



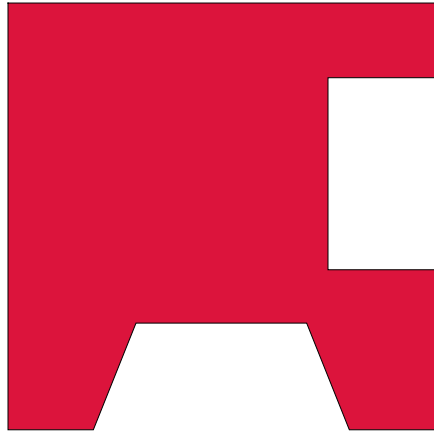
Late Binding for Software Contracts

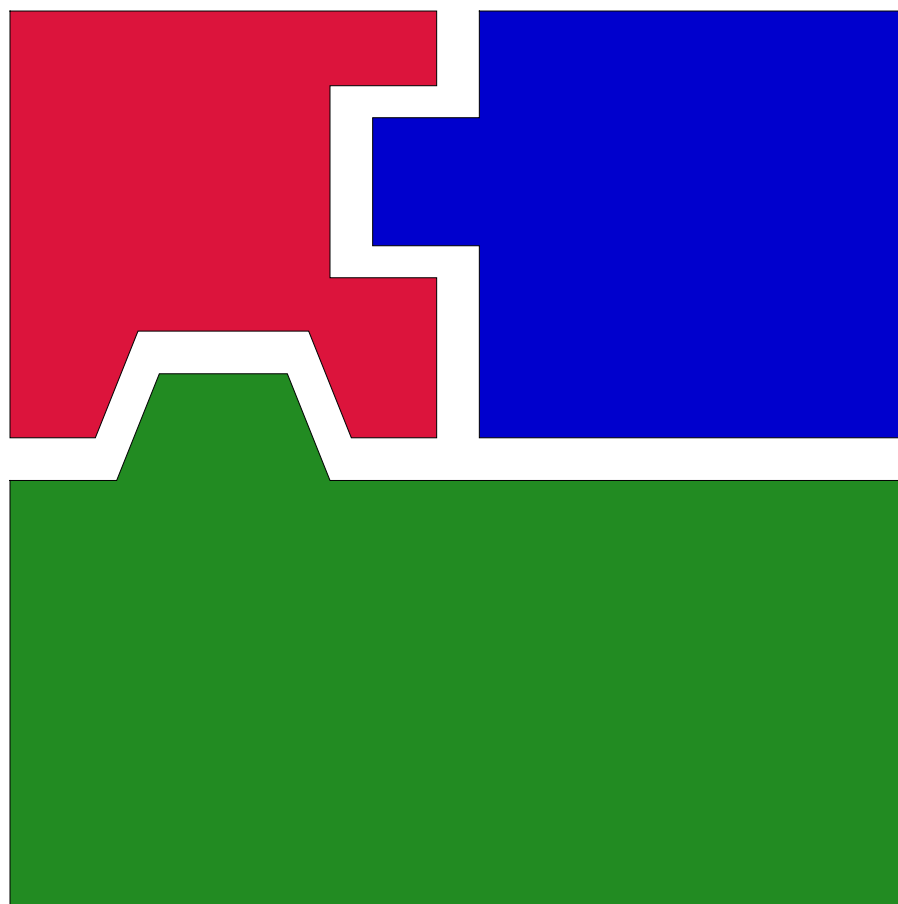
Robby Findler
University of Chicago

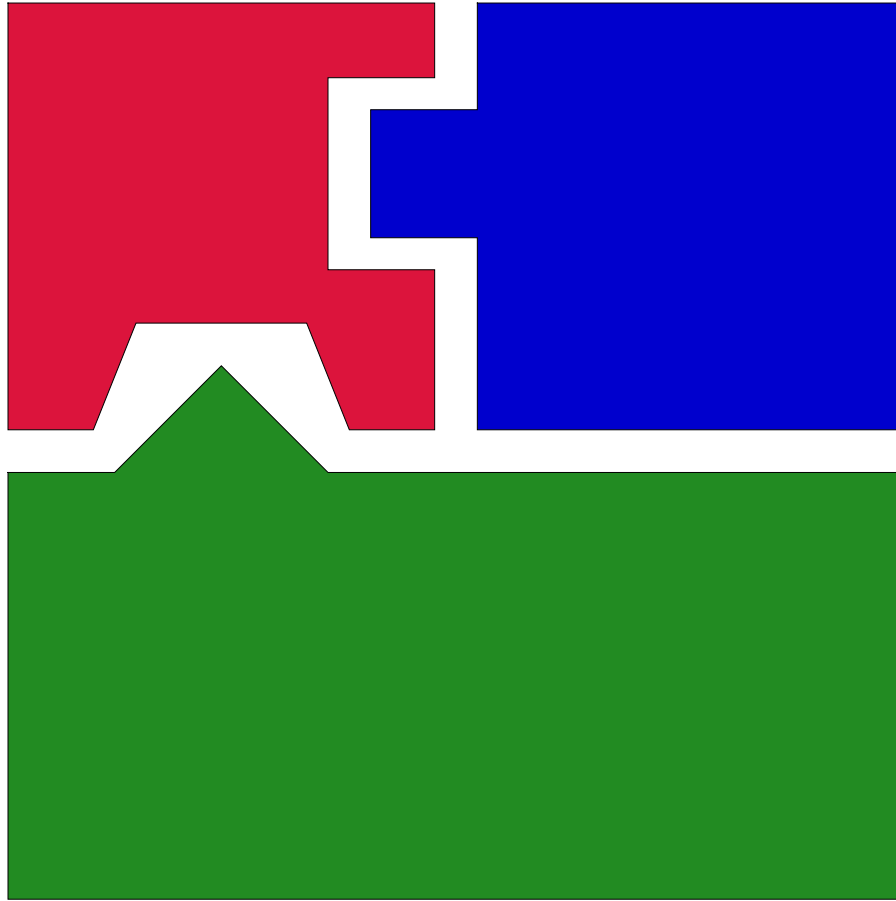


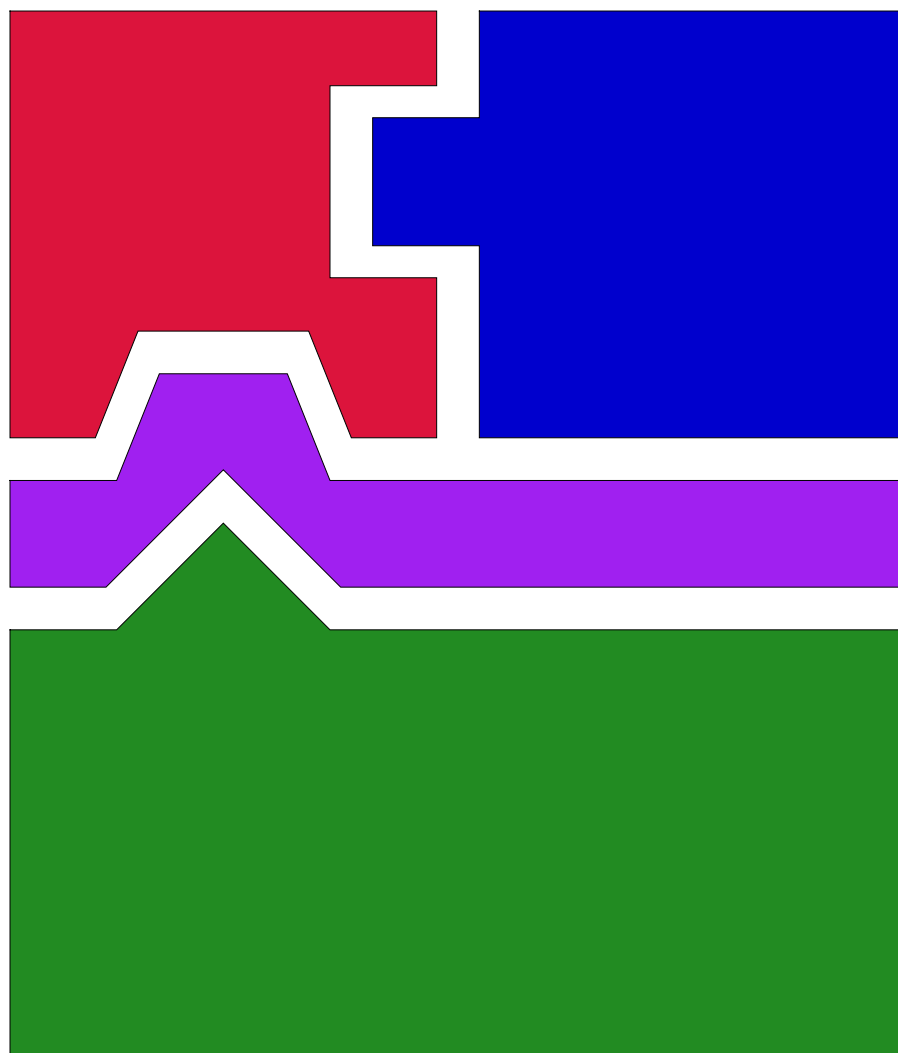
Component marketplace

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

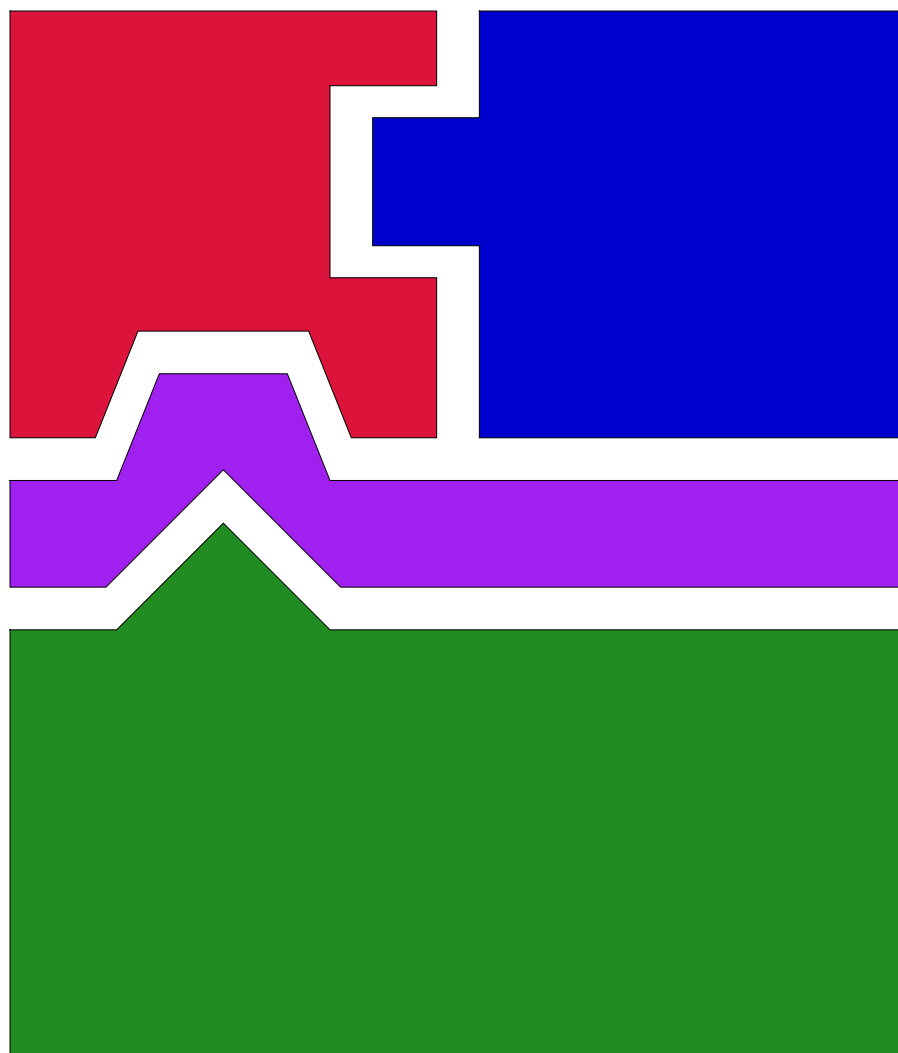


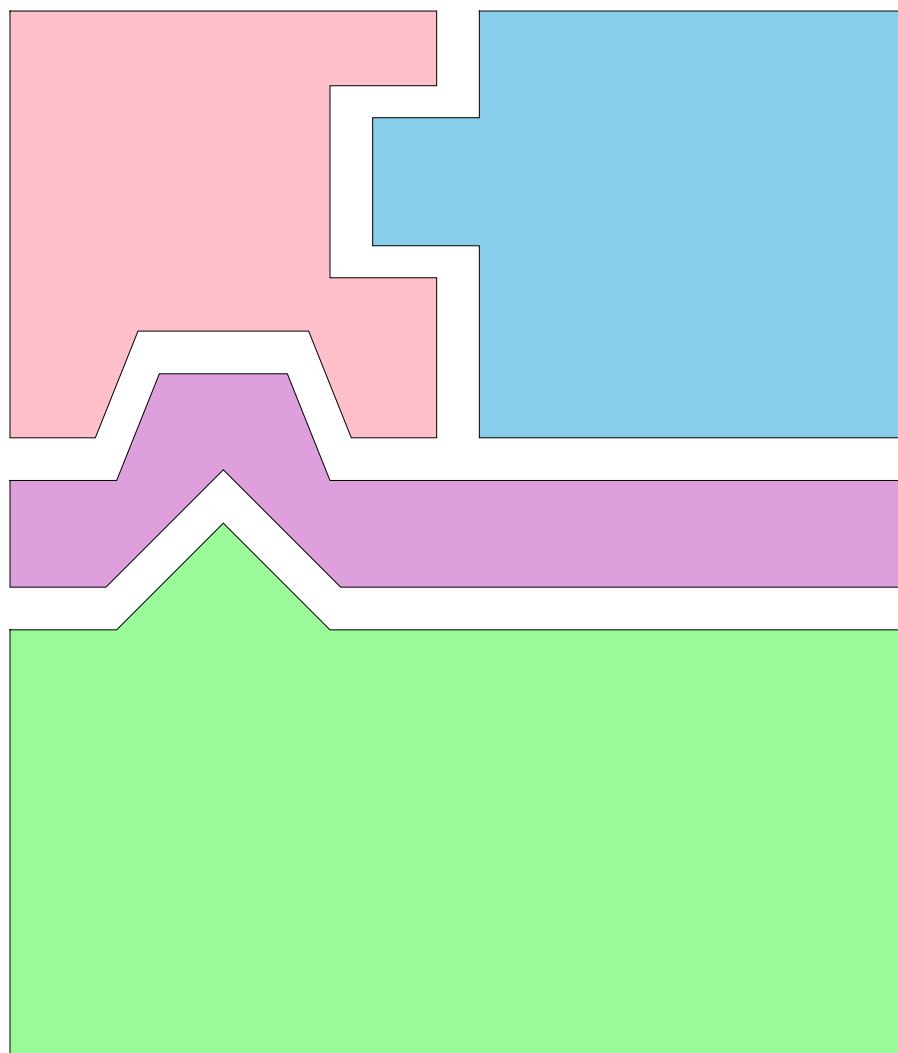


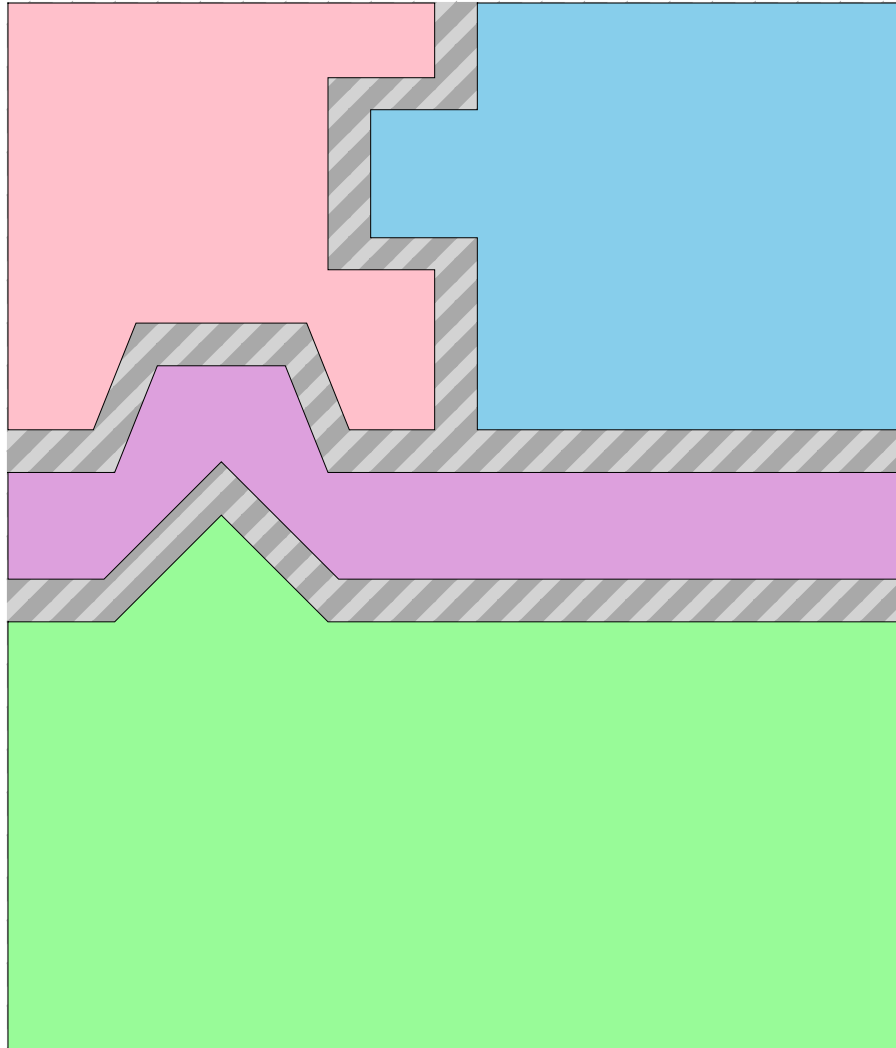














What is a contract?

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



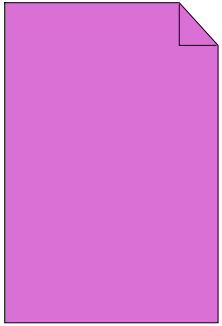
Why check contracts?

- Find faulty components
- Accountability supports component economy

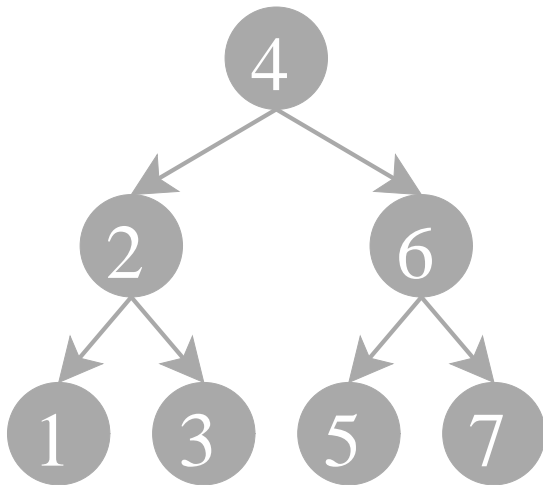
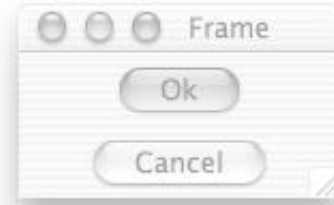


Contracts [Beugnard et al. 1999]

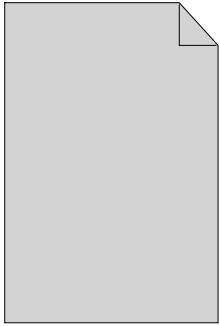
- **Syntactic: types**
int f(int[] x, int k)
- **Semantic level 1: behavioral**
int f(int[] x, int k) // $0 \leq k < x.length()$
- **Semantic level 2: sequencing, concurrency**
finalize is called for all objects
- **Quality of service: space, time**
