Membranes as Ownership Boundaries

Tom Van Cutsem (Alcatel-Lucent Bell Labs and VUB)

Context

- ***Contracts*** use transitive wrappers (proxies) at module boundaries to test pre and postconditions

- ***Membranes*** use transitive wrappers to wrap strategic objects to enforce confinement of object graphs

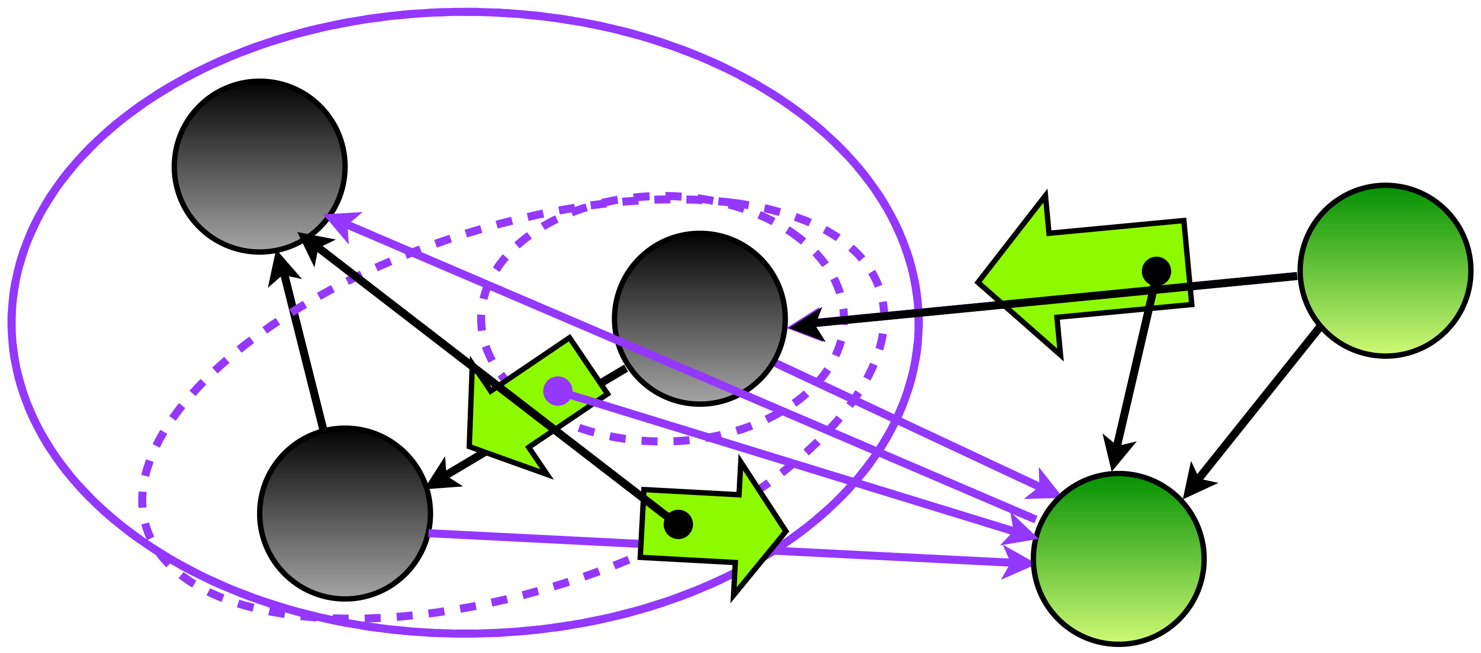
- Using membranes as the basis for controlled sharing of object references
  Use cases similar to those of ownership types

- Report on work-in-progress experiments in JavaScript

Example: revocable membranes

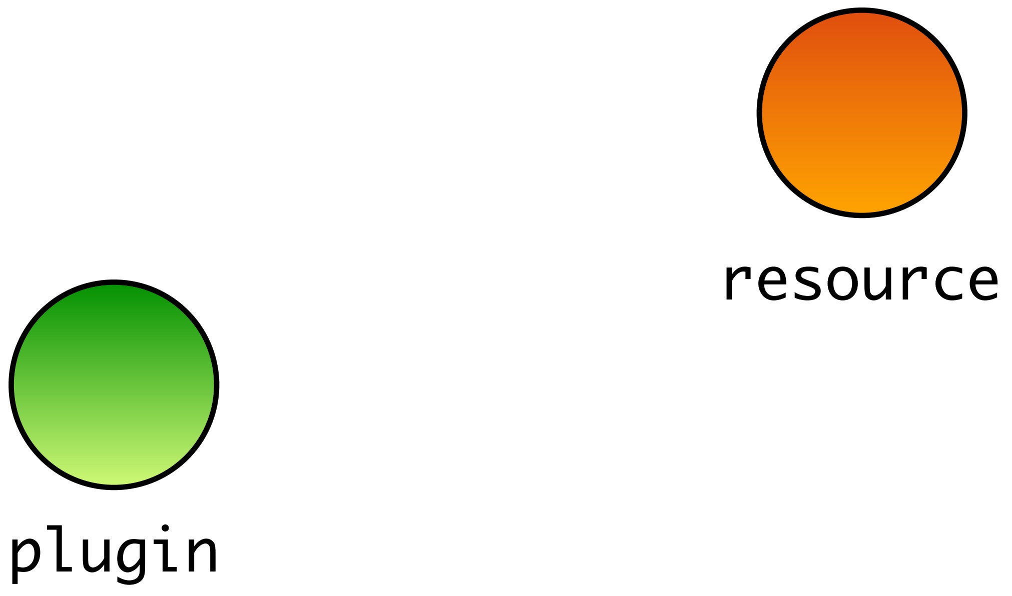
- Origin: Miller’s E and Caja languages. Use object-capabilities to express access control. Isolate graphs of untrusted third-party objects.

