Dependency Injection at AeroFS

Weihan, Feb 21, 2012

What is DI?

- A class declares other classes it depends on.
- These classes are instantiated and linked (injected) to the dependent at runtime instead of compile time.

For example

No DI

```
class B {
  A _a = new A();
}

main()
{
  B b = new B();
}
```

Manual DI

```
class B {
  B(A a) {
    _a = a;
  }
}

main()
{
  A a = new A();
  B a = new B(a);
}
```

For example

Automatic DI

```
class B {
  @Inject B(A a) {
    _a = a;
  }
}

main()
{
  Injector inj = ...;
  B b = inj.get(B.class);
}
```

Why DI?

- Enables unit testing => testable code
- Avoids boilerplates => clean code
- No dependency passing => Law of Demeter

How to DI?

- We use a customized version of Guice. Please read the manual at http://code.google.com/p/google-guice/
- Stay away from advanced functions (Because it’s evolving and not performant)

Why customize Guice?

- To allow singletons by default => less boilerplates => less errors

```
class FooModule extends ModuleAbstract {
  void configure()
  {
    bind(Scoping.class).toInstance(
        Scoping.SINGLETON_INSTANCE);
    ...
  }
}
```

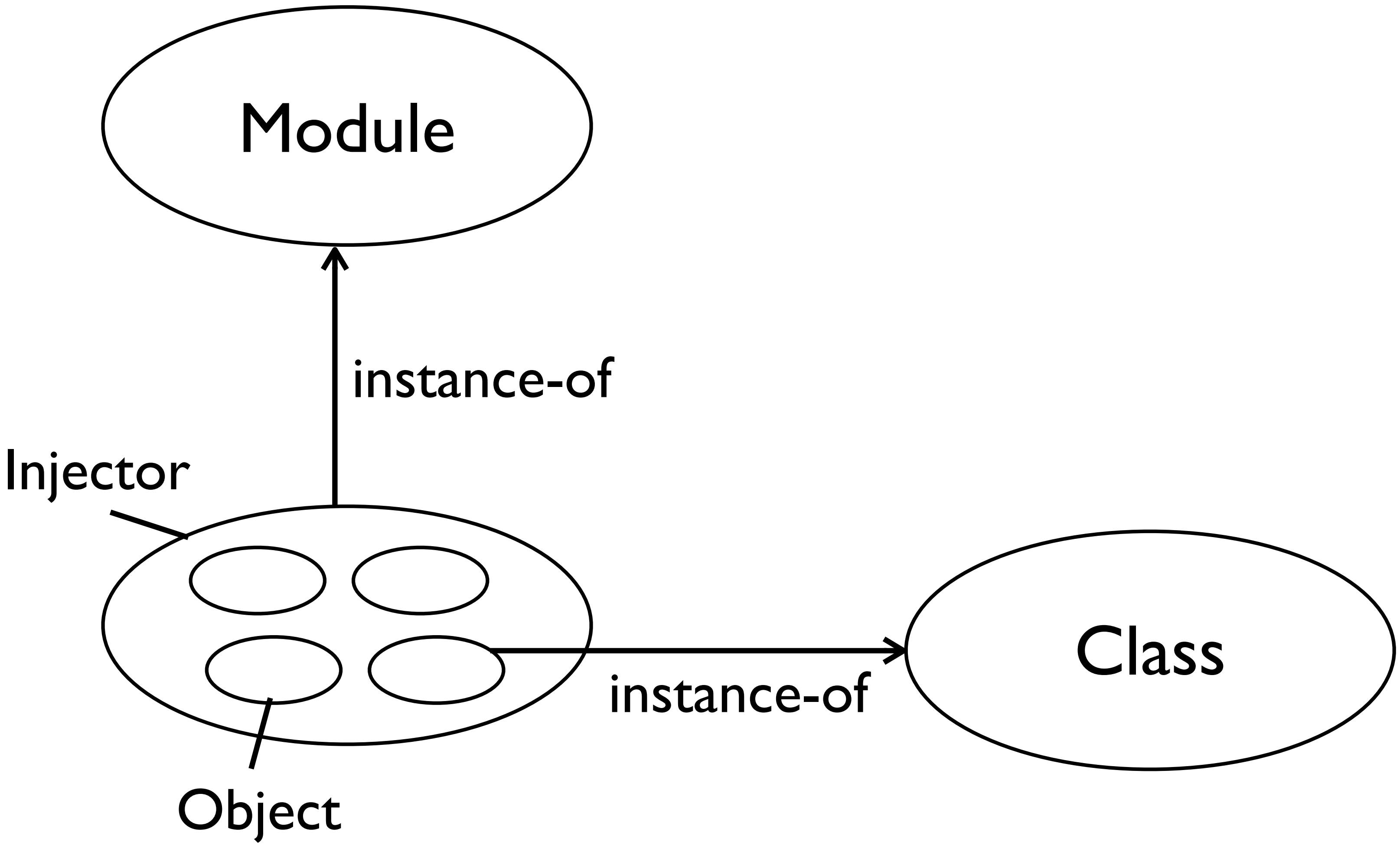
More conceptual stuff coming up...

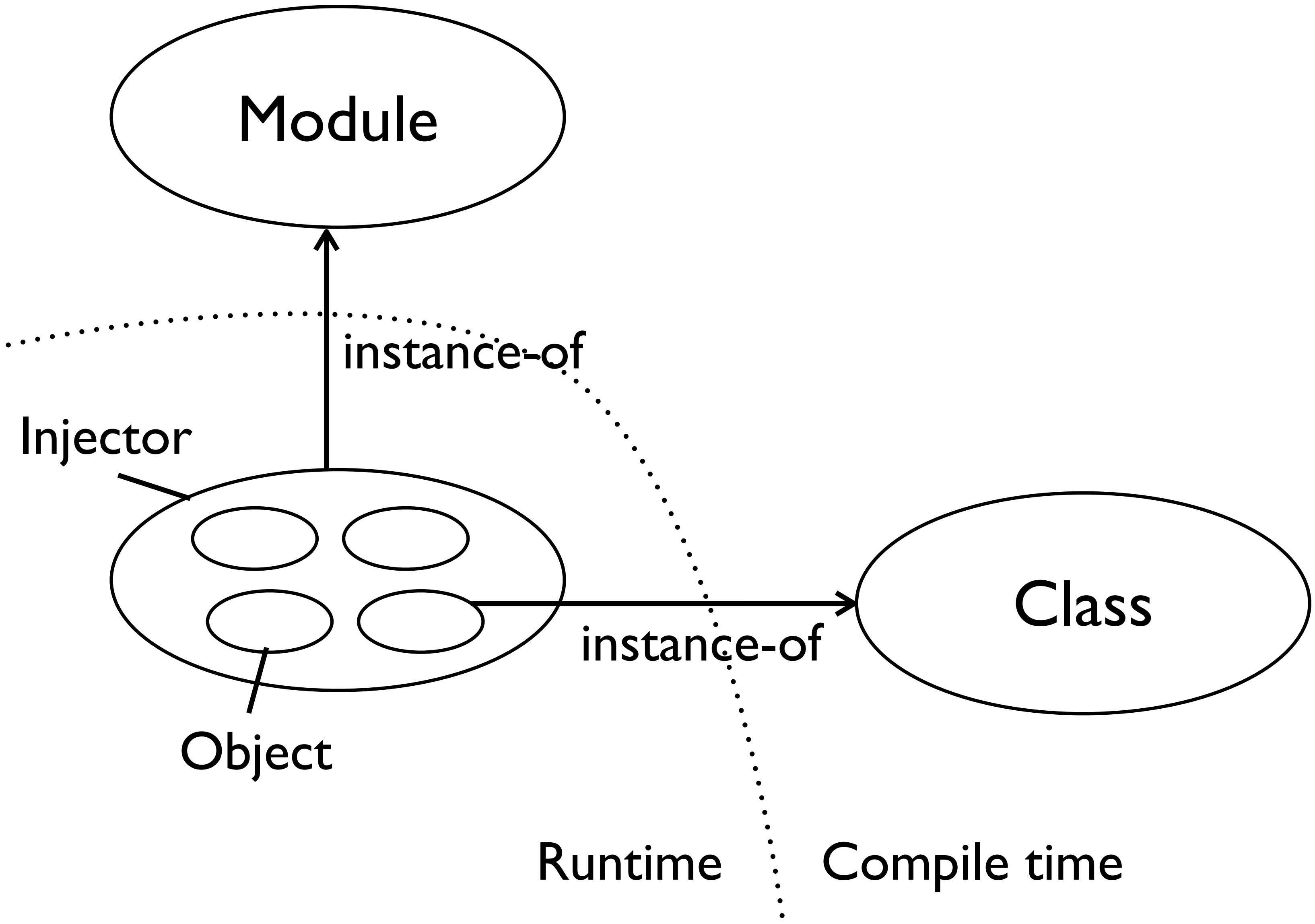
Packages vs modules

- Java packages
  > are compile-time namespaces
  > scope classes
- Guice modules
  > are runtime configurations
  > scope objects
- Packages and modules should correspond to each other

To be more precise...

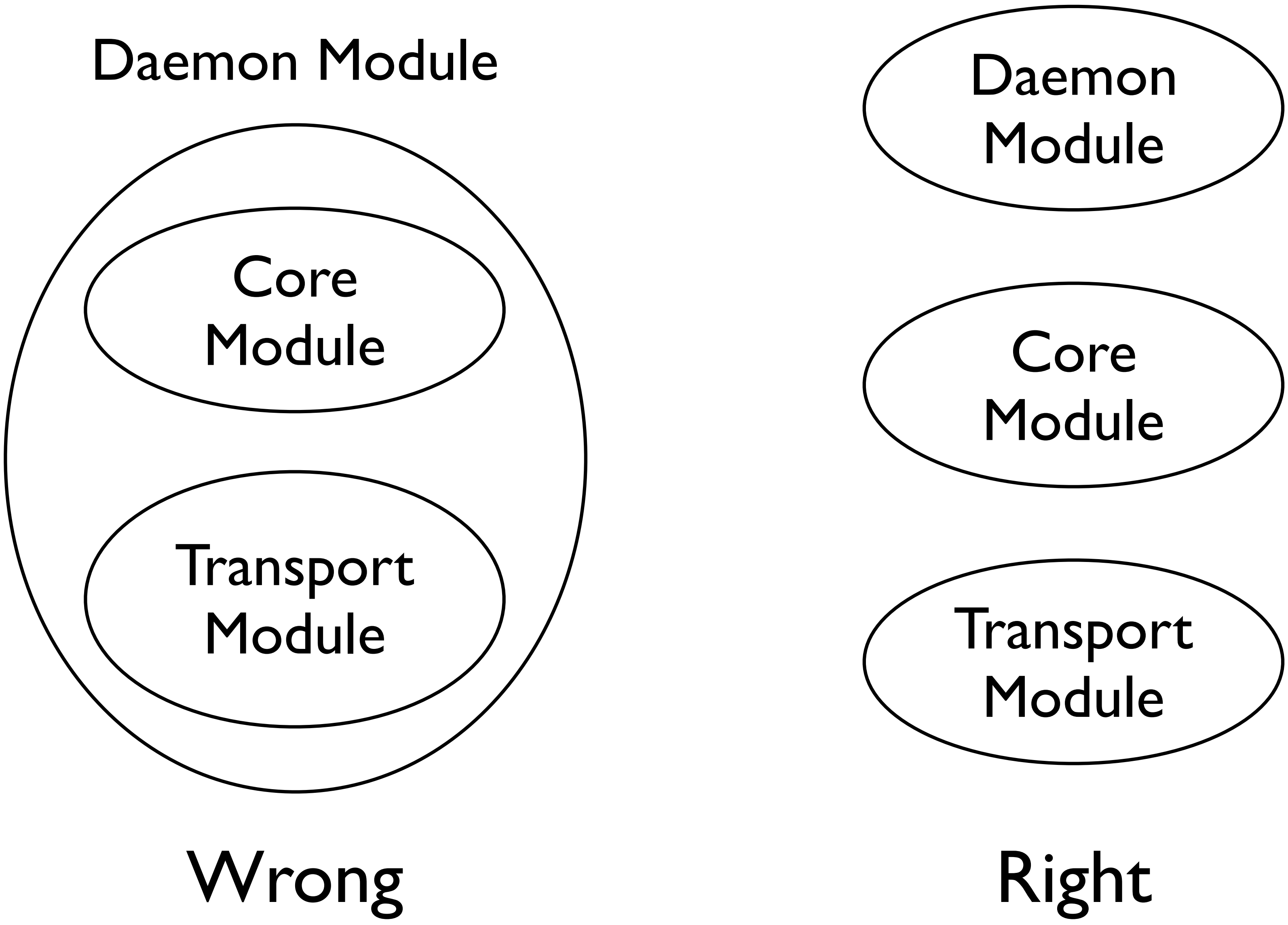
- Guice modules are
  > compile-time blueprint of configurations
  > similar to classes!
- Guice injectors are
  > runtime instantiation of configurations
  > similar to objects!


Module
instance-of
Injector
Class
instance-of
Object


Module
instance-of
Injector
Class
instance-of
Object
Runtime
Compile time

A corollary from module == class

- Observation: Classes refer to each other only to access static fields.
- Observation: Modules don't have static fields.
- Therefore, modules should not refer to each other.
- For example:


Daemon Module
Core Module
Transport Module
Wrong
Daemon Module
Core Module
Transport Module
Right

But what if one module interacts with another?

- It's the instances of modules, not modules themselves, that are interacting!
- So, refer to an injector instead:

```
class FooModule extends ModuleAbstract {

  public FooModule(Injector injBar)
  {
    ...
  }

}
```

But what if one module interacts with another?

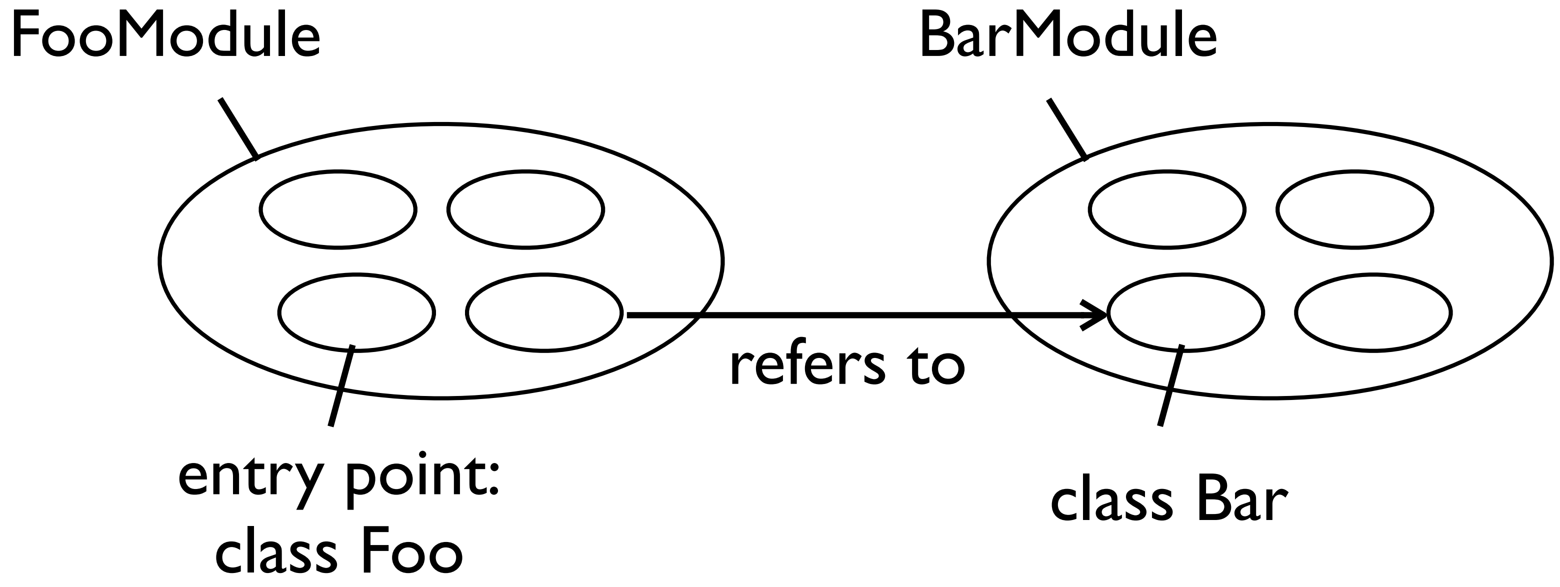
- It’s similar to classes referring to objects:

```
class Foo {

  public Foo(Bar bar)
  {
    ...
  }

}
```

Example


FooModule
BarModule
refers to
entry point:
class Foo
class Bar

Example

```
class BarModule extends AbstractModule {
  configure() {...}
}

class FooModule extends AbstractModule {
  Injector _injBar;
  Foo(Injector injBar) {
    _injBar = injBar;
  }
  @Provide
  Bar provideBar() {
    return _injBar.getInstance(Bar.class);
  }
  configure() {...}
}
```

Example

```
void main() {
  injBar = Guice.createInjector(
             new BarModule());
  injFoo = Guice.createInjector(
             new FooModule(injBar));
  injFoo.getInstance(Foo.class).start();
}
```

See DaemonProgram$inject for more examples

Factories & DI

- Both factories and DI can replace “new”
- So, when to use what?

Factories & DI

- DI describes **depend-on** relationship
- Factories describe **create** relationship
- Use DI if "A depends on B" is more sensible than "A creates/owns B"; use factories otherwise.

How to write a factory

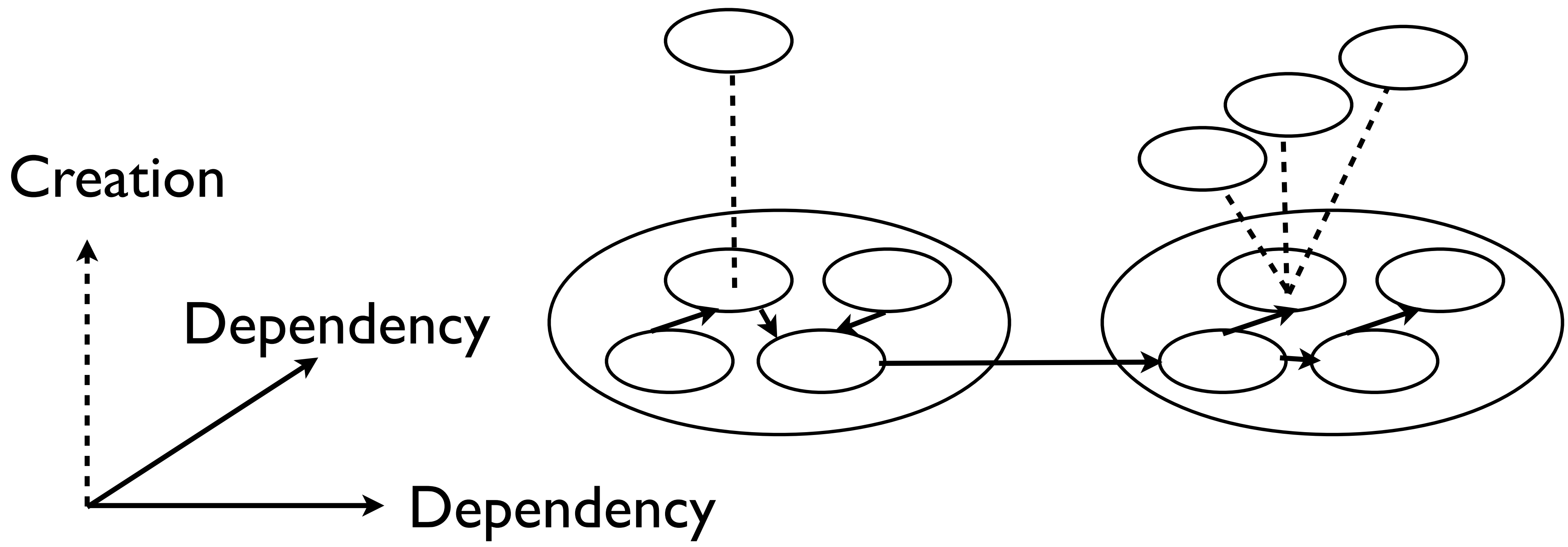
```
class B {
  Dep1 _d1; Dep2 _d2;
  B(Dep1 d1, Dep2 d2, int param) {...}

  public static Factory {
    Dep1 _d1, Dep2 _d2;
    @Inject Factory(Dep1 d1, Dep2 d2) {...}
    B create(int param) {
      return new B(d1, d2, param);
    }
  }
}

class A {
  B.Factory _factB;
  @Inject A(B.Factory factB) {...}
}
```

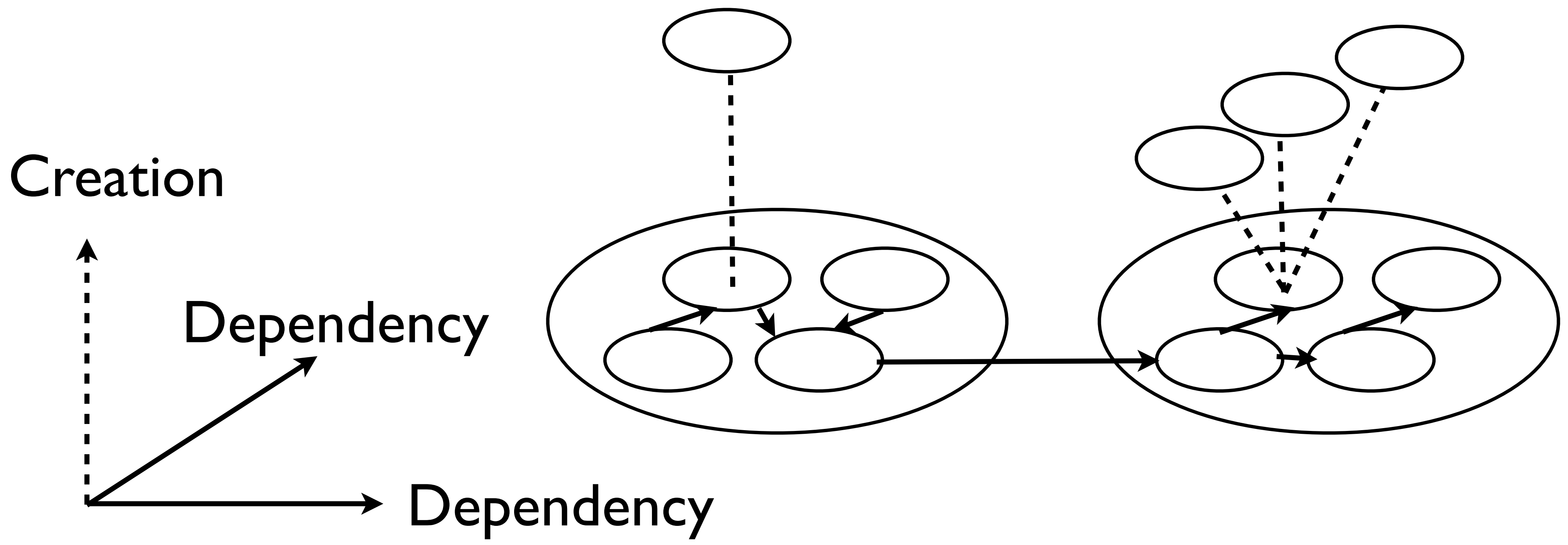
More on factories

- I propose the following model that unifies depend-on and create relationships



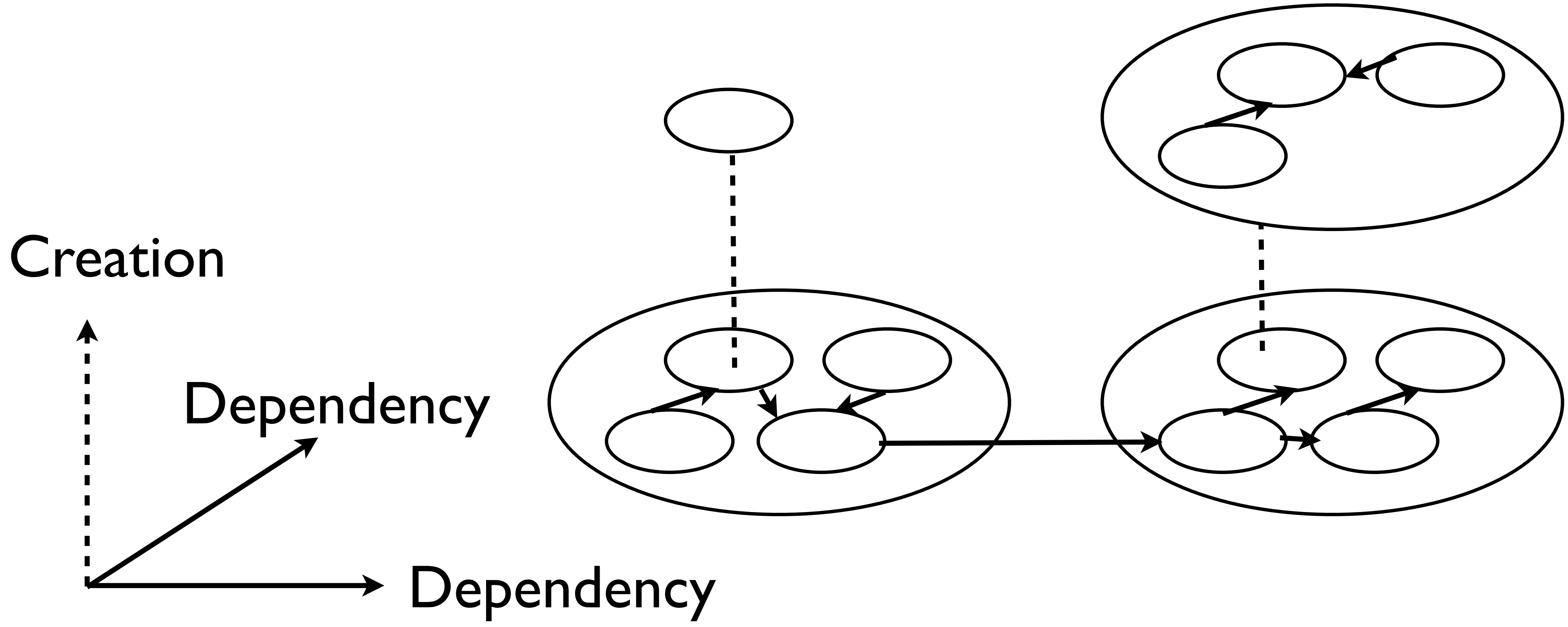
More on factories

- DI operates within a **dependency plane**
- Each object creation generates a new plane
- So, no DI is possible among created objects



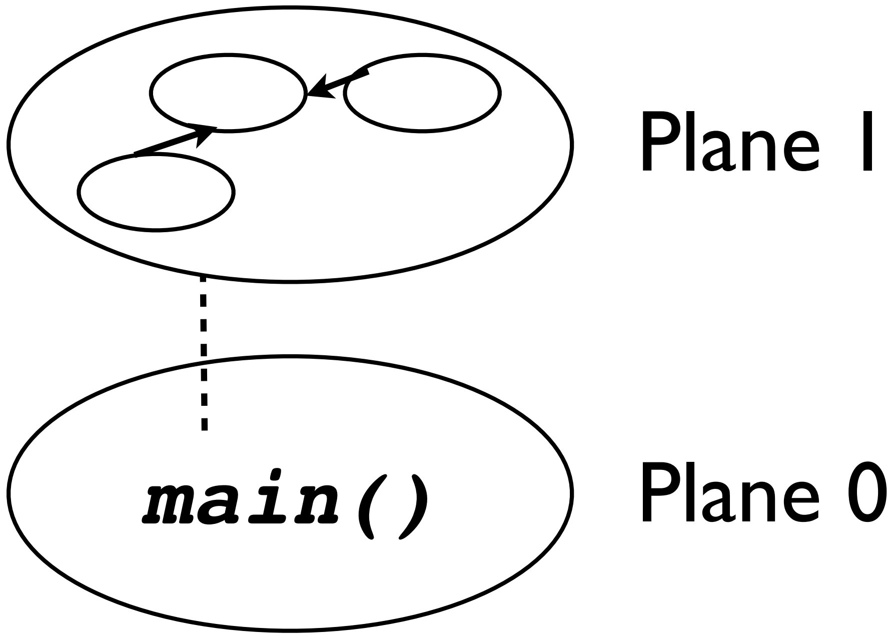
More on factories

- Solution? The factory creates injectors instead of individual objects.



More on factories

- In fact, main() runs on one plane, manufacturing the first injector onto another.



Singleton or not?

- Me: singleton should be the default in DI
- You: WTF? I need non-singletons!
- Me: when?
- You: when there are dynamically created objects of the same type, ...
- You: ... or when my classes depend on multiple objects of the same type.

When there're dynamically created objects of the same type ...

- It usually indicates the need for factories.
- Solution: use factories to create either objects or injectors
- Factories are singletons on the current dependency plane

When classes depend on multiple objects of the same type ...

- They are actually different types!

```
class Person {
  Person(String name, String address);
}
```

- The two parameters control the system's behavior in different ways
  => they are not interchangeable
  => this violates LSP
  => they have different *behavioral* types!

When classes depend on multiple objects of the same type ...

- Guice's way to solve the problem: using annotations:

```
class Person {
  Person(@Name String name, @Address String address);
}
```

- But I think the type system is here to solve type problems.

Type wrappers

- How about type wrappers:

```
class Person {
  Person(NameString name, AddressString address);
}
```

- Type wrappers add behavioral typing on top of literal types.
- Each wrapper only need one instance.
- Singleton!

Type wrapper: example

```
class Queue {...}

class SubsysAQueue extends Queue {}

class SubsysBQueue extends Queue {}

class SubsysA {
  @Inject SubsysA(SubsysAQueue q) {...}
}

class SubsysB {
  @Inject SubsysB(SubsysBQueue q) {...}
}
```

By-product:
a stronger type system
=> better error prevention

Summary

- Non-singletons are rarely needed in DI

Cyclic dependencies

- Guice resolves cyclic dependencies using proxy objects
- Impossible if depending on concrete classes
- To workaround, use:

  ```
  @inject inject(Dep1 dep1, Dep2 dep2) {...}
  ```

  instead of constructors
- Even better, avoid cycles!