Managing dependencies

Andrew P. Black

Based on Chapter 3 of POODR

Why Dependencies?

- A single object can't do everything, so it will have to talk to other objects

For any desired behaviour:

- an object can either:
  1. know it itself,
  2. inherit it, or
  3. know another object that knows it.
- This chapter is about 3.

Collaboration

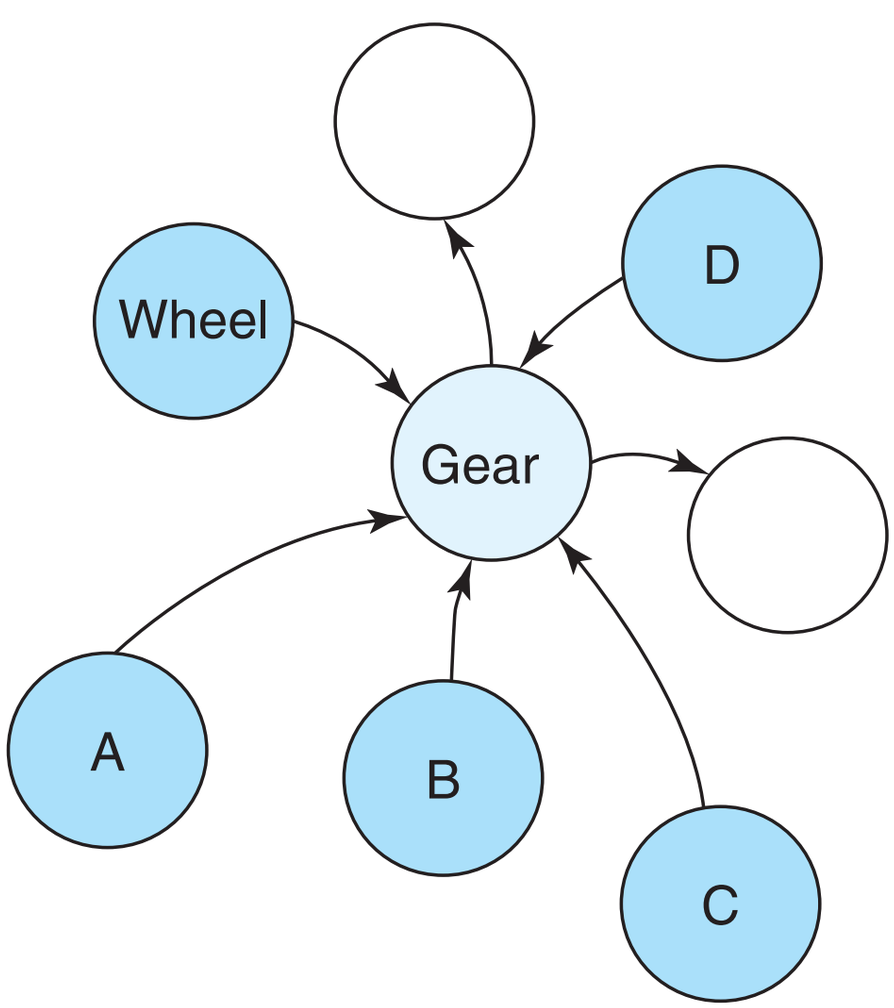
- Collaborating with another object introduces a dependency
- That is, if the other object changes, you might be forced to change too.

Dependencies exist when:

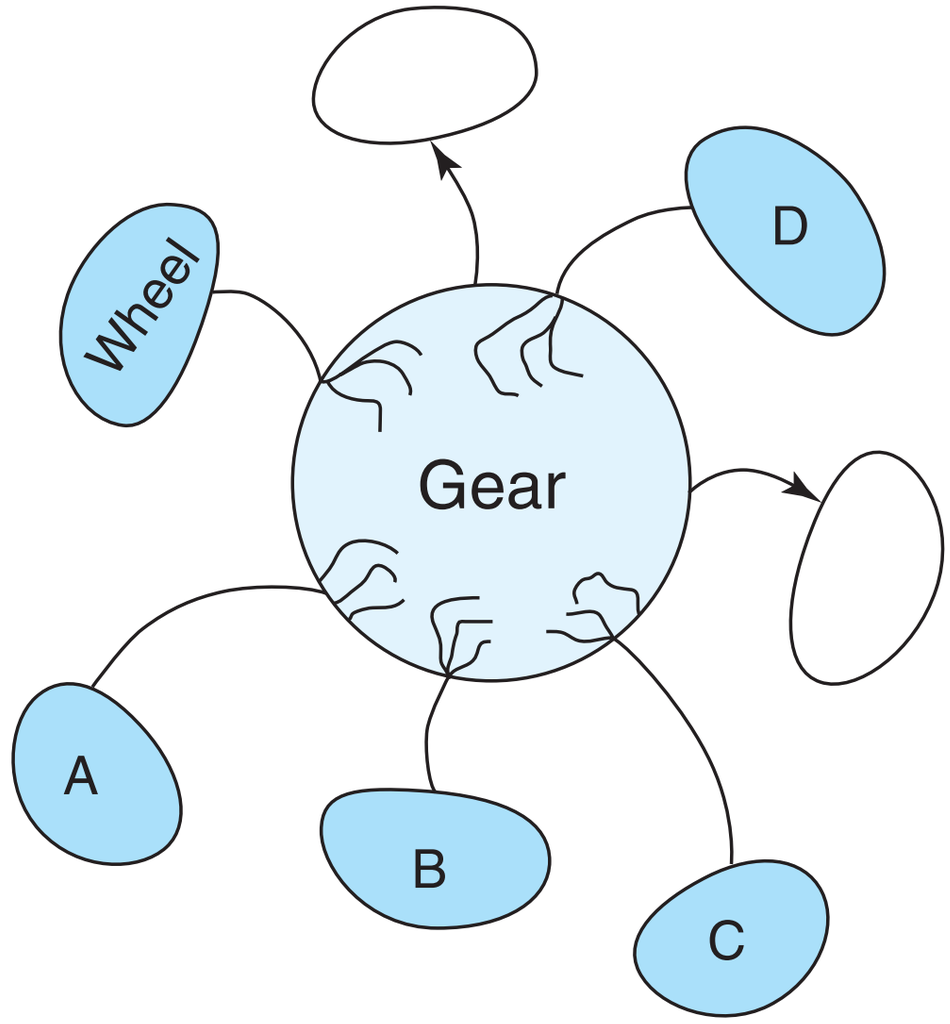
- an object has a dependency when it knows:
  - The name of another object (Metz says “class”)
  - the name of a *request* that it makes on someone other than **self**
  - the arguments of a request (number and position)

Why limit dependencies?

- The more dependencies you have, the greater are the chances that minor tweaks turn into major undertakings.
- Dependencies create coupling
  - an object and its dependencies act like a single big object; you can't reuse (or test) the object without also reusing (or testing) its dependencies too.



Gear depends on wheel, A, B, C and D



Gear and its dependencies act like one thing

Figure 3.1 Dependencies entangle objects with one another.

Law of Demeter

- Law of Demeter violation is a particular case of case 3: knowing another object that knows … another object that can respond to your request.
- Design **interfaces** to avoid it

Dependencies in Testing

- Tests *must* depend on code
- write tests to avoid over-coupling

Dependencies in Gear

- Gear depends on *Wheel* class
  - Metz: “Gear becomes less useful when it knows too much about other objects; if it knew less, it could do more”
  - My paraphrase: smart objects know how to delegate
- Instead, give a gear a wheel instance when the gear is created.
  - Called: “Dependency Injection”

(Removed) Dependencies in Gear

```
1  method gearWithChainRing(chain) cog(c) {
2      gearWithChainRing(chain) cog(c) wheel(defaultWheel)
3  }
4
5  class gearWithChainRing(chain) cog(c) wheel(w) {
6      method chainring { chain }
7      method cog { c }
8      method wheel { w }
9
10     method ratio {
11         chainring / cog
12     }
13
14     method gearInches {
15         ratio * wheel.diameter
16     }
17 }
18
19 class defaultWheel {
20     method diameter {
21         EnvironmentException.raise "the user should have provided a wheel if they want a
               diameter"
22     }
23     method asString {
24         "please set the wheel"
25     }
26 }
27
```

- If you can't remove class dependencies, isolate them by:
  - moving them to instance initialization, or
  - moving them to their own method.

Gear knows:

- what requests some *other objects* understand
- replace these requests (*wheel.diameter*) with self-requests:
- add a *diameter* method to self
  - *isolates* the knowledge that *wheel* understands *diameter*

Argument–Parameter Dependencies

- Knowing the parameters of a request is a dependency
- Sometimes, you can pass a dictionary containing the arguments.
- This may be a good approach if the parameters are likely to grow or shrink, or if you need defaults.

Pros and Cons of Dictionary Parameters

- Fixed named arguments are simpler today, but increase the risk that changes will be harder tomorrow.
- There is still a dependency on the keys used for the parameters
  - and it can’t be checked statically

- Using *new* for instance creation means that confusion over the meaning and position of parameters is likely
- Grace avoids this problem by encouraging you to give *intention revealing names* for instance creation method (aka classes)

- Using many creation-time arguments means that confusion over the meaning and position of parameters is likely
- Having a *long parameter list* is a code smell: get rid of it
  - use *replace parameter with method*, *preserve whole object*, and *introduce parameter object*.

- Dictionaries make it easy to make arguments optional and to have defaults
- Fixed parameter lists can lead to a combinatorial explosion of variants with and without the optional parameters

Defaults

- Default parameters are best specified:
  - in a separate method for each default value
  - in a defaults method (requires merge of dictionaries)

External modules

- if you don't control the interface of the offending request,
  - wrap it in a factory method that you *do* own and control.
- put that method on a singleton object (Ruby module)
  - In Grace, you can just leave it as a factory method at the top-level of your own module

Direction of dependencies

- The direction of a dependency matters
- gear depends on wheel ≠ wheel depends on gear
  - You can make either “work”
- getting direction “right” means that your application will be pleasant to work on and easy to maintain into the future.

