SOLID Principles II

Table of contents

Outline . . . 1
Why SOLID (recap) . . . 1
Liskov Substitution Principle . . . 2
Interface Segregation Principle . . . 9
Dependency Inversion Principle . . . 10
Summary and key takeaways . . . 15
Further reading . . . 15
Applied class . . . 15

See csse2002[1] for course logistics. Continues 2026-04-02-refactoring[2] (Week 6), which has Single Responsibility and Open-Closed. The remaining three SOLID principles are below. The code-level treatment of all five is in java-solid-principles[3].

Outline

1. Liskov Substitution Principle (and substitution with contracts)
2. Interface Segregation Principle
3. Dependency Inversion Principle

Why SOLID (recap)

Large software tends towards becoming **rigid** and **fragile**.

- **Rigid software** — difficult to change. Changes, even small changes, require many modifications in many parts of the codebase.

[1] courses/csse2002/csse2002.pdf
[2] courses/csse2002/2026-04-02-refactoring.pdf
[3] courses/csse2002/java-solid-principles.pdf

- **Fragile software** — likely to break when a change is made. Sometimes the breakage occurs in parts of the codebase that *appear to have no relation to the change*.

SOLID principles are a way to help prevent rigidity and fragility. Many of them were floating around before becoming "SOLID"; Robert Martin combined them in *Design Principles and Design Patterns* (2000).

	Principle	Notes
S	Single Responsibility	Week 6 — 2026-04-02-refactoring[4]
O	Open-closed Principle	Week 6 — 2026-04-02-refactoring[5]
L	Liskov Substitution Principle	Week 7 — below
I	Interface Segregation Principle	Week 7 — below
D	Dependency Inversion	Week 7 — below

Liskov Substitution Principle

> Subclasses should be substitutable for their parent classes.

Defined by Barbara Liskov in a 1987 keynote. In full: an instance of a subclass type can be used wherever an instance of a superclass type is expected, **without breaking the correctness of the program**. That last clause is the whole content of the principle.

The motivating example is an `Animal` supertype with `Cat` and `Mouse` subclasses, each declaring `+ eat(f: Food) : void`, and a client that only ever talks to the supertype (see java-polymorphism[6], java-inheritance[7]):

```
List<Animal> animals = new ArrayList<>();
animals.add(new Cat());
animals.add(new Mouse());
for (Animal animal : animals) {
    animal.eat(new Food());
}
```

A violation: the postgraduate GPA

> Different types of students (undergraduates and postgraduates) should behave consistently when calculating GPA.

[4] courses/csse2002/2026-04-02-refactoring.pdf
[5] courses/csse2002/2026-04-02-refactoring.pdf
[6] courses/csse2002/java-polymorphism.pdf
[7] courses/csse2002/java-inheritance.pdf

```
class Student {
    protected List<Course> courses;

    public double calculateGPA() {
        double totalPoints = 0;
        for (Course course : courses) {
            totalPoints += course.getGradePoints();
        }
        return totalPoints / courses.size();
    }
}

class PostgraduateStudent extends Student {
    @Override
    public double calculateGPA() {
        throw new UnsupportedOperationException("Postgraduates have a different GPA
        ↪ system.");
    }
}
```

The failure is entirely in the client code:

```
Student undergraduate = new UndergraduateStudent();
double gpa = undergraduate.calculateGPA();

Student postgraduate = new PostgraduateStudent();
// This will throw an exception!
gpa = postgraduate.calculateGPA();
```

Both variables are declared `Student`. The second one is not usable where a `Student` is expected, so `PostgraduateStudent` is not a substitutable subclass. This compiles perfectly and only breaks at run time — see java-exceptions[8].

Substitution with contracts

Substitutability is made precise using the method's contract (see java-specification[9]):

```
class Parent {
    /**
     * @requires x > 0
     * @ensures \result > 'A' && \result < 'Z'
```

[8] courses/csse2002/java-exceptions.pdf
[9] courses/csse2002/java-specification.pdf

```
     */
    char f(int x);
}
```

A subclass of `Parent` can override `f`. Even when the implementation changes, **the override must still follow the contract commitments of the original version.**

> **Note**
>
> The course spells these tags two ways: `@requires` / `@ensures` as above, and `@require` / `@ensure` (no trailing `s`) as in the examples further down. The Week 4 tag table and the exam preparation both use `@requires` / `@ensures`. Neither spelling is "the" course form — see csse2002-diagram-notation[10].

Preconditions

Suppose `Child extends Parent`, but `Child.f()` can deal with negative numbers as well. What should the precondition of `Child.f()` be? There are two options:

1. Leave it as `x > 0`.
2. Change it to `x != 0`.

A possible problem with option 1 is confusion for readers about why the limitation is there when the code is more capable.

Option 2 does not violate the substitution principle. Compare the **sets of inputs each precondition allows**. Everything in the first set (`x > 0`) is also in the second set (`x != 0`), so the second precondition is **weaker**, that is, less strict. It does not reject any state the first one would allow. Stated as an implication:

`x > 0` $\Rightarrow$ `x != 0`

> Preconditions in a subclass must be **no stronger** than those in the superclass; they may be weaker or remain the same.

Postconditions

What if `Child.f()` only returned `'K'`, `'L'`, or `'M'` — would `\result > 'J' && \result < 'N'` be an acceptable postcondition?

---

10 courses/csse2002/csse2002-diagram-notation.pdf

Yes. Every result still fits within the original postcondition, so the new postcondition is **stronger** (more restrictive). Stated as an implication, the `Child` postcondition implies the `Parent` postcondition:

`\result > 'J' && \result < 'N'` ⇒ `\result > 'A' && \result < 'Z'`

> Postconditions in a subclass must be **no weaker** than those in the superclass; they may be stronger or remain the same.

The rule

> **A subclass must not strengthen preconditions or weaken postconditions.**

The implications run in opposite directions: for preconditions the *parent's* condition implies the *child's*; for postconditions the *child's* implies the *parent's*.

The funnel picture

The rule can be drawn as well as stated. A method is a box; the cone above it is the set of inputs it will accept, the cone below is the set of outcomes it may produce. Width means permissiveness, