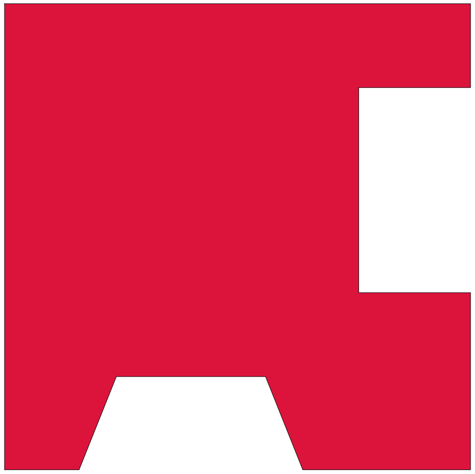
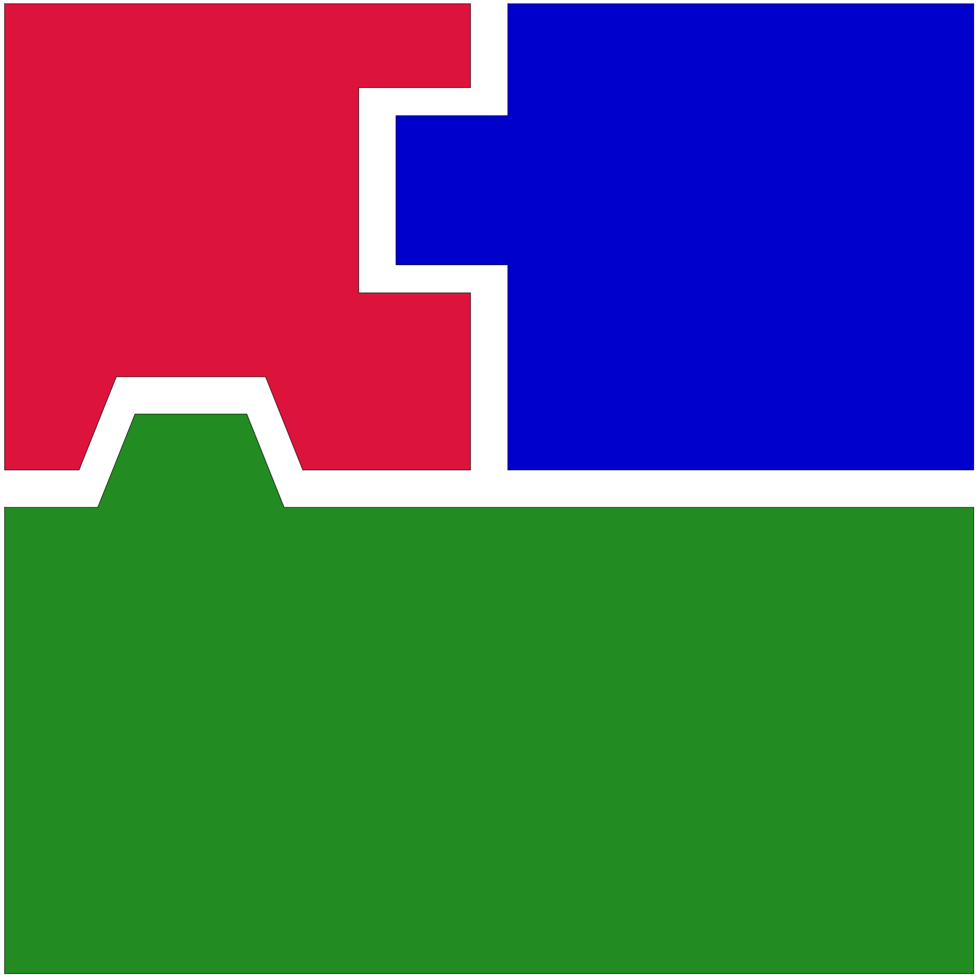
Contracts and Subtyping

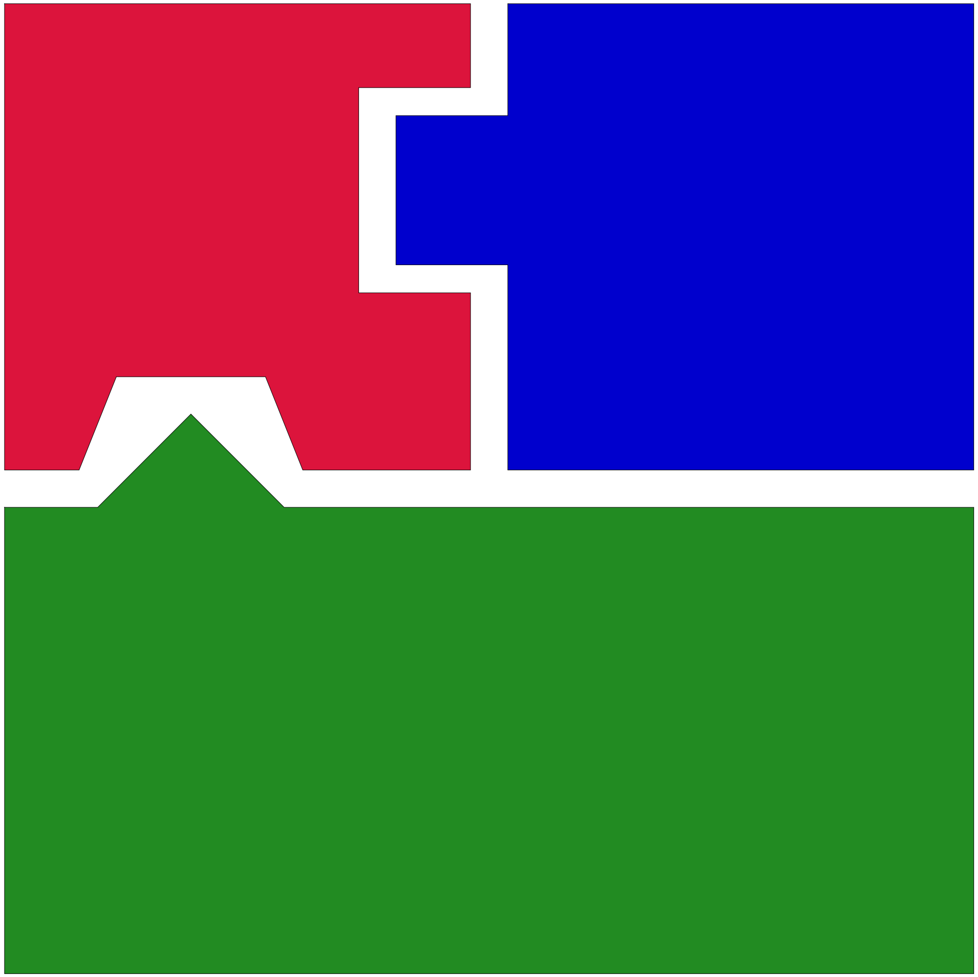
Robby Findler
University of Chicago

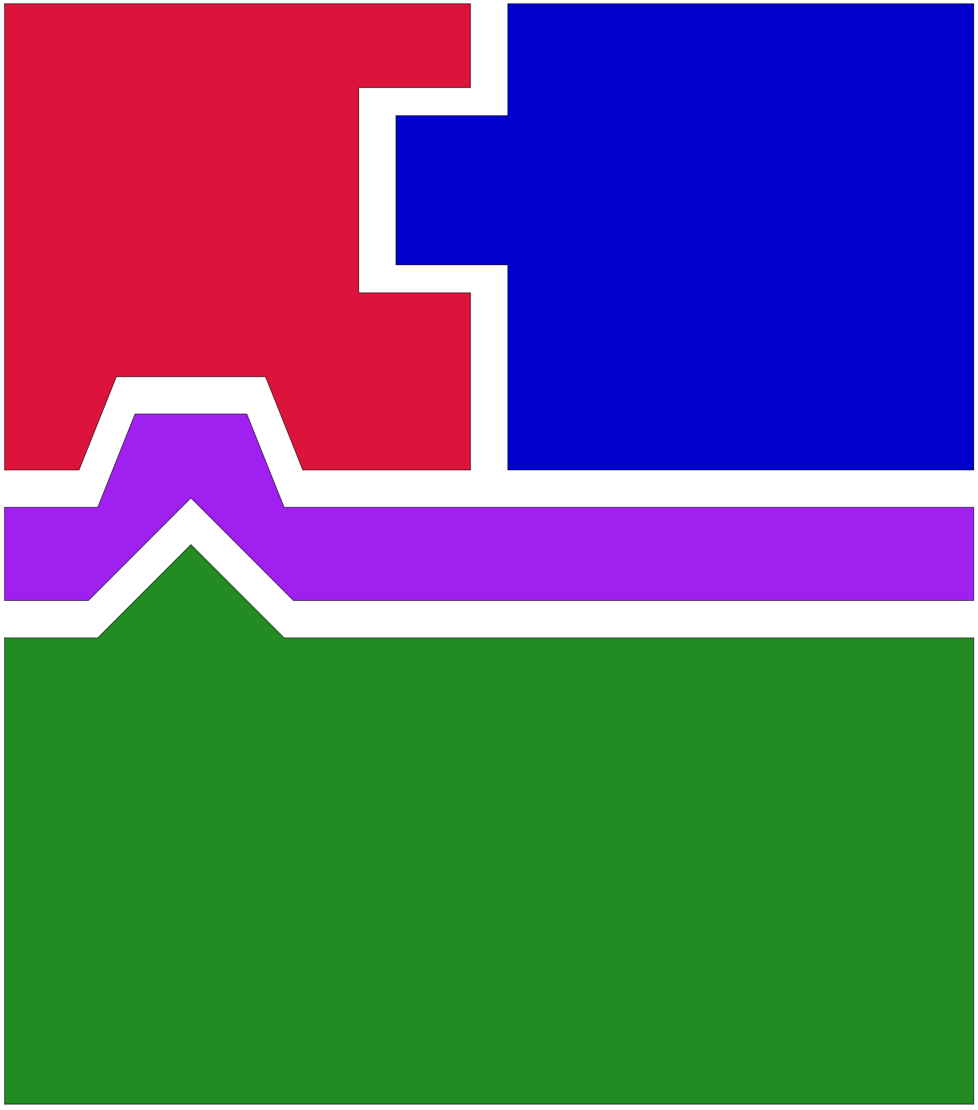
Component marketplace

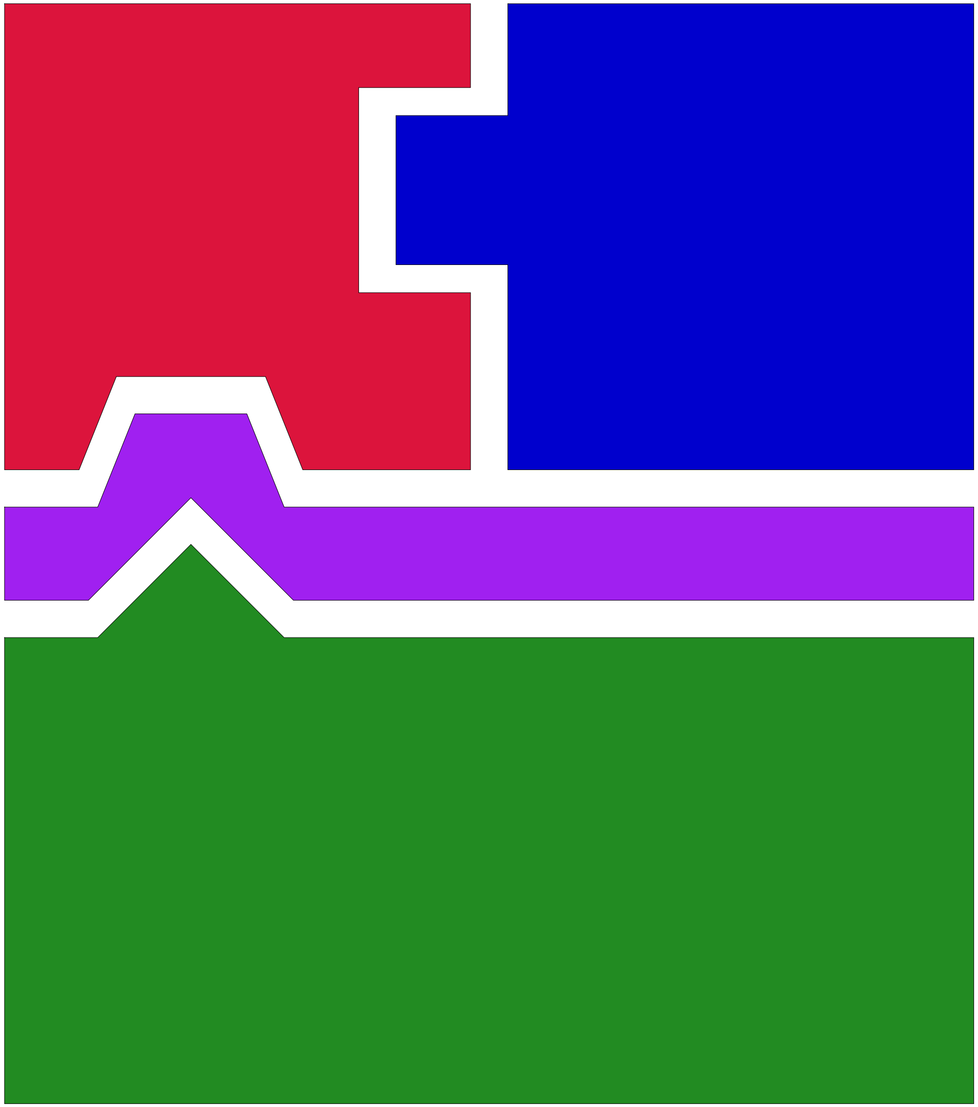
- McIlroy’s vision (1969)
- Independent developers produce pieces of programs (components)
- 3rd parties compose the components
- Economic benefits: division of labor and competition
- Software construction: merely plug & play

