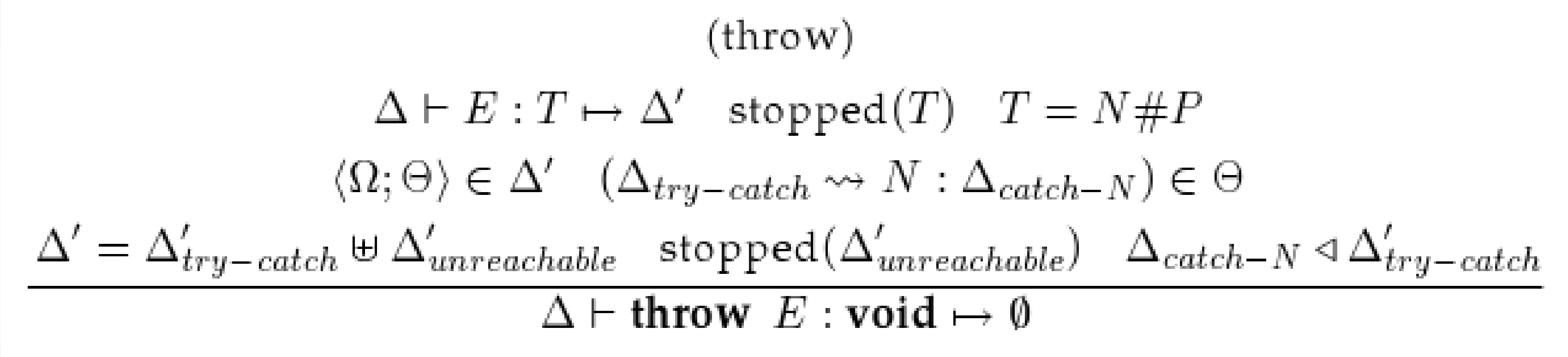
$$\text{(if else)}$$

$$\frac{\Delta \vdash E^{cond} : \mathbf{boolean} \mapsto \Delta_{cond} \quad \Delta_{cond} \vdash E^{if} : T \mapsto \Delta_{if} \quad \Delta_{cond} \vdash E^{else} : T \mapsto \Delta_{else}}{\Delta \vdash \mathbf{if}(E^{cond})\ E^{if}\ \mathbf{else}\ E^{else} : T \mapsto \Delta_{if} \sqcap \Delta_{else}}$$

Type System – try-catch and throw



$$\frac{\begin{array}{c} \Delta_{try} = \Delta \uplus \langle \Omega; \Theta + (\Delta \leadsto N : \Delta_N) \rangle \\ \Delta_{try} \vdash E_{try} : T_{try} \mapsto \Delta'_{try} \quad \mathsf{stopped}(T_{try}) \\ \Delta'_{try} = \Delta' \uplus \langle \Omega; \Theta + (\Delta \leadsto N : \Delta_N) \rangle \\ \Delta_{catch} = \Delta_N \uplus \langle \Omega; \Theta \rangle \uplus (n : N\#\mathbf{stop} \times N\#\mathbf{stop}) \\ \Delta_{catch} \vdash E_{catch} : T_{catch} \mapsto \Delta'_N \uplus \langle \Omega; \Theta \rangle \quad \mathsf{stopped}(T_{catch}) \end{array}}{\Delta \uplus \langle \Omega; \Theta \rangle \vdash \mathbf{try}\ \ E_{try}\ \ \mathbf{catch}(N\ \ n)\ \ E_{catch} : \mathbf{void} \mapsto \Delta' \sqcap \Delta'_N \uplus \langle \Omega; \Theta \rangle}$$



(throw)

$$\frac{\begin{array}{c} \Delta \vdash E : T \mapsto \Delta' \quad \mathsf{stopped}(T) \quad T = N\#P \\ \langle \Omega; \Theta \rangle \in \Delta' \quad (\Delta_{try-catch} \leadsto N : \Delta_{catch-N}) \in \Theta \\ \Delta' = \Delta'_{try-catch} \uplus \Delta'_{unreachable} \quad \mathsf{stopped}(\Delta'_{unreachable}) \quad \Delta_{catch-N} \lhd \Delta'_{try-catch} \end{array}}{\Delta \vdash \mathbf{throw}\ \ E : \mathbf{void} \mapsto \emptyset}$$

Part 3 – Closing Points (~3 min)

- Related Work
- Conclusions
- Future Work

Related Work (I)

Behavioral verification is a very broad topic.