- A *membrane* represents a confined object graph that can be controlled as a whole



Simple ownership policy: revocable references

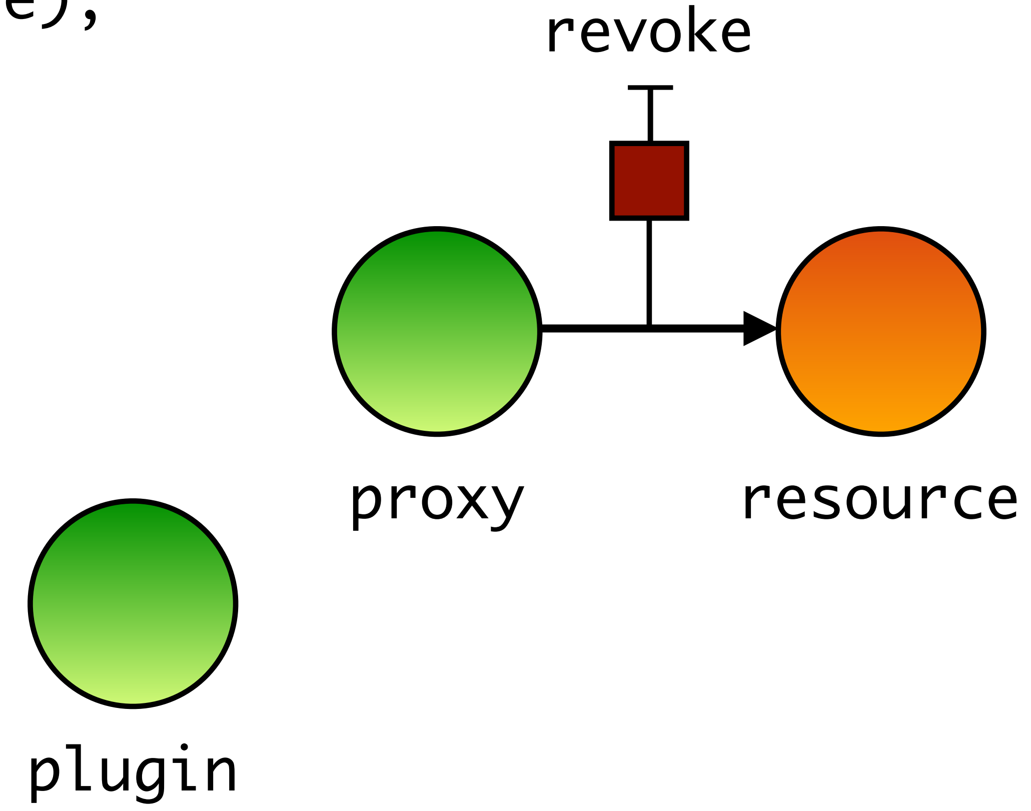
- Provide temporary access to a resource
- Useful for explicit memory management or expressing security policy



Simple ownership policy: revocable references

- Provide temporary access to a resource
- Useful for explicit memory management or expressing security policy

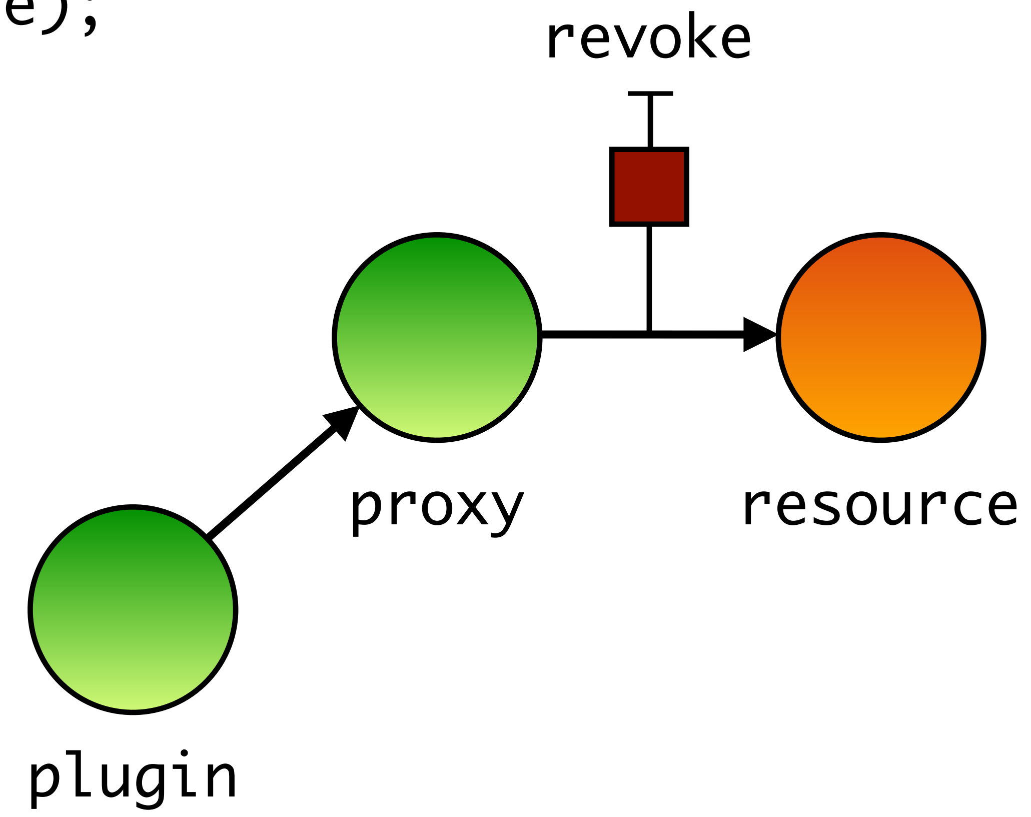
```
var {proxy,revoke} = makeRevocable(resource);
```