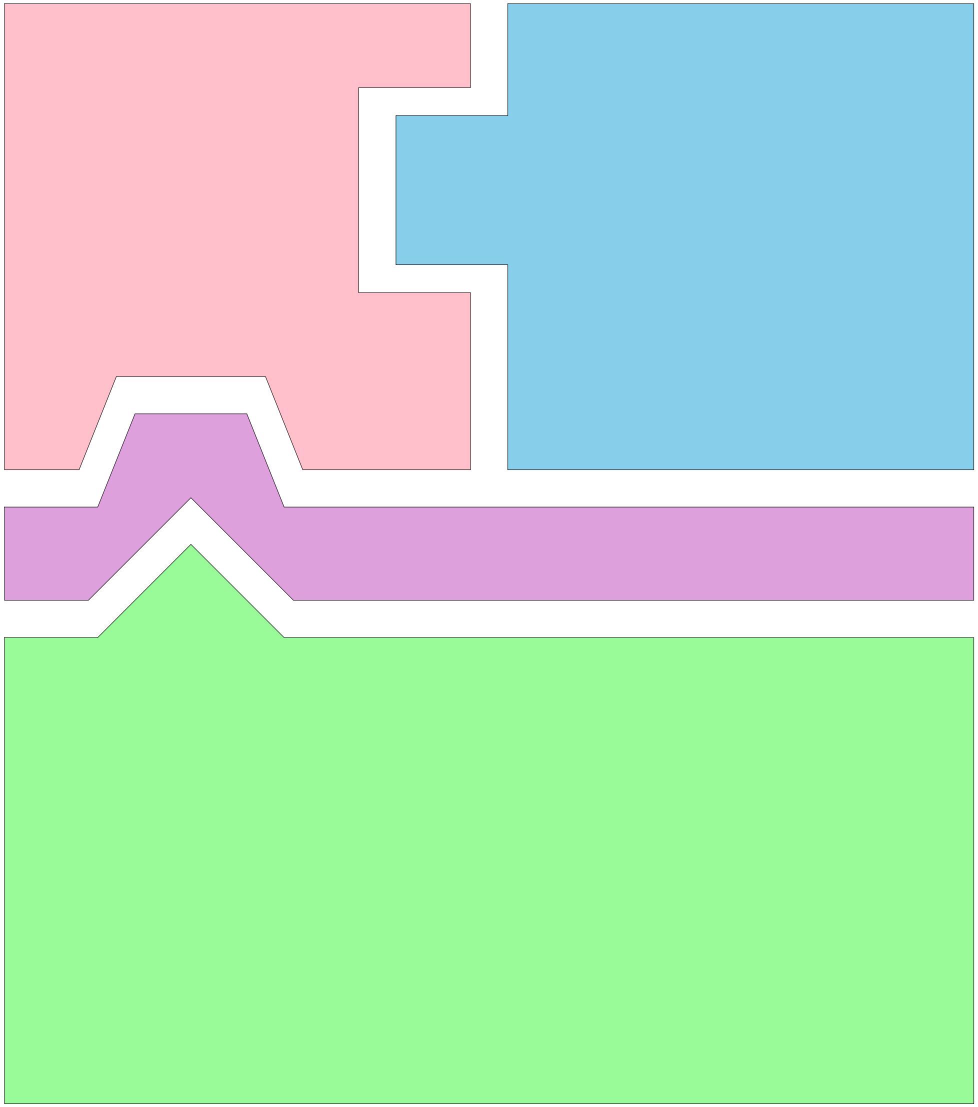


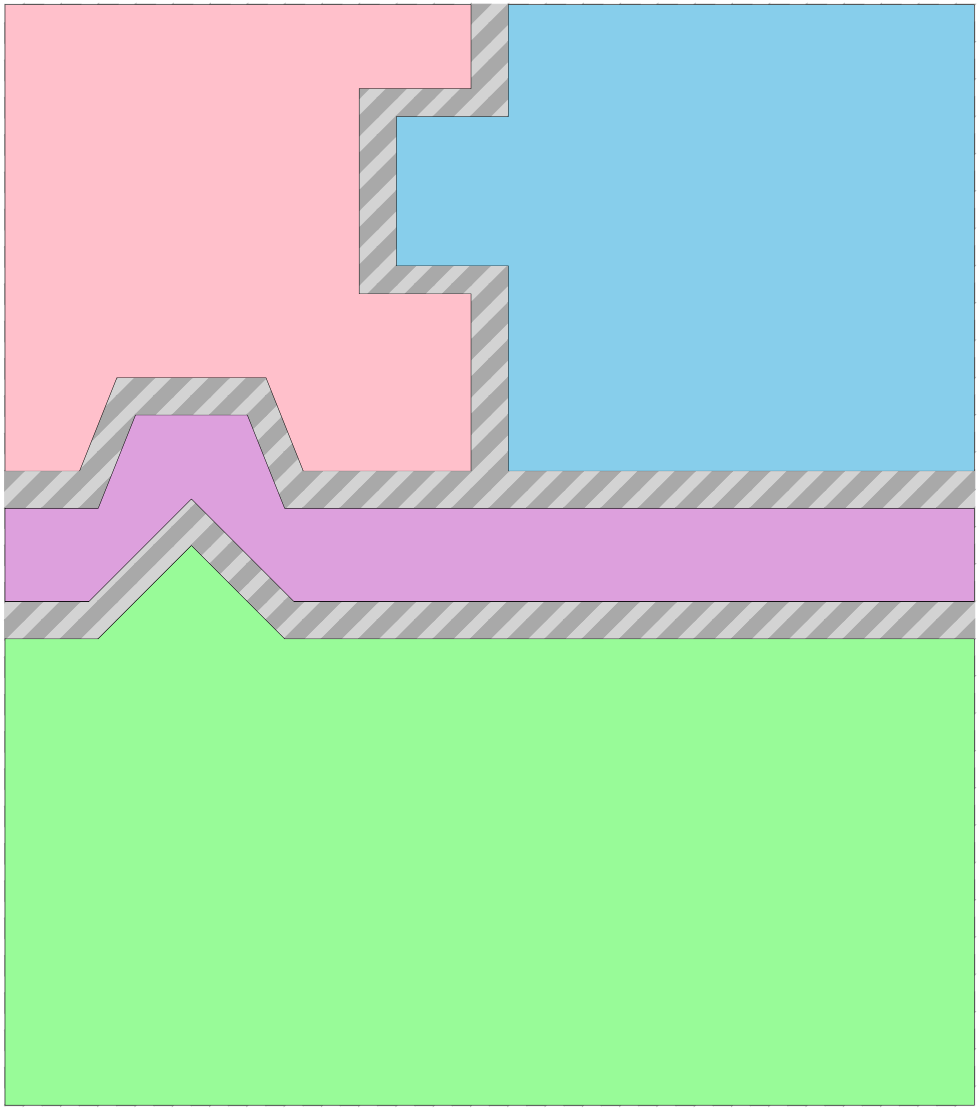












What is a contract?

- Agreement between two components
- Only allows certain patterns of interactions

Why check contracts?

- Find faulty components
- Accountability supports component economy

Contracts [Beugnard et al. 1999]

- Syntactic: types

  int f(int[] x, int k)

- Semantic level 1: behavioral

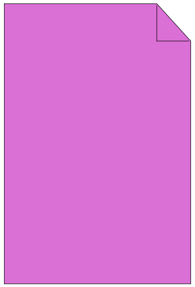
  int f(int[] x, int k) // 0 <= k < x.length()

- Semantic level 2: sequencing, concurrency

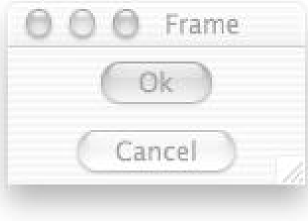
  finalize is called for all objects

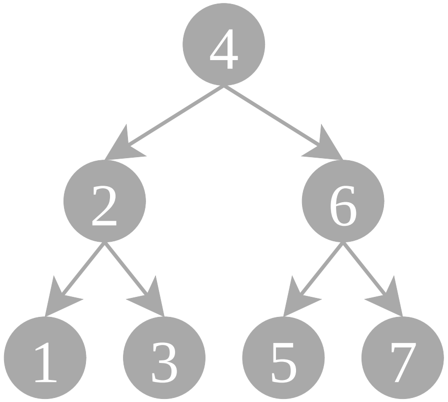
- Quality of service: space, time

  web server handles at least 1000 GET/sec



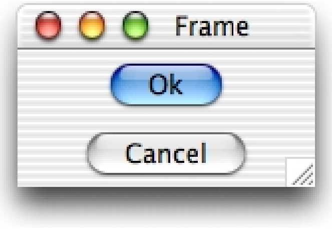
```
fopen();
...
fputs();
```

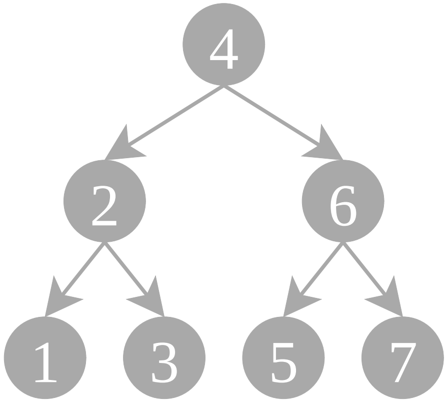




```
1/0
a[i+1]
o.m()
```

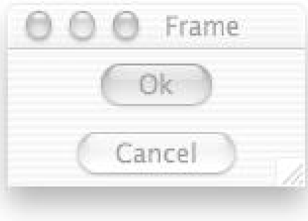
```
fopen();
...
fputs();
```

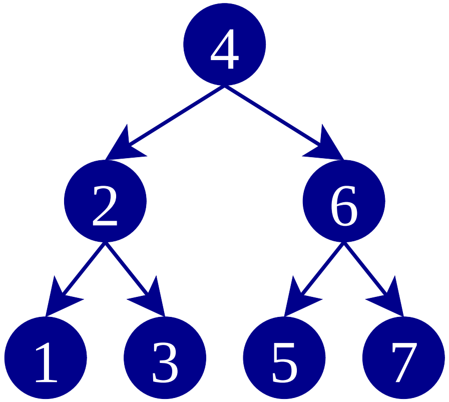




```
1/0
a[i+1]
o.m()
```

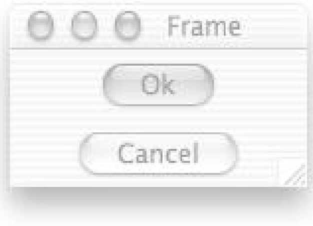
```
fopen();
...
fputs();
```

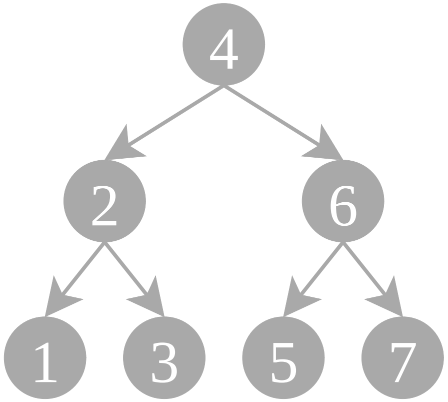




```
1/0
a[i+1]
o.m()
```

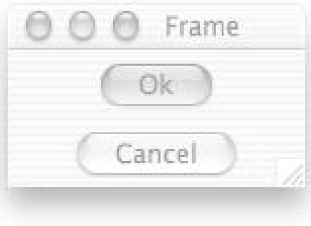
```
fopen();
...
fputs();
```

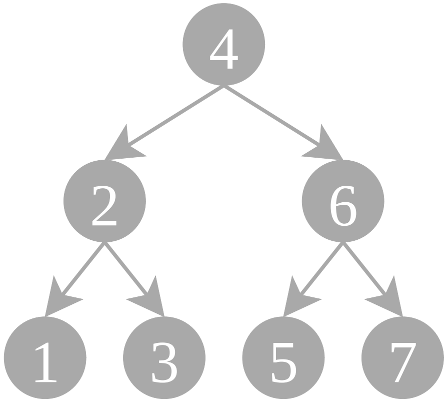




```
1/0
a[i+1]
o.m()
```

```
fopen();
...
fputs();
```

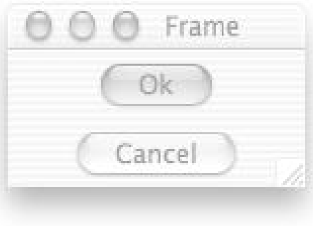


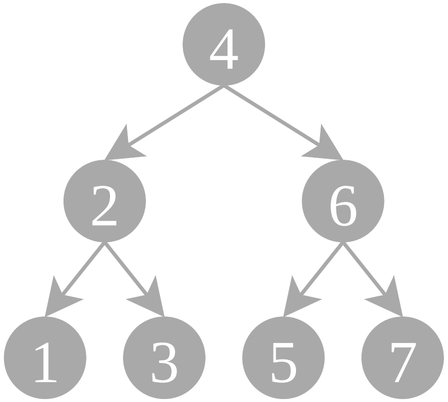


```
1/0
a[i+1]
o.m()
```

File Writer

```
fopen();
...
fputs();
```

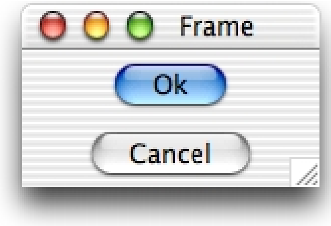


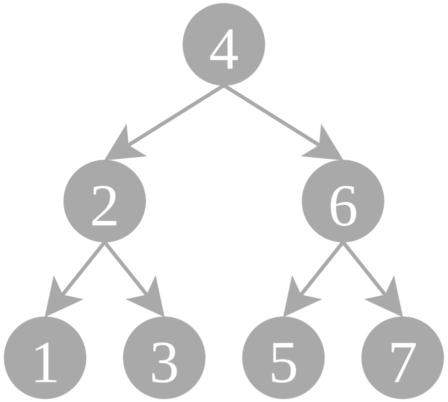


```
1/0
a[i+1]
o.m()
```

File Writer

```
fopen();
...
fputs();
```



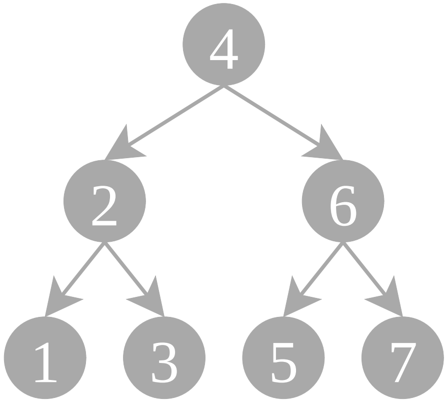


```
1/0
a[i+1]
o.m()
```

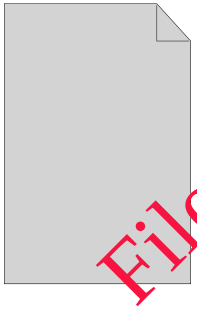
File Writer

```
fopen();
...
fputs();
```

Button Clicker



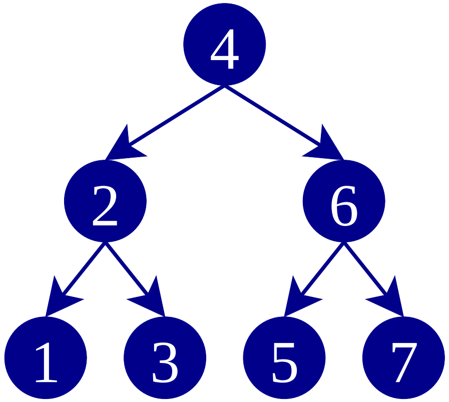
```
1/0
a[i+1]
o.m()
```



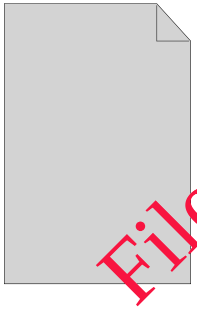
```
fopen();
...
fputs();
```

File Writer

Button Clicker



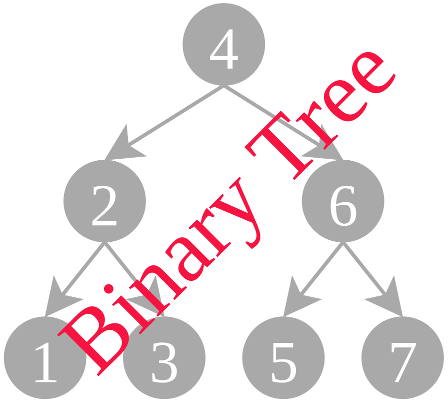
```
1/0
a[i+1]
o.m()
```



```
fopen();
...
fputs();
```

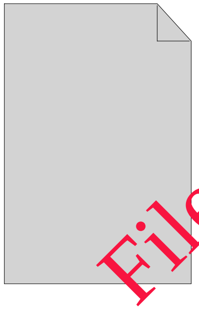
File Writer

Button Clicker



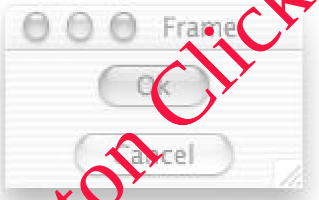
Binary Tree

```
1/0
a[i+1]
o.m()
```

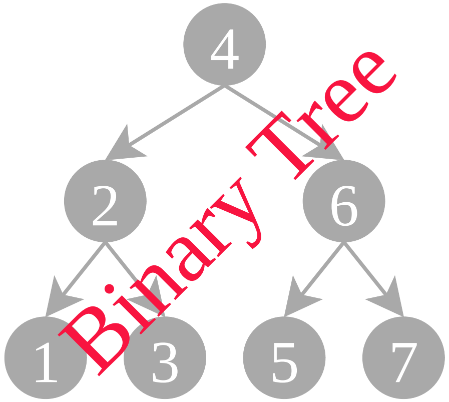


```
fopen();
...
fputs();
```

File Writer



Button Clicker



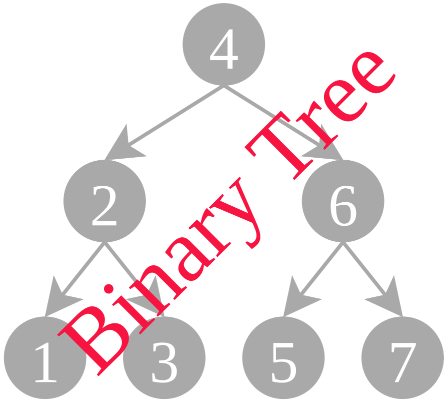
Binary Tree

```
1/0
a[i+1]
o.m()
```

```
fopen();
...
fputs();
```

File Writer

Button Clicker



Binary Tree

```
1/0
a[i+1]
orm()
```

Program

Behavioral contract desiderata

- Simplicity
- Precise enforcement
- Blame

Behavioral contract history

- Parnas: 1972
- Luckham: ANNA for Ada
- Meyer: Eiffel
- ...