There are several different approaches to the same core problem.

A quick overview of some of the most closely related work...

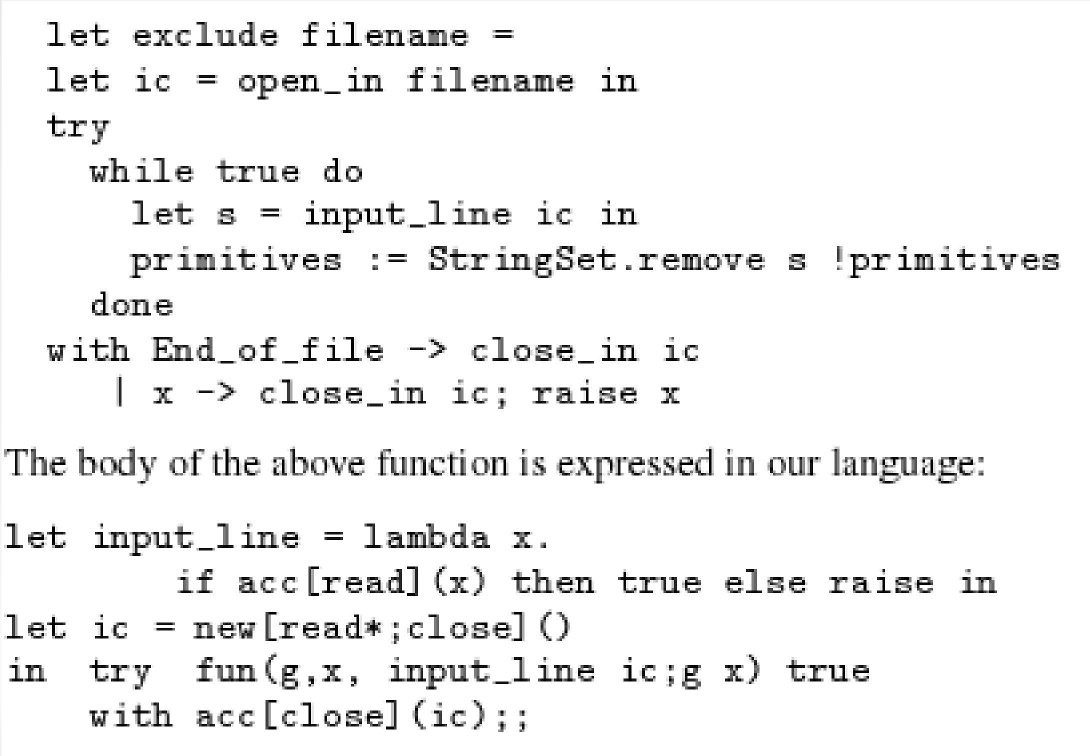
Related Work (II)

- *Atsushi Igarashi* and *Naoki Kobayashi*.
  Resource usage analysis. 2002.

- *Futoshi Iwama*, *Atsushi Igarashi*, and *Naoki Kobayashi*.
  Resource usage analysis for a functional language with exceptions. 2006.

+ complex protocol expressiveness (even tough somewhat confusing)
+ concurrency
+ (some) exception handling

- ML based (functional language)
- no practical algorithms for checking (only formal type system)