Simple ownership policy: revocable references

- Provide temporary access to a resource
- Useful for explicit memory management or expressing security policy

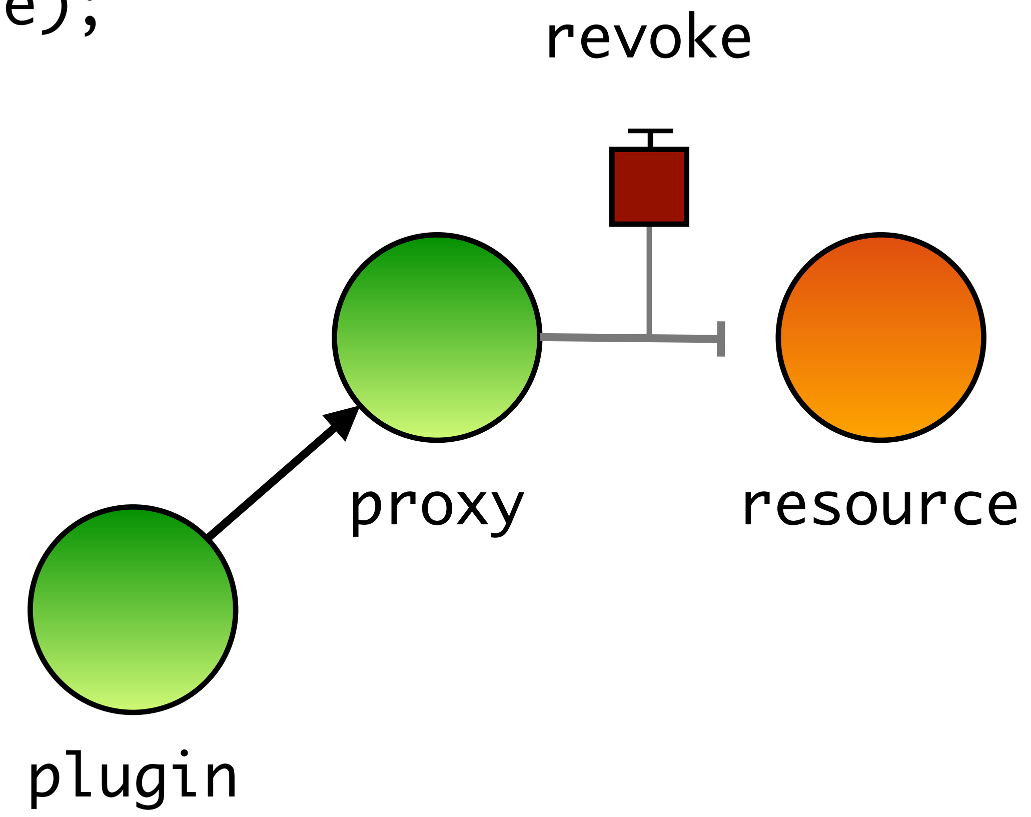
```
var {proxy,revoke} = makeRevocable(resource);

plugin.configure(proxy)
```



Simple ownership policy: revocable references

- Provide temporary access to a resource
- Useful for explicit memory management or expressing security policy

```
var {proxy,revoke} = makeRevocable(resource);

plugin.configure(proxy)

...
revoke();
```