web server handles at least 1000 GET/sec



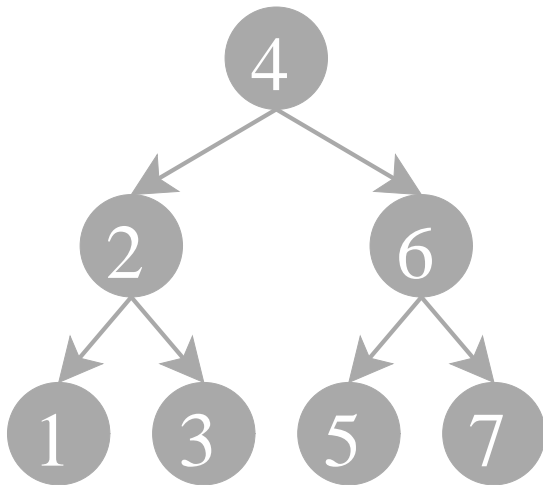
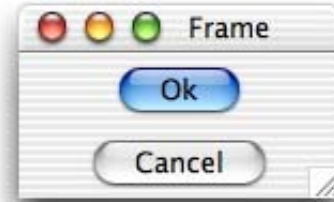
```
fopen( ) ;  
...  
fputs( ) ;
```



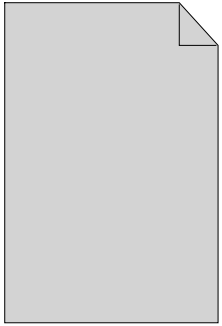
```
1/0  
a[i+1]  
o.m( )
```



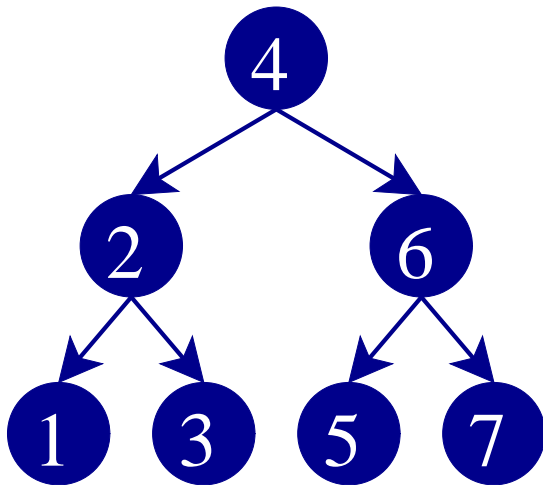
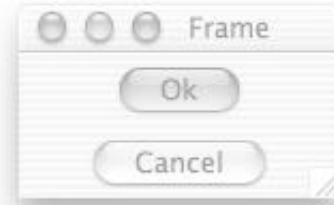
```
fopen( ) ;  
...  
fputs( ) ;
```



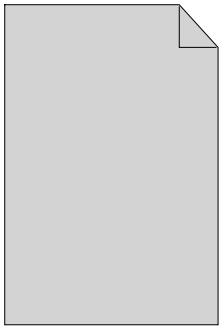
```
1 / 0  
a[i+1]  
o.m( )
```



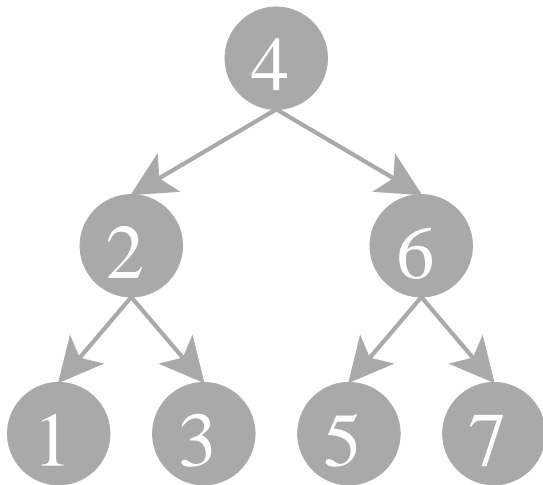
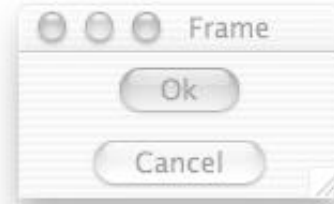
```
fopen( ) ;  
...  
fputs( ) ;
```



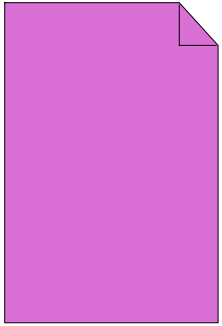
```
1/0  
a[i+1]  
o.m( )
```



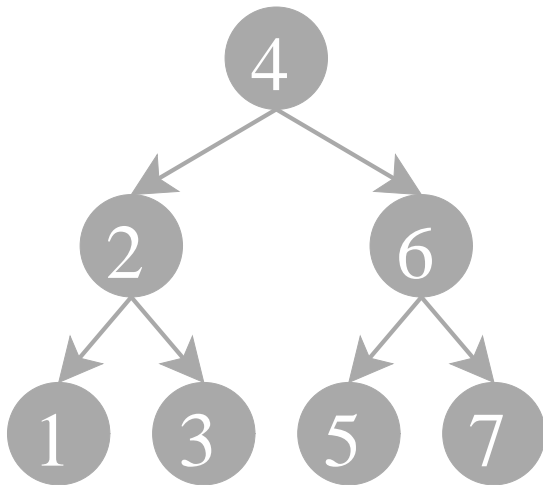
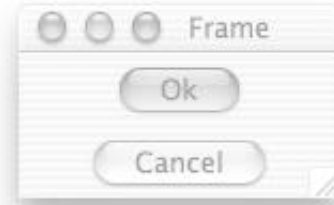
```
fopen( ) ;  
...  
fputs( ) ;
```



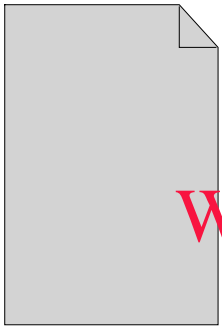
```
1 / 0  
a[i+1]  
o.m( )
```



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



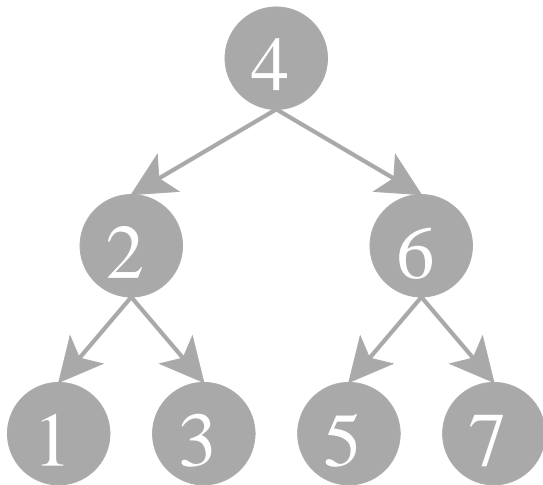
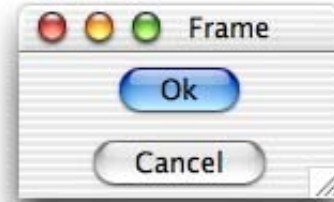
```
1/0  
a[i+1]  
o.m( )
```



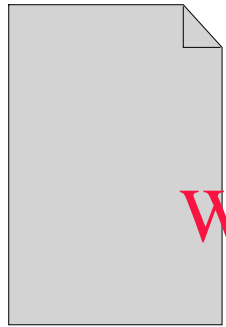
File
Writer

```
fopen( ) ;
```

```
fputs( ) ;
```

