Dependency Inversion Principle

Depend on abstractions not on concrete classes.

Martin, 1994

Some forms of dependency are desirable

* … because we want objects to interact
* … other forms of dependency are undesirable
* … because they break reuse, flexibility, and maintenance.

Simply using objects is insufficient to create robust, reusable, and maintainable systems.

* A certain pattern of dependence is necessary which
* … 1. Supports communication among objects
* … 2. Isolates reusable from non-reusable elements, and
* … 3. Blocks the propagation of change due to maintenance

Dependency:

* Consider a copy program,
* … in which a Copy class depends on Keyboard and Printer classes.
* The problem: The Copy class does not work with other input and output protocols.
* The solution: make the Copy class depend on Input and Output abstractions.
* Now, we can reuse Copy with any implementations of the Input/Output abstractions.

Good dependencies are stable: they are highly unlikely to change.

Achieving Stability: the most stable classes are

* Independent: they have no dependencies (or as few as possible).
* Responsible: they have many dependents (changing them has a large impact).
* Stable classes have few reasons to change, and many reasons not to change.

Class Categories:

* Categories are groups of classes that can be reused only in collaboration with each other.
* These highly cohesive classes for the three rules (in order of importance):
* 1. Change together; if one changes, they likely all must change.
* 2. Require each other for reuse; using one means using all of them.
* 3. Share some common goal.
* 4. Should be released with release numbers.

The most important dependencies to manage are those between categories.

* **Focus on the stability, responsibility, and independency of categories not of classes.**

**Metric of Stability (I)**

* Count the dependencies that interact with a class category.
* Afferent Couplings (Ca) – the number of classes outside the category, that depend on classes inside the category.
* Efferent Couplings (Ce) – the number of classes inside the category, that depend on classes outside the category.
* Instability = (Ce / (Ca + Ce)), 0 is very stable and 1 is very unstable.

Not all categories ought to be stable.

* To make a stable category also be flexible, compose the category with abstract classes.
* To discourage dependence on unstable categories, compose the category with concrete classes.
* There are two good kinds of class categories:
* 1. maximally stable and maximally abstract
* 2. maximally instable and maximally concrete.

**Metric of Abstractness (A)**

* Abstractness = (# of abstract classes / # of classes), 0 means concrete and 1 means abstract.

Class Category Classification and the "Main Sequence"

|  |  |  |
| --- | --- | --- |
| Abstractness | Stability | Implication? |
| 0 – very concrete | 0 – very stable | Rigid design. |
| 1 – very abstract | 0 – very stable |  |
| 0.5 - abstract | 0.5 - somewhat stable | Balanced design |
| 0 – very concrete | 1 – very unstable |  |
| 1 – very abstract | 1 – very unstable | Impossible design: abstract, no dependents. |

**Metric of Distance from the Main Sequence (Dn)**

* Dn = |(A + I - 1)|, where better designs have a Dn that is close to zero.

Martin, 1996

Claims that the DIP principle is a structural implication of the Open-Closed and the Liskov Substitution principles.

"The structure that results from rigorous use of [the Open-Closed and the Liskov Substitution] principles can be generalized into a principle all by itself. I call it “The Dependency Inversion Principle” (DIP)."

What are the characteristics of a bad design?

* Rigidity: every change affects too many parts of the system.
* Fragility: when we make a change, unexpected parts of the system break.
* Immobility: it's hard to reuse, because we cannot disentangle it from its current application.

The opposite: flexible, robust, reusable

The interdependence of modules make a design rigid, fragile, and immobile.

Dependency Inversion

* Both high-level and low-level classes should depend on abstractions
* Neither should know about each other.
* I.e. Copy should take a Reader and Writer abstraction,
* Instead of taking KeyboardReader and a PrinterWriter classes.

The Dependency Inversion Principle:

A. HIGH LEVEL MODULES SHOULD NOT DEPEND UPON LOW LEVEL MODULES. BOTH SHOULD DEPEND UPON ABSTRACTIONS.

B. ABSTRACTIONS SHOULD NOT DEPEND UPON DETAILS. DETAILS SHOULD DEPEND UPON ABSTRACTIONS.

Why "inversion?"

* Traditional procedural design methods
* ... tend to have high-level modules depend on low-level modules,
* … and to have abstractions depend on details.
* … thus, good Object Oriented design inverts this tradition.

The problem with the traditional approach

* ...is that changes to low-level modules,
* ...have effects on high-level modules,
* ...and high-level modules become hard to reuse
* ... without bringing along the low-level modules.

Layering

* When layering, avoid making dependencies transitive.
* Do this by depending on abstractions not on concrete classes.
* Doing this makes the design more flexible, durable, and mobile.

Why would a change in X have any effect at all on Y?

* A change in X can have an effect on Y,
* … when the interface is not separated from the implementation.
* The way to avoid this is via a pure abstract class (aka an interface).

Example: Lamp <--> Button

* Naïve approach:
  + ...the Button contains a Lamp and sends an On/Off message to it.
  + ...and the Button class depends on the Lamp class.
* Inverted approach:
  + … the Button contains a ButtonClient
  + … a ButtonImplementation implements Button
  + … a Lamp implements ButtonClient
  + … and, a LampAdapter class can wrap third-party lamps.

Vocabulary

* Policy, high level
* Mechanism, mid level
* Utility, low level