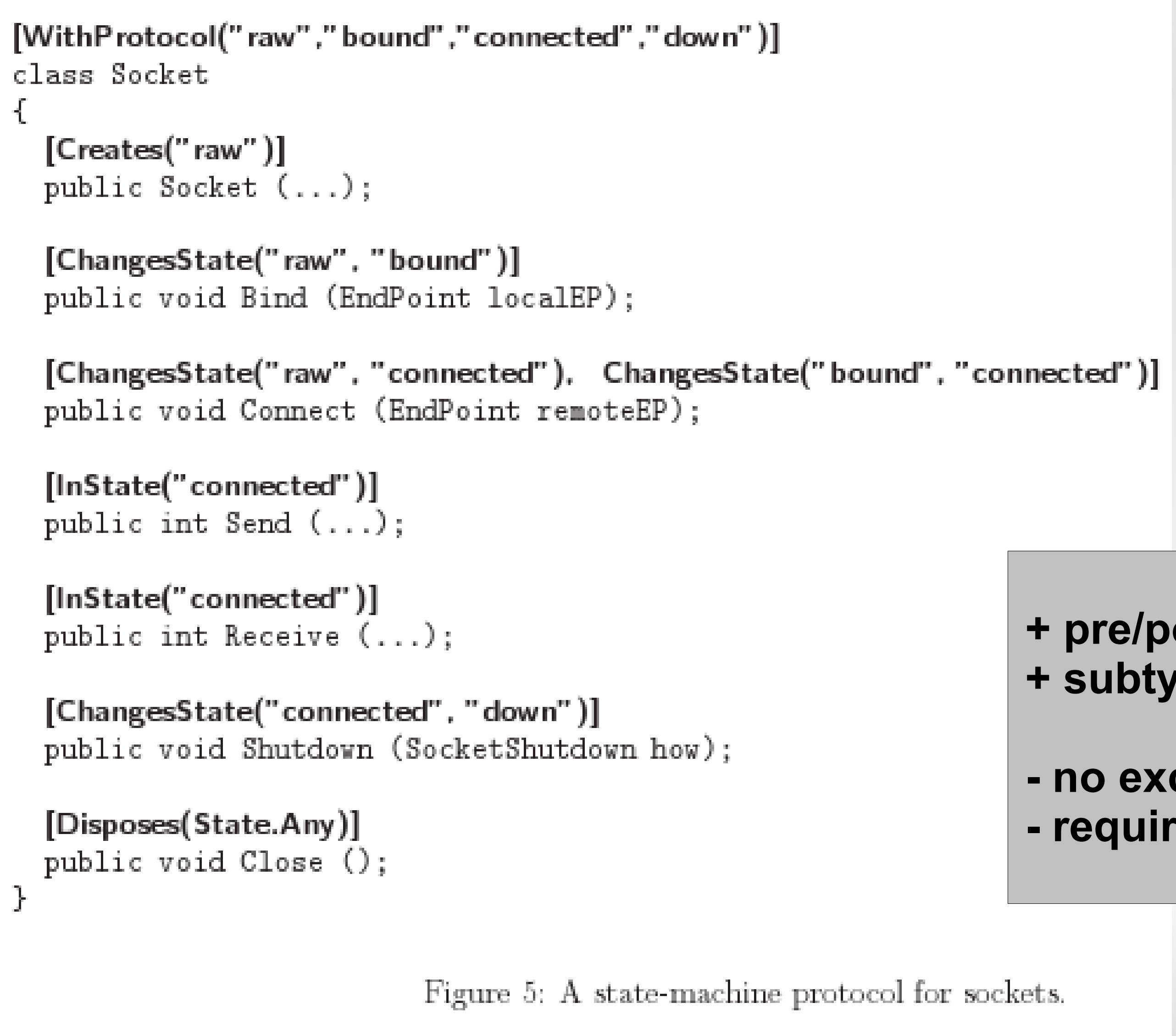
```
let exclude filename =
let ic = open_in filename in
try
  while true do
    let s = input_line ic in
    primitives := StringSet.remove s !primitives
  done
with End_of_file -> close_in ic
   | x -> close_in ic; raise x
```

The body of the above function is expressed in our language:

```
let input_line = lambda x.
        if acc[read](x) then true else raise in
let ic = new[read*;close]()
in  try  fun(g,x, input_line ic;g x) true
    with acc[close](ic);;
```

Related Work (III)

- *R. DeLine* and *M. Fahndrich*.
  The fugue protocol checker: Is your software baroque. 2003.



```
[WithProtocol("raw","bound","connected","down")]
class Socket
{
  [Creates("raw")]
  public Socket (...);

  [ChangesState("raw", "bound")]
  public void Bind (EndPoint localEP);

  [ChangesState("raw", "connected"),  ChangesState("bound", "connected")]
  public void Connect (EndPoint remoteEP);

  [InState("connected")]
  public int Send (...);

  [InState("connected")]
  public int Receive (...);

  [ChangesState("connected", "down")]
  public void Shutdown (SocketShutdown how);

  [Disposes(State.Any)]
  public void Close ();
}
```

Figure 5: A state-machine protocol for sockets.