What's "Right"?

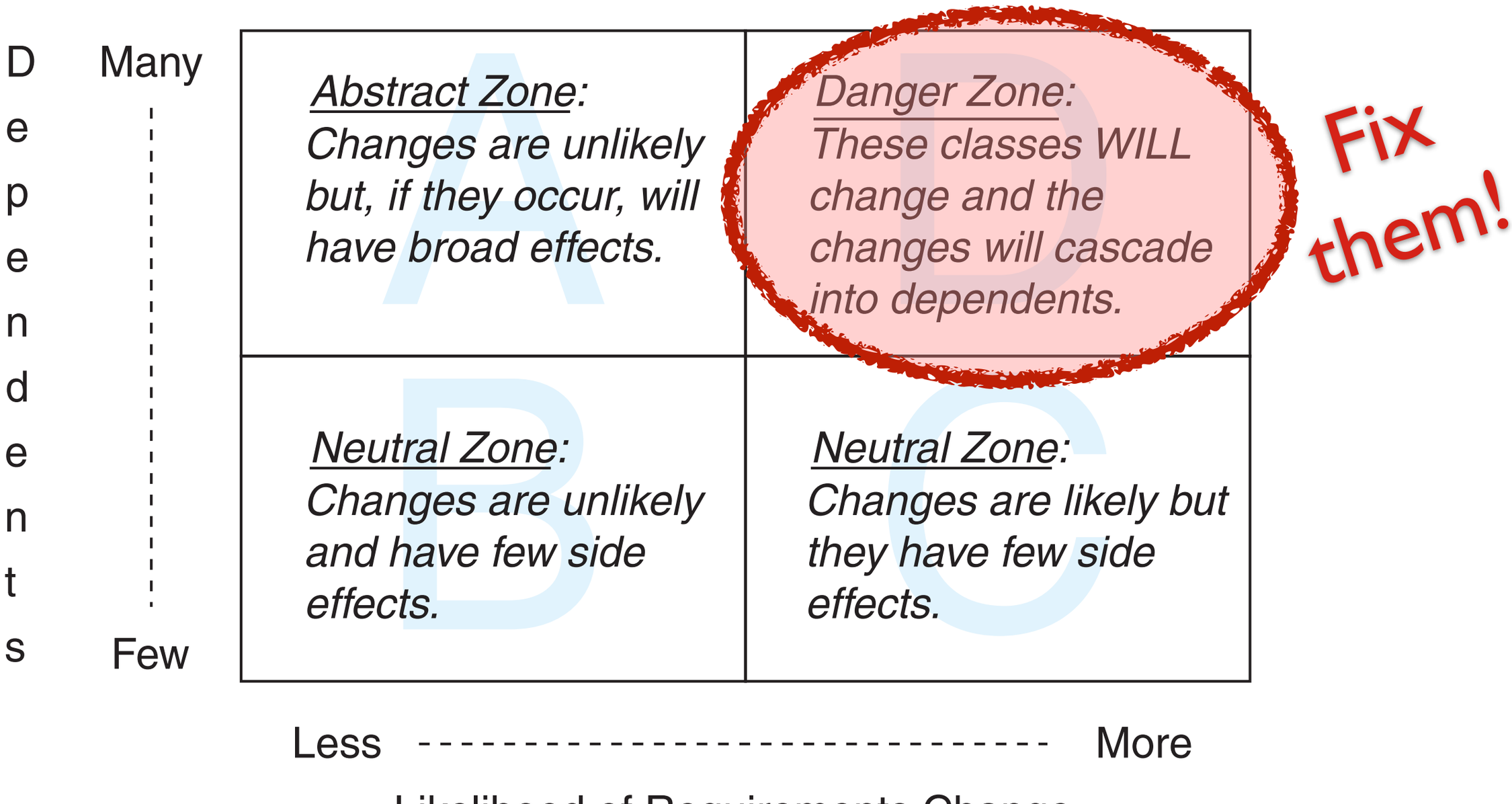
- Key idea: depend on things that will change less often than you do.
- Some objects are more likely than others to have changes in requirements
  - Basic libraries, vendor frameworks, application code
- implementations are more likely to change than interfaces
- depend on abstractions rather than on concretions

Beware!

- Classes with lots of dependents are unlikely to ever change!
- Metz: “Your application will be permanently handicapped by your reluctance to pay the price required to make a change to this class”

Classifying dependencies

- Put them on this grid:



Summary

1. *Injecting* dependencies creates self-contained objects that can be reused in ways that you might never have guessed
2. *Isolating* dependencies lets you react to changes, when they come, quickly and easily
3. Depend on things that change less often than you do.
   - When possible, depend on abstractions rather than concretions.