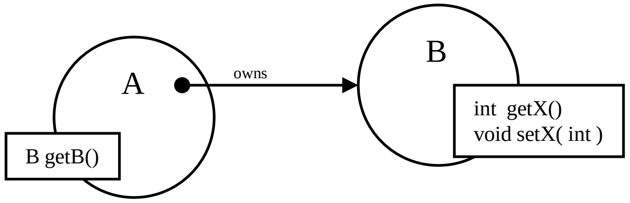
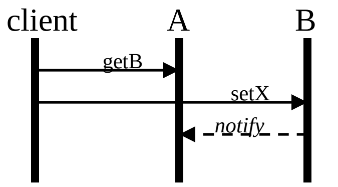
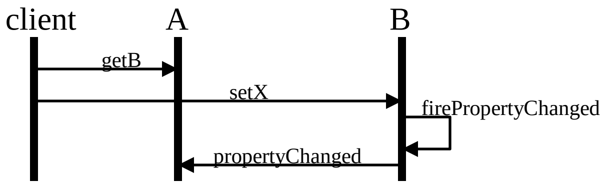
Scenario
A owns B.
B is exposed to A's clients.
A needs to monitor changes in B.
B knows nothing about A.

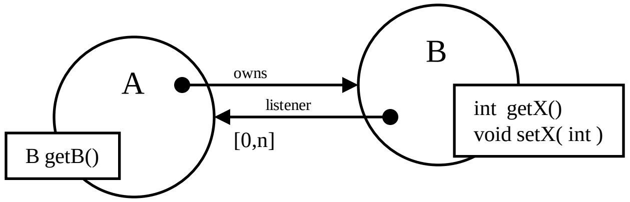




Solution: property listener
If B is observable then A can register as a listener in B.

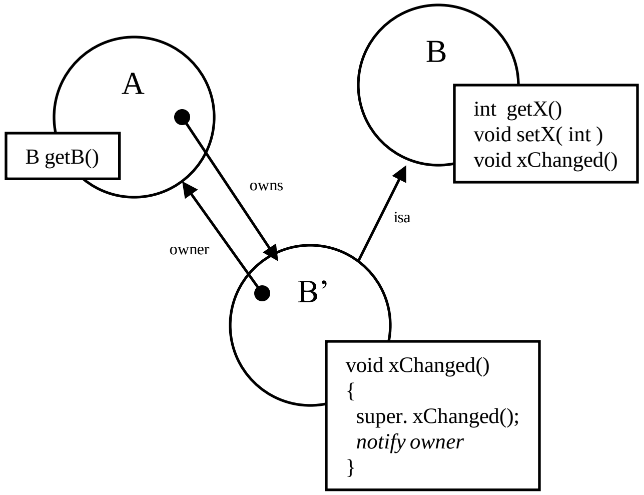
- Only works if B is observable.
- If A is the only observer then a general property listening mechanism is a bit heavyweight.



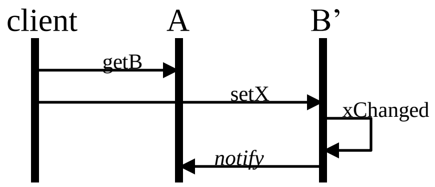


Solution: interrupting B's internal notification
If B has some kind of internal change notification that is overridable then A can expose a subclass of B where this notification mechanism is overridden to notify A.

- Only possible if B has an internal change notification that can be overridden.
- If A has a setB(B) method then the required invariant [A owns a B'] can't be assured.



```
void xChanged()
{
  super. xChanged();
  notify owner
}
```



note: if B' is an inner class of A the reference
owner is implicitly generated by the compiler

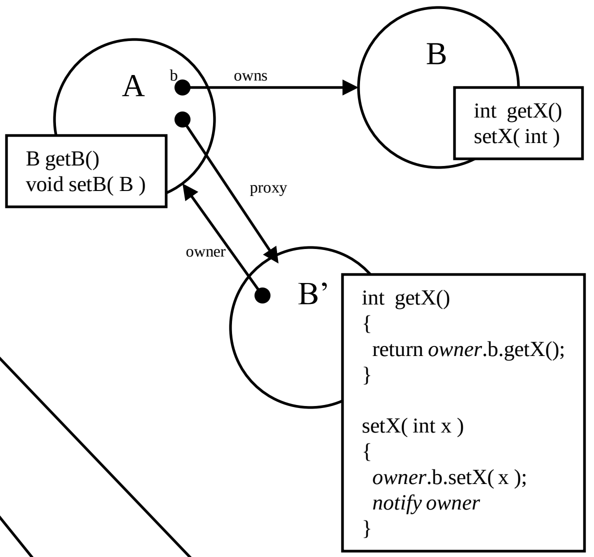
Solution: mutable proxy
Instead of exposing B let A expose a proxy that catches mutating calls to B.