Queues: an example

Queue implementation

```
class Q implements IQueue {
  void enq(int X) {...}


  int deq() {...}


  boolean empty() {...}
}
```

Queue implementation

```
class Q implements IQueue {
  void enq(int X) {...}
  // @post !this.empty()

  int deq() {...}
  // @pre !this.empty()

  boolean empty() {...}
}
```

Queue implementation

```
class Q implements IQueue {
  void enq(int X) {...}
  // @post !this.empty()

  int deq() {...}
  // @pre !this.empty()

  boolean empty() {...}
}
```

Good client

```
IQueue q = new Q();
q.enq(1);
q.deq();
```

Queue implementation

```
class Q implements IQueue {
  void enq(int X) {...}
  // @post !this.empty()

  int deq() {...}
  // @pre !this.empty()

  boolean empty() {...}
}
```

Good client

```
IQueue q = new Q();
q.enq(1);
q.deq();
```

Bad client

```
IQueue q = new Q();
q.deq();
q.enq(1);
```

Queue implementation

```
class Q implements IQueue {
  void enq(int X) {...}
  // @post !this.empty()

  int deq() {...}
  // @pre !this.empty()

  boolean empty() {...}
}
```

Good client

```
IQueue q = new Q();
q.enq(1);
q.deq();
```

Bad client

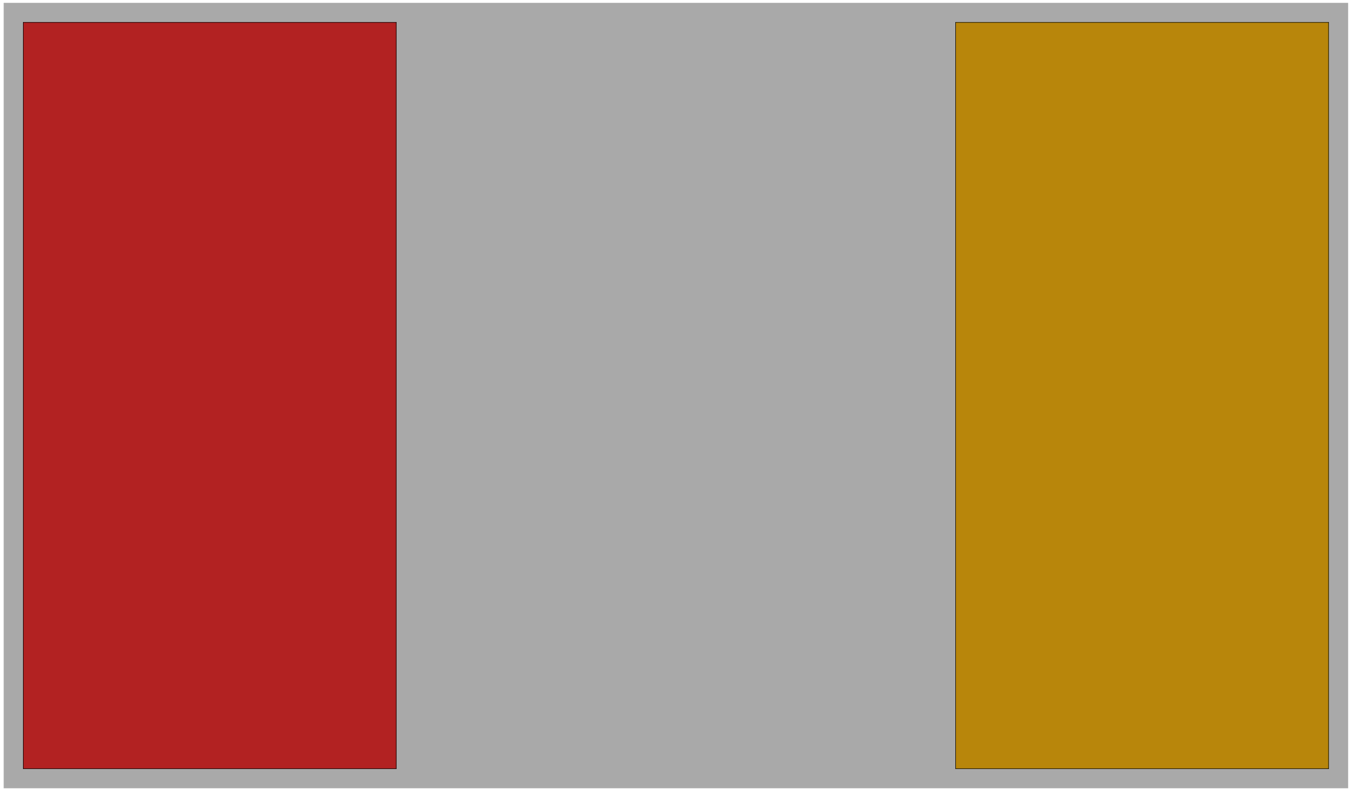
```
IQueue q = new Q();
q.deq();
q.enq(1);
```

Blame q.deq(); in Bad Client

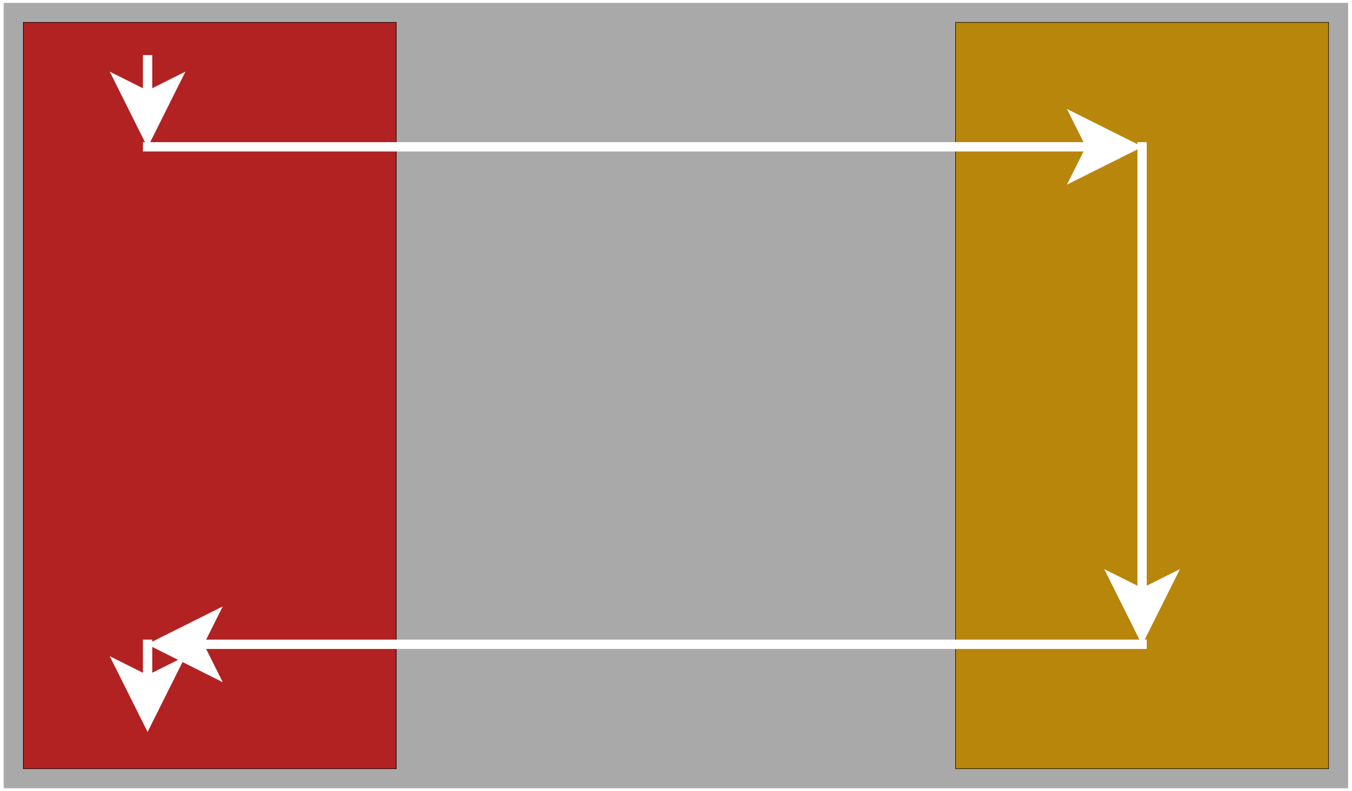
"The effective coupling of two dynamically selected parties via a well-defined interface is a powerful concept called late binding and is right at the heart of object-oriented programming."

Szyperski, 1998

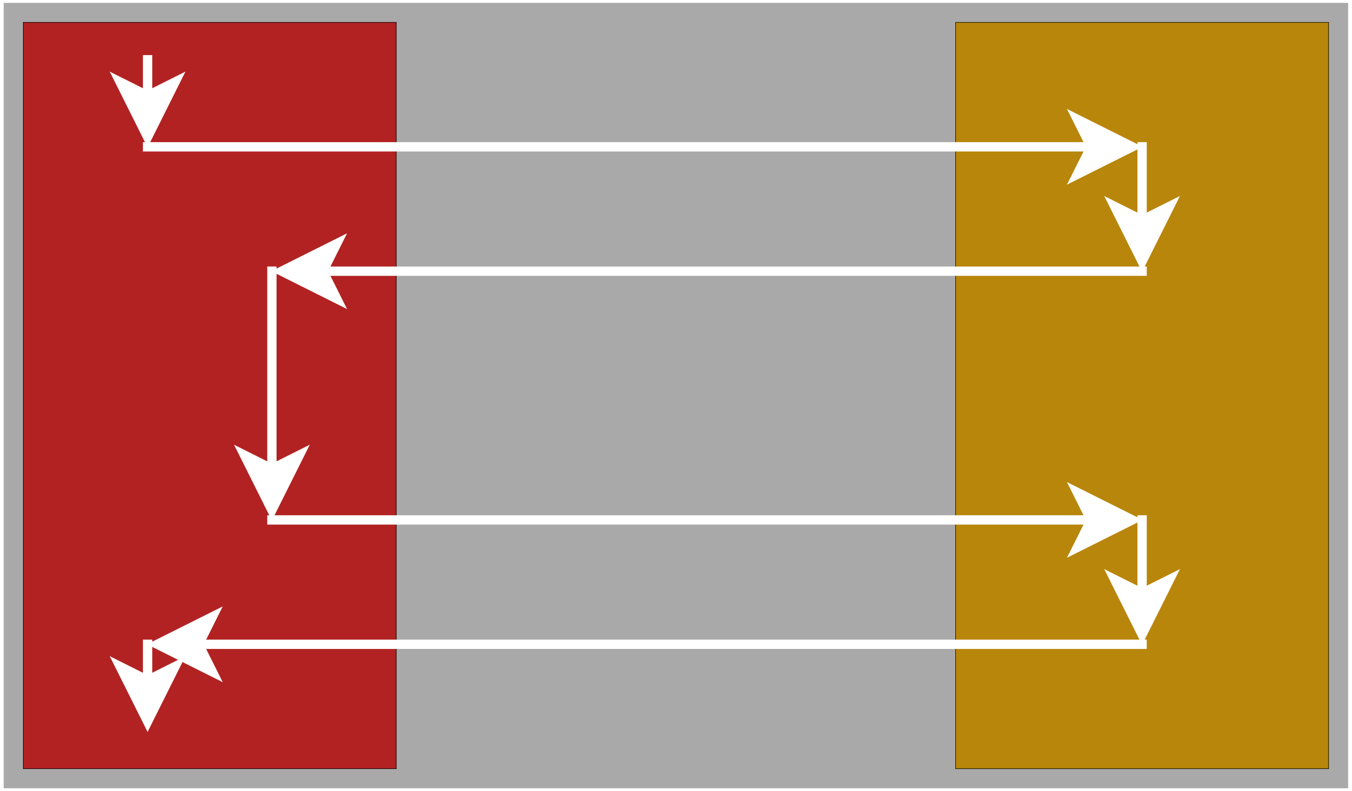
Callbacks



Callbacks



Callbacks



Queue with observer

```
class Q implements IQueue {
  Obs o;

  void enq(int X) {...}
  // @post !this.empty()
  // effect: o.onEnq(this)

  int deq() {...}
  // @pre !this.empty()
  // effect: o.onDeq(this)

  void register(Obs _o) {o = _o;}
  // please: a "good" Observer
}
```

Good observer

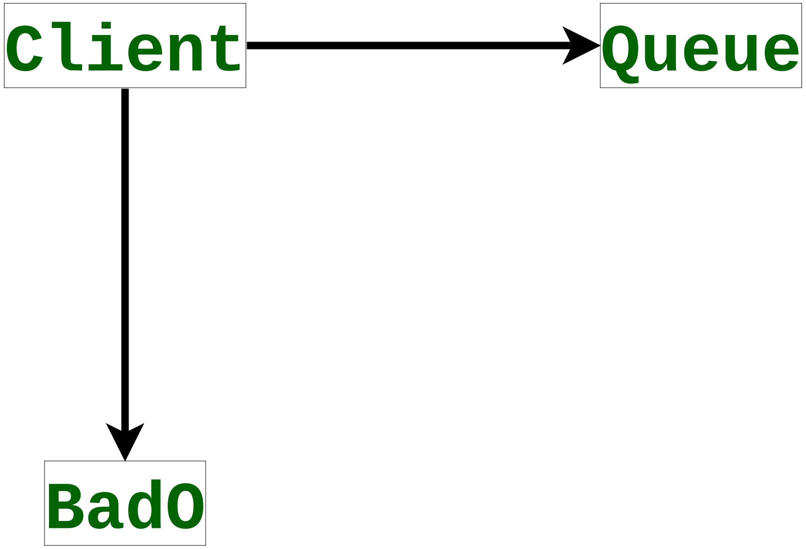
```
class GoodO
  implements Obs {
  void init() {...}

  void onEnq(IQueue q)
     {...}
  // @post !q.empty()

  void onDeq(IQueue q)
     {...}
}
```

Good observer

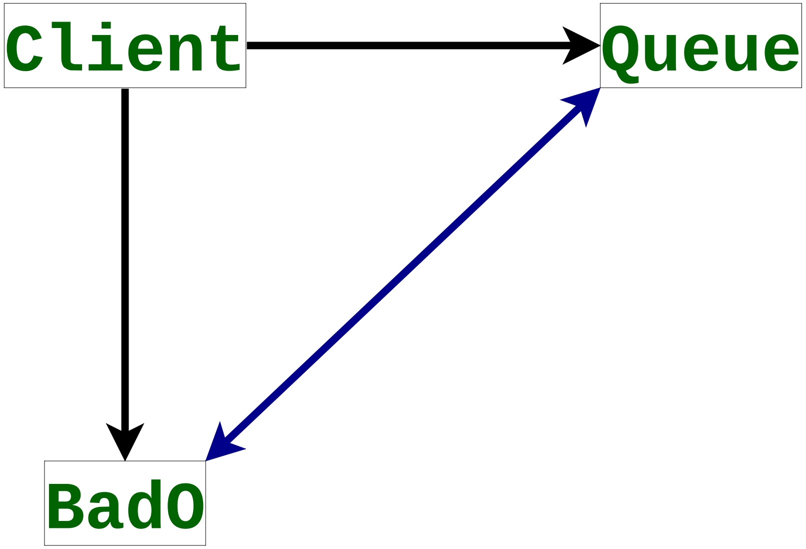
```
class GoodO
  implements Obs {
  void init() {...}

  void onEnq(IQueue q)
     {...}
  // @post !q.empty()

  void onDeq(IQueue q)
     {...}
}
```

Bad Observer

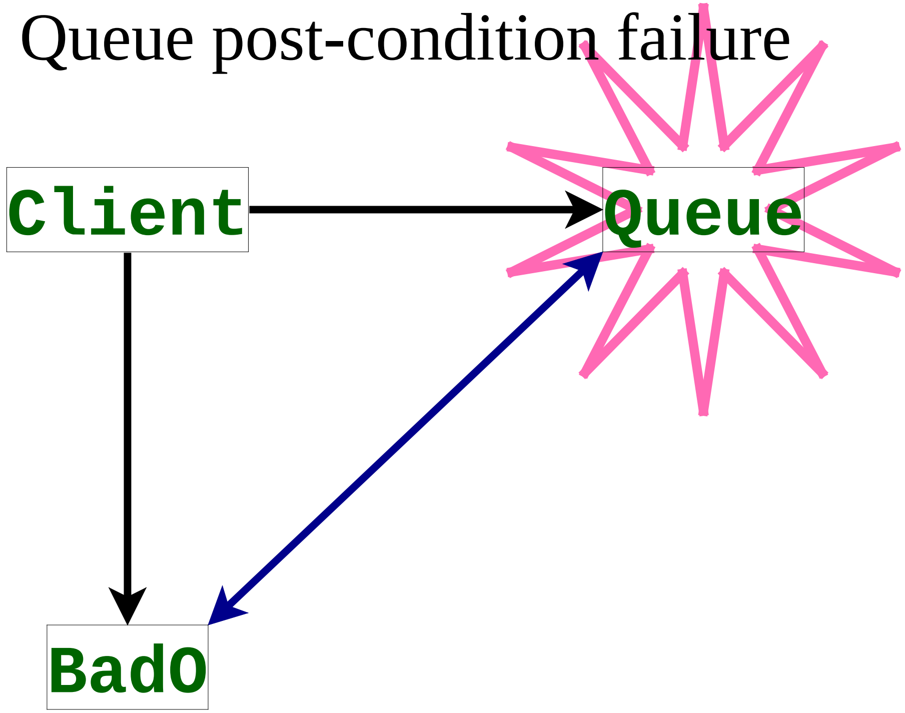
```
class BadO
  implements Obs {
  void init() {...}

  void onEnq(IQueue q)
    { q.deq() }


  void onDeq(IQueue q)
    {...}
}
```