+ pre/post + state-machine
+ subtyping and parameter check

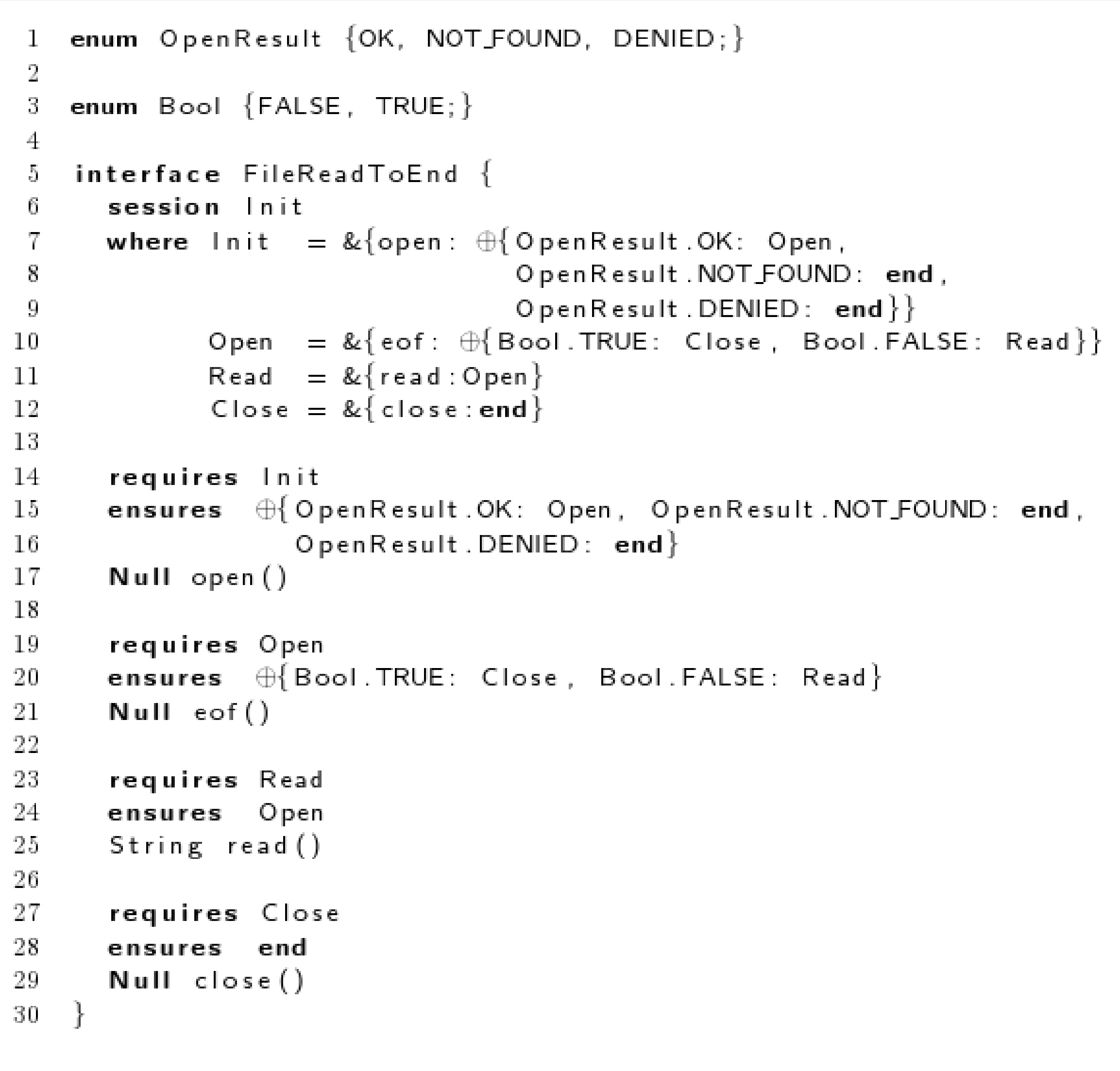
- no exception handling
- requires extensive annotations

Related Work (IV)

- *Simon Gay*, *Vasco T. Vasconcelos*, and *Antonio Ravara*. **Dynamic interfaces**. 2007.