Generic revocable references using proxies

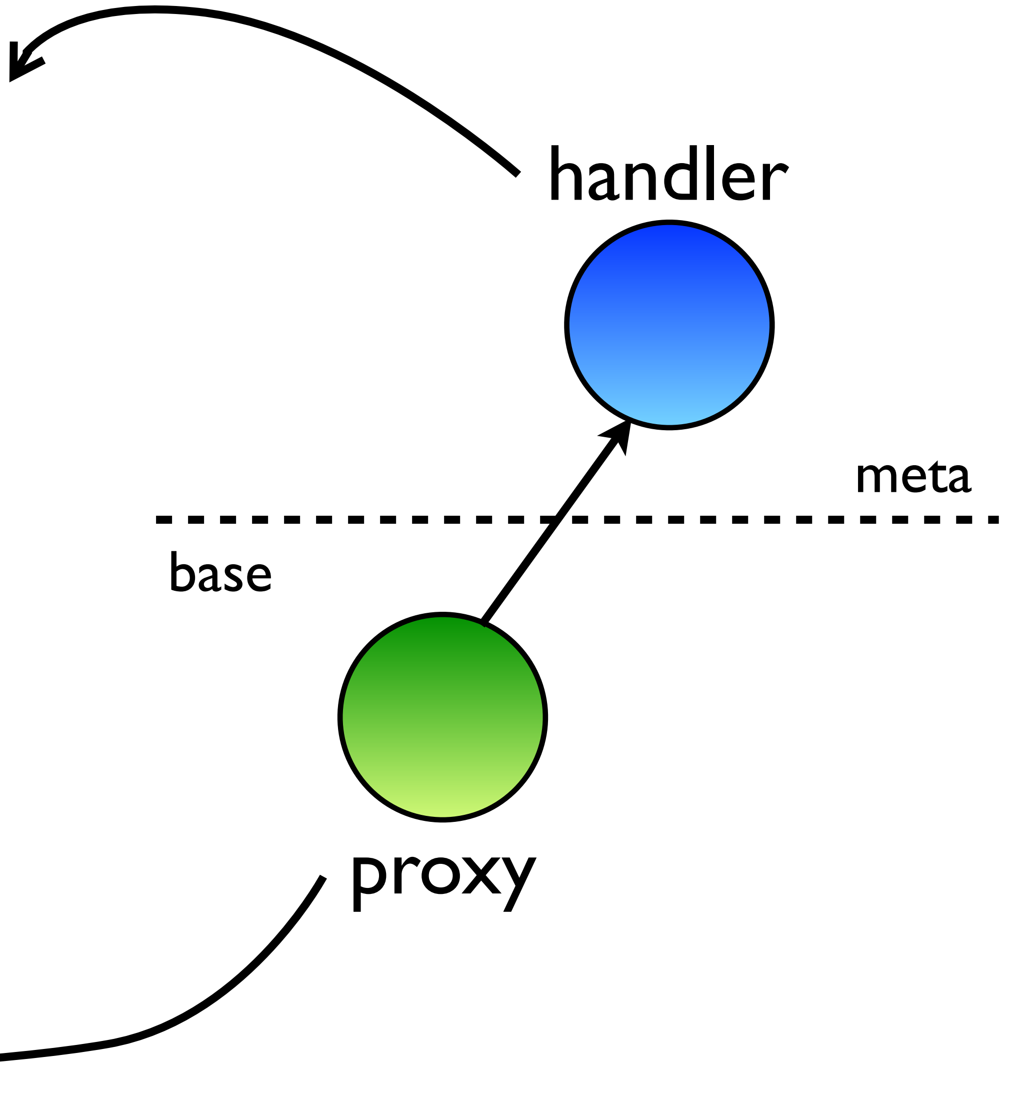
```
function makeRevocable(target) {
  var enabled = true;
  var proxy = new Proxy(target, {









  });
  return {
     proxy: proxy,
    revoke: function() { enabled = false; }
  }
}
```

Generic revocable references using proxies

```
function makeRevocable(target) {
  var enabled = true;
  var proxy = new Proxy(target, {
    get: function(tgt, name) {
      if (!enabled) throw Error(“revoked”)
      return target[name];
    },
    set: function(tgt, name, val) {
      if (!enabled) throw Error(“revoked”)
      target[name] = val;
    },
    ...
  });
  return {
     proxy: proxy,
    revoke: function() { enabled = false; }
  }
}
```

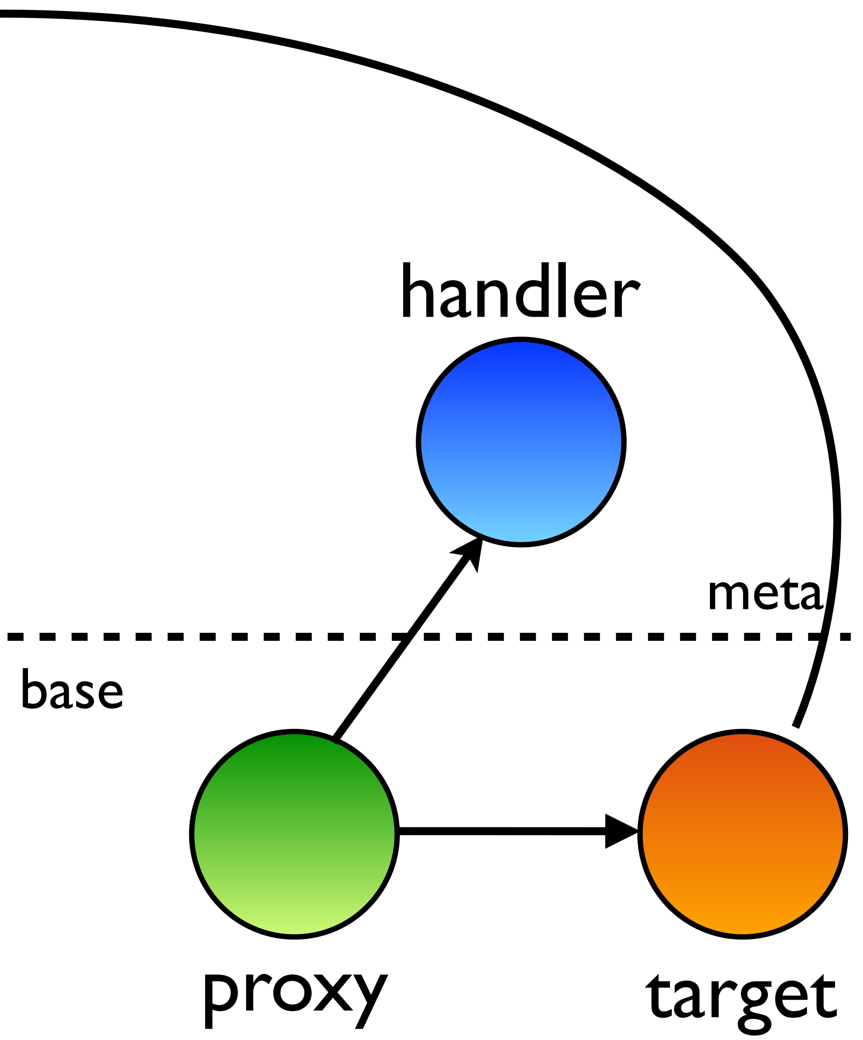
Generic revocable references using proxies

```
function makeRevocable(target) {
  var enabled = true;
  var proxy = new Proxy(target, {
    get: function(tgt, name) {
      if (!enabled) throw Error(“revoked”)
      return target[name];
    },
    set: function(tgt, name, val) {
      if (!enabled) throw Error(“revoked”)
      target[name] = val;
    },
    ...
  });
  return {
     proxy: proxy,
    revoke: function() { enabled = false; }
  }
}
```