Client
Queue
BadO

Client links BadO and Queue



Queue post-condition failure

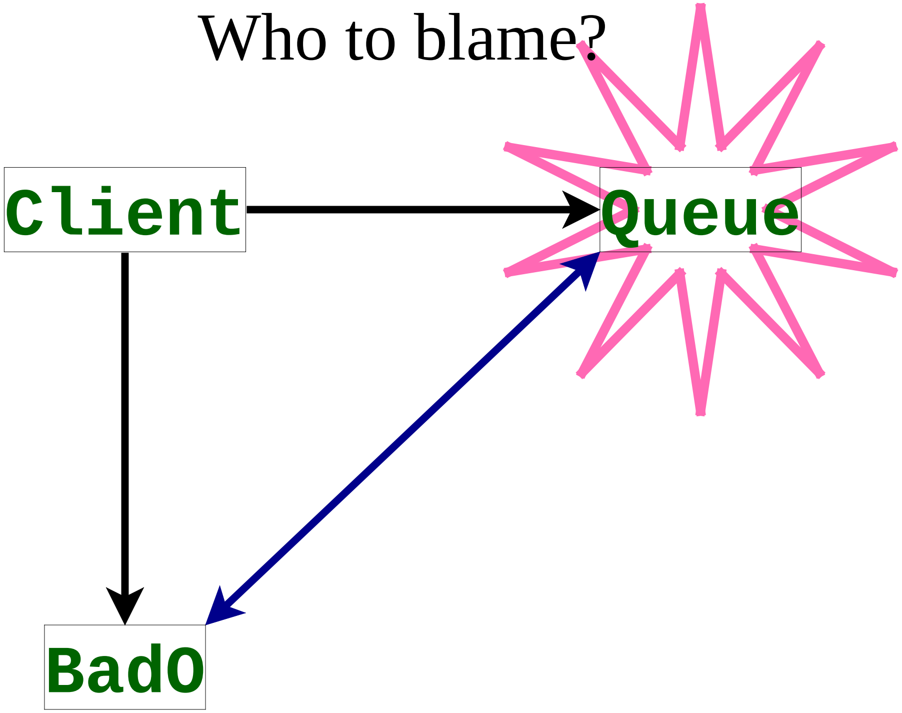


Client

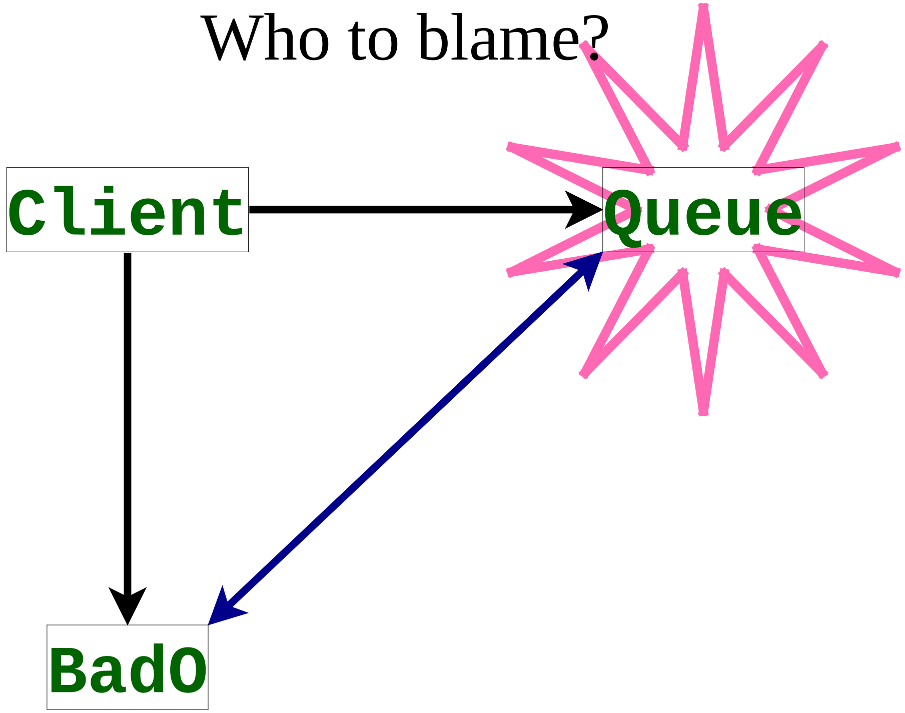
Queue

BadO

Who to blame?



Who to blame?



- Client combines mis-matched components
- BadO violates informal contract
- Queue is blamed

Queue with observer

```
class Q implements IQueue {
  Obs o;

  void enq(int X) {...}
  // @post !this.empty()
  // effect: o.onEnq(this)

  int deq() {...}
  // @pre !this.empty()
  // effect: o.onDeq(this)

  void register(Obs _o) {o = _o;}
  // please: a "good" Observer
}
```

Queue with observer

```
class Q implements IQueue {
  Obs o;

  void enq(int X) {...}
  // @post !this.empty()
  // effect: o.onEnq(this)

  int deq() {...}
  // @pre !this.empty()
  // effect: o.onDeq(this)

  void register(Obs _o) {o = _o;}
  //     @pre _o.onEnq(...)
}
```

Contracts in interfaces?

Observer contracts

```
interface Obs {
  void init();

  void onEnq(IQueue q);
  // @post !q.empty()

  void onDeq(IQueue q);
  // @pre !q.empty()
}
```

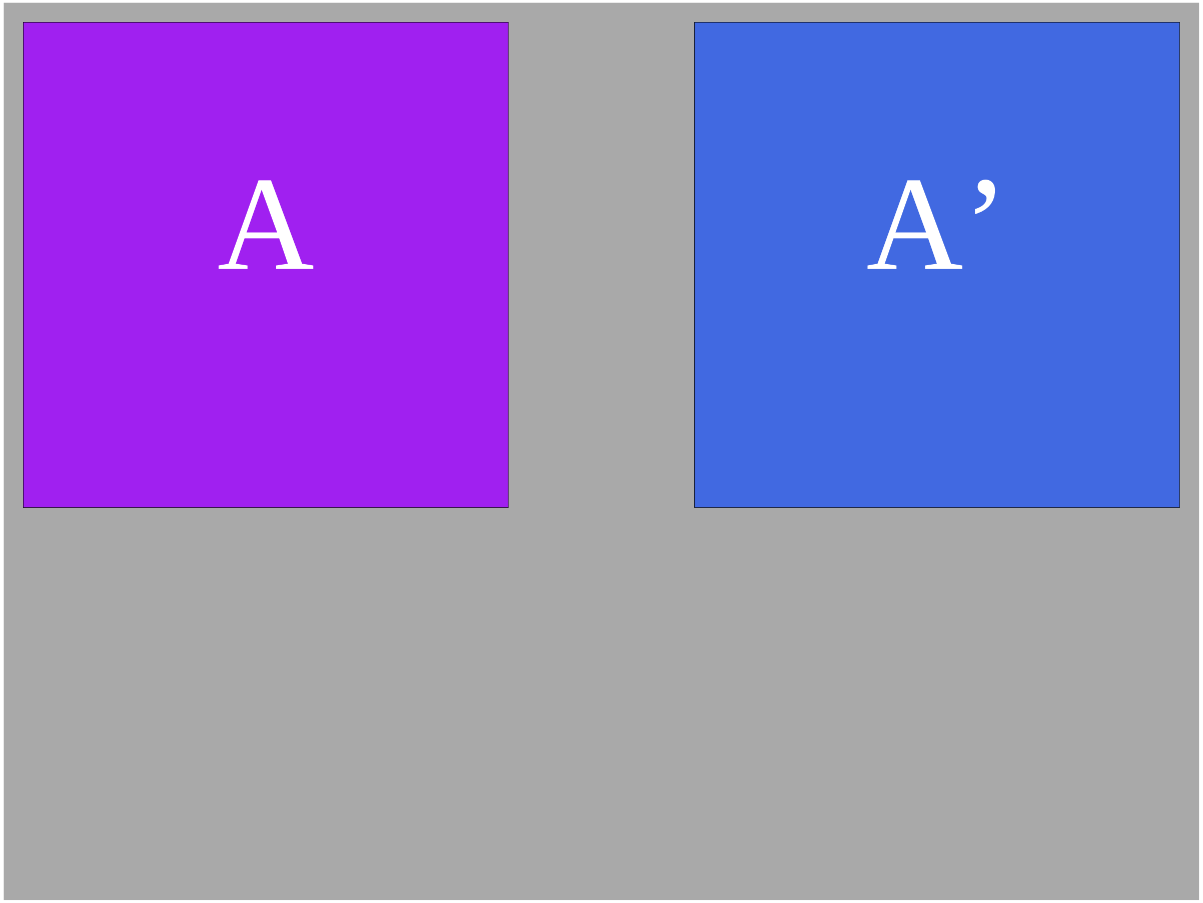
Force observers to meet pre- and post-conditions that Queue needs

Controlling BadO

```
class BadO
  implements Obs {
  void init() {...}

  void onEnq(IQueue q)
    { q.deq() }


  void onDeq(IQueue q)
    {...}
}
```

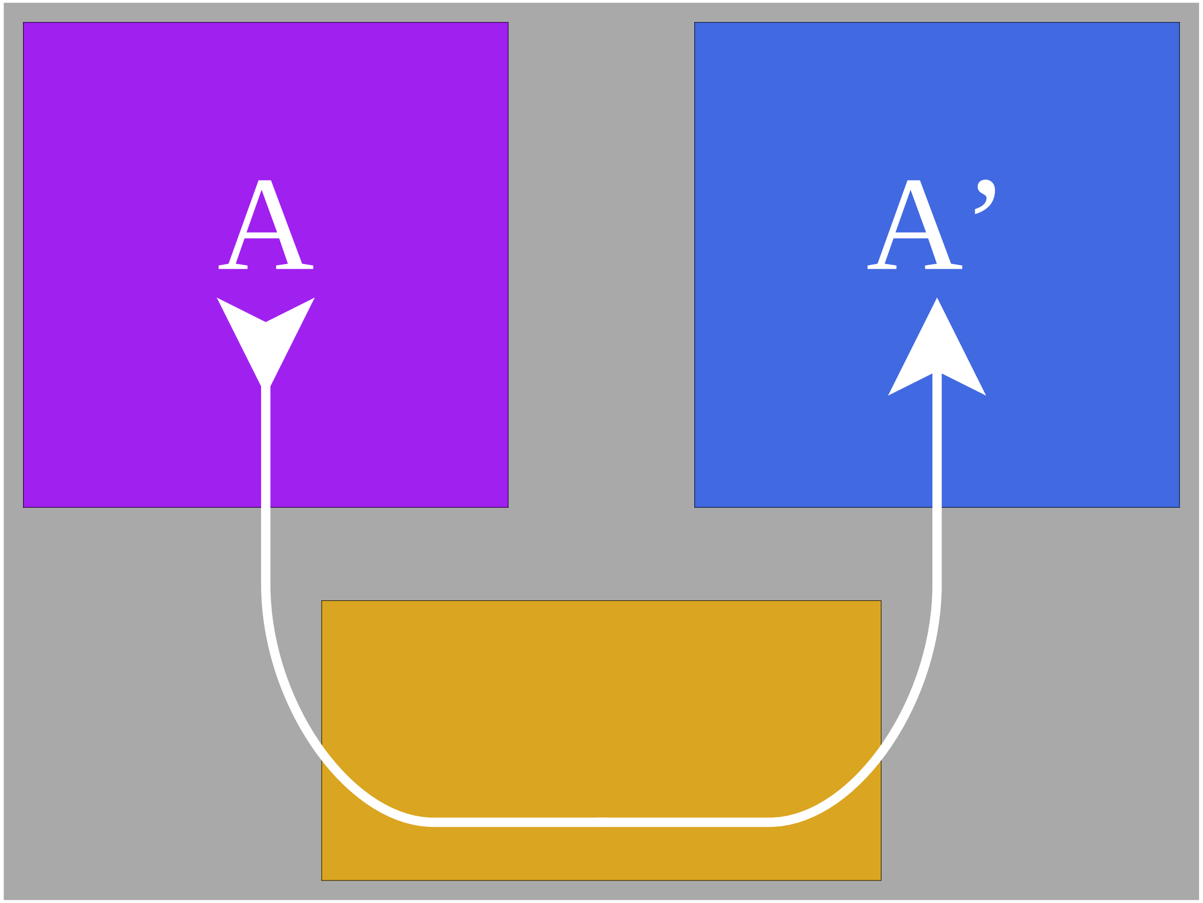
Controlling BadO

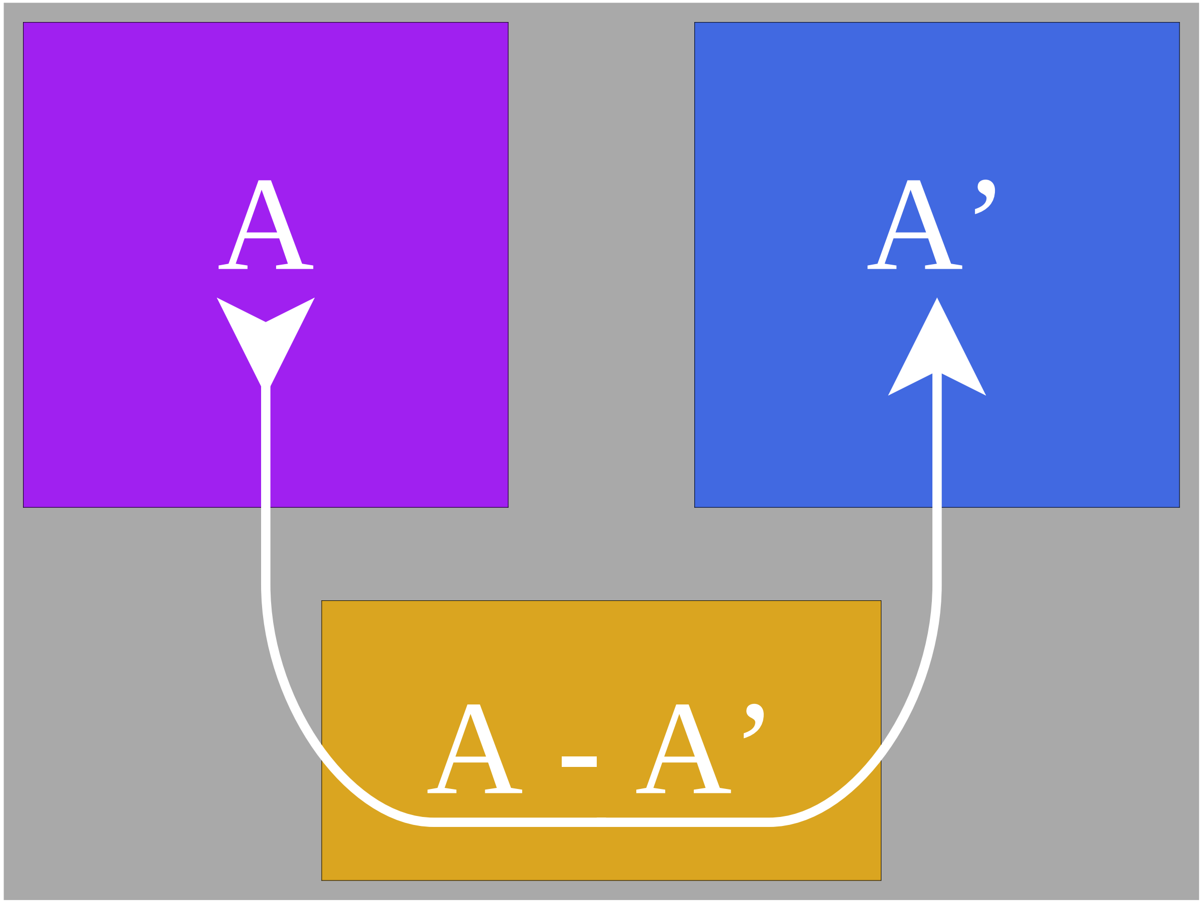
```
class BadO
  implements Obs {
  void init() {...}

  void onEnq(IQueue q)
    { q.deq() }
  // @post !q.empty()

  void onDeq(IQueue q)
    {...}
}
```

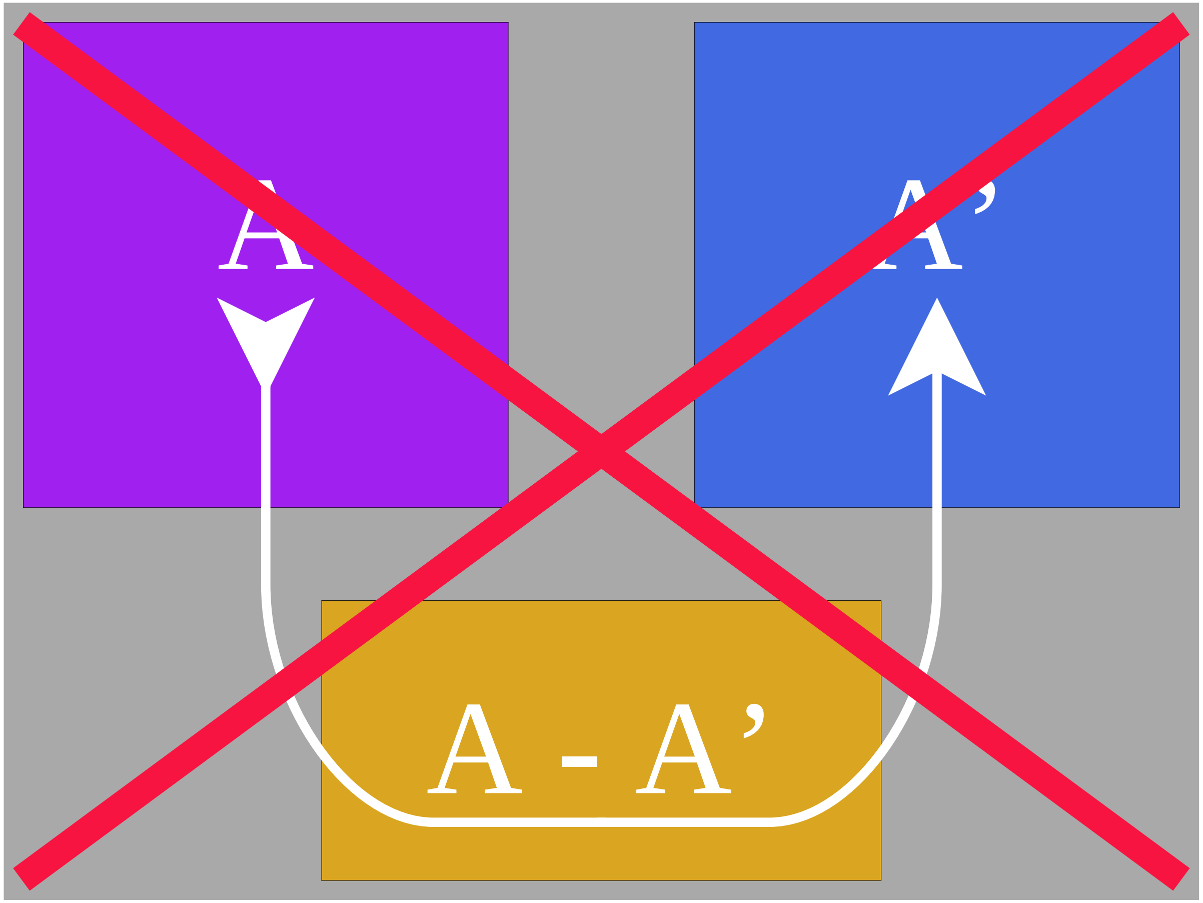


A

A’