For this to work B’ must be type compatible with B. This can be achieved in two ways:
1) B’ subclasses B. B’ inherits attributes from B but doesn’t use them.
2) Instead of exposing B, let A expose an interface BIF that defines the protocol of B. Both B and B’ implements this interface.

Using an interface has the advantage of statically forcing B’ to implement the entire protocol. If a method is added to the protocol then the compiler will require B’ to implement it. If B’ subclasses B then the code will compile just fine but the intended semantics if B’ will be compromised.

Note that B’ must be able to access A.b without using A.getB (which would return B’). If B’ is an inner class of A then this is guaranteed by the compiler.

Changing the value of A.b is not a problem because the proxy always access its delegate through A.

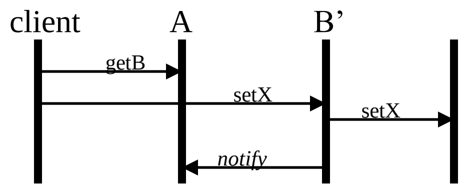


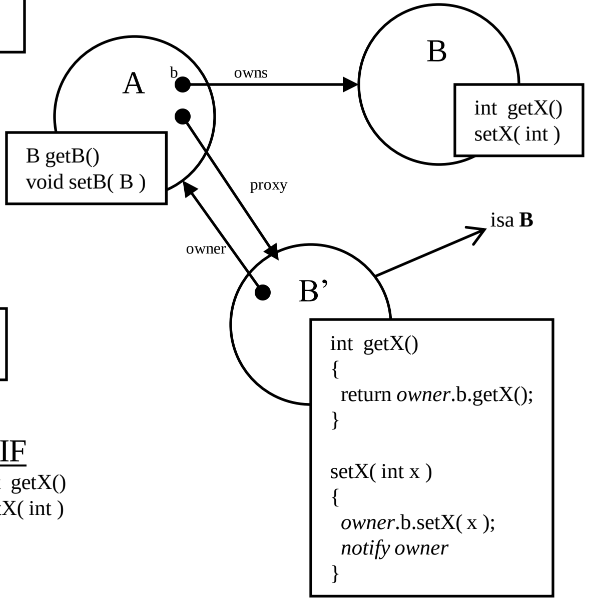
```
B getB()
void setB( B )
```

```
int  getX()
setX( int )
```

```
int  getX()
{
  return owner.b.getX();
}

setX( int x )
{
  owner.b.setX( x );
  notify owner
}
```



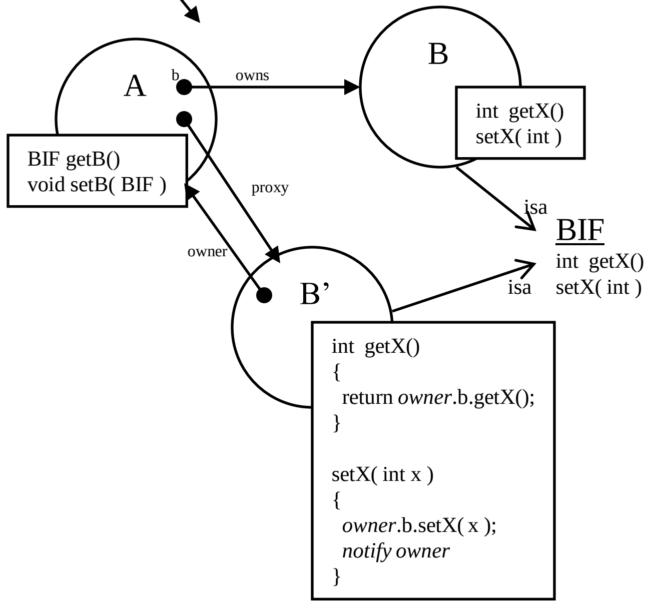


```
B getB()
void setB( B )
```

```
int  getX()
setX( int )
```

```
int  getX()
{
  return owner.b.getX();
}

setX( int x )
{
  owner.b.setX( x );
  notify owner
}
```



```
BIF getB()
void setB( BIF )
```

```
int  getX()
setX( int )
```