Generic revocable references using proxies

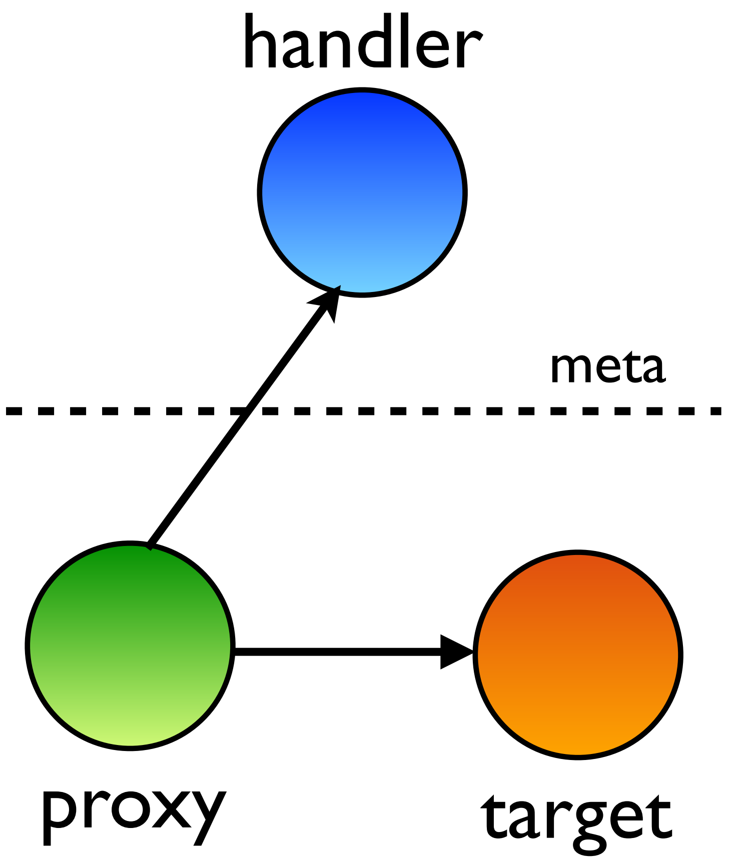
```
function makeRevocable(target) {
  var enabled = true;
  var proxy = new Proxy(target, {
    get: function(tgt, name) {
      if (!enabled) throw Error(“revoked”)
      return target[name];
    },
    set: function(tgt, name, val) {
      if (!enabled) throw Error(“revoked”)
      target[name] = val;
    },
    ...
  });
  return {
     proxy: proxy,
    revoke: function() { enabled = false; }
  }
}
```



JavaScript’s Proxy API

```
var proxy = new Proxy(target, handler);
```

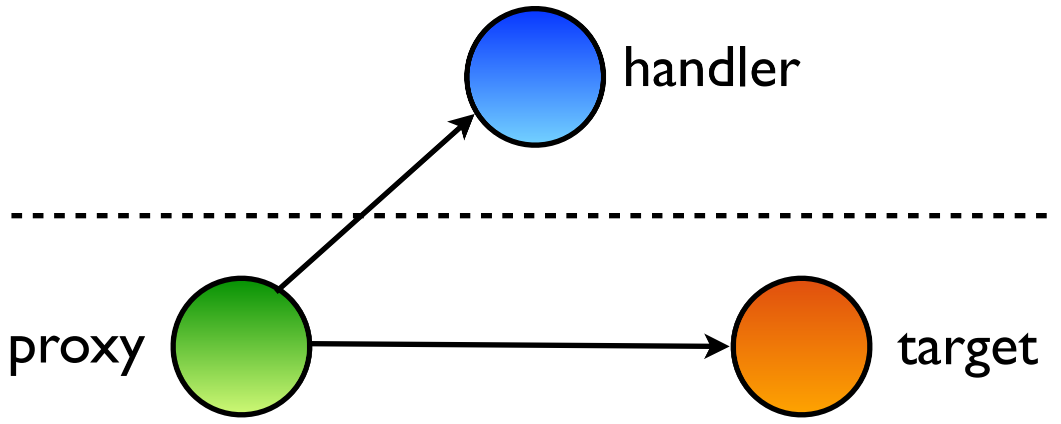
```
handler.get(target, ‘foo’)
handler.set(target, ‘foo’, 42)
```