A
A’


A
A’
A - A’

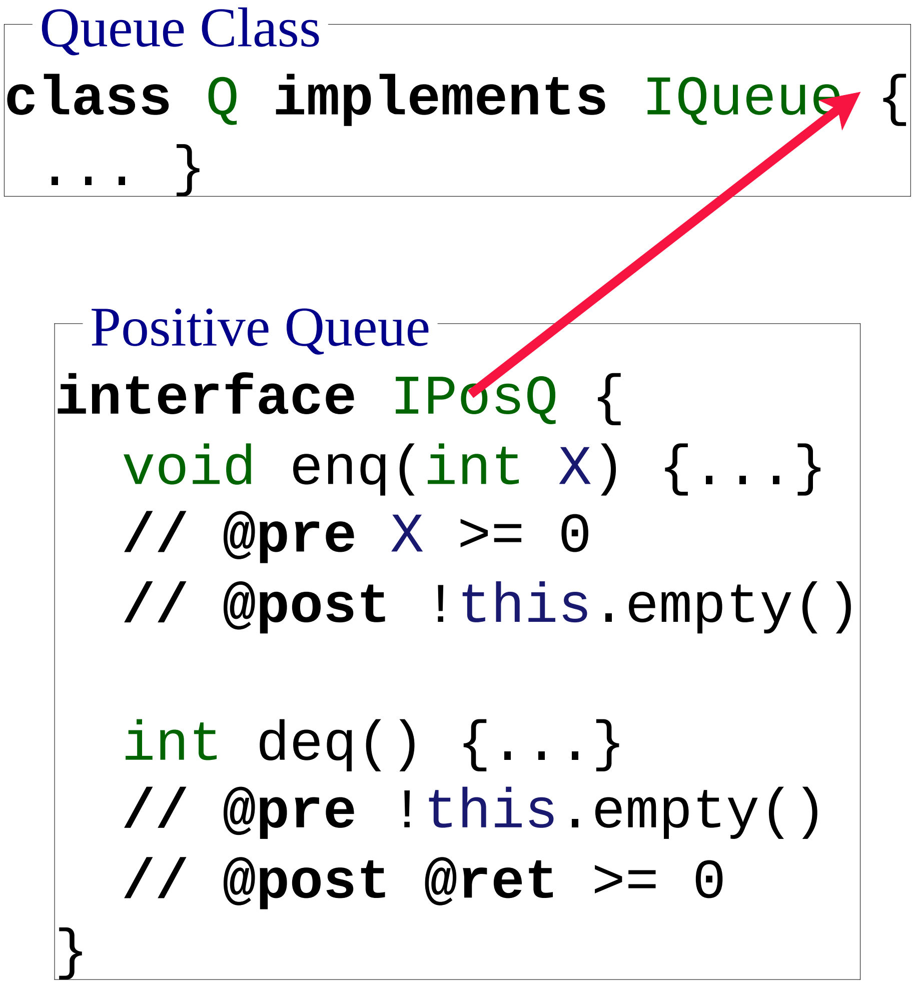

A
A’
A - A’

Queue Class

```
class Q implements IQueue {
 ... }
```

Positive Queue

```
interface IPosQ {
  void enq(int X) {...}
  // @pre X >= 0
  // @post !this.empty()

  int deq() {...}
  // @pre !this.empty()
  // @post @ret >= 0
}
```



Queue Class

```
class Q implements IQueue {
 ... }
```

Positive Queue

```
interface IPosQ {
  void enq(int X) {...}
  // @pre X >= 0
  // @post !this.empty()

  int deq() {...}
  // @pre !this.empty()
  // @post @ret >= 0
}
```

Structural vs Nominal Subtyping

Subtyping

- Type system policy
- Determines which types match
- Determines which values flow where

Nominal subtyping

- Hierarchy explicit
- Conventional OO PLs: C++, C#, Eiffel, Java

Structural subtyping

- Hierarchy implicit
- Research OO PLs: Moby, OML, OCaml, LOOM, PolyTOIL

Nominal subtyping

- Simple to implement
- Simple type-error messages
- Inhibits re-use

Structural subtyping

- Harder to implement
- Complex type-error messages
- Permits flexible re-use

QClass

```
class Q implements IQueue {
 ... }
```

IQueue

```
interface IQueue {
  void enq(int X) {...}
  // @post !this.empty()


  int deq() {...}
  // @pre !this.empty()

}
```

IPosQ

```
interface IPosQ {
  void enq(int X) {...}
  // @pre X >= 0
  // @post !this.empty()

  int deq() {...}
  // @pre !this.empty()
  // @post @ret >= 0
}
```

Structural Subtyping for Contracts

Goals

- Bring structural subtyping to contracts
- Leave behind complexity
- Use in a nominal context

```
wrap(obj, Int, <fromStr>, <toStr>)
```

A structural subtype "cast"

```
wrap(obj, Int, <fromStr>, <toStr>)
```

The object that gets casted, now has additional contracts

```
wrap(obj, Int, <fromStr>, <toStr>)
```

The interface that describes
the additional contracts

```
wrap(obj, Int, <fromStr>, <toStr>)
```

The name of the component
where the object is from;

Responsible for post-conds

```
wrap(obj, Int, <fromStr>, <toStr>)
```

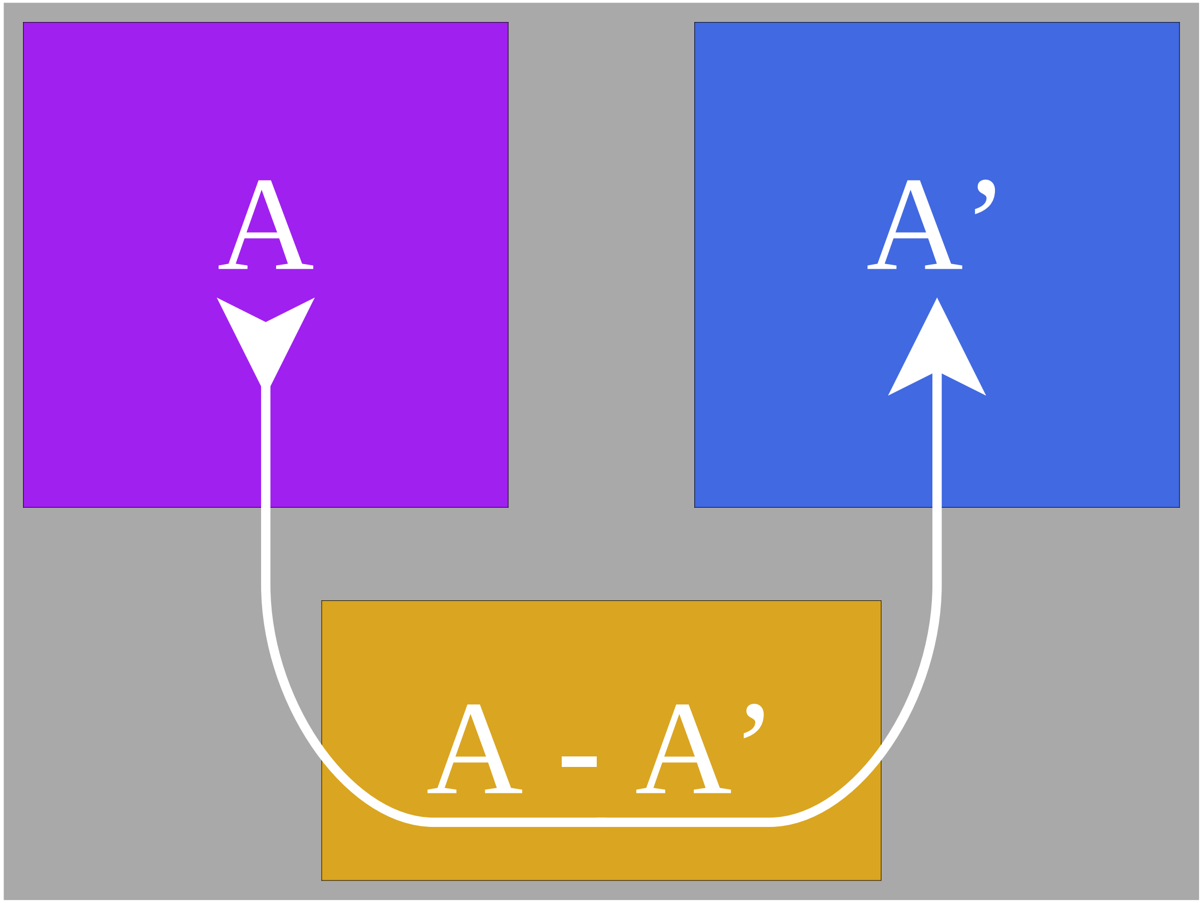
The name of the component
where the object is sent;

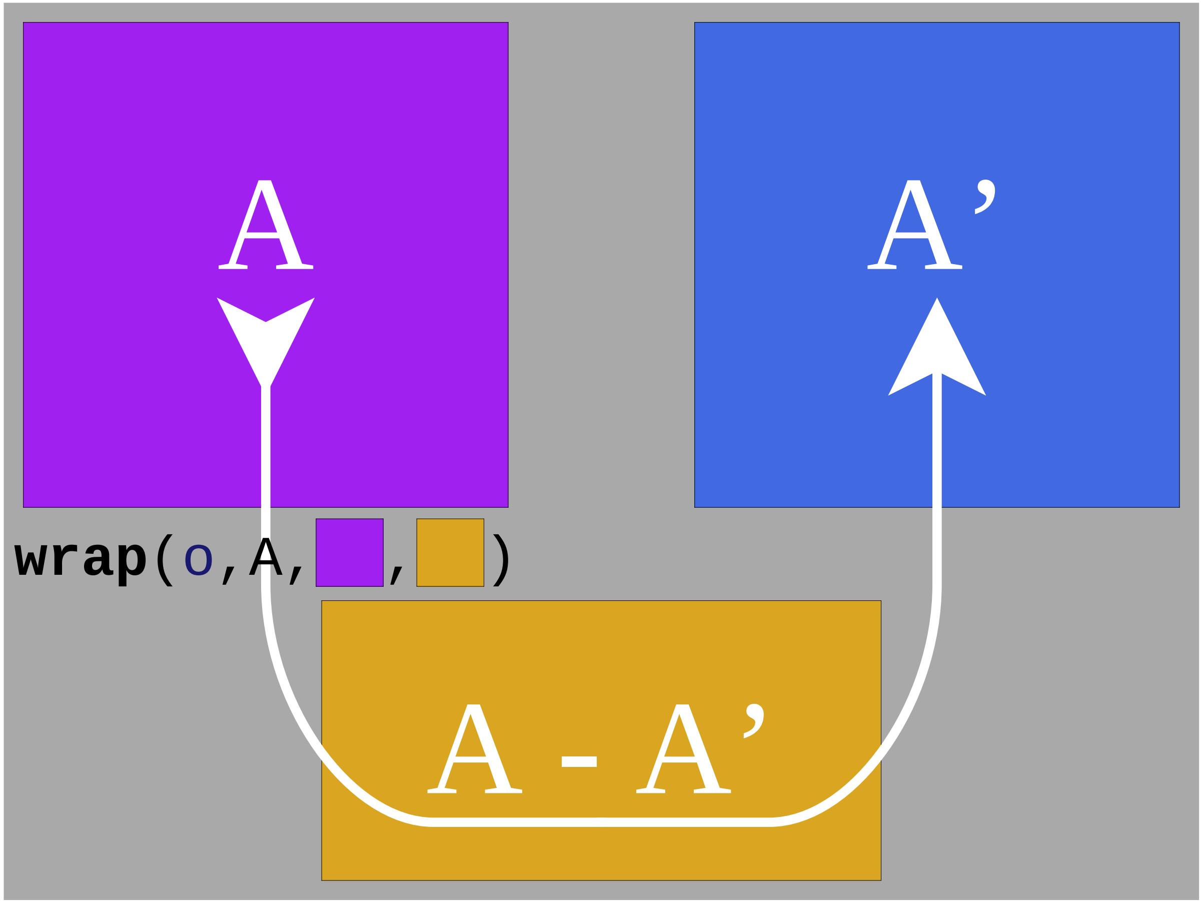
Responsible for pre-conds

```
wrap(obj, Int, <fromStr>, <toStr>)
```

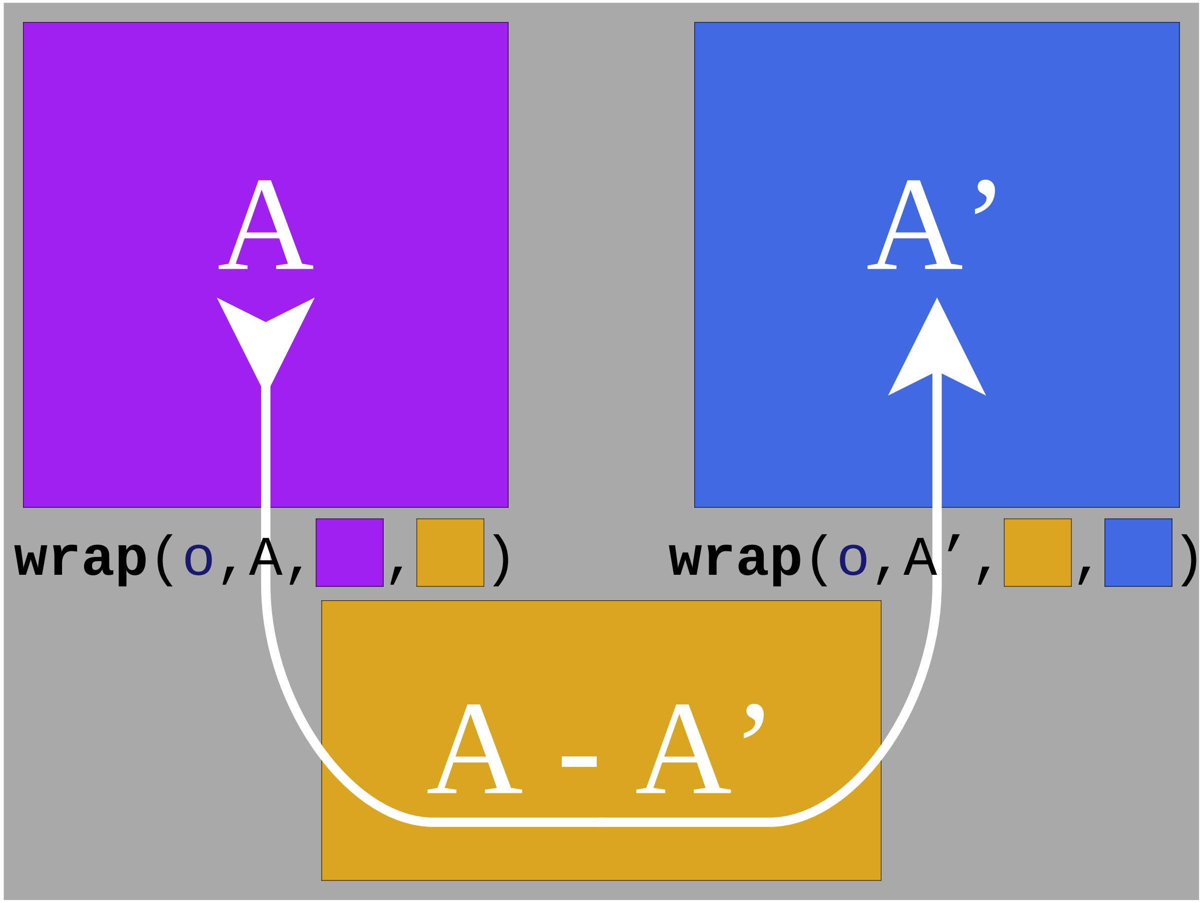
Result ensures Int’s contracts
but otherwise identical to obj

Has type Int, even if obj doesn’t


A
A’
A - A’



```
wrap(o,A,🟪,🟨)
```


A
A’
wrap(o,A,🟪,🟨)
wrap(o,A’,🟨,🟦)
A - A’