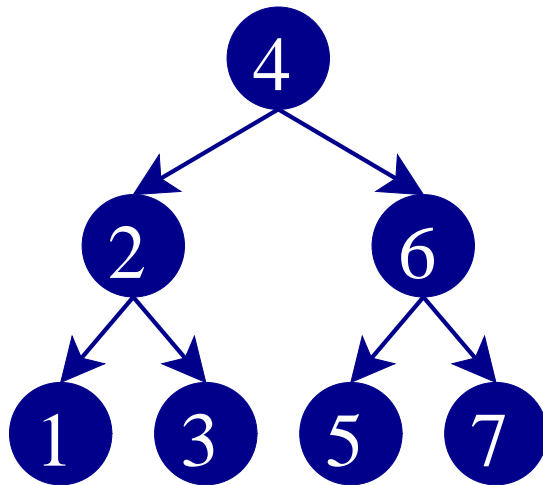
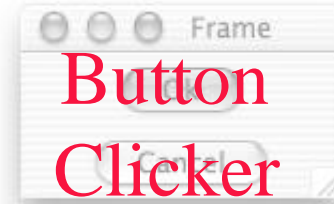


```
1/0  
a[i+1]  
o.m( )
```

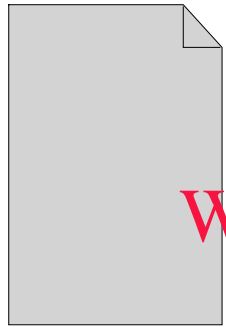


File
Writer

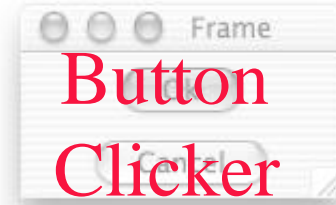
```
fopen( ) ;  
fputs( ) ;
```



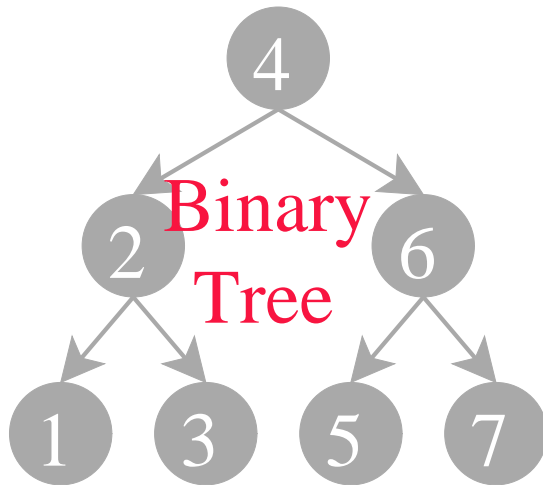
```
1 / 0  
a[i+1]  
o.m( )
```



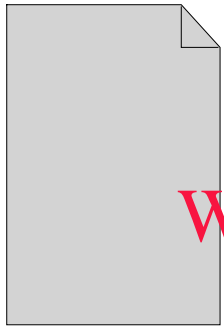
**File
Writer**
`fopen() ;`
`fputs() ;`



**Button
Clicker**

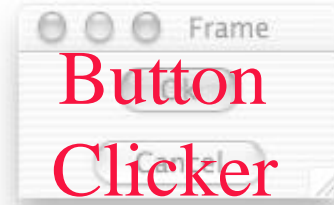


`1 / 0`
`a[i+1]`
`o.m()`

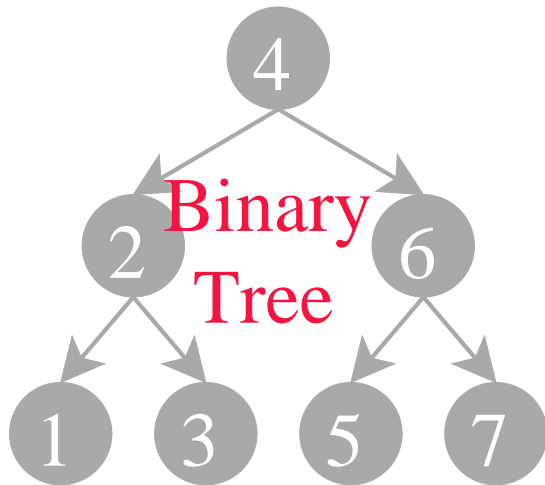


**File
Writer**

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



**Button
Clicker**



1 / 0
Program
o . m ()



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 Queue {  
    void enq(int X) {...}  
  
    int deq() {...}  
  
    boolean empty() {...}  
}
```



Queue implementation

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

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

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



Queue implementation



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

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

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

Good client

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

Queue implementation



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

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

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

Good client

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

Bad client

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



Queue implementation

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

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

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

Good client

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

Bad client

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

Blame q.deq(); in Bad Client

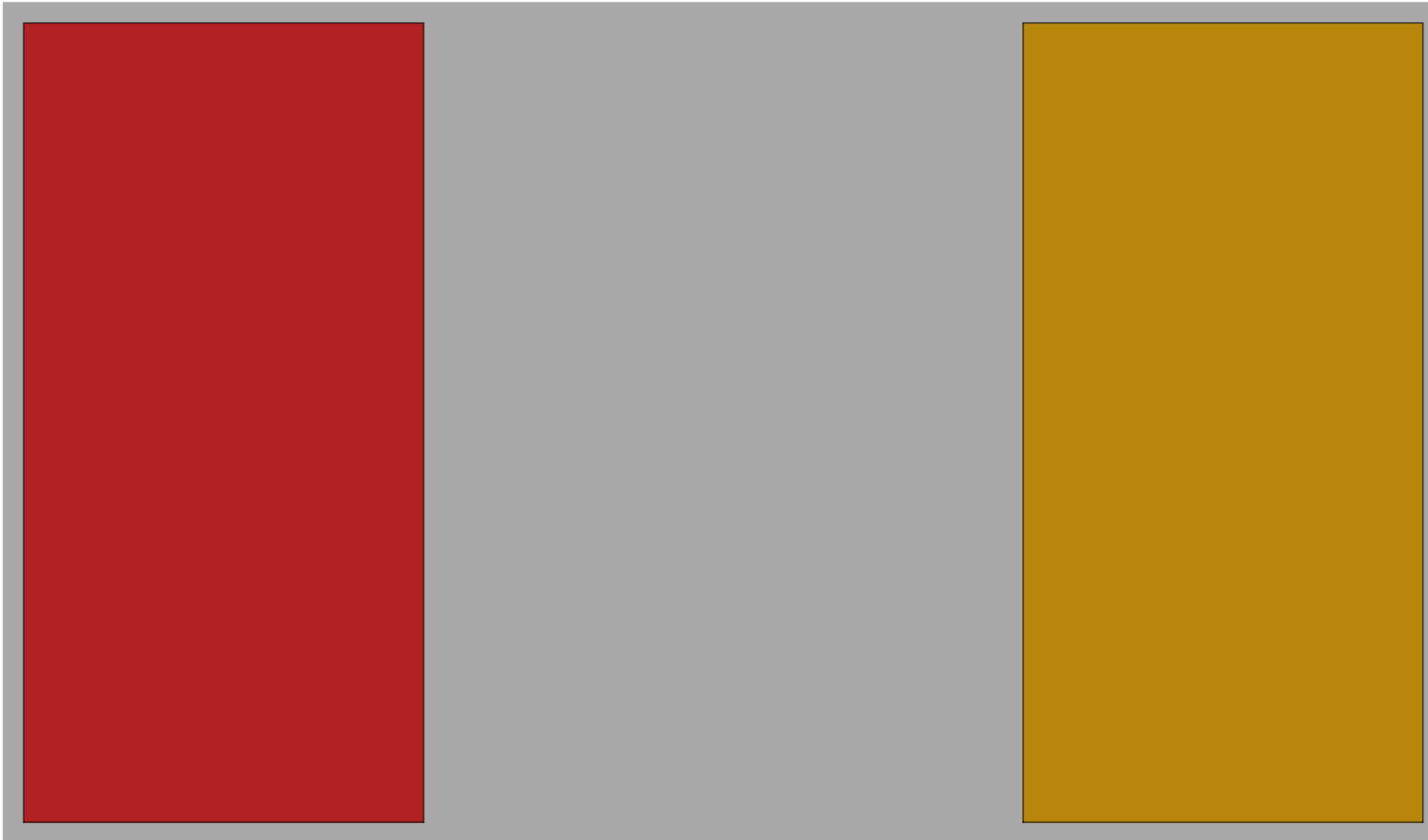


"The effective coupling of two dynamically selected parties via a well-defined interface is a powerful concept called late binding and is right at the heart of object-oriented programming."

Szyperski, 1998

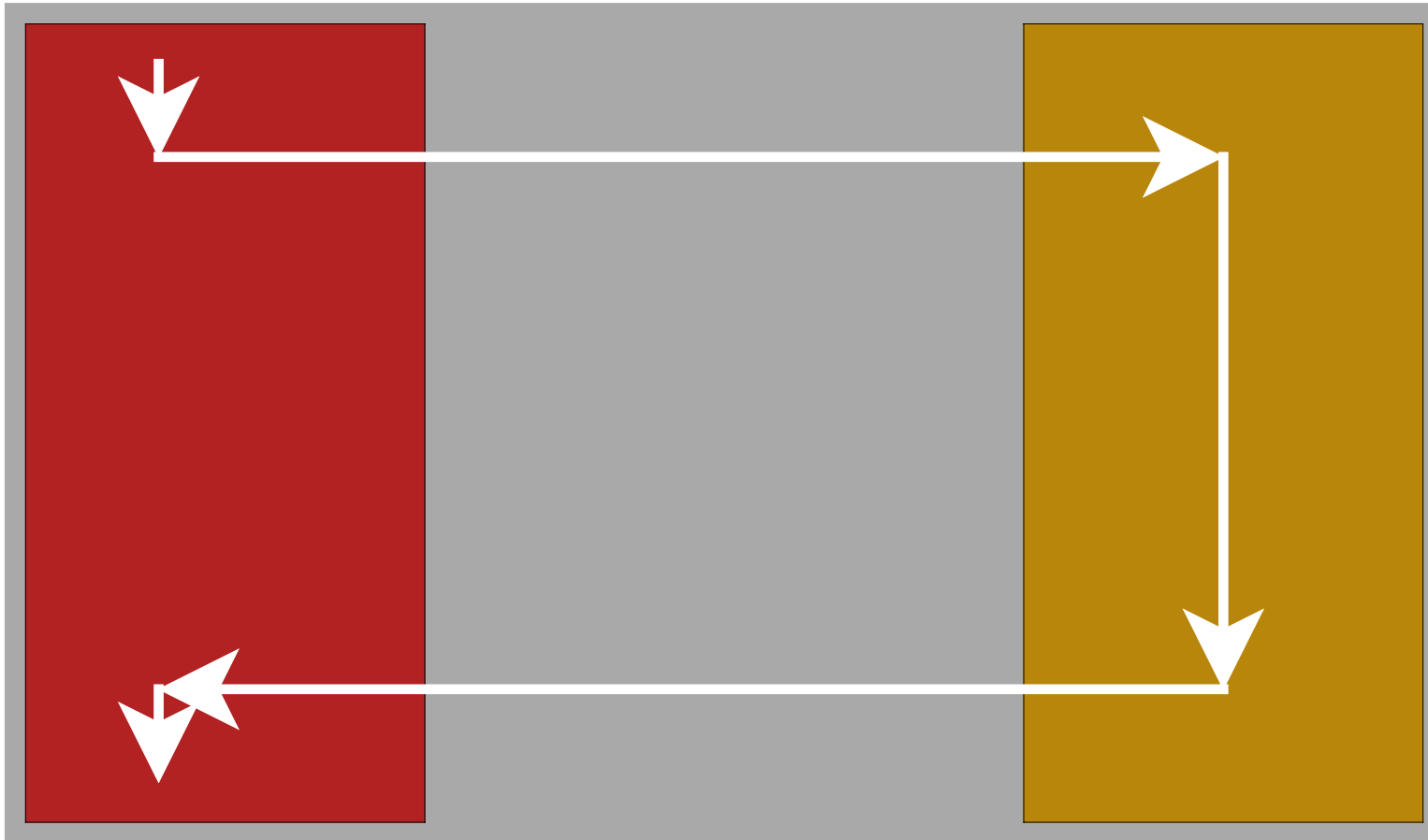


Callbacks





Callbacks





Queue with observer

```
class Q implements Queue {
  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(Queue q)
    {...}
  // @post !q.empty()