base

```
proxy.foo
proxy.foo = 42
```

The meta-object protocol of a scripting language…

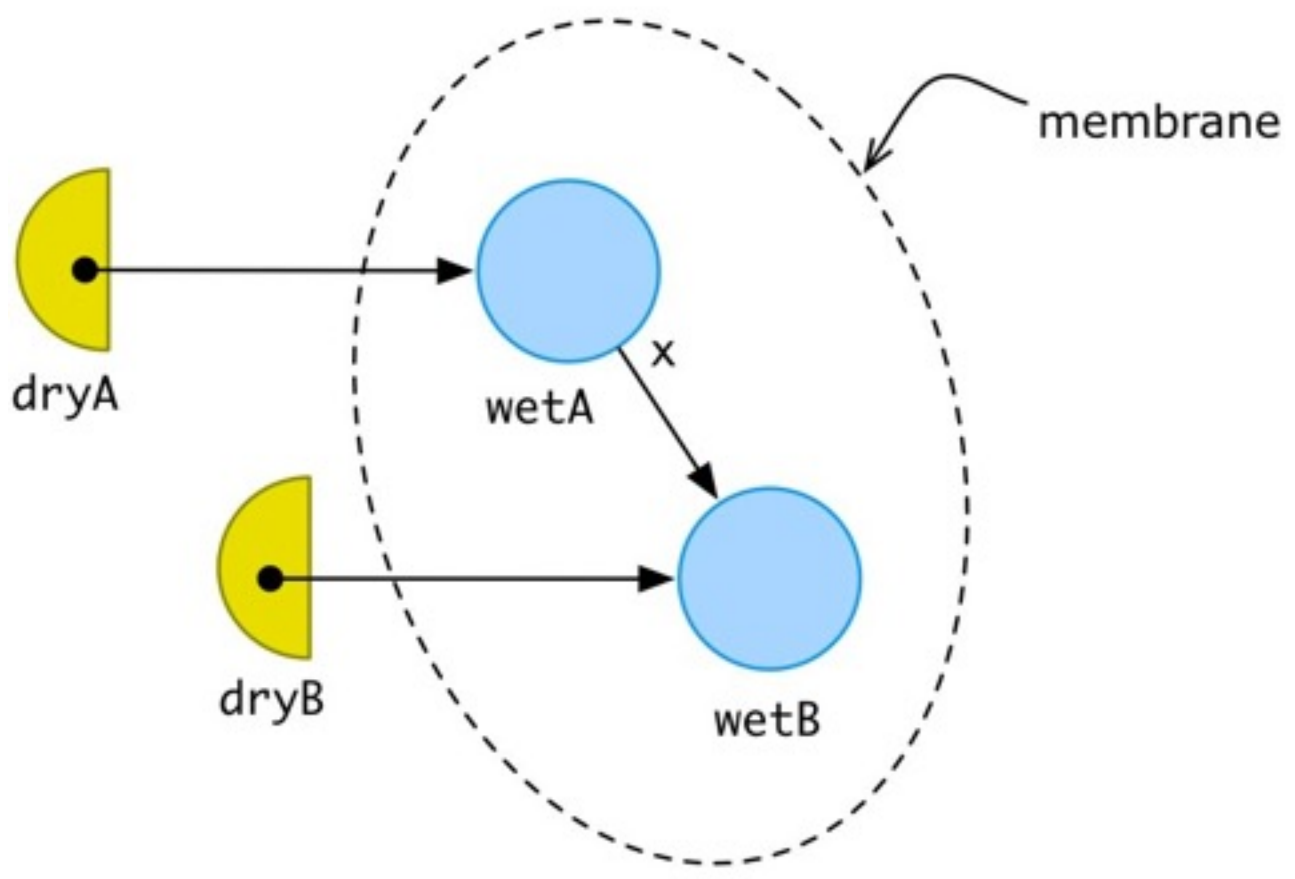


```
Object.getPrototypeOf(proxy)
Object.setPrototypeOf(proxy, proto)
Object.getOwnPropertyDescriptor(proxy,name)
Object.defineProperty(proxy,name,pd)
Object.getOwnPropertyNames(proxy)
delete proxy.name
for (name in proxy) { ... }
Object.preventExtensions(proxy)
Object.isExtensible(proxy)
name in proxy
Object.keys(proxy)
proxy.name
proxy.name = val
proxy(...args)
new proxy(...args)
```

```
handler.getPrototypeOf(target)
handler.setPrototypeOf(target,proto)
handler.getOwnPropertyDescriptor(target,name)
handler.defineProperty(target,name,pd)
handler.getOwnPropertyNames(target)
handler.deleteProperty(target,name)
handler.enumerate(target)
handler.preventExtensions(target)
handler.isExtensible(target)
handler.has(target,name)
handler.ownKeys(target)
handler.get(target,name,receiver)
handler.set(target,name,value,receiver)
handler.apply(target,receiver,args)
handler.construct(target,args)
```