<Queue>

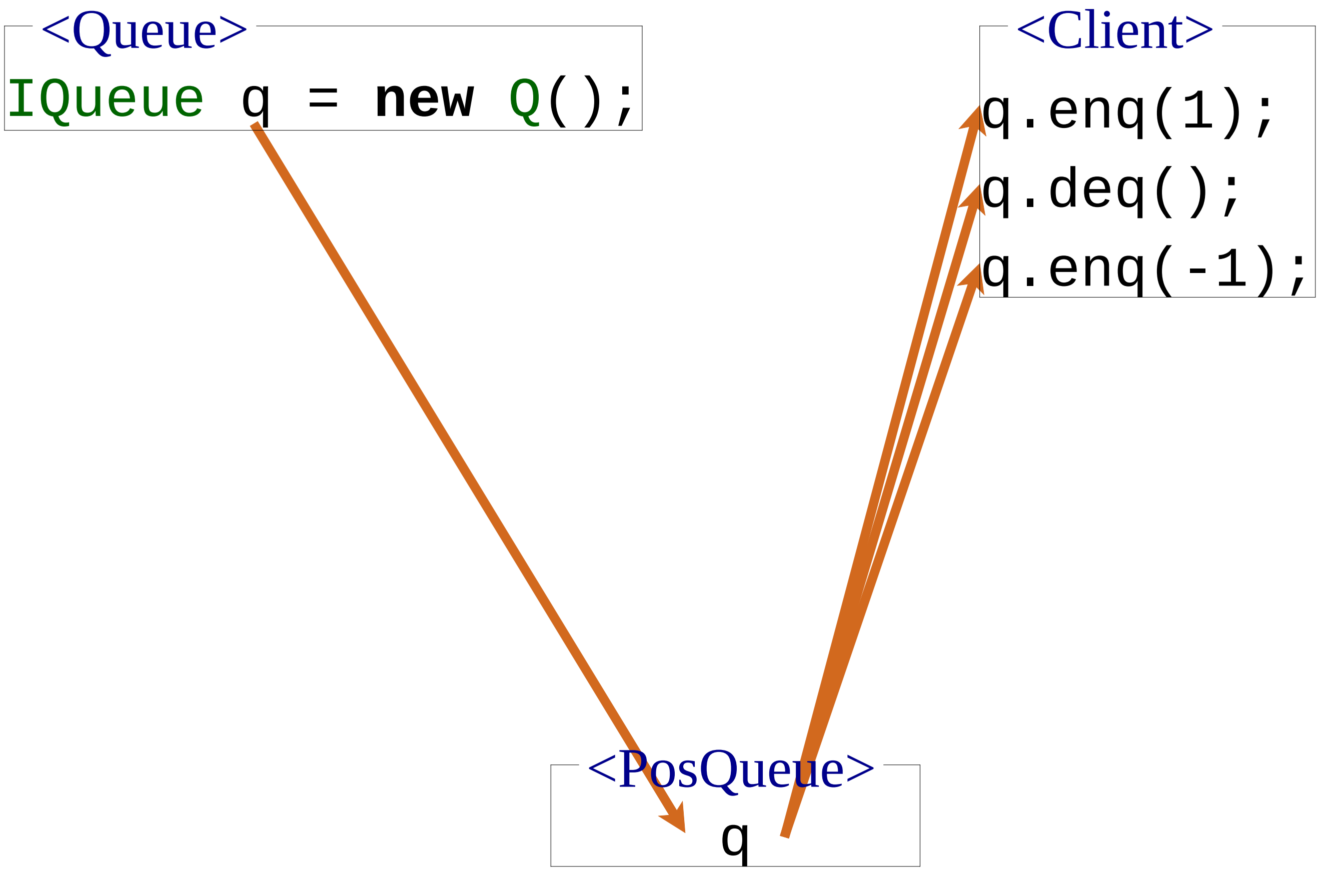
```
IQueue q = new Q();
```

<Client>

```
q.enq(1);
q.deq();
q.enq(-1);
```

<PosQueue>

```
q
```



<Queue>

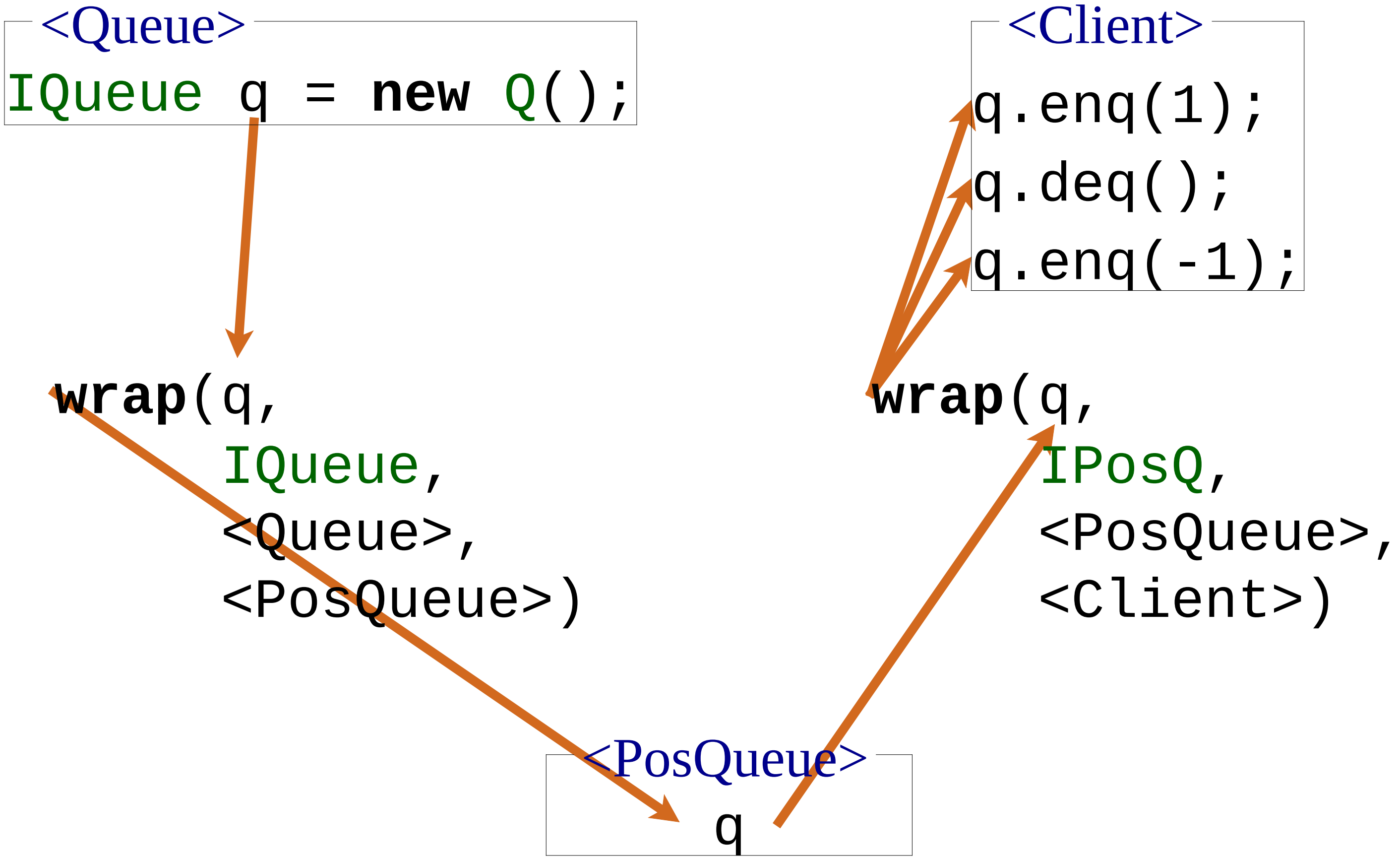
```
IQueue q = new Q();
```

<Client>

```
q.enq(1);
q.deq();
q.enq(-1);
```

<PosQueue>

q



<Queue>

```
IQueue q = new Q();
```

<Client>

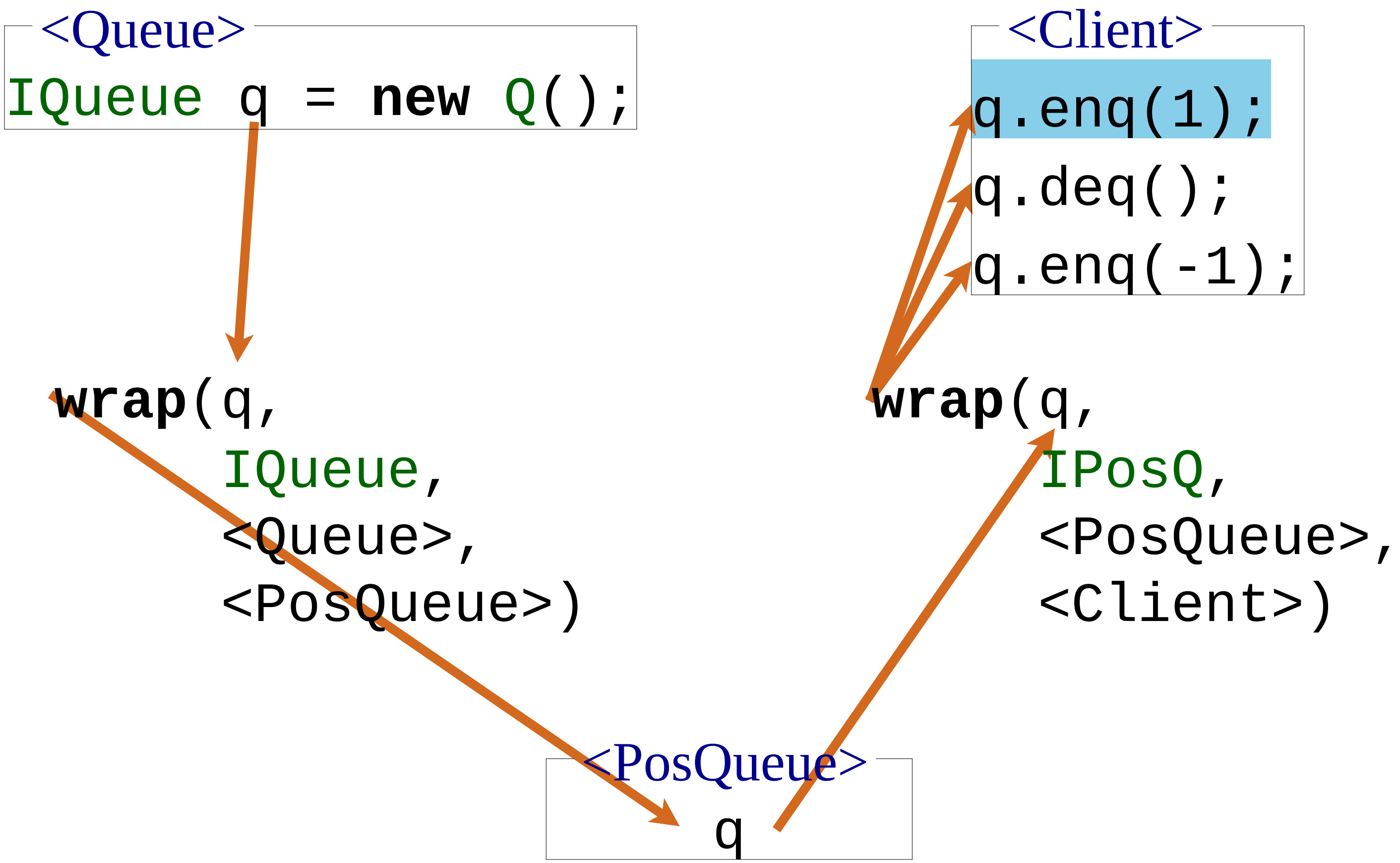
```
q.enq(1);
q.deq();
q.enq(-1);
```

```
wrap(q,
     IQueue,
     <Queue>,
     <PosQueue>)
```

```
wrap(q,
     IPosQ,
     <PosQueue>,
     <Client>)
```

<PosQueue>

```
q
```



<Queue>

```
IQueue q = new Q();
```

<Client>

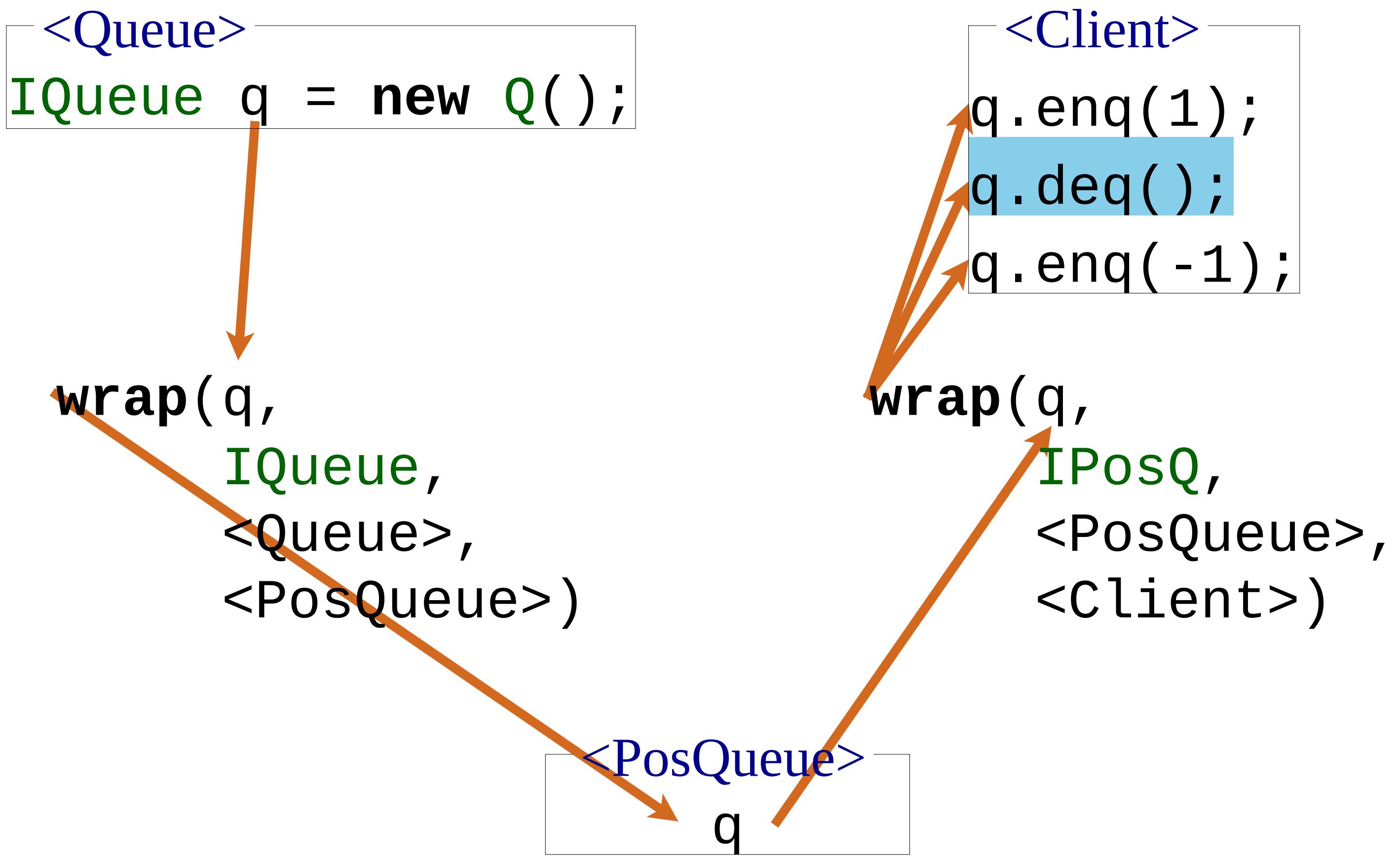
```
q.enq(1);
q.deq();
q.enq(-1);
```

```
wrap(q,
     IQueue,
     <Queue>,
     <PosQueue>)
```

```
wrap(q,
     IPosQ,
     <PosQueue>,
     <Client>)
```