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



Good observer

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

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

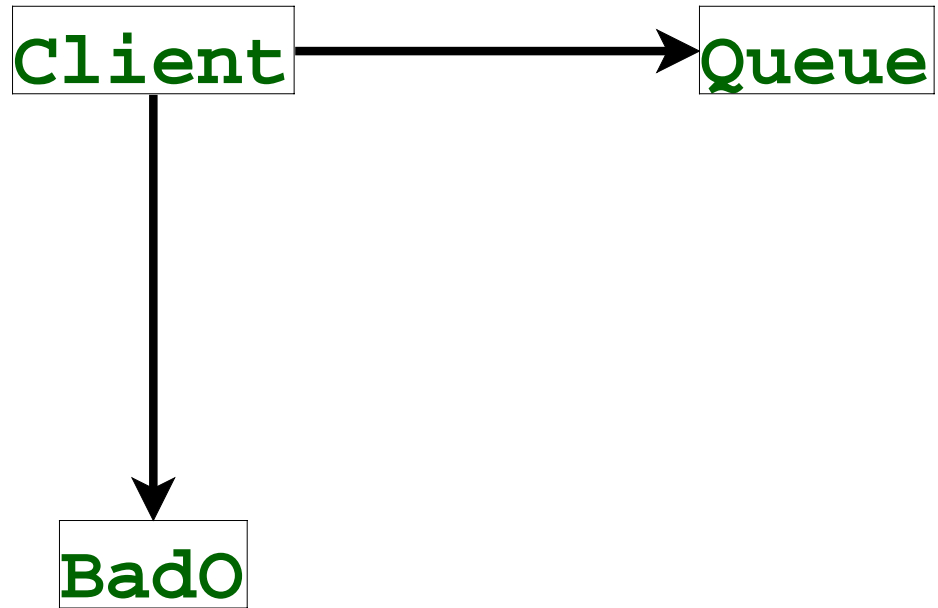
  void onDeq(Queue q)
    {...}
}
```

Bad Observer

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

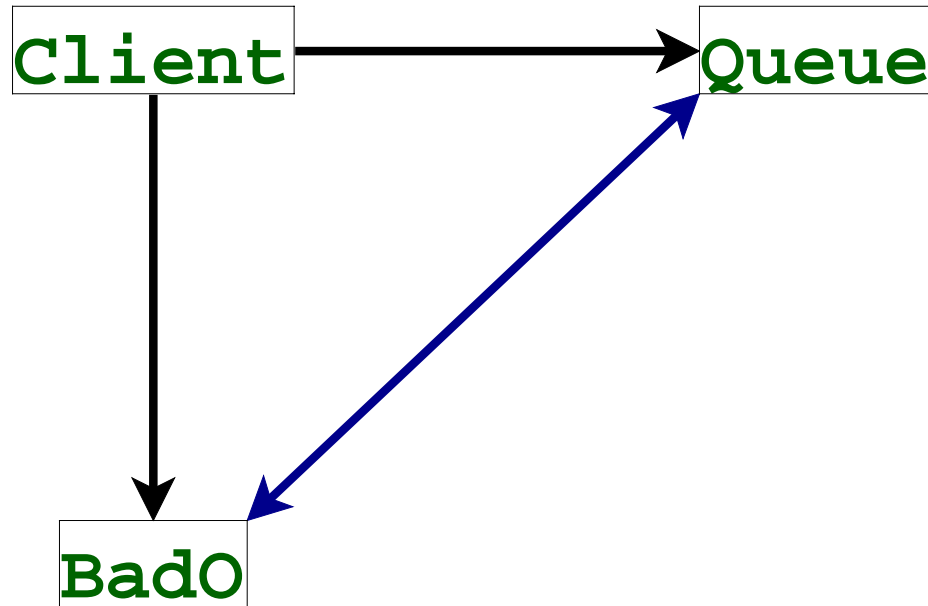
  void onEnq(Queue q)
    { q.deq() }

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



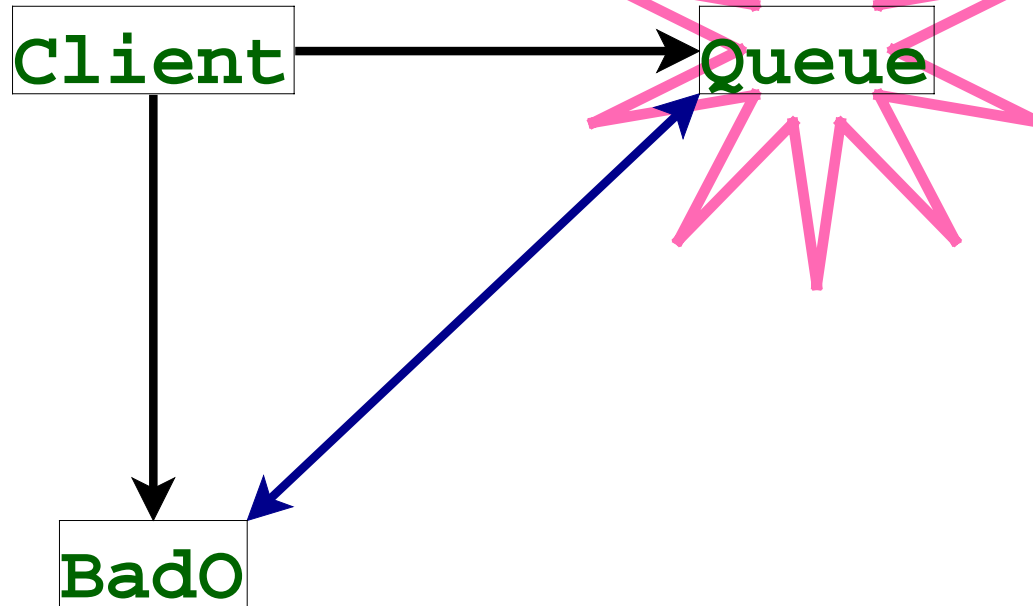


Client links BadO and Queue



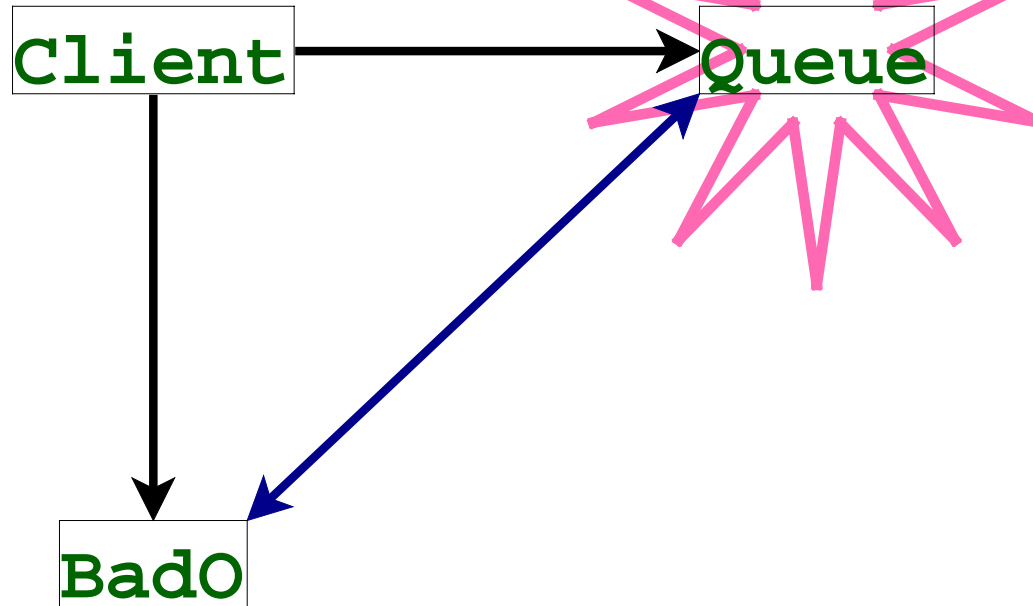


Queue post-condition failure



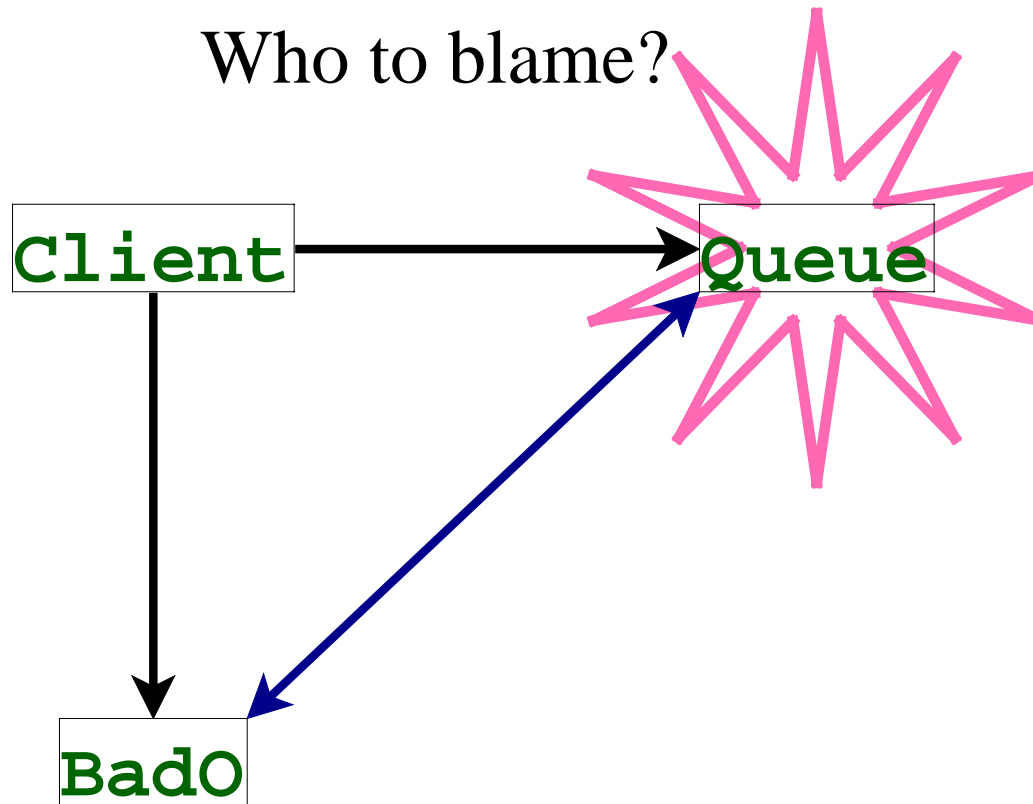


Who to blame?





Who to blame?



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



Queue with observer

```
class Q implements Queue {
    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 Queue {
    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 interface contracts

```
interface Obs {  
    void init();  
  
    void onEnq(Queue q);  
    // @post !q.empty()  
  
    void onDeq(Queue q);  
    // @pre !q.empty()  
}
```

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



Controlling the observer

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

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

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



Controlling the observer

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

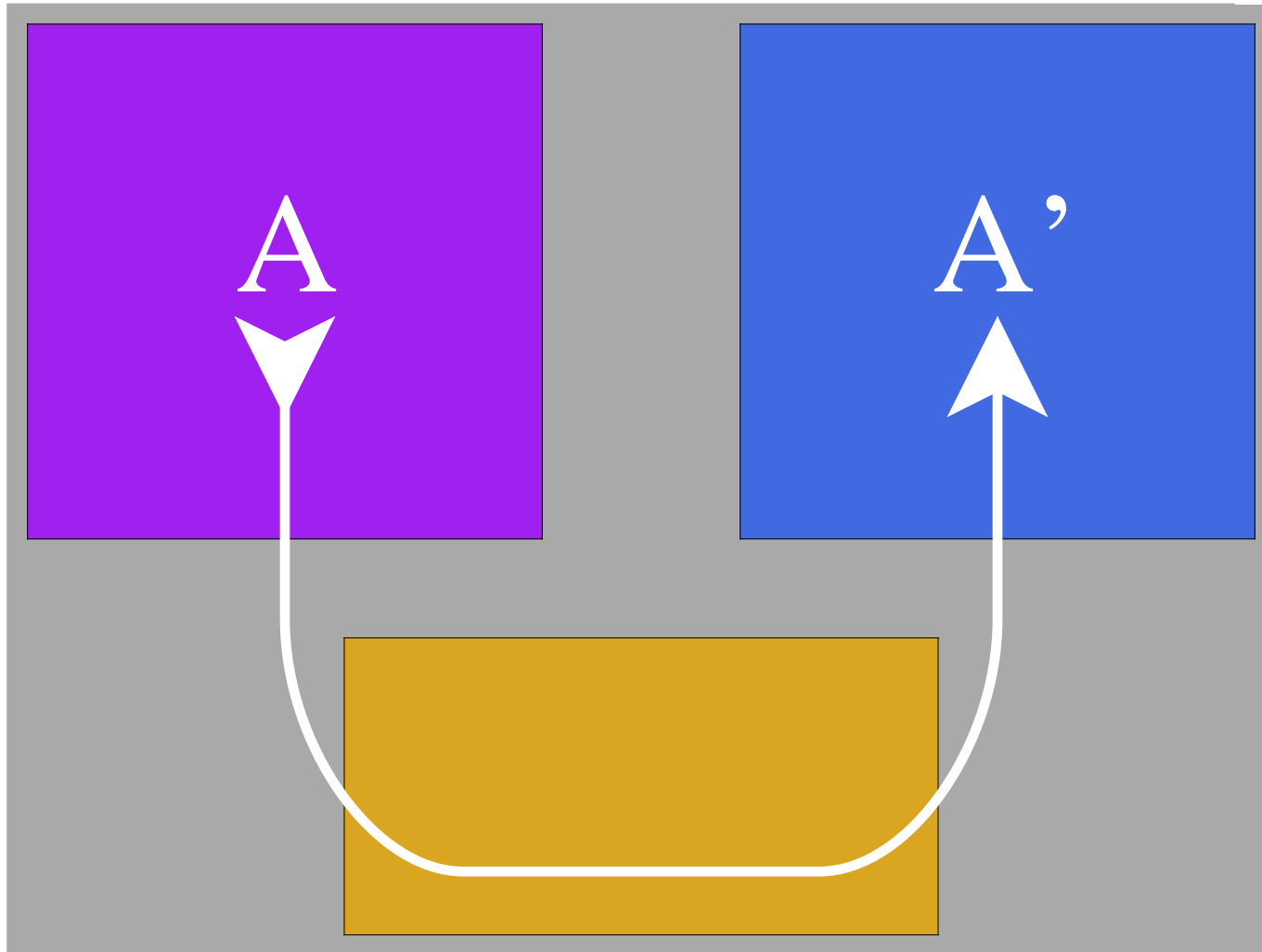
  void onEnq(Queue q)
    { q.deq() }
  // @post !q.empty()

