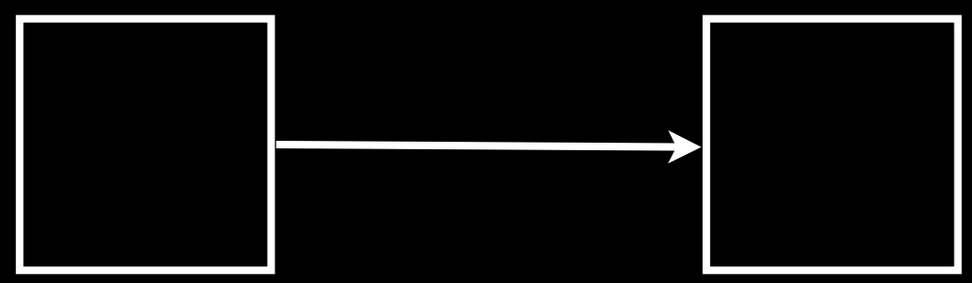
Dependency Injection

Fancy Term... simple concept!

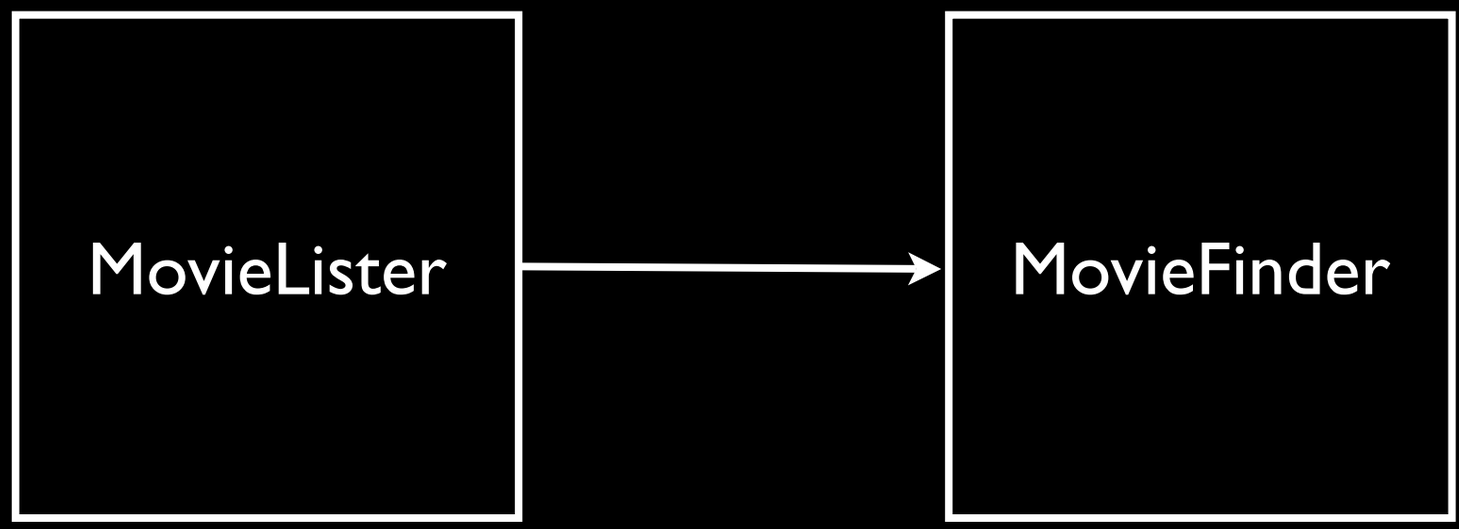
Dependecy

... any time one class references another...

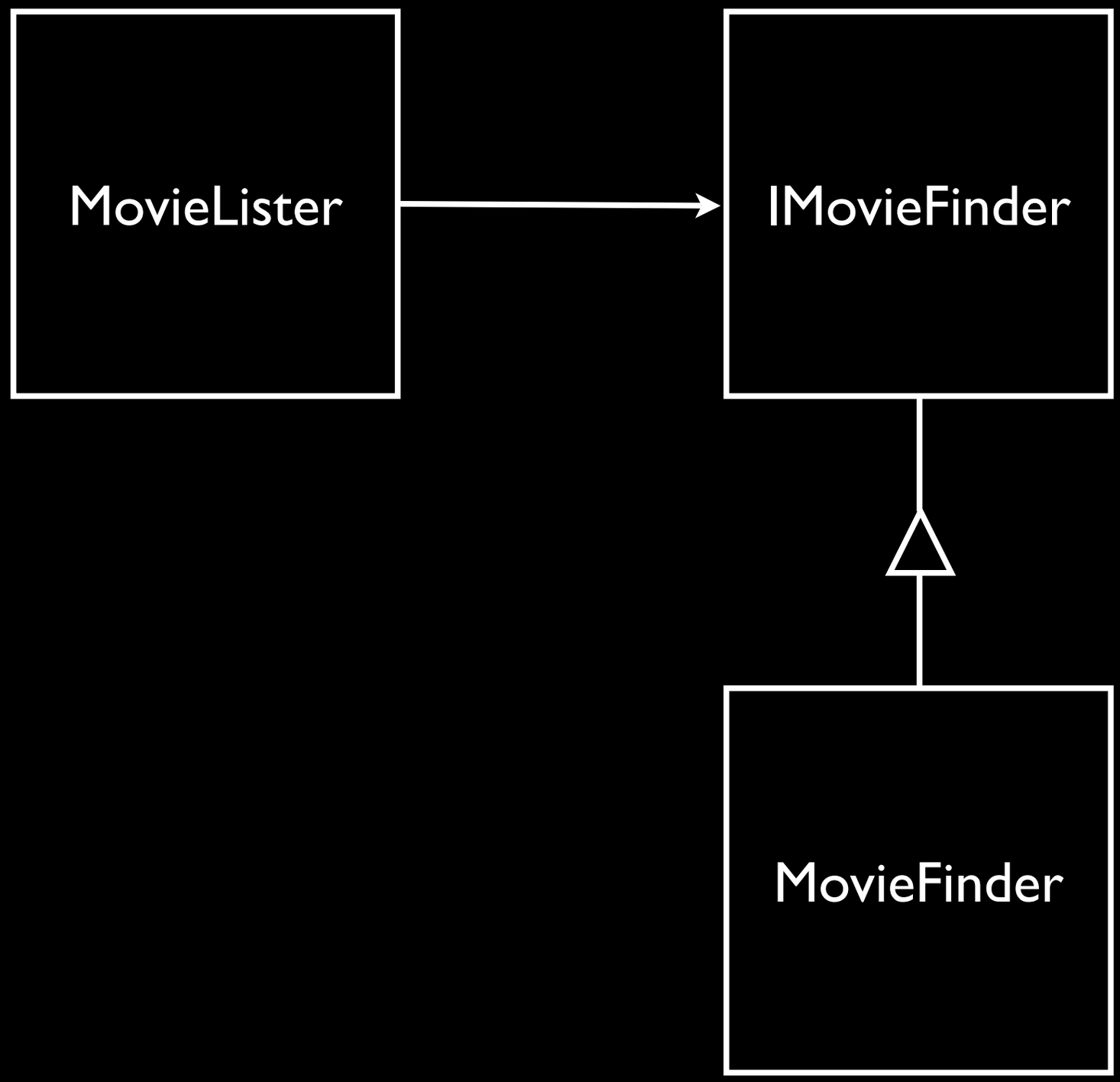


```
class MovieLister
{
 MovieFinder finder;

 public MovieLister()
 {
  finder = new MovieFinder();
 }
```

```
 pubilc Movie[] ListMoviesByActor(string
 actor)
 {
  Movie[] allMovies = finder.FindAll();
  // do some filtering and return only
  // the right movies...
 }
}
```


MovieLister
MovieFinder

Open Closed Principle (OCP)


MovieLister
IMovieFinder
MovieFinder

```
class MovieLister
{
  IMovieFinder finder;

  public MovieLister()
  {
    finder = new MovieFinder();
  }
```

```
  public Movie[] ListMoviesByActor(string
  actor)
  {
    Movie[] allMovies = finder.FindAll();
    // do some filtering and return only
    // the right movies...
  }
}
```

Injection

```
class MovieLister
{
  IMovieFinder finder;

  public MovieLister(IMovieFinder finder)
  {
    this.finder = finder;
  }
```

```
  pubilc Movie[] ListMoviesByActor(string
  actor)
  {
    Movie[] allMovies = finder.FindAll();
    // do some filtering and return only
    // the right movies...
  }
}
```

We have
Separation of Concerns
(SoC)

1. How to filter movies

2. How to find movies.

3. Which IMovieFinder strategy to use.

Good OO Design leads to a complex network of simple objects.

Dependency Injection separates the network from the objects.

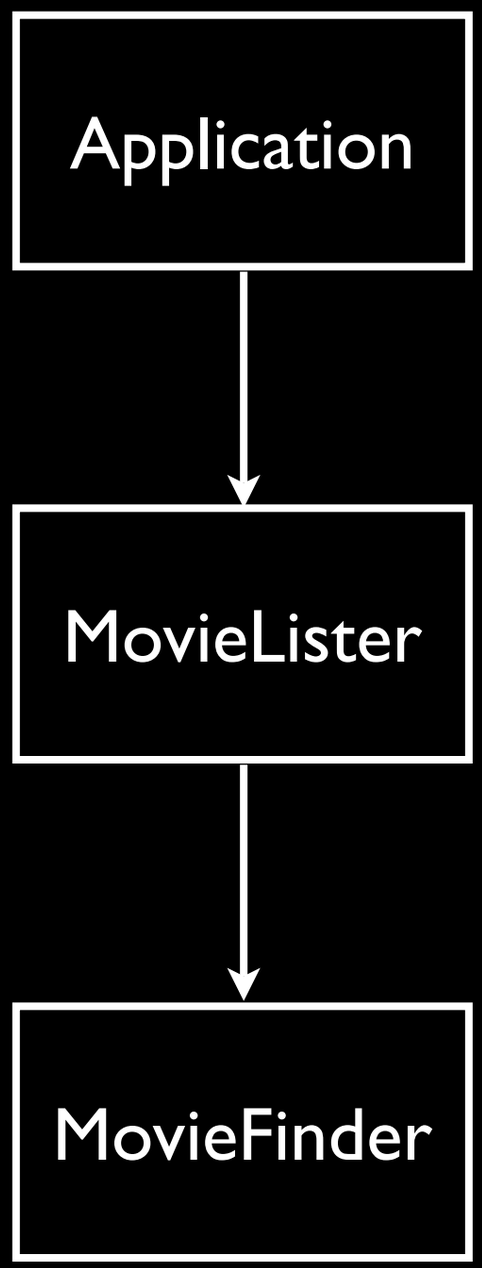
That’s all folks...

... almost

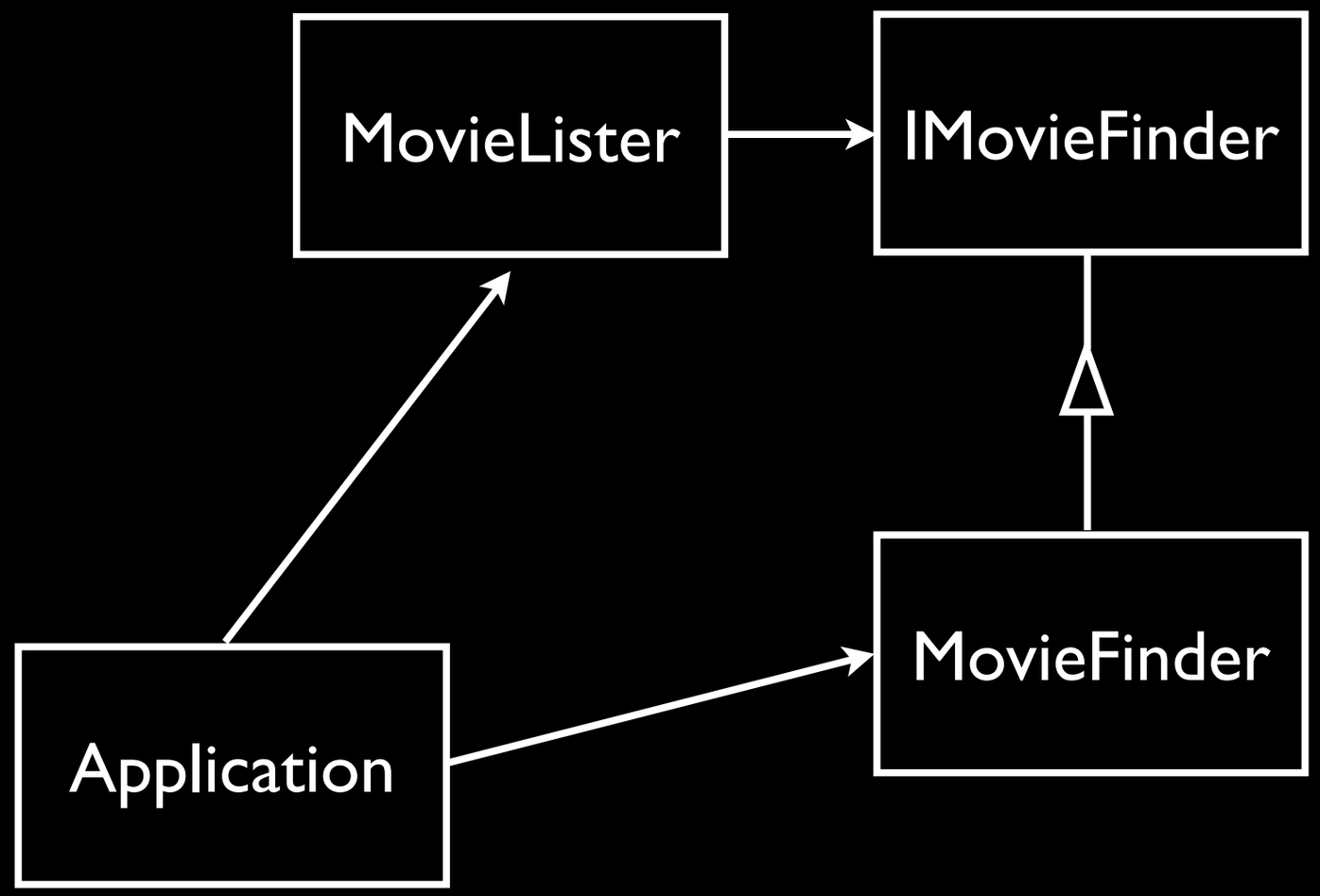
Where is decision 3

being made?

```
IMovieFinder finder =
  new CSVFileMovieFinder
   (applicationSettings.CSVMovieListFilena
   me);

IMovieLister lister =
  new MovieLister(finder);
```


Application
MovieLister
MovieFinder

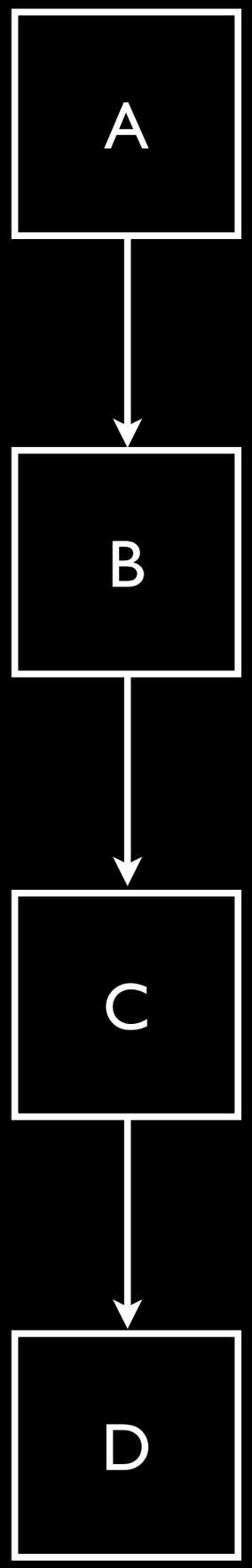
Becomes

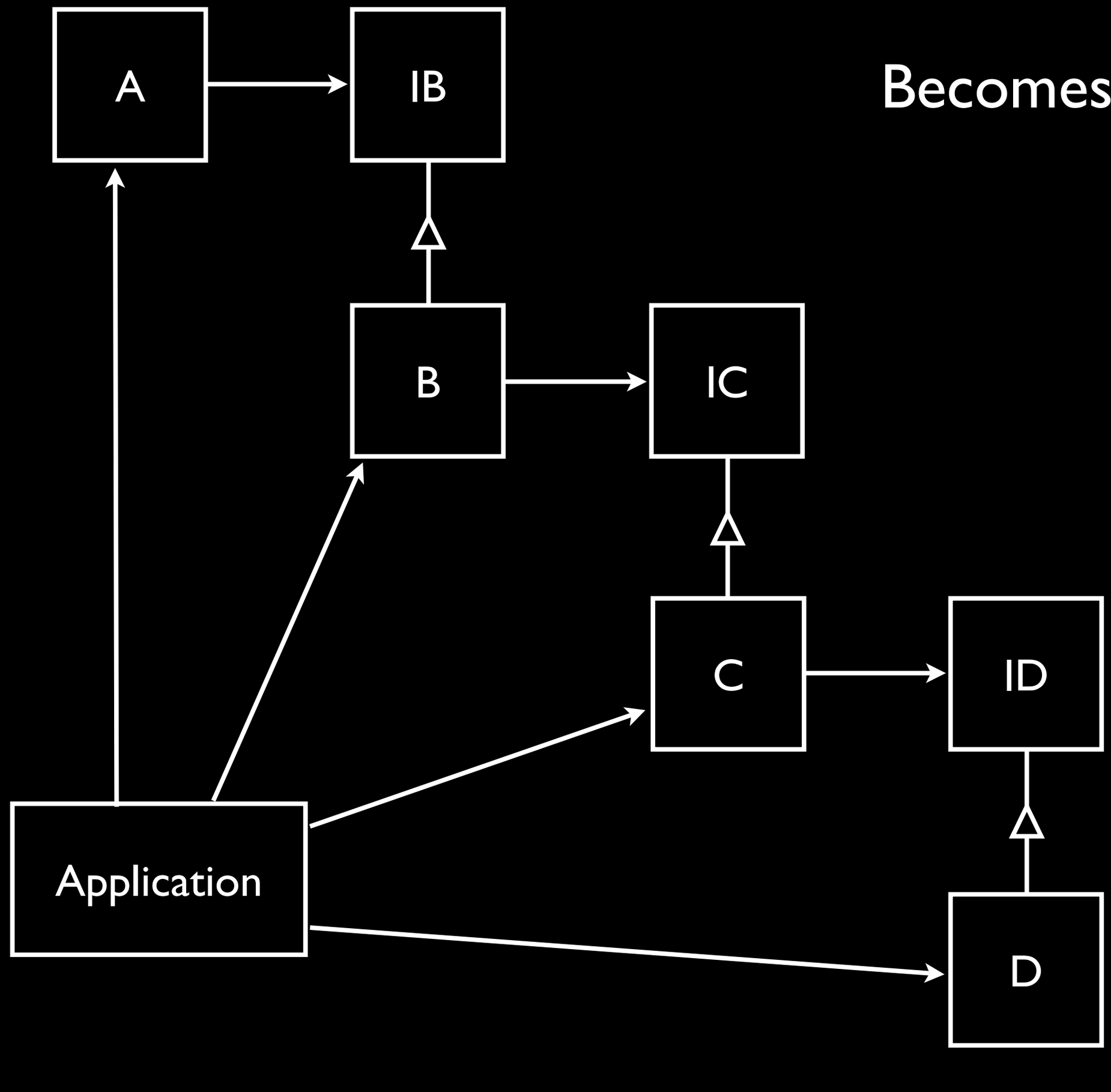


Wiring

The decision as to which implementation to use in which situation.

Problem:
Wiring Complexity


A
B
C
D


A
IB
Becomes
B
IC
C
ID
Application
D

```
IA a = new A();
```

becomes ...

```
ID d = new D();
IC c = new C(d);
IB b = new B(c);
IA a = new A(b);
```

Can get complex

```
IValve valv1 = new Valve();
IValve valv2 = new Valve();
IValve valv3 = new Valve();
IValve valv4 = new Valve();
...
IWheel wheel1 = new Wheel();
IWheel wheel2 = new Wheel();
IWheel wheel3 = new Wheel();
IWheel wheel4 = new Wheel();
...
IMotor motor = new V8Motor(...);
...
ICar c = new Car(axel1, axel2,
motor, ...);
```

Problem:
Circular Dependencies

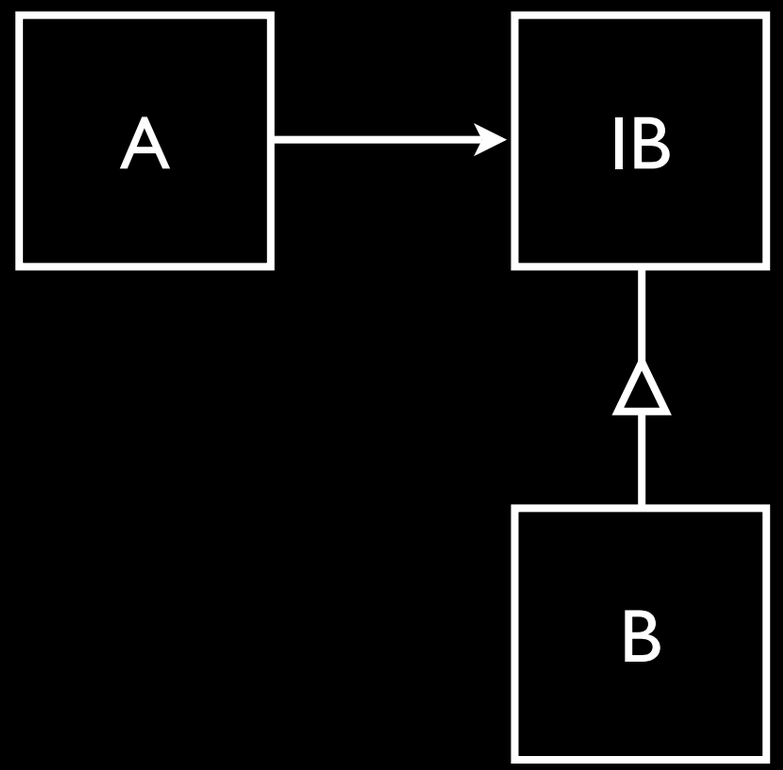
```
class View : IView
{
 private IController controller;
  ...
}
```

```
class Controller : IController
{
  private IView view;
  ...
}
```

```
IView v = new View(controller);

IController controller = new Controller
(view);

// ERROR!
```

NDependencyInjection

Declarative Wiring

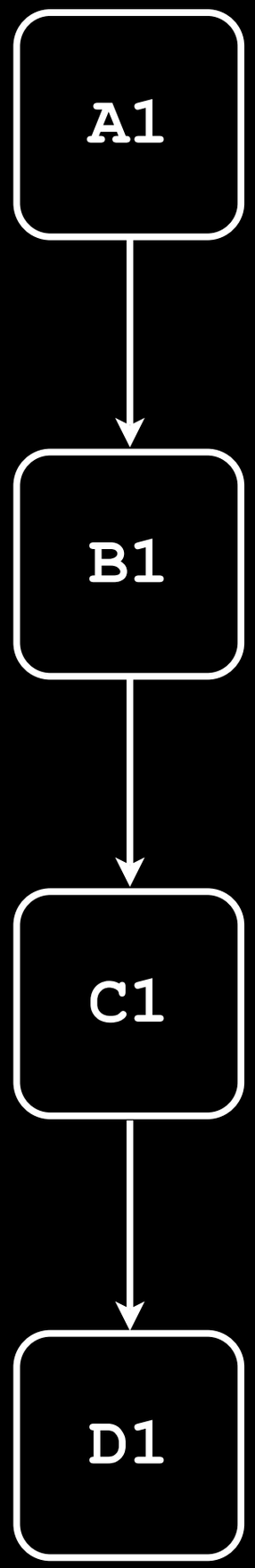

A
IB
B

```
// Classes define which dependencies
(services) they use

class A: IA
{
 public A(IB b) {...}
}

class B: IB
{}
```

```
// Wiring defines which classes provide
each service

SystemDefinition system = new
SystemDefinition();

system.HasFactory<A>().Provides<IA>();
system.HasFactory<B>().Provices<IB>();

IA a = system.Get<IA>();
```

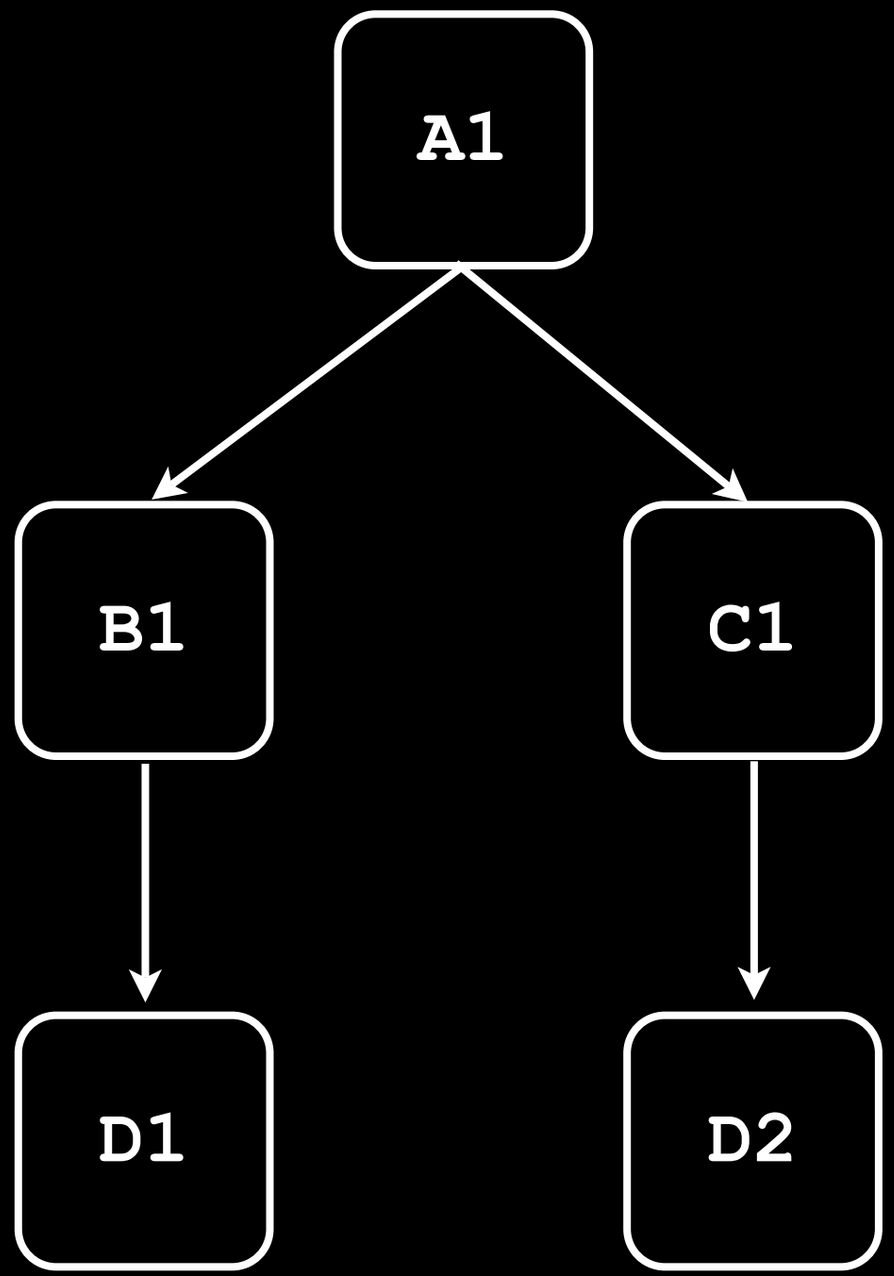
Object Diagram

Things this shape are instances **of classes**



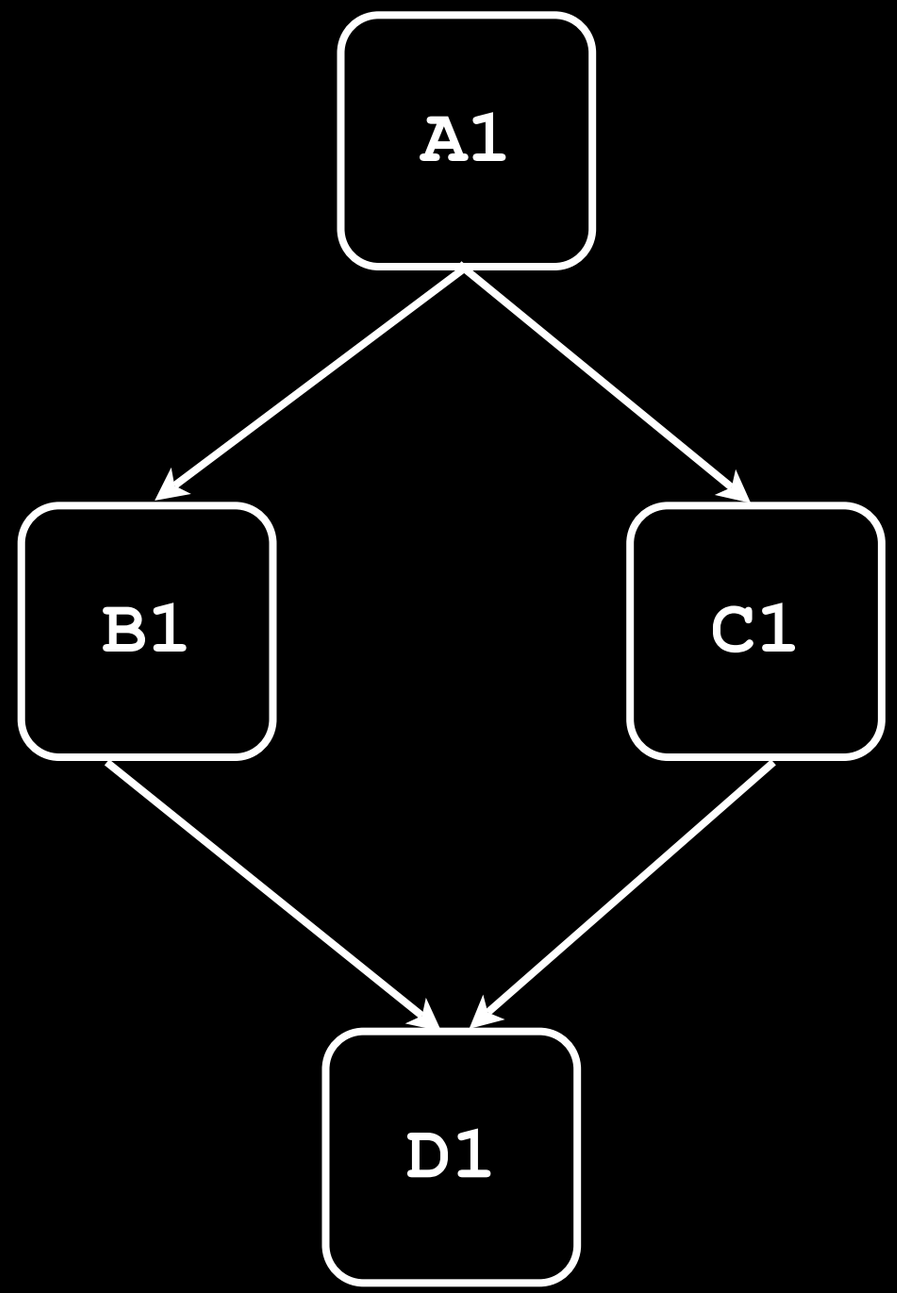
```
// Classes define which dependencies
(services) they use
class A: IA
{
 public A(IB b) {...}
}
class B: IB
{
 public B(IC c) {...}
}
class C: IC
{
 public C(ID d) {...}
}
class D: ID
{
 public D() {...}
}
```

```
// Wiring defines which classes provide
each service

ISystemDefinition system = new
SystemDefinition();

system.HasFactory<A>().Provides<IA>();
system.HasFactory<B>().Provides<IB>();
system.HasFactory<C>().Provides<IC>();
system.HasFactory<D>().Provides<ID>();

IA a = system.Get<IA>();
```


A1
B1
C1
D1
D2

```
// Classes define which dependencies
(services) they use
class A: IA
{
 public A(IB b, IC c) {...}
}
class B: IB
{
 public B(ID d) {...}
}
class C: IC
{
 public C(ID d) {...}
}
class D: ID
{
 public D() {...}
}
```

```
// Wiring defines which classes provide
each service

ISystemDefinition system = new
SystemDefinition();

system.HasFactory<A>().Provides<IA>();
system.HasFactory<B>().Provides<IB>();
system.HasFactory<C>().Provides<IC>();
system.HasFactory<D>().Provides<ID>();

IA a = system.Get<IA>();
```


A1
B1
C1
D1

same dependencies => classes are unchanged

```
ISystemDefinition system = new
SystemDefinition();

system.HasFactory<A>().Provides<IA>();
system.HasFactory<B>().Provides<IB>();
system.HasFactory<C>().Provides<IC>();
system.HasSingleton<D>().Provides<ID>();

IA a = system.Get<IA>();
```

D.S.L

Domain Specific Language

.Provides<IA>

Use this service each time you need an IA.

.HasFactory<A>

Create a new instance of A each time it is used.

```
system.HasFactory<A>().Provides<IA>();

IA a1 = system.Get<IA>();
IA a2 = system.Get<IA>();

Assert.AreNotSame(a1,a2);
```

.HasSingleton<A>

Use same instance of A for all services it provides.

```
system.HasSingleton<A>().Provides<IA>();

IA a1 = system.Get<IA>();
IA a2 = system.Get<IA>();

Assert.AreSame(a1,a2);
```

Circular dependencies

solved automatically

```
system.HasSingleton<View>()
  .Provides<IView>();

system.HasSingleton<Controller>()
  .Provides<IController>()
  .Provides<IViewListener>();

IController c = system.Get<IController>();

c.ShowDialog();
```

```
interface IViewListener
{
  void OnButtonXClicked();
}
```

```
interface IView
{
  void ShowStatusMessage(string message);
}
```

```
class View: Form, IView
{ public View(IViewListener listener){...}

  // some code to call the listener when the
  // button is clicked

  public void ShowStatusMessage(string
  message)
  {
    status.Text = message;
  }
}
```

```
class Controller: IController, IViewListener
{
 public Controller(IView view)
 {...}

 public void OnButtonXClicked()
 {
  // do something magic
  view.ShowStatusMessage("Magic has been
  done");
 }

 ...

}
```

Constructors must have no side effects.

.HasInstance(x)

Use a specific instance of x for all services it provides.

```
A a = new A();
system.HasInstance(a).Provides<IA>();

IA a1 = system.Get<IA>();
IA a2 = system.Get<IA>();

Assert.AreSame(a1,a2);
```

.HasSubsystem (builder)

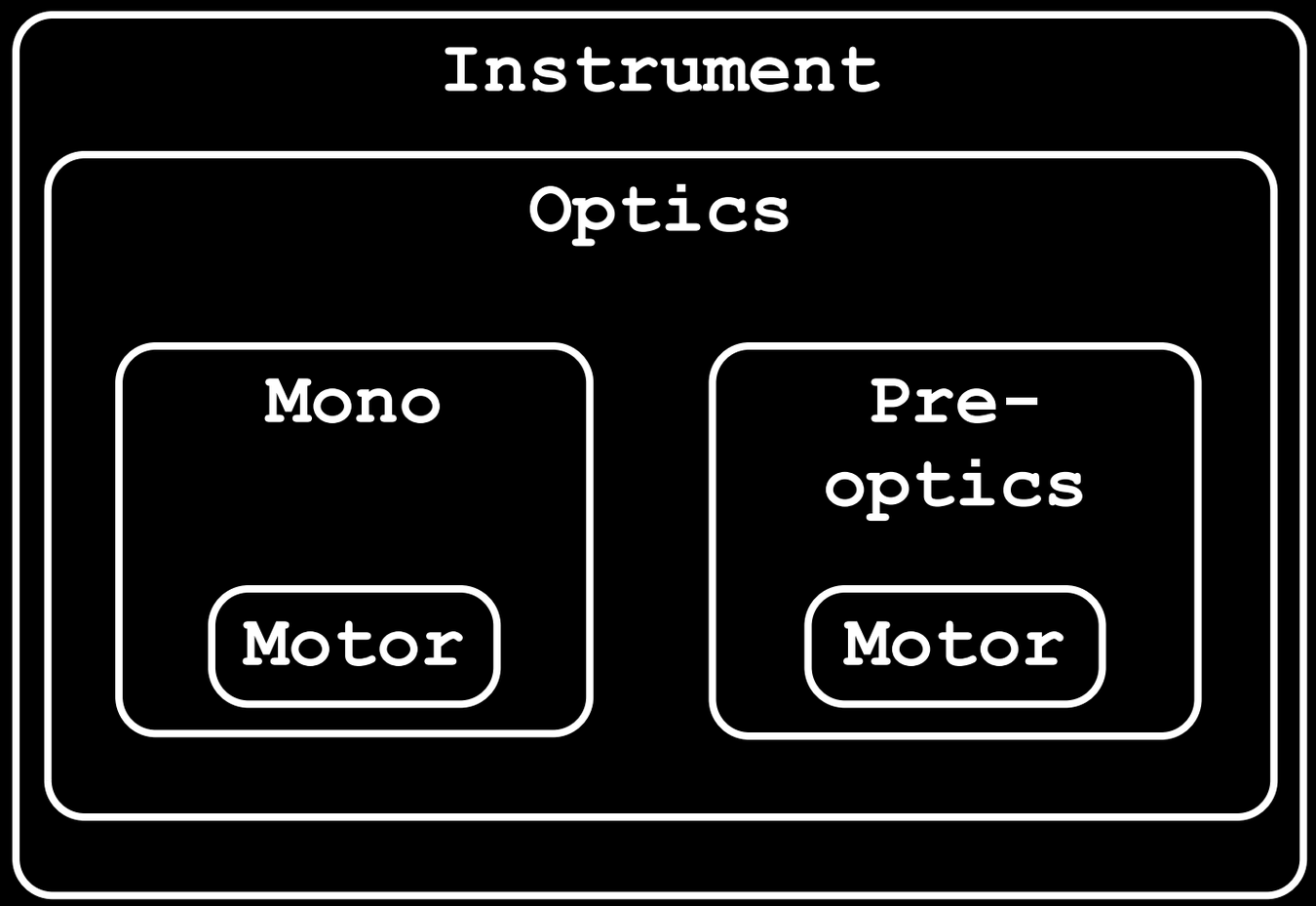
Use a specific builder to populate a subsystem with services which it can later provide.

```
ABuilder aBuilder = new ABuilder();

system.HasSubsystem(aBuilder)
  .Provides<IA>()
  .Provides<IB>();

IA a = system.Get<IA>();
IB b = system.Get<IB>();
```

```
class ABuilder : ISubsystemBuilder
{
 public void Build(ISystemDefinition system)
  {
    // define your subsystem here.
  }
}
```

Subsystems introduce layering

Systems & Subsystems



Layering solves wiring complexity.

.HasCollection
(builder , builder, ...)

.Provides<Thing[]>();
requires each builder to provide ‘Thing’

There is more...

.Broadcasts<IListener>

&

.ListensTo<IListener>

.BroadcastReceiver<

&

.ListenableBroadcast<

Not

Today!