<PosQueue>

```
q
```



<Queue>

```
IQueue q = new Q();
```

<Client>

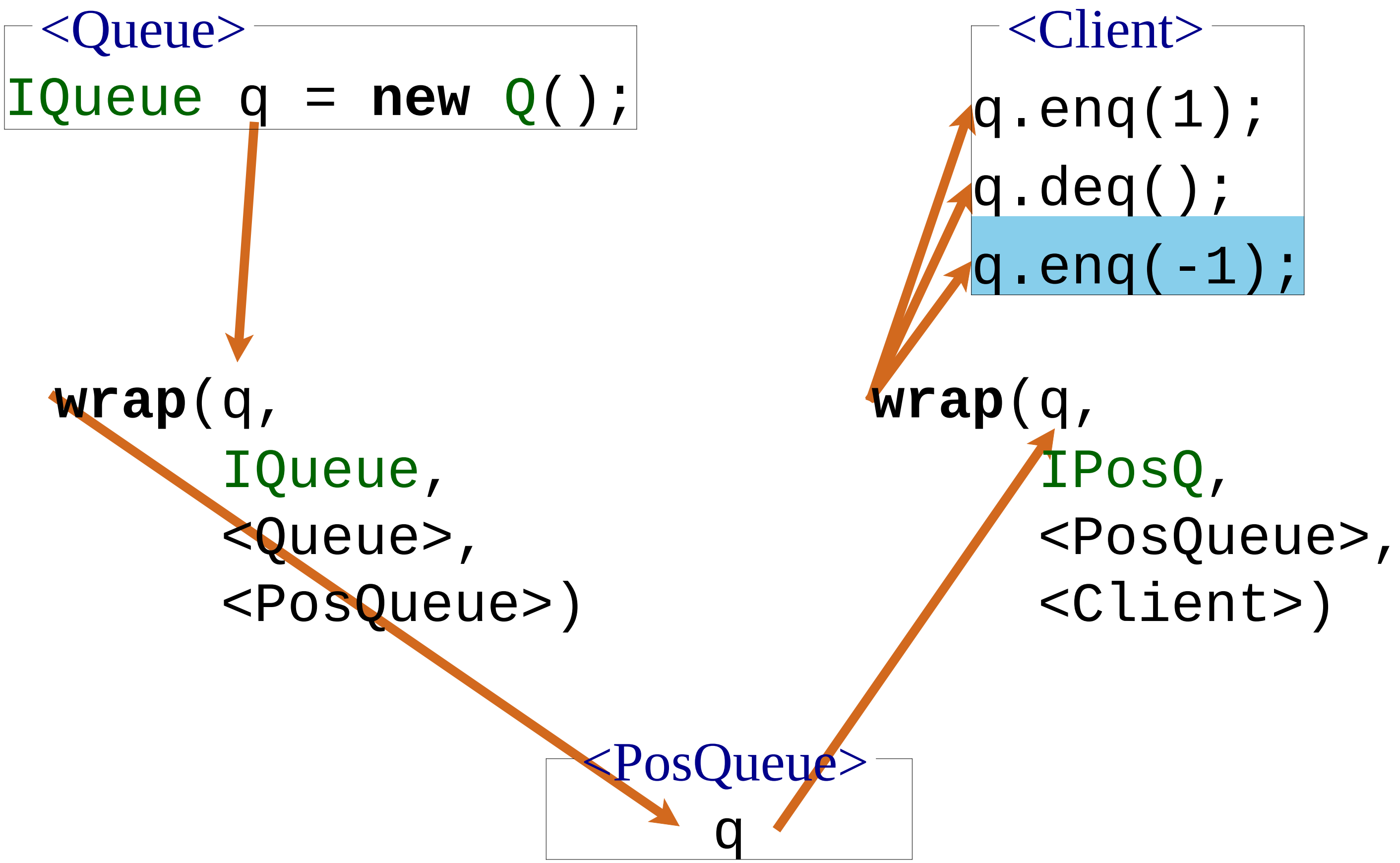
```
q.enq(1);
q.deq();
q.enq(-1);
```

```
wrap(q,
     IQueue,
     <Queue>,
     <PosQueue>)
```

```
wrap(q,
     IPosQ,
     <PosQueue>,
     <Client>)
```

<PosQueue>

```
q
```



<Queue>

```
IQueue q = new Q();
```

<Client>

```
q.enq(1);
q.deq();
q.enq(-1);
```

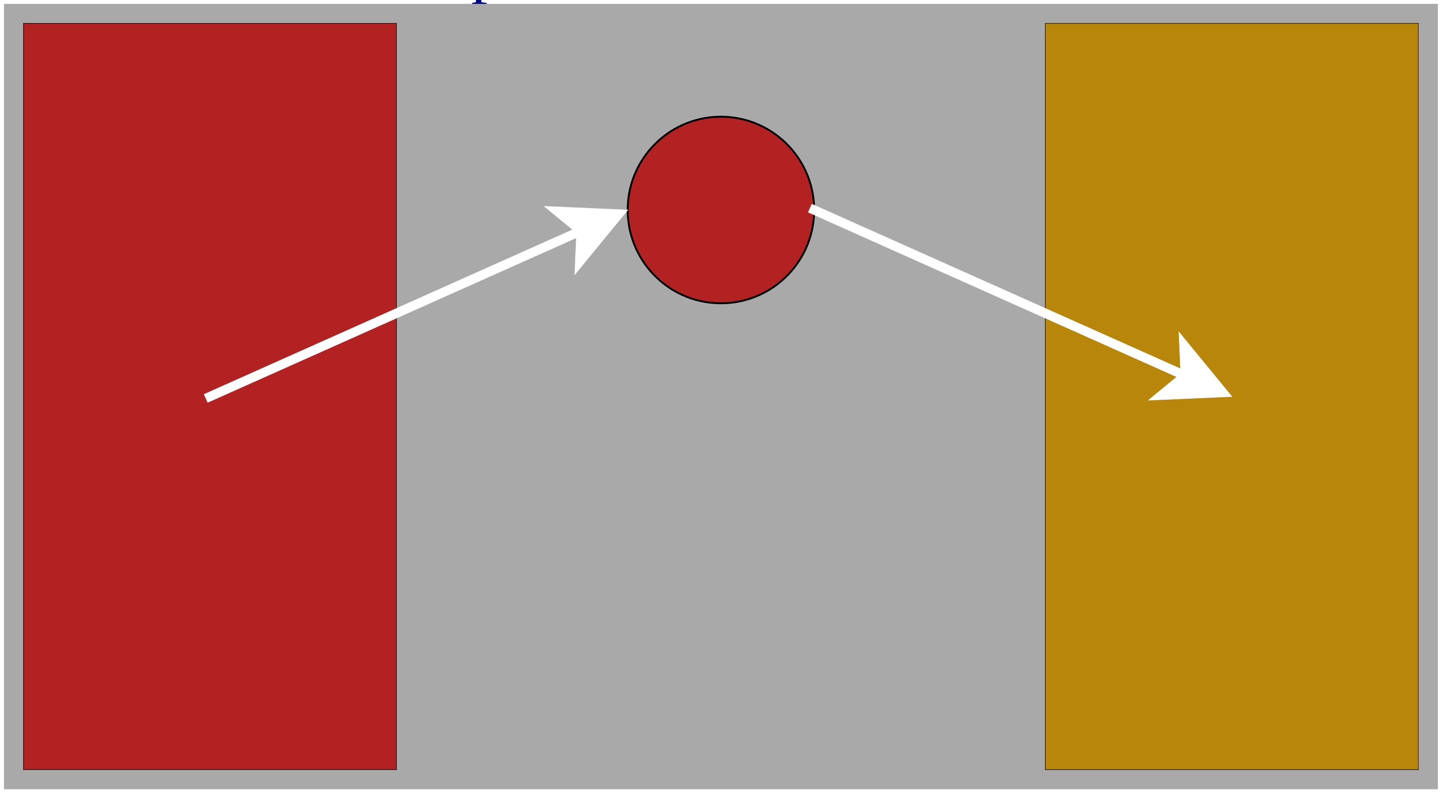
```
wrap(q,
     IQueue,
     <Queue>,
     <PosQueue>)
```

```
wrap(q,
     IPosQ,
     <PosQueue>,
     <Client>)
```