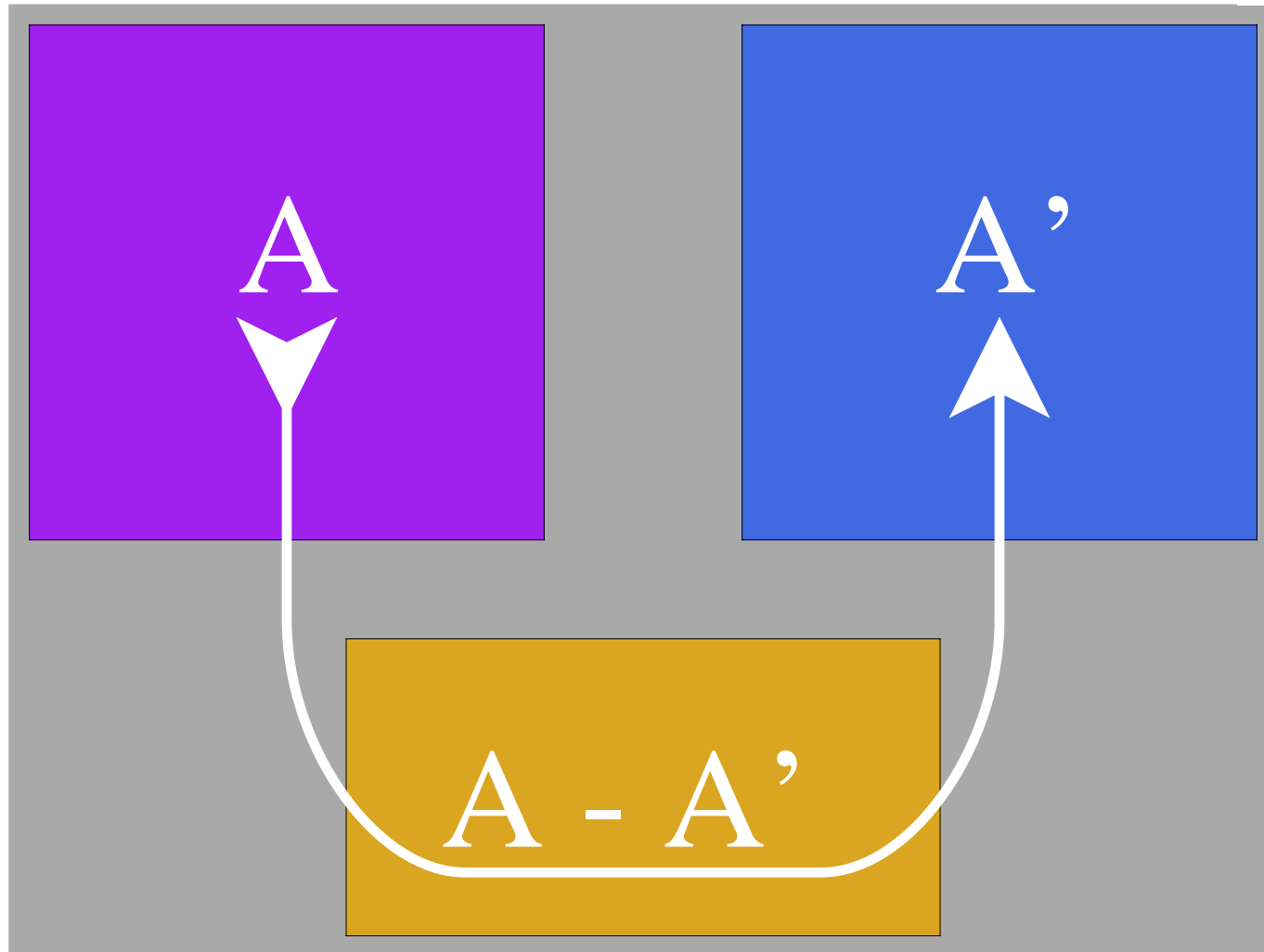
  void onDeq(Queue q)
    {...}
}
```

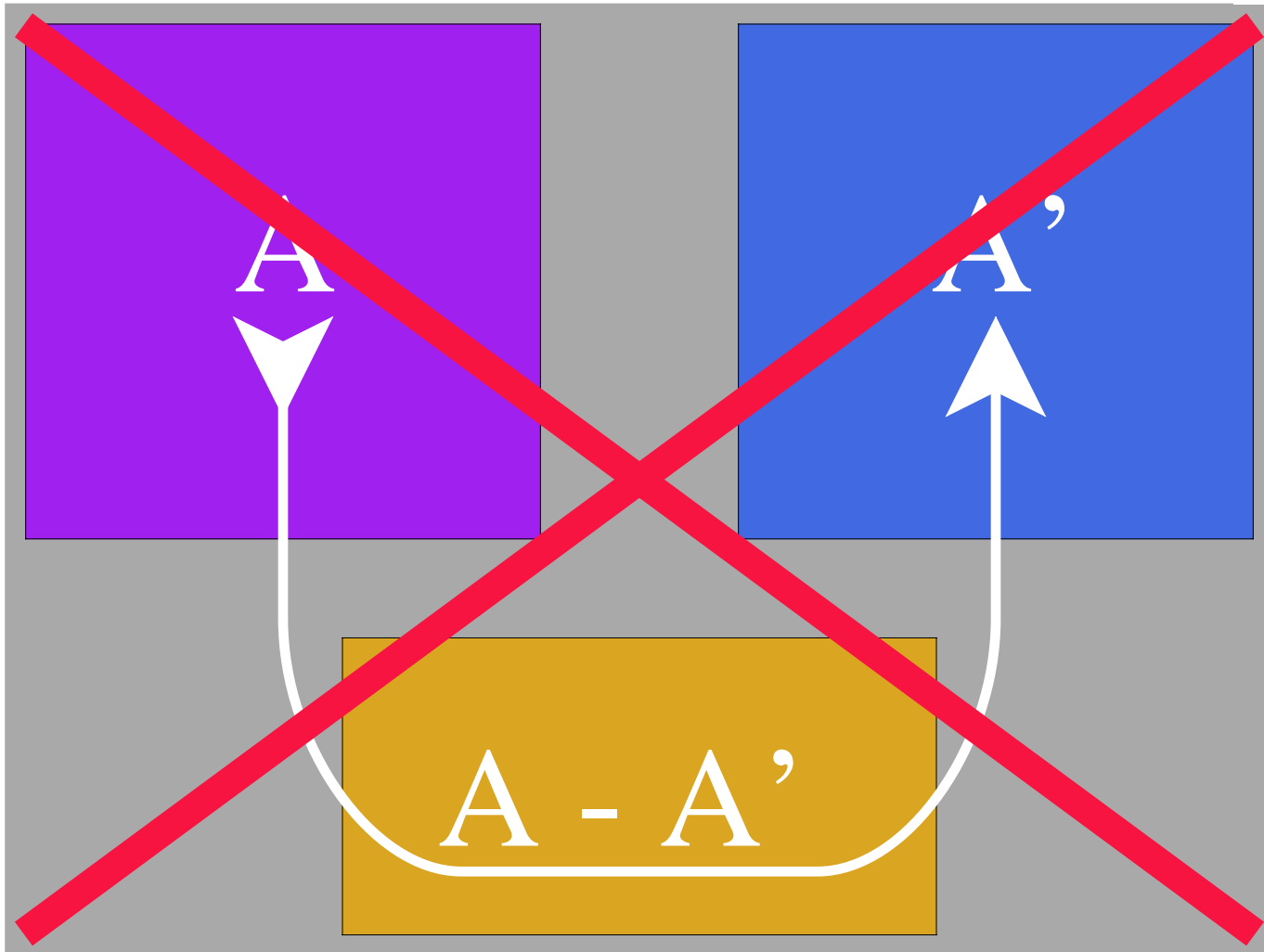


A

A'









Queue Class

```
class Q implements Queue {  
    ...  
}
```

Positive Queue

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



Queue Class

```
class Q implements Queue {  
    ...  
}
```

Positive Queue

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



Late Binding for Contracts



```
wrap(o, I, <name>, <name>)
```

Sits at boundary between
a pair of components



```
wrap(o, I, <name>, <name>)
```

The object that gets
additional contracts



```
wrap(o, I, <name>, <name>)
```

The interface that describes
those contracts



```
wrap(o, I, <name>, <name>)
```

The name of the component
where the object is from;

Responsible for post-conds



```
wrap(o, I, <name>, <name>)
```

The name of the component
where the object is sent;

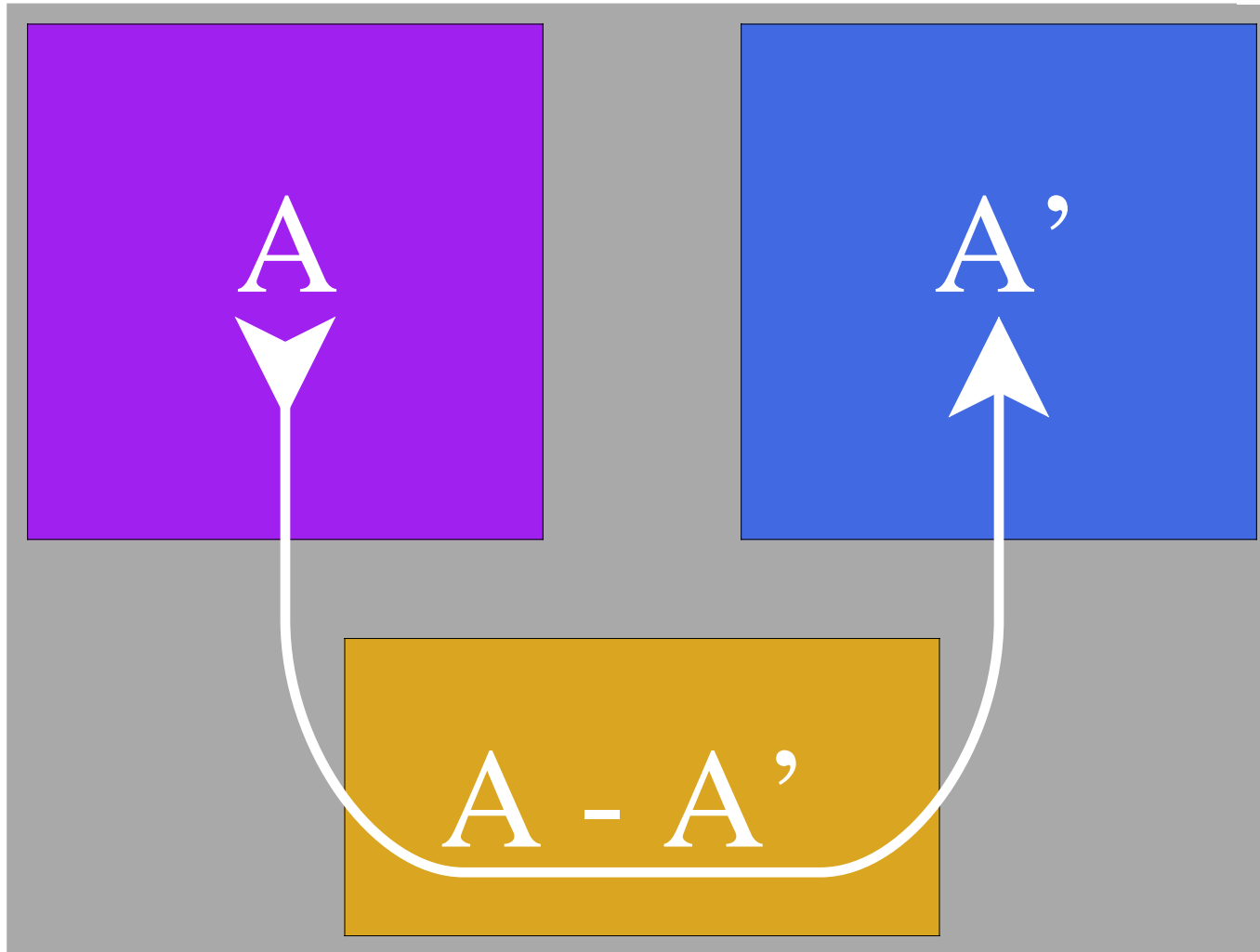
Responsible for pre-conds

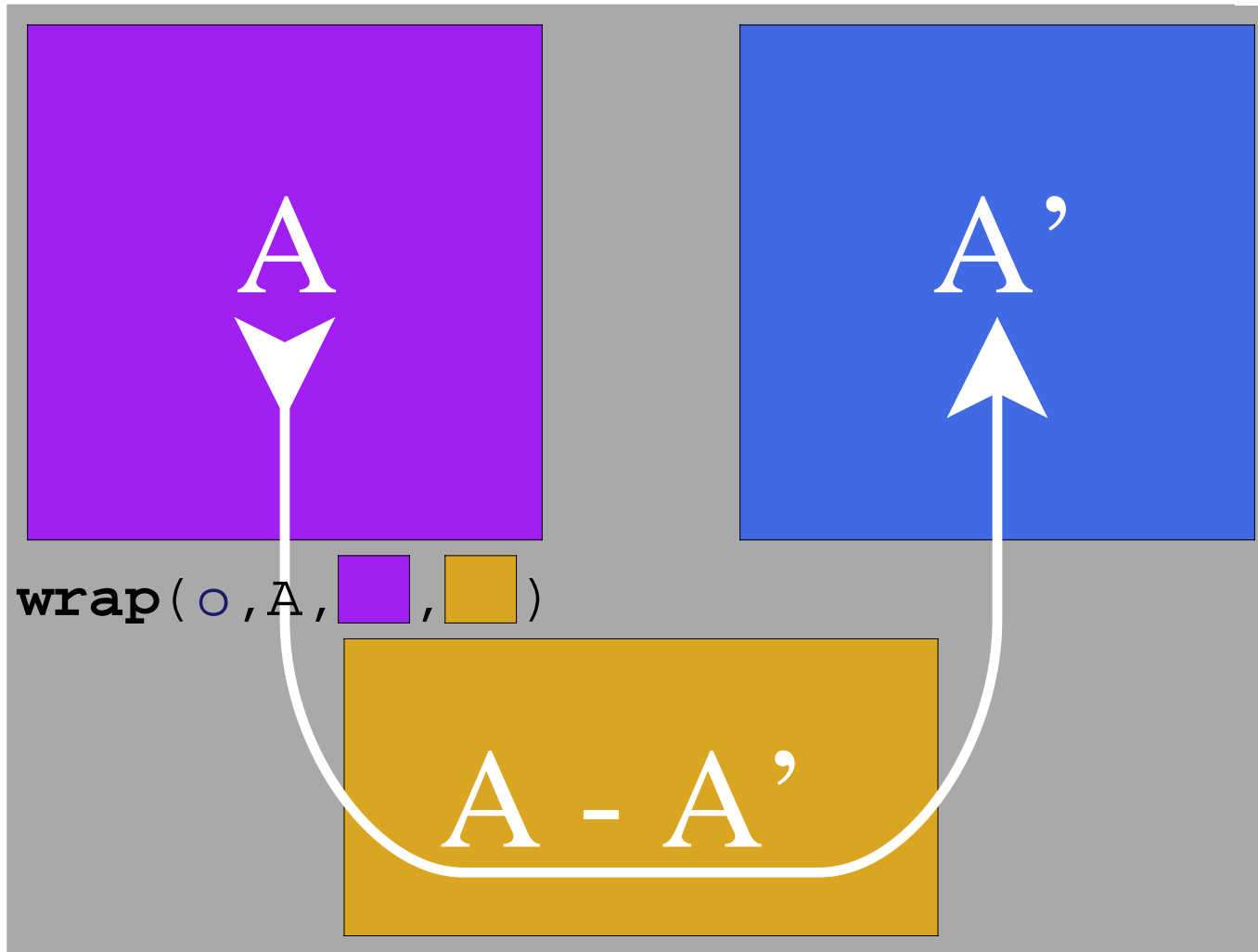


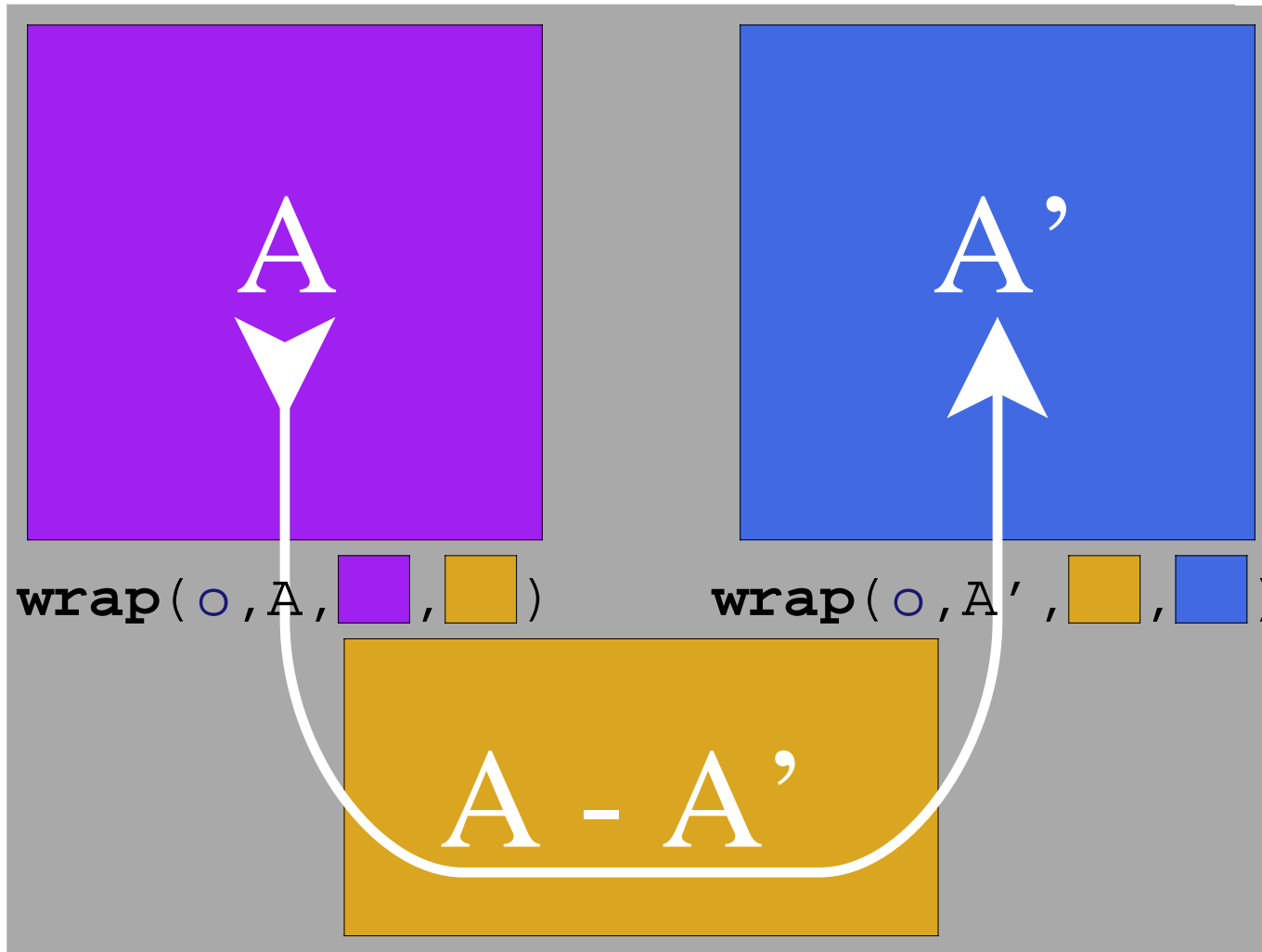
```
wrap(o, I, <name>, <name>)
```

Result ensures I's contracts
but otherwise identical to o

Has type I, even if o doesn't









<Queue>

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

<Client>

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

<PosQueue>

```
q
```



<Queue>

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

<Client>

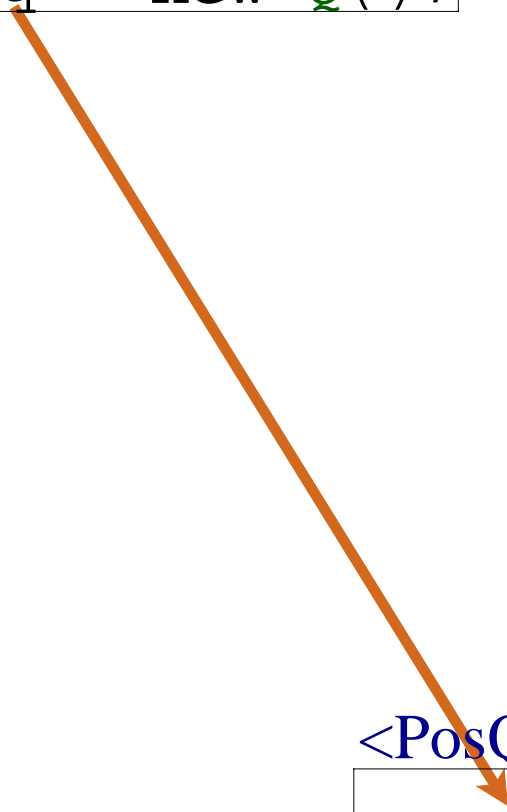
```
q.enq(1);
```

```
q.deq();
```

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

<PosQueue>

```
q
```





<Queue>

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

<Client>

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

```
q.deq();
```

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

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

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

<PosQueue>

```
q
```



<Queue>

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

<Client>

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

```
q.deq();
```

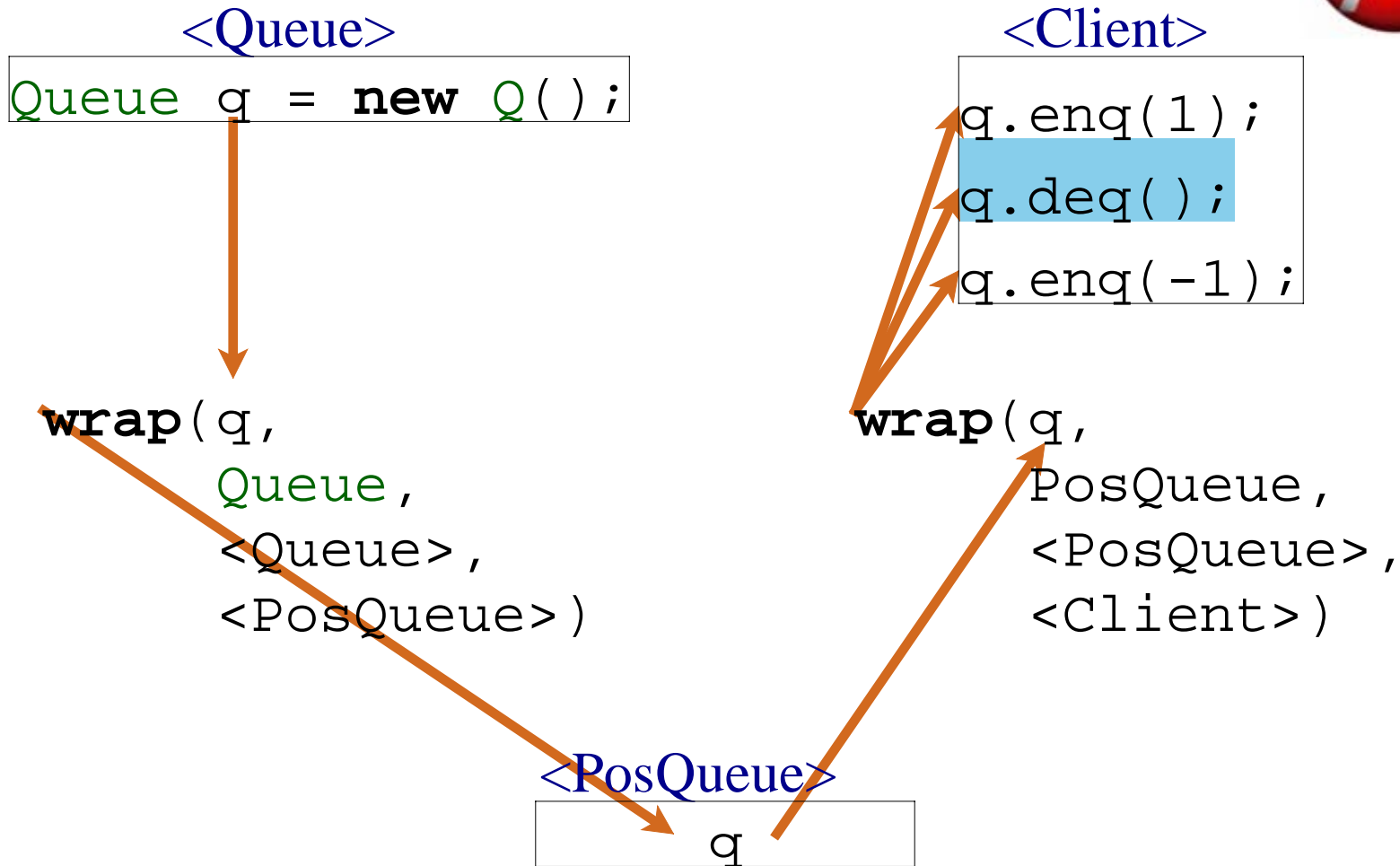
```
q.enq(-1);
```

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

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

<PosQueue>

```
q
```





<Queue>

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

<Client>

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

```
q.deq();
```

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

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

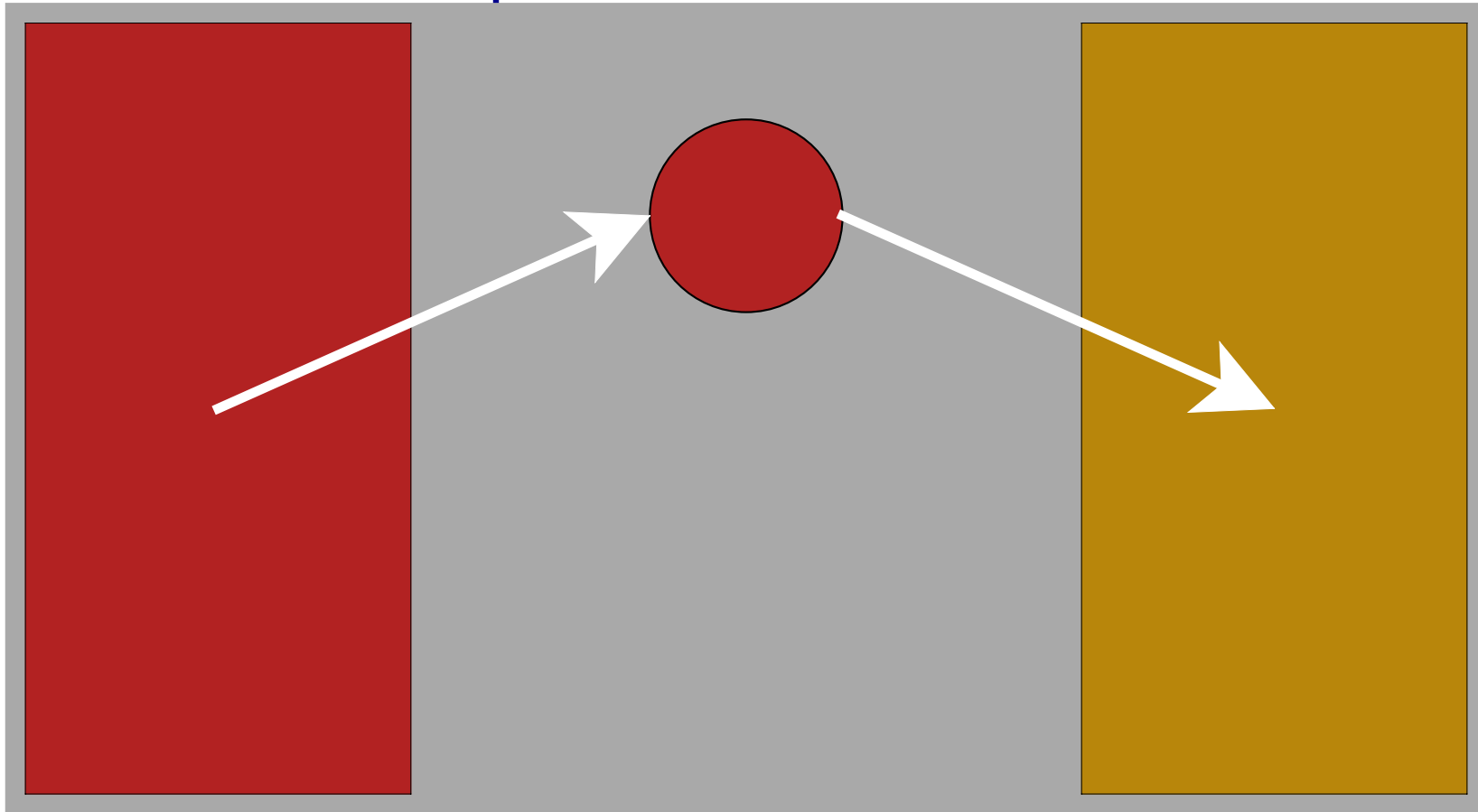
```
wrap(q,  
PosQueue,  
<PosQueue>,  
<Client>)
```

<PosQueue>