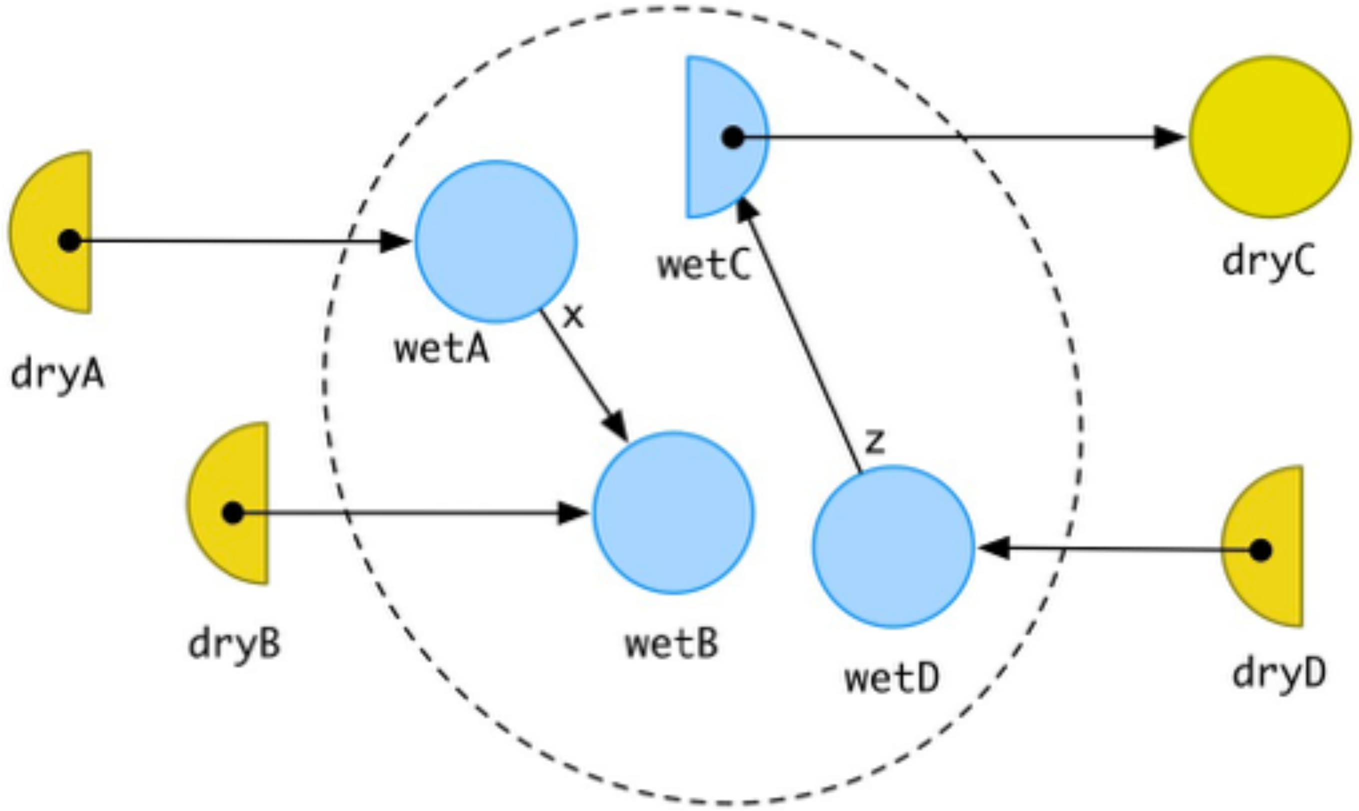
Membranes transitively interpose proxies

```
var wetB = {};
var wetA = { x: wetB };

var dryA = wet2dry(wetA);
var dryB = dryA.x;
```



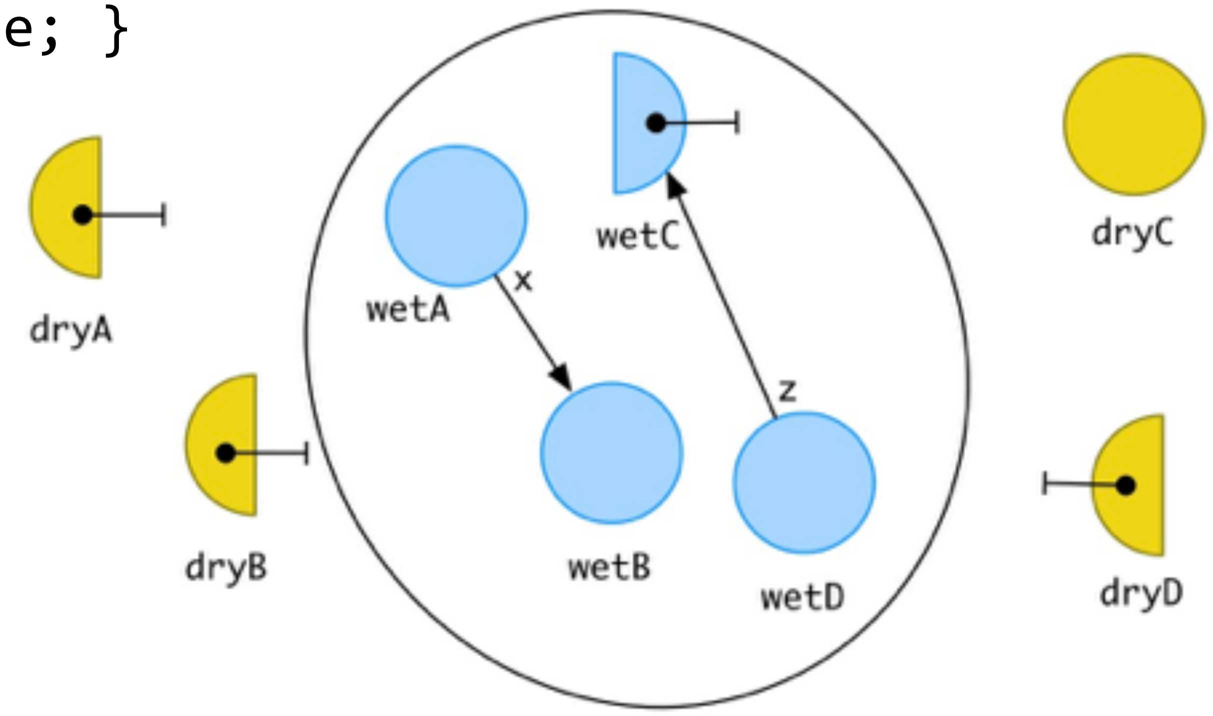
Membranes transitively interpose proxies

```
var wetB = {};
var wetA = { x: wetB };

var dryA = wet2dry(wetA);
var dryB = dryA.x;

wetA.m = function(wetC) {
  var wetD = { z: wetC };
  return wetD;
}

var dryC = {};
var dryD = dryA.m(dryC);
```



Membrane proxies can share state

```
function makeMembrane(initDryTarget) {
  var enabled = true;
  var wetProxies = new WeakMap();
  var dryProxies = new WeakMap();
  ...
  function wet2dry(wetTarget) { … }
  function dry2wet(dryTarget) { … }
  ...
  return {
    proxy: dry2wet(initDryTarget),
    revoke: function() { enabled = false; }
  };
}
```



Generalizing membranes

- A revocable membrane is just one possible kind of membrane
- wet / dry distinction very similar to negative / positive blame labels? Two-way filtering.
- Basis for controlled sharing and confinement of object references, like ownership types
- Inspired by Erwann Wernli et al.'s paper: "Ownership, Filters and Crossing Handlers", *DLS* 2012
    - But… their system doesn't use proxies. Instead, each object is explicitly owned by at most one other object.

Membranes as ownership boundaries

```
function Box(init) {
  this.state = init;
}
Box.prototype.read = function(){
  return this.state;
}
Box.prototype.write = function(v) {
  this.state = v;
}
```

```
var aBox = new Box(42)
aBox.read() // 42
aBox.write(0)
aBox.read() // 0
```

Membranes as ownership boundaries

- *Classify* methods according to zero or more “topics”

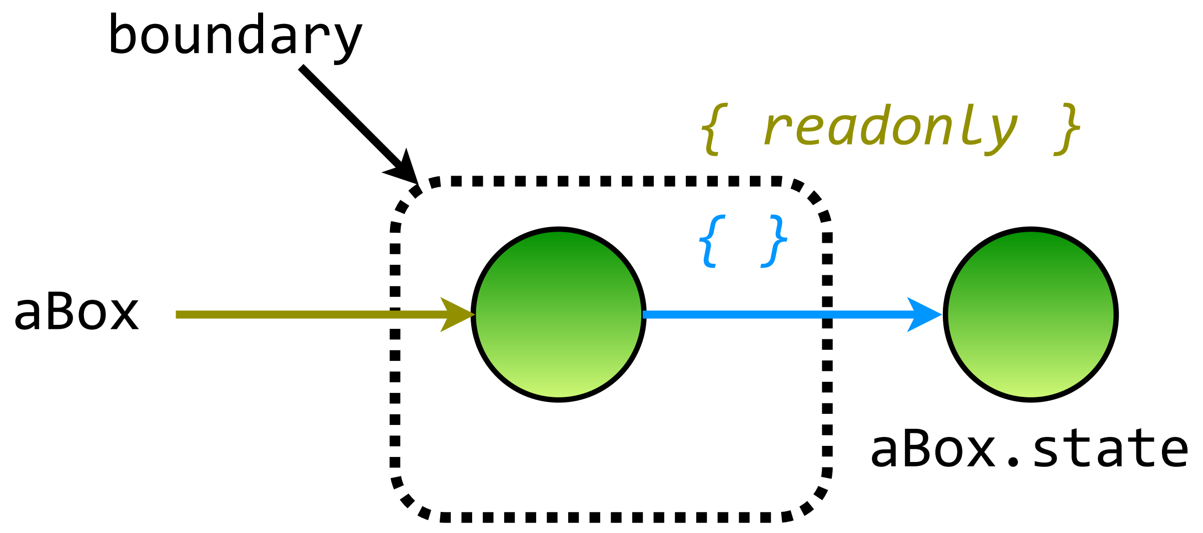
```
function Box(init) {
  this.state = init;
}
Box.prototype.read = function(){
  return this.state;
}.class("readonly");
Box.prototype.write = function(v) {
  this.state = v;
}.class("mutator");
```

```
var aBox = new Box(42)
aBox.read() // 42
aBox.write(0)
aBox.read() // 0
```

Membranes as ownership boundaries

- *Wrap* strategic object in an ownership boundary and *apply filters*

```
var boundary = new Boundary({
  in: [],
  out: [".readonly"],
  entry: Box
});
var BoundBox = boundary.entry;

var aBox = new BoundBox(42);
boundBox.read() // 42
boundBox.write(0) // Error: method 'write' does not match out-filter
```



Summary

- Like contracts, membranes transitively interpose wrappers.
- Use membranes to control sharing of object references (like ownership types, but dynamic):
  - Can introduce asymmetric filters on messages flowing across the membrane
  - Runtime checks only when crossing boundaries
  - Objects can belong to multiple, overlapping boundaries