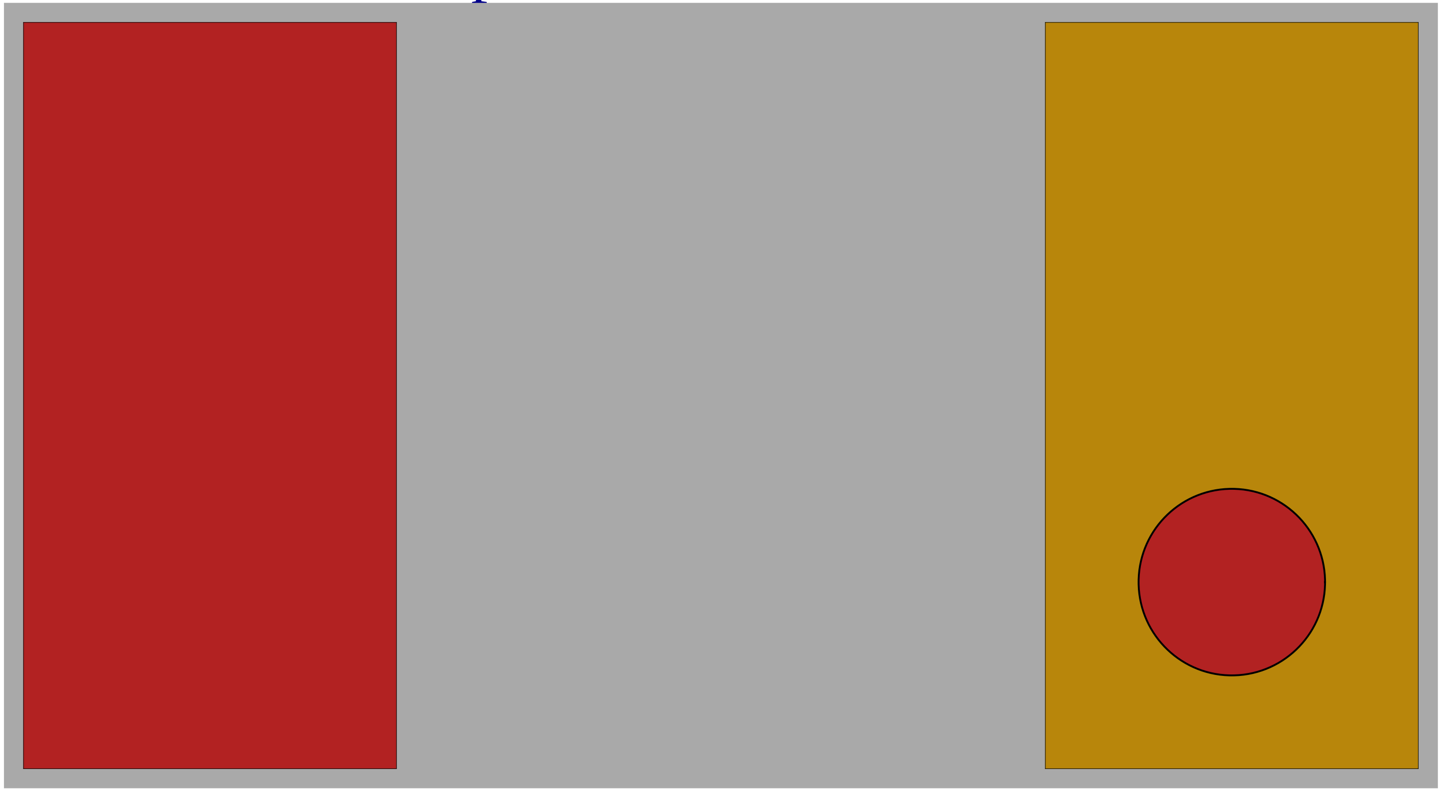
<PosQueue>

```
q
```

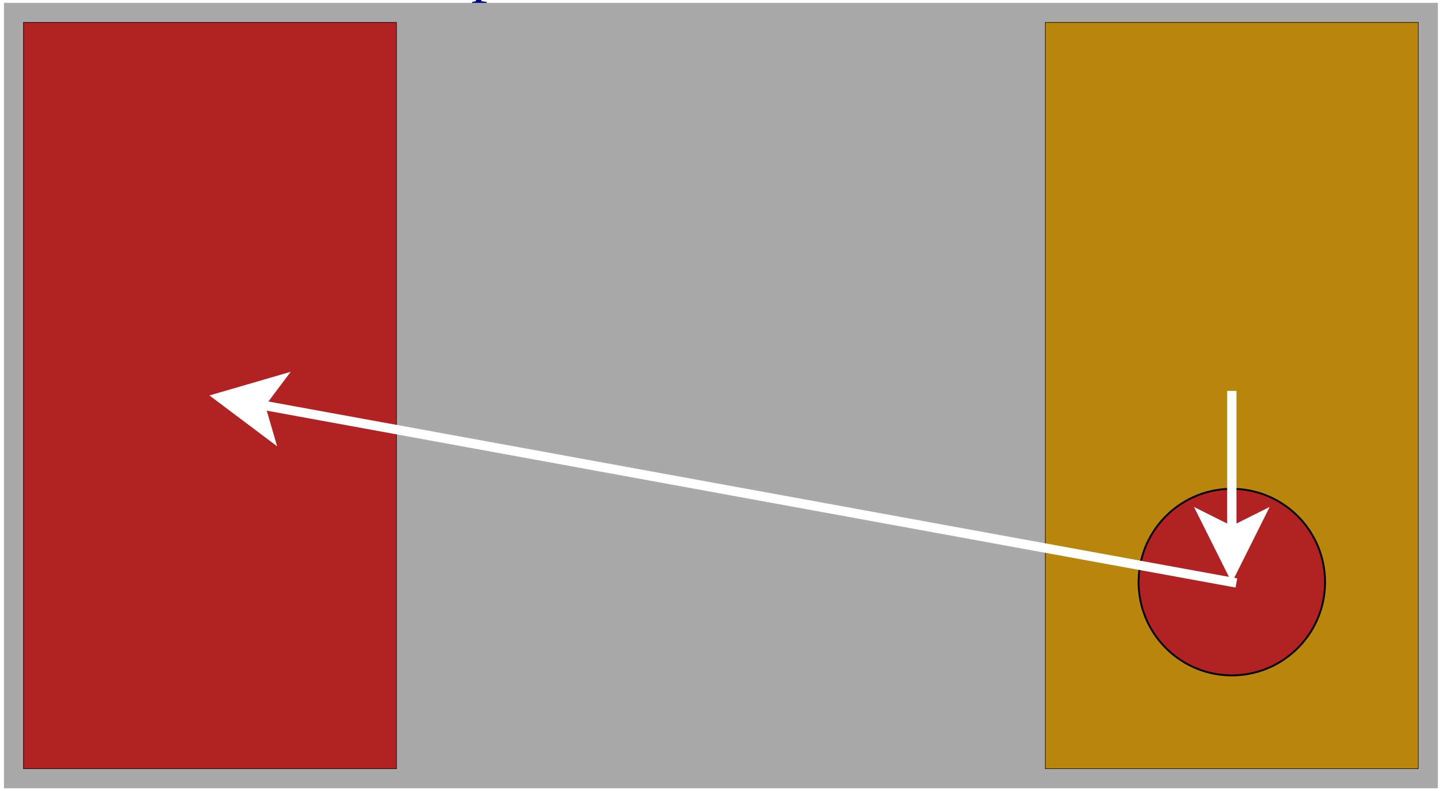
Semantics of wrap



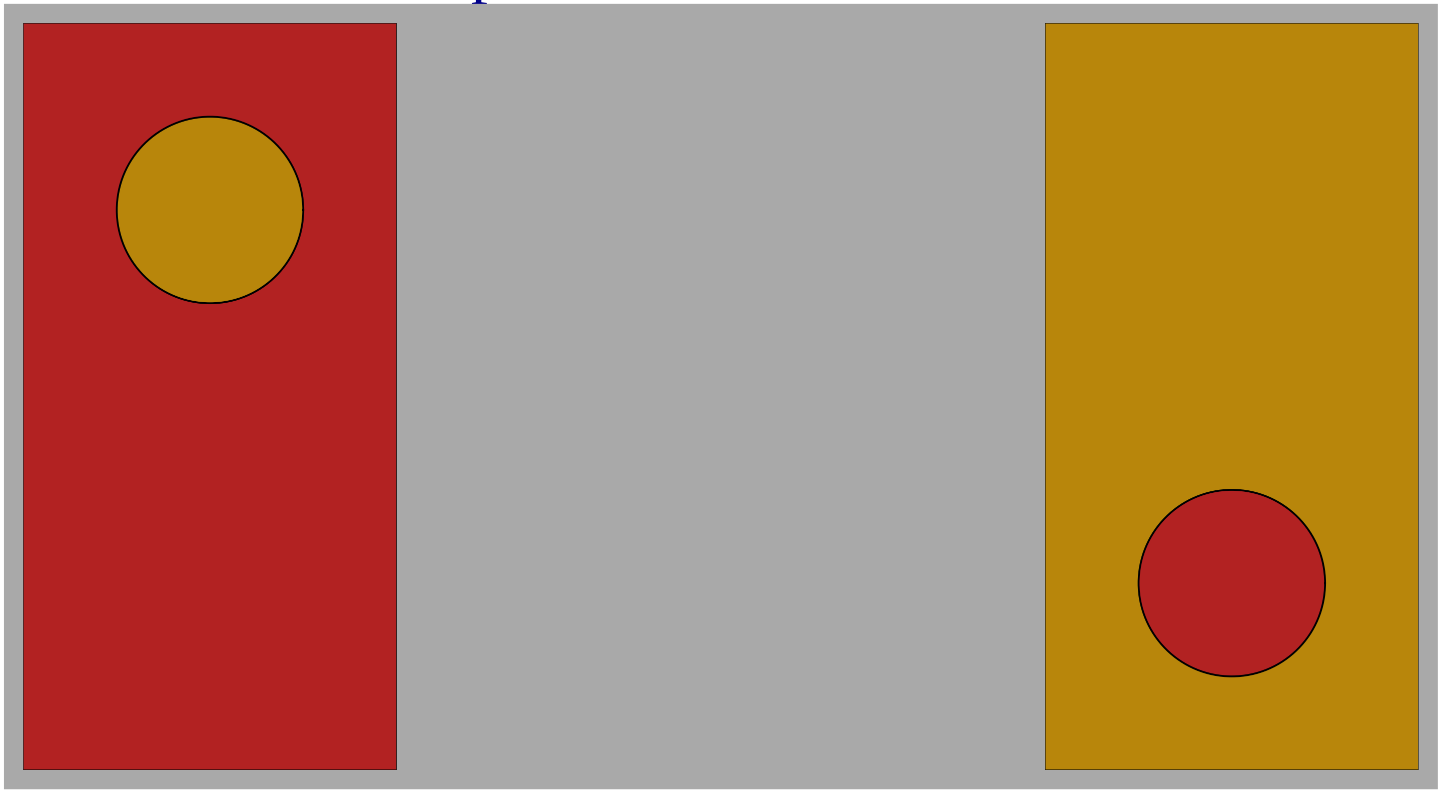
Semantics of wrap



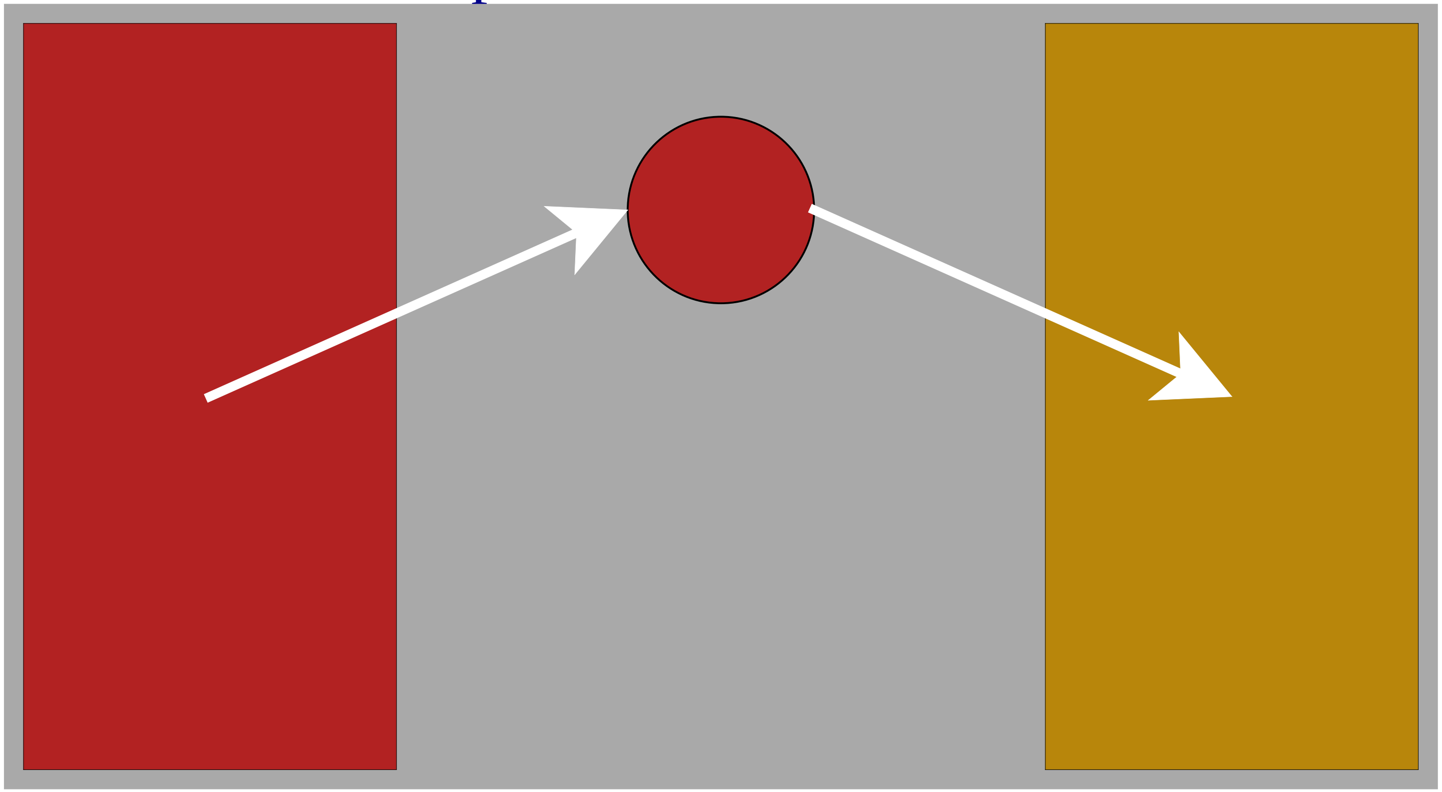
Semantics of wrap



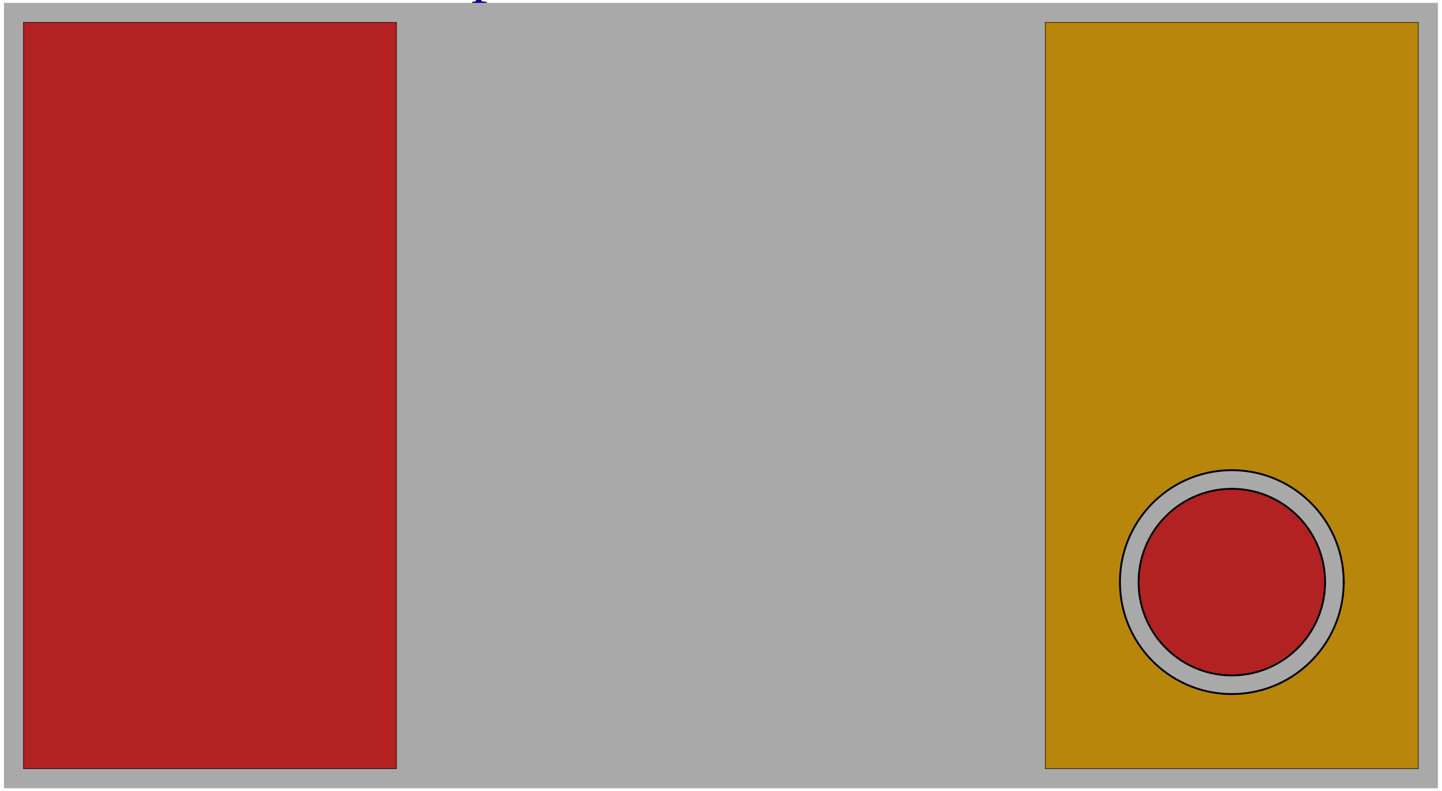
Semantics of wrap



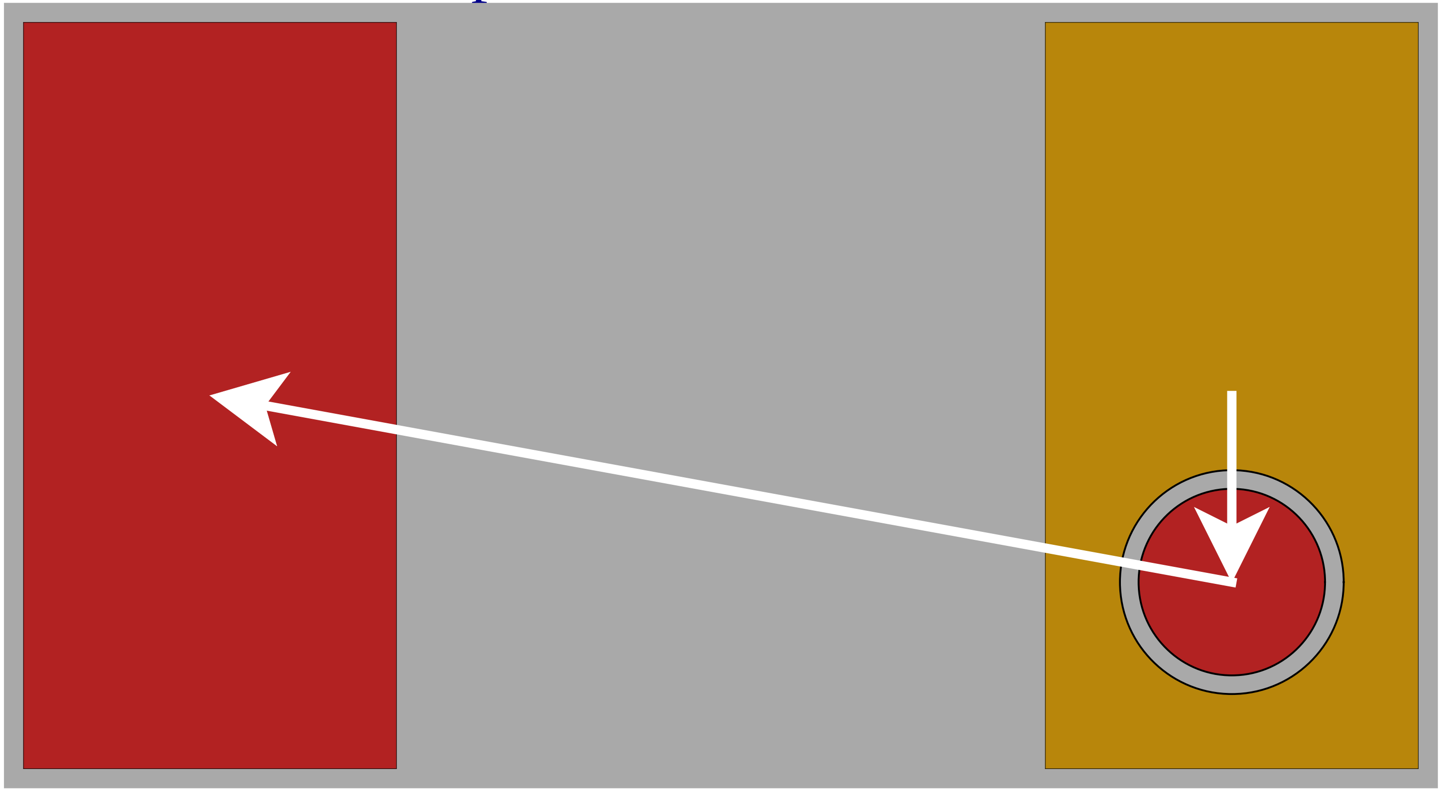
Semantics of wrap



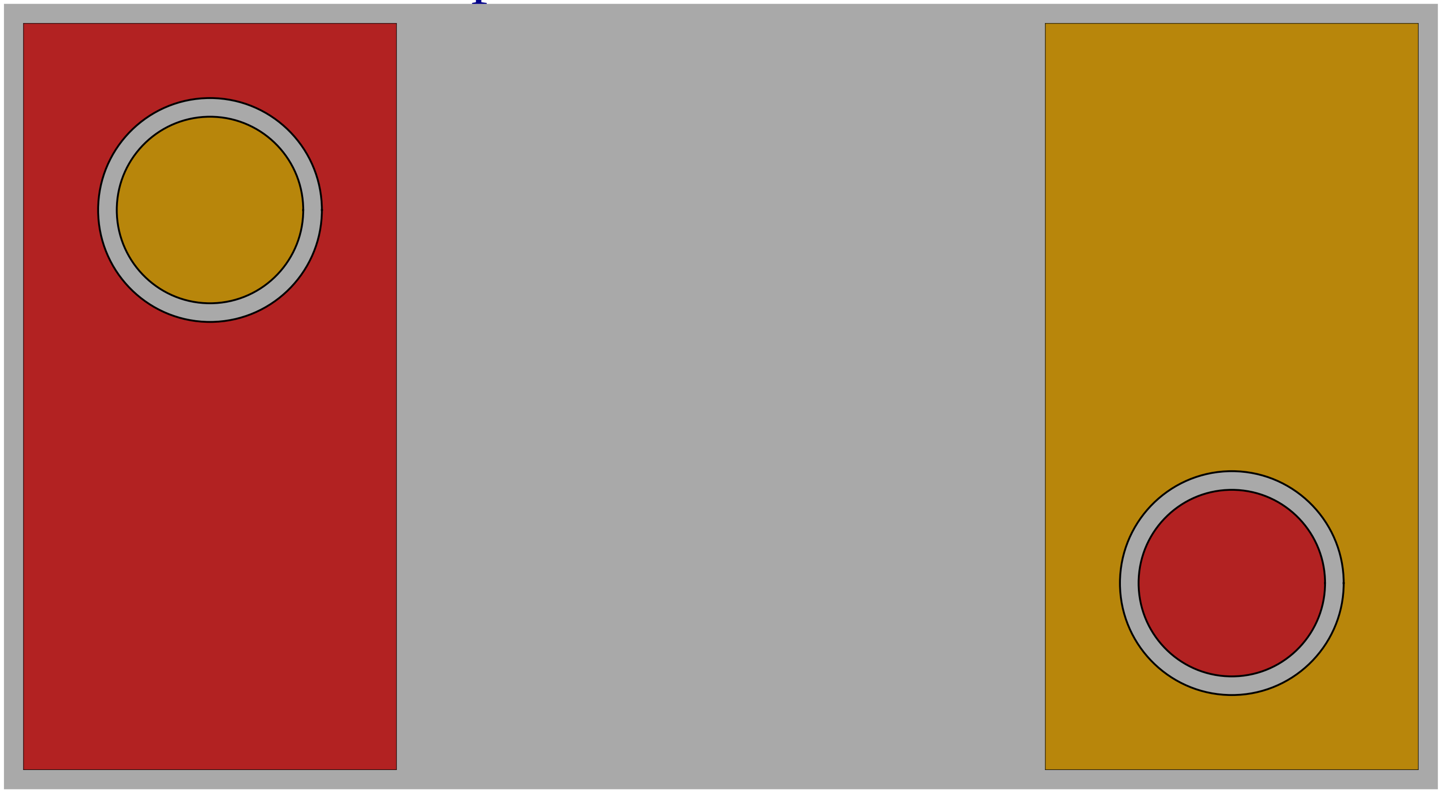
Semantics of wrap



Semantics of wrap



Semantics of wrap



```
wrap(o, I, <from>, <to>).m(o’)
=
wrap(o.m(wrap o’, J, <to>, <from>),
     K,
     <from>,
     <to>)

interface I { K m(J x); }
interface J { ... }
interface K { ... }
```

Implementation

- Proxies
- Construct new proxies at method calls

```
interface I { A1 m(B1 x); }
interface J { A2 m(B2 x); }
J o = ...
I o = wrap(o, I, <frm>, <to>)
```

```
interface I { A1 m(B1 x); }
interface J { A2 m(B2 x); }
J o = ...
I o = new JtoI(o, "frm", "to")
```

```
interface I { A1 m(B1 x); }
interface J { A2 m(B2 x); }
J o = ...
I o = new JtoI(o, "frm", "to")

class JtoI implements I {
  J o; String frm; String to;

  JtoI(J o, String frm, String to) {
    this.o=o; this.frm=frm; this.to=to;
  }
}
```

```
interface I { A1 m(B1 x); }
interface J { A2 m(B2 x); }
J o = ...
I o = new JtoI(o, "frm", "to")

class JtoI implements I {
  J o; String frm; String to;

  A1 m(B1 x) {
    // check J pre-conditions, blame to
    B2 b2 = new B1toB2(x, to, frm);
    A2 a2 = o.m(b2);
    A1 res = new A2toA1(a2, frm, to);
    // check J post-conditions, blame frm
    return res;
  }
}
```

Object identity

- Proxied objects are not == to originals
- Introduce a new form of equality that unwraps the objects
- More expensive, not yet a problem in DrScheme

Wrap up

Structural subtyping for contracts

- Adds flexibility to conventional languages
- Provides mechanism for assigning blame
- Still simple expressions of type boolean

Thank you.