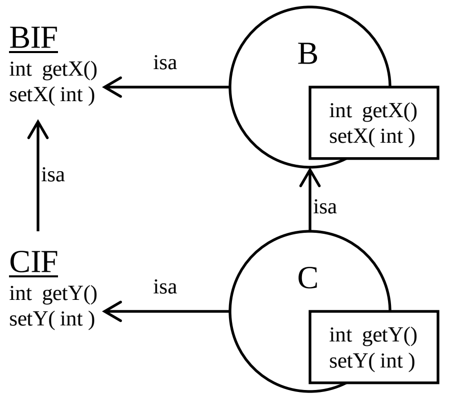
```
int  getX()
{
  return owner.b.getX();
}

setX( int x )
{
  owner.b.setX( x );
  notify owner
}
```

note: If B’ is an inner class of A the reference *owner* is implicitly generated by the compiler and B’ is given access to the private scope of A.

note: Beyond this point the discussion only considers the alternative where BIF exists.

Multiple protocols
What if BIF is only the root of a tree protocols?
Lets introduce the interface CIF and a corresponding class C.

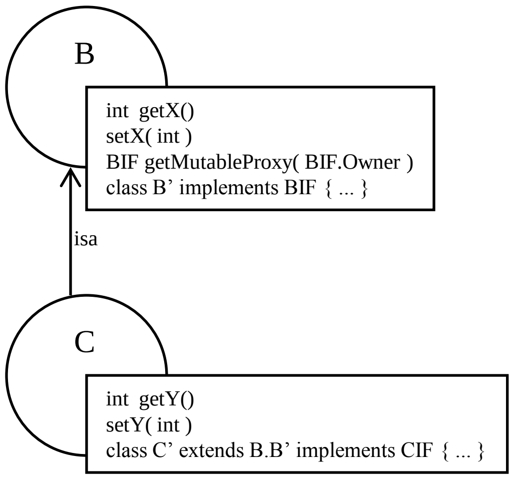


Suddenly it gets hard to create the proxy. A doesn’t know if A.b refers to an instance of BIF or CIF. Only the object referred to by A.b knows this, so the responsibility of creating the proxy falls upon it.

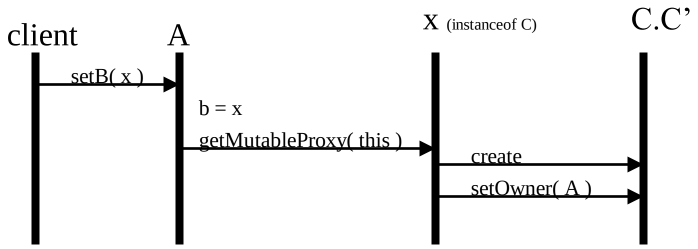
This excludes the possibility of the proxy being an inner class of A. Instead the proxy must have an explicit reference to A.
However B isn’t allowed to know anything about A. The way around this is to introduce the interface BIF.Owner that A must implement.