```
q
```

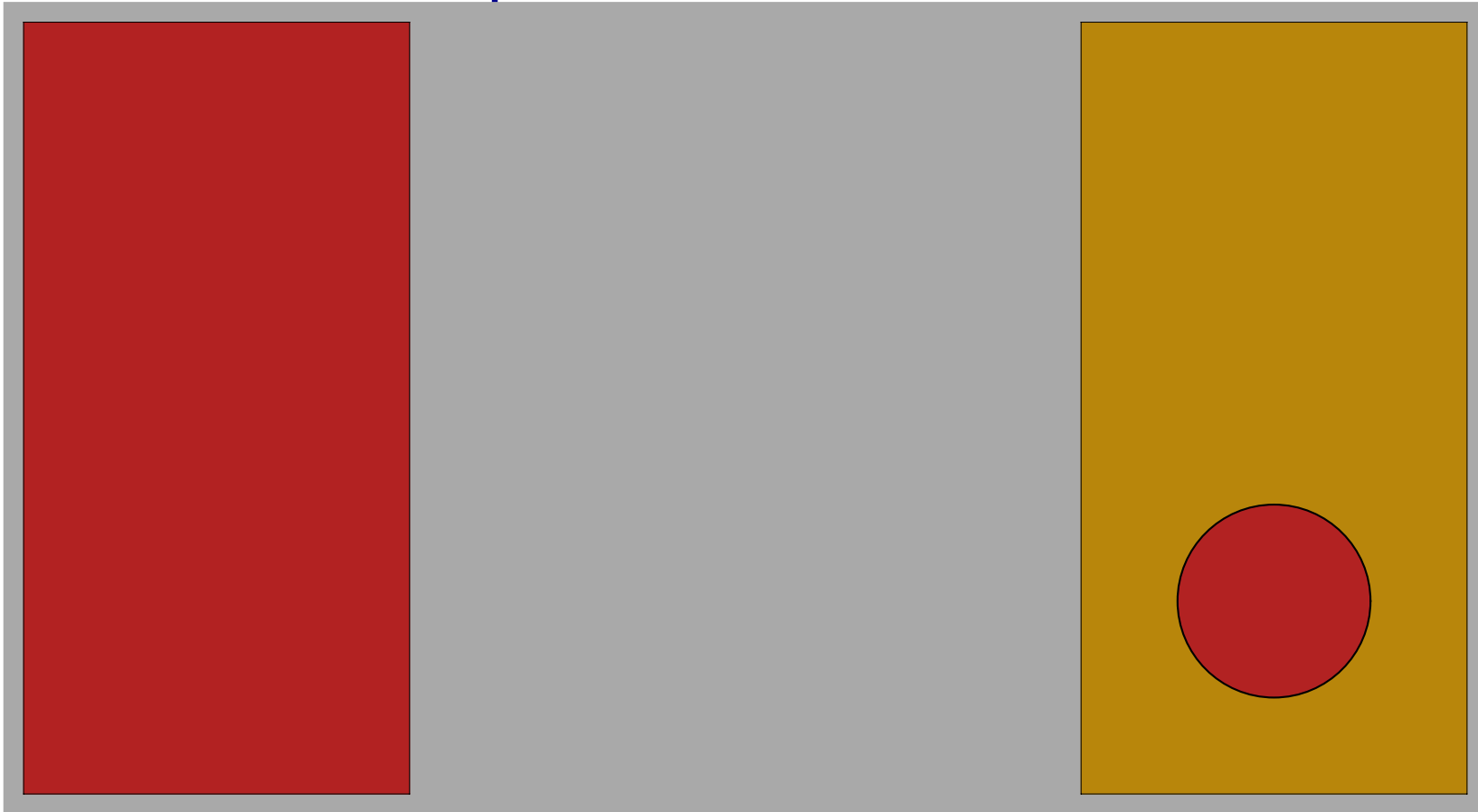


Semantics of wrap



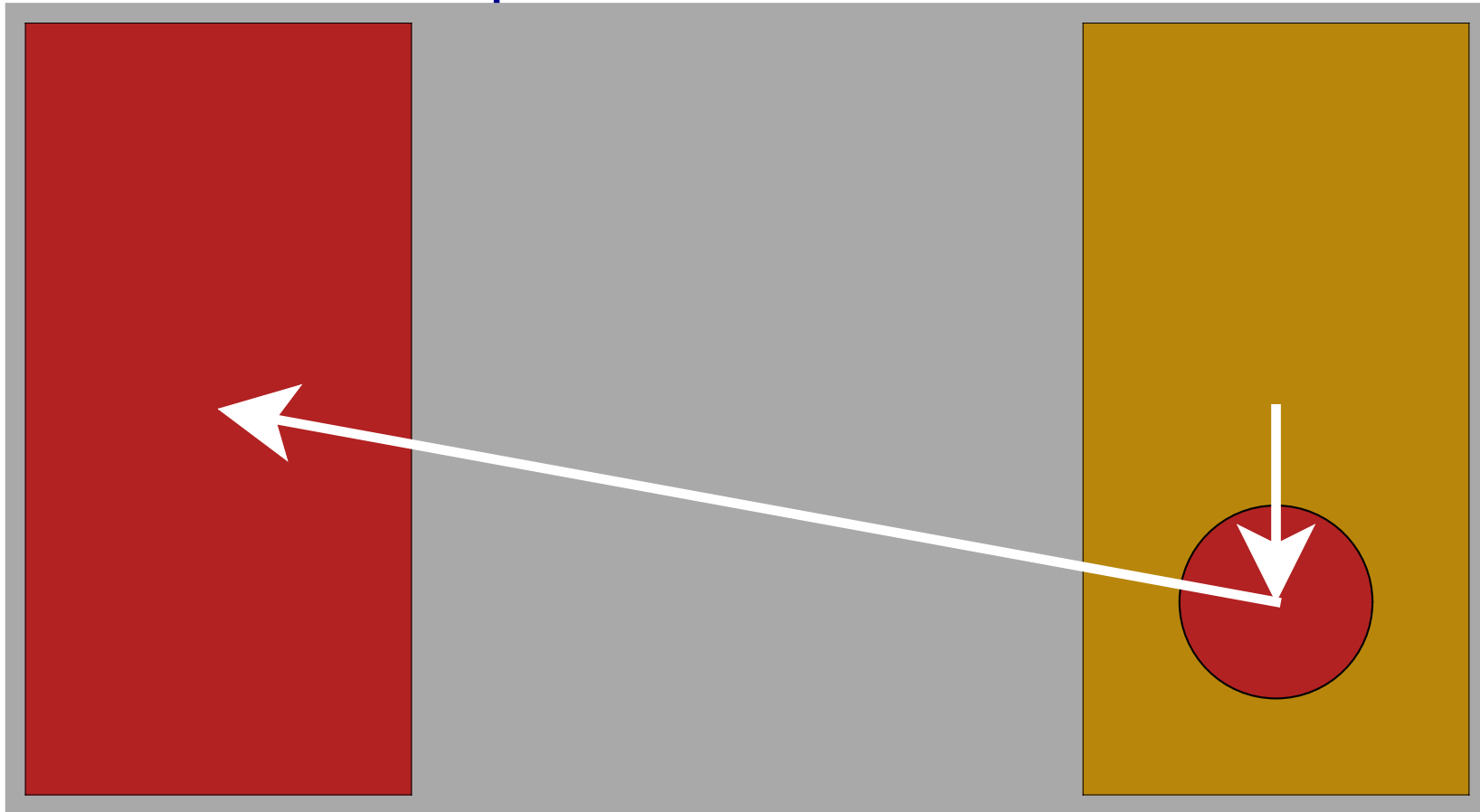


Semantics of wrap



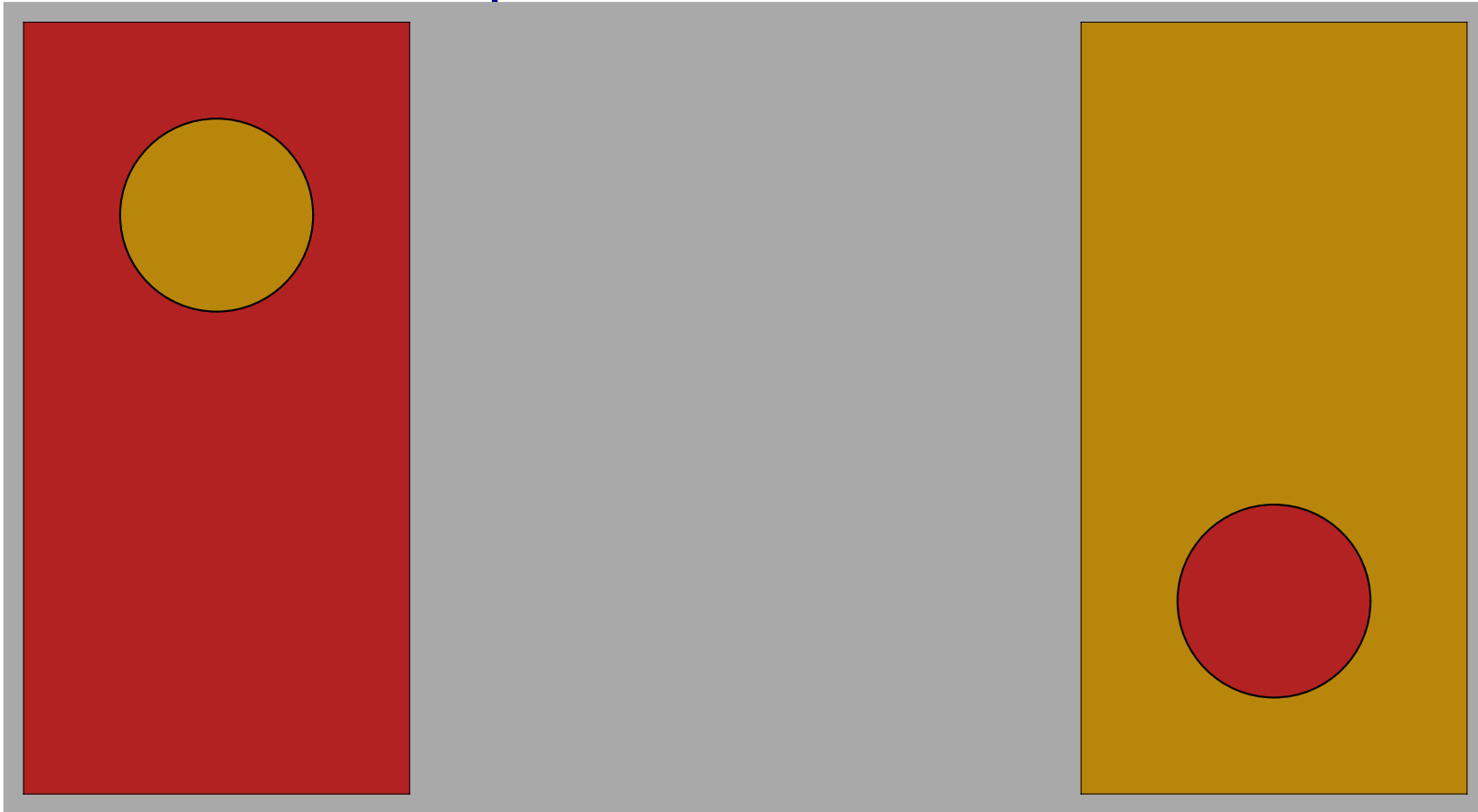


Semantics of wrap



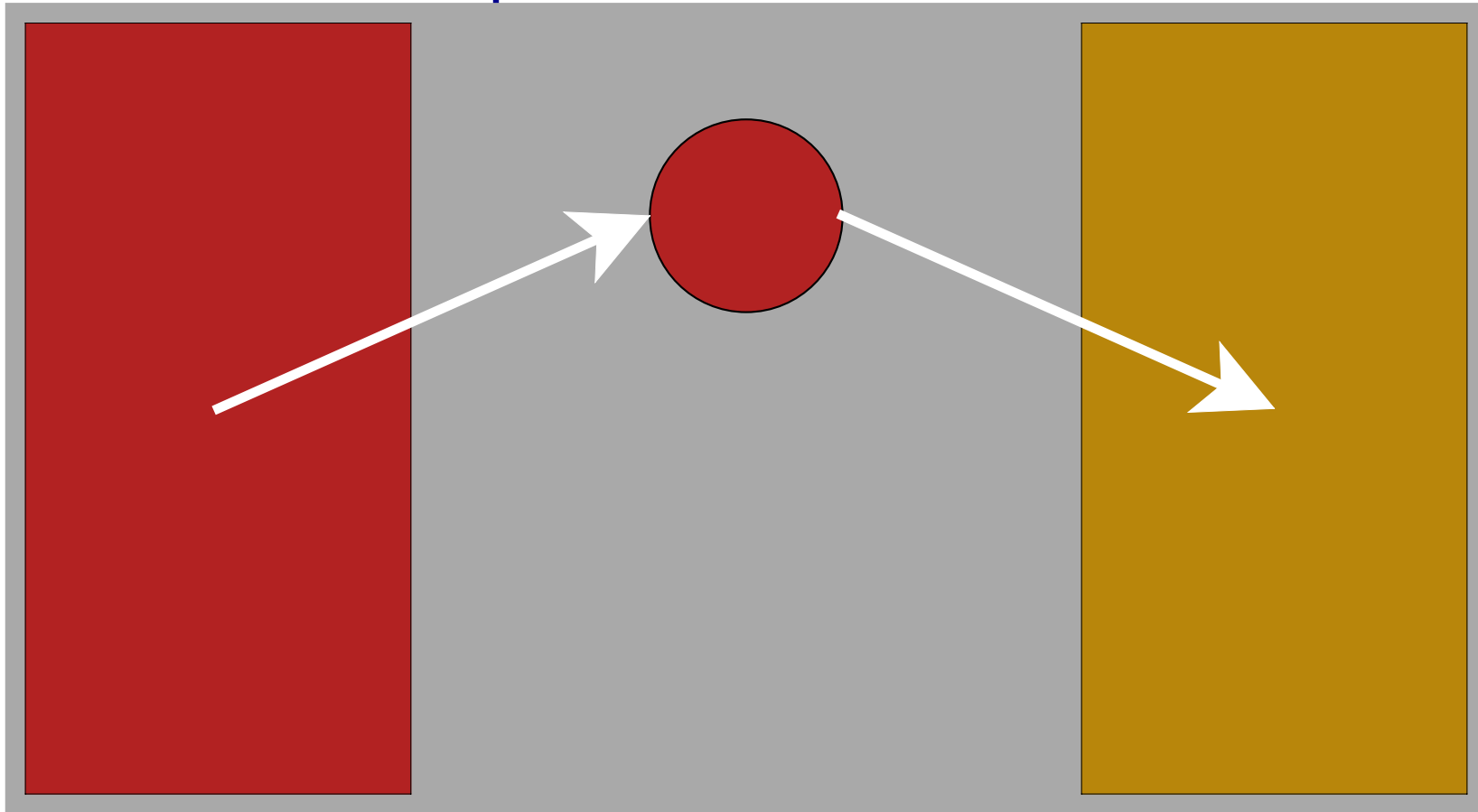


Semantics of wrap



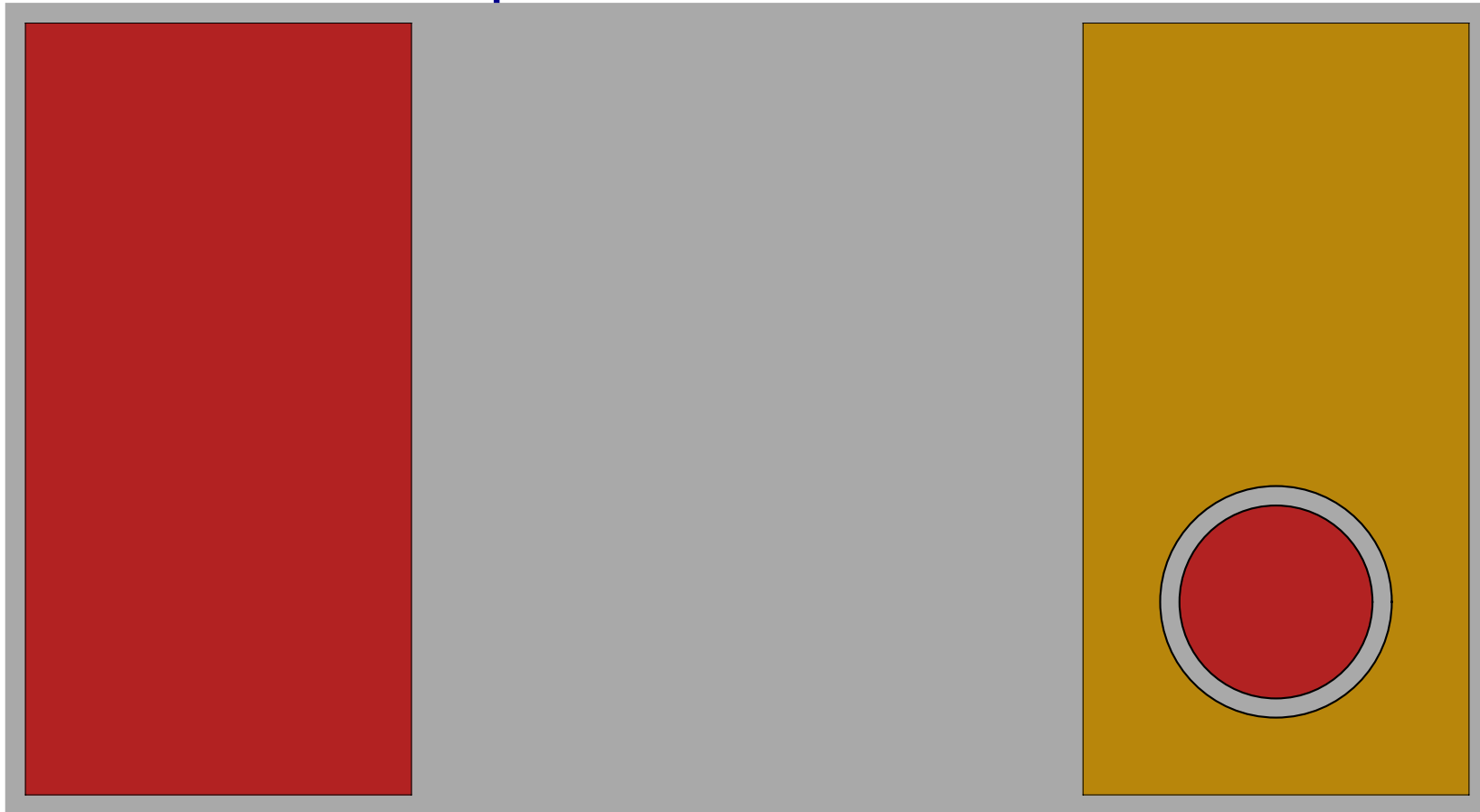


Semantics of wrap



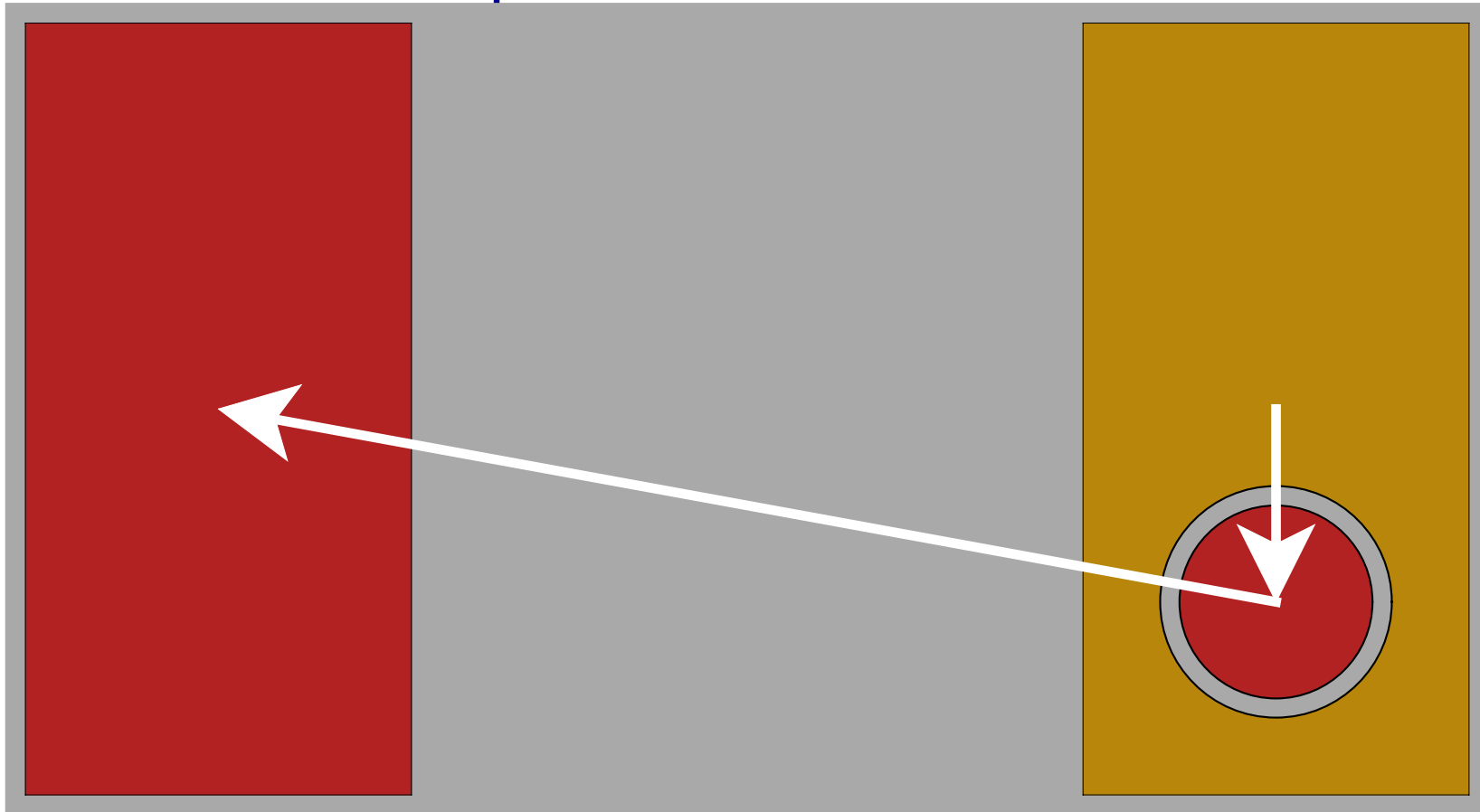


