1. Use Interfaces

Only show what needs to be shown. If `A` only needs one method from `B`, don't hand it the whole kitchen sink. Instead, define an interface (or an abstract base class) that includes just that method, and let `B` implement it.

> Think of this like giving someone a vending machine, not access to your kitchen.

```
interface IWorkable
{
    void DoWork();
}

class B : IWorkable
{
    public void DoWork()
    {
        // complex internal logic hidden
    }
}

class A
{
    private readonly IWorkable _worker;

    public A(IWorkable worker)
    {
        _worker = worker;
    }

    public void Execute()
    {
        _worker.DoWork();
    }
}
```

`A` depends only on the interface `IWorkable`, which exposes exactly what it needs (`DoWork`), hiding all other details inside `B`.

2. Apply Abstraction

Only show what needs to be shown. If `A` only needs one method from `B`, don't hand it the whole kitchen sink. Instead, define an interface (or an abstract base class) that includes just that method, and let `B` implement it.

> Think of this like giving someone a vending machine, not access to your kitchen.

Hide complexity behind clear, meaningful method names. `B.DoWork()` might do 100 things internally — but `A` doesn't care how it's done, only that it's done.

This is the **black-box principle**: you tell the machine what you want, not how to do it.

```
// Class A only calls a simple method on B
class A
{
    private B _b = new B();

    public void DoWorkInA()
    {
        // A doesn't know or care about B's internal details
        _b.DoWork();
    }
}

// Class B does a lot of internal stuff, but exposes a simple method
class B
{
    public void DoWork()
    {
        Step1();
        Step2();
        Step3();
        // Potentially many more steps, complex logic, etc.
    }

    private void Step1()
    {
        Console.WriteLine("Step 1 done.");
        // complex stuff here
    }

    private void Step2()
```

```
    {
        Console.WriteLine("Step 2 done.");
        // complex stuff here
    }

    private void Step3()
    {
        Console.WriteLine("Step 3 done.");
        // complex stuff here
    }
}
```

How this fits the black-box principle:

- Class `A` only sees a simple interface: `_b.DoWork()`.
- Class `B` handles all the internal complexity in private methods.
- Class `A` doesn't care how `B` does the work, only that it is done.

3. Embrace Dependency Injection

Let `A` receive its collaborators (`B`, etc.) from the outside—rather than reaching out and constructing them itself. This makes `A` flexible, testable, and decoupled from specific implementations.

> Think of it as someone bringing you a new teammate, instead of you going out and hiring one on your own.

```
interface IWorker
{
    void DoWork();
}

class B : IWorker
{
    public void DoWork()
    {
        // complex internal logic hidden
    }
}

class A
{
    private readonly IWorker _worker;

    // Dependency injected from outside
    public A(IWorker worker)
    {
        _worker = worker;
    }

    public void Execute()
    {
        _worker.DoWork();
    }
}
```

`A` doesn't create `B` itself; it receives any `IWorker` implementation from the outside.