+ pre/post + session-types
+ inheritance and subtyping

- more limited approach:
no self calls
no behavioral termination
no behavioral exceptions
no ownership

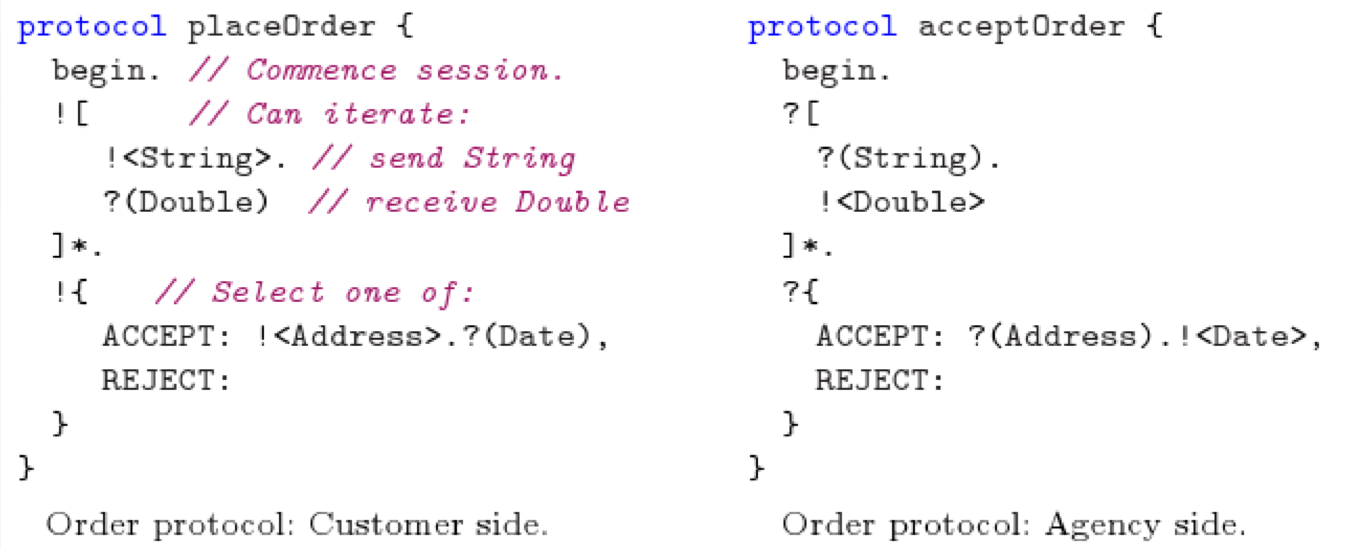


```
enum OpenResult {OK, NOT_FOUND, DENIED;}

enum Bool {FALSE, TRUE;}

interface FileReadToEnd {
  session Init
  where Init  = &{open: ⊕{OpenResult.OK: Open,
                           OpenResult.NOT_FOUND: end,
                           OpenResult.DENIED: end}}
        Open  = &{eof: ⊕{Bool.TRUE: Close, Bool.FALSE: Read}}
        Read  = &{read:Open}
        Close = &{close:end}

  requires Init
  ensures  ⊕{OpenResult.OK: Open, OpenResult.NOT_FOUND: end,
             OpenResult.DENIED: end}
  Null open()

  requires Open
  ensures  ⊕{Bool.TRUE: Close, Bool.FALSE: Read}
  Null eof()

  requires Read
  ensures  Open
  String read()

  requires Close
  ensures  end
  Null close()
}
```

Fig. 1. The interface of a file that must be read to the end-of-file.

Related Work (V)

- Raymond Hu, Nobuko Yoshida, and Kohei Honda.
  Session-based distributed programming in java. 2008.

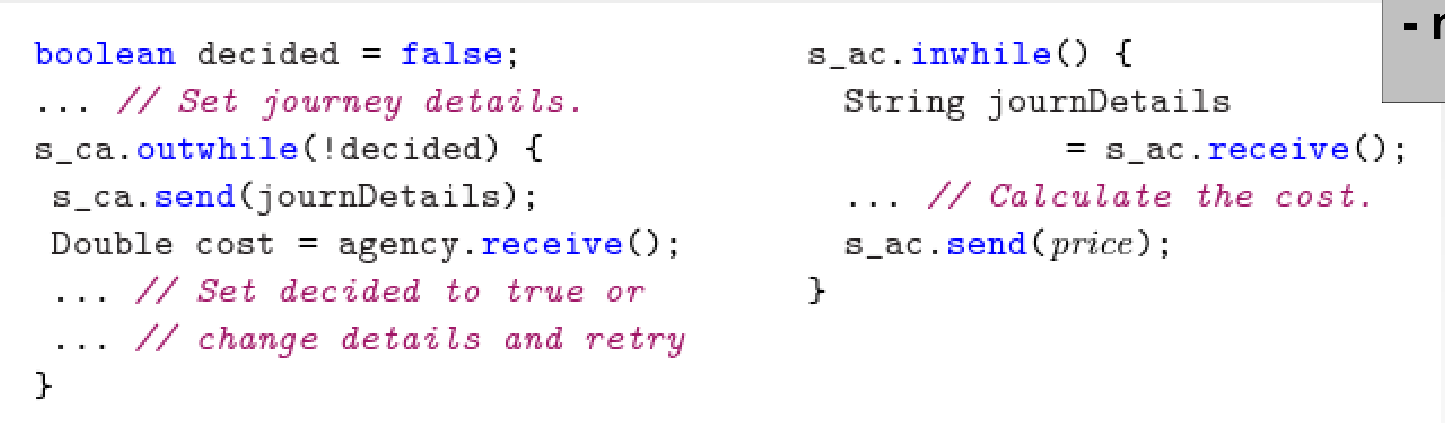


```
protocol placeOrder {
  begin.  // Commence session.
  ![      // Can iterate:
     !<String>. // send String
     ?(Double)  // receive Double
  ]*.
  !{    // Select one of:
     ACCEPT: !<Address>.?(Date),
     REJECT:
  }
}
```