<u>BIF</u>
int getX()
setX(int)
interface Owner { void notify(); }

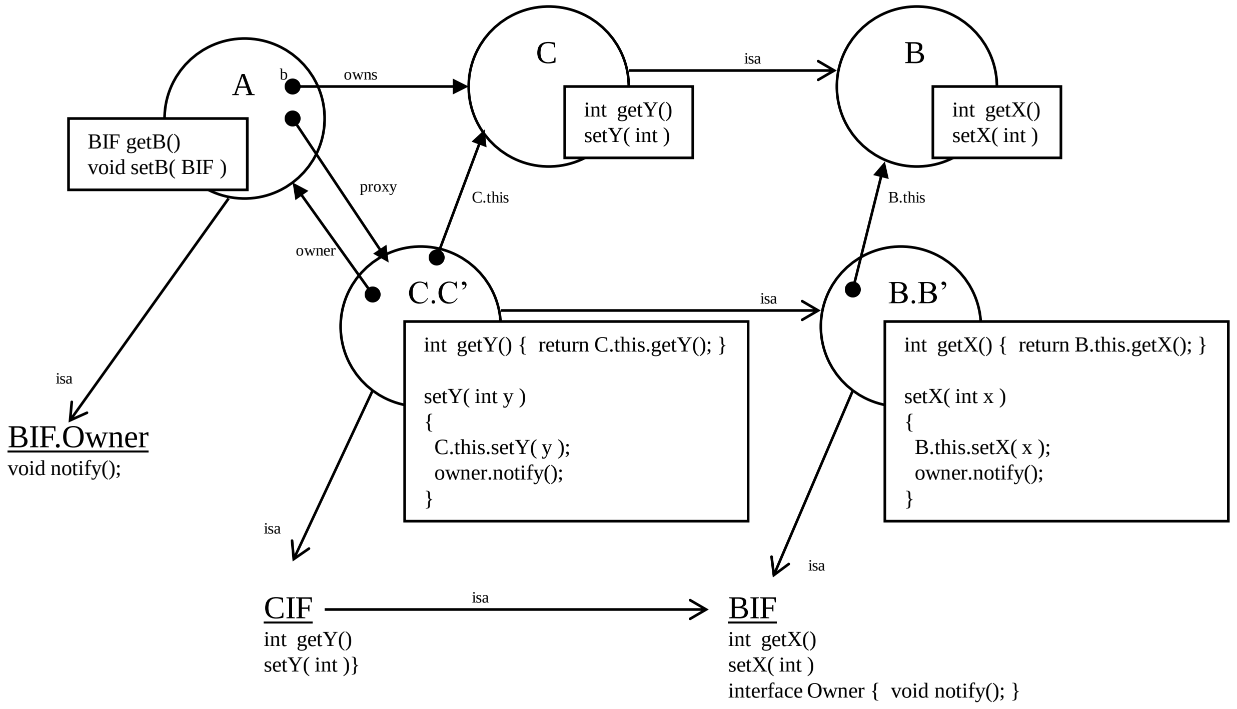
Since B is now responsible for creating the proxy it must have a method for doing so. The proxy gains access to B by being an inner class in B.


B
int getX()
setX(int)
BIF getMutableProxy(BIF.Owner)
class B’ implements BIF { ... }
isa
C
int getY()
setY(int)
class C’ extends B.B’ implements CIF { ... }

Now A can create a new proxy when A.b changes value.



The resulting structure is: (new page)



```
BIF getB()
void setB( BIF )
```

```
int getY()
setY( int )
```

```
int getX()
setX( int )
```

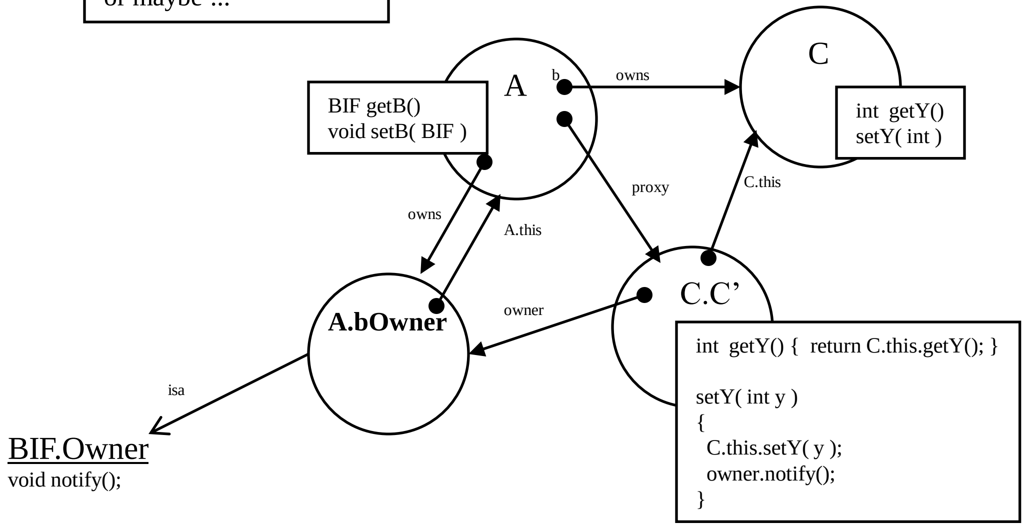
```
int getY() { return C.this.getY(); }

setY( int y )
{
  C.this.setY( y );
  owner.notify();
}
```

```
int getX() { return B.this.getX(); }

setX( int x )
{
  B.this.setX( x );
  owner.notify();
}
```

or maybe ...



```
BIF getB()
void setB( BIF )
```

```
int getY()
setY( int )
```

```
int getY() { return C.this.getY(); }

setY( int y )
{
  C.this.setY( y );
  owner.notify();
}
```