Semantics of wrap



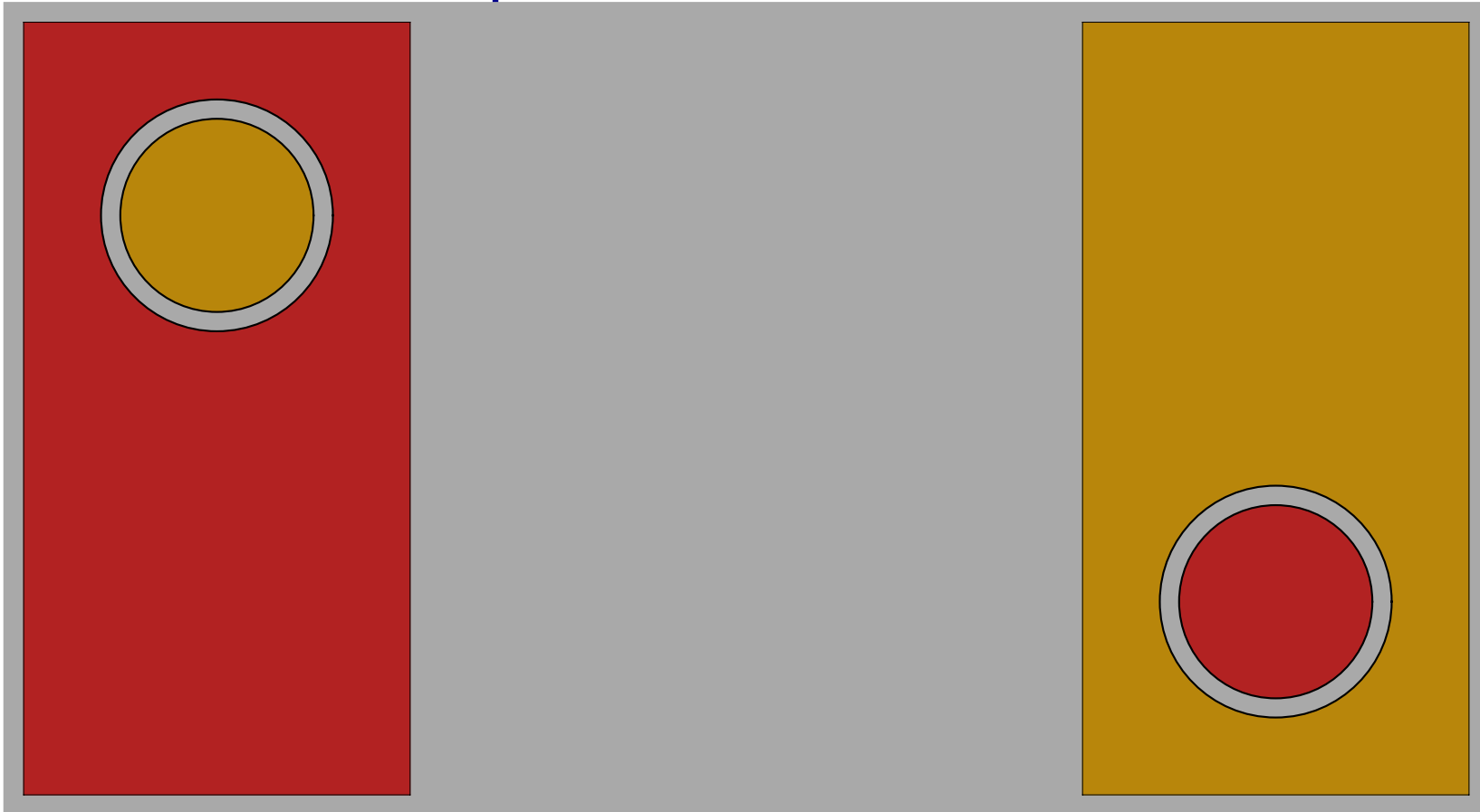


Semantics of wrap





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



Late binding for contracts

- Implementation is subtle; using it is not
- Allows flexible component composition
- Provides mechanism for assigning blame
- Still simple expressions of type boolean



Thank you.