Order protocol: Customer side.

```
protocol acceptOrder {
  begin.
  ?[
    ?(String).
    !<Double>
  ]*.
  ?{
    ACCEPT: ?(Address).!<Date>,
    REJECT:
  }
}
```

Order protocol: Agency side.



```
boolean decided = false;
... // Set journey details.
s_ca.outwhile(!decided) {
 s_ca.send(journDetails);
 Double cost = agency.receive();
 ... // Set decided to true or
 ... // change details and retry
}
```

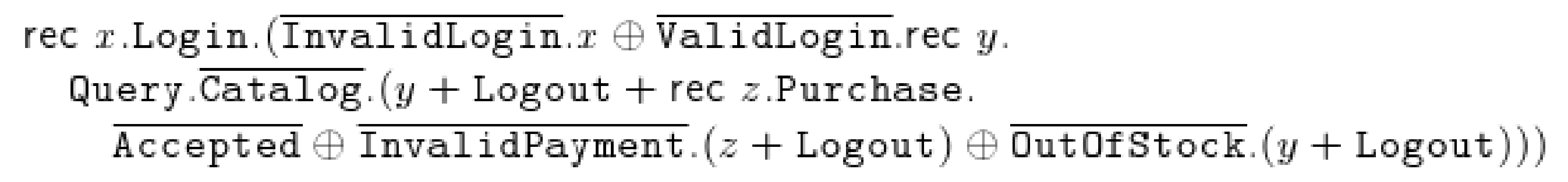
```
s_ac.inwhile() {
  String journDetails
              = s_ac.receive();
  ... // Calculate the cost.
  s_ac.send(price);
}
```

+ session-type based

- focused on channel communication (only)
- complex syntax
- not language transparent
- pairwise composition of protocols
- no behavioral exceptions

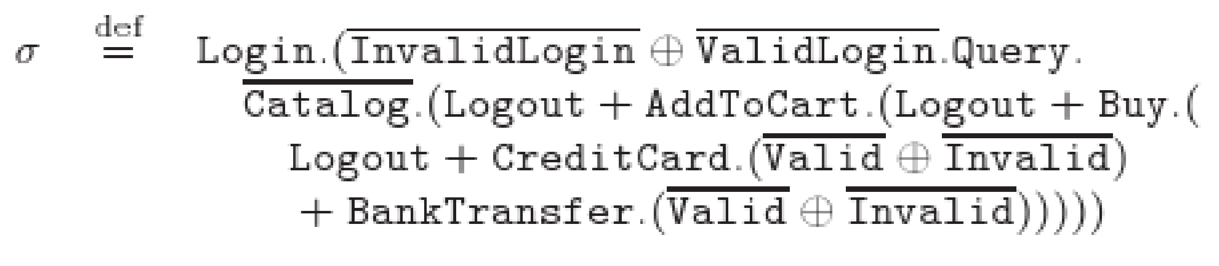
Related Work (VI)

- *Cosimo Laneve* and *Luca Padovani*.
  The must preorder revisited. 2007.



$$\begin{aligned}
&\mathtt{rec}\ x.\mathtt{Login}.(\overline{\mathtt{InvalidLogin}}.x \oplus \overline{\mathtt{ValidLogin}}.\mathtt{rec}\ y.\\
&\quad \mathtt{Query}.\overline{\mathtt{Catalog}}.(y + \mathtt{Logout} + \mathtt{rec}\ z.\mathtt{Purchase}.\\
&\qquad \overline{\mathtt{Accepted}} \oplus \overline{\mathtt{InvalidPayment}}.(z + \mathtt{Logout}) \oplus \overline{\mathtt{OutOfStock}}.(y + \mathtt{Logout})))
\end{aligned}$$

- *Giuseppe Castagna*, *Nils Gesbert*, and *Luca Padovani*.
  A theory of contracts for web services. 2008.



$$\begin{aligned}
\sigma \stackrel{\text{def}}{=}\ & \mathtt{Login}.(\overline{\mathtt{InvalidLogin}} \oplus \overline{\mathtt{ValidLogin}}.\mathtt{Query}.\\
&\quad \overline{\mathtt{Catalog}}.(\mathtt{Logout} + \mathtt{AddToCart}.(\mathtt{Logout} + \mathtt{Buy}.(\\
&\qquad \mathtt{Logout} + \mathtt{CreditCard}.(\overline{\mathtt{Valid}} \oplus \overline{\mathtt{Invalid}})\\
&\qquad + \mathtt{BankTransfer}.(\overline{\mathtt{Valid}} \oplus \overline{\mathtt{Invalid}})))))
\end{aligned}$$

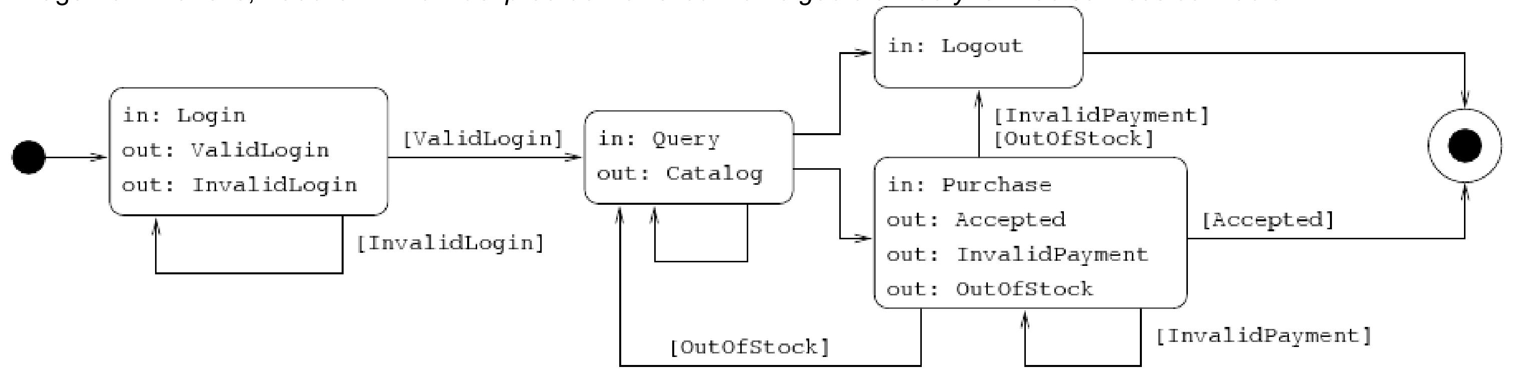
- (only) focused on the contract layer

+ flexible and interesting operations

+ sub-contract very similar to our behavioral sub-typing

Example – WS-CDL (wrapper)

Image from: Laneve, Padovani. *The must preorder revisited – an algebraic theory for web services contracts*



```
class Service{
 usage &l( login [ InvalidLogin: l ] ;
   &q( query;
     ( q + logout +
      &p( purchase[ InvalidPayment: p+logout | OutOfStock: q+logout ] )
     ) ) )

 login(string username, string password)
       throws InvalidLogin { ... }

 Catalog query(string query) { ... }

 string purchase(Purchase purchase)
       throws InvalidPayment, OutOfStock { ... }

 logout() { ... }
}
```

Conclusions

+ minimalistic experimental language

+ formal description of the type system

+ working prototype (and publicly available)

(parser + interpreter + type checker + run-time system)

+ some interesting examples

Future Work

> soundness proof

- concurrency

- prototype improvements:
  query (object pool) by protocol
  protocol expressiveness
  error messages friendliness
  improve/simplify code base
  ...

The End.

Yak prototype

(http://ctp.di.fct.unl.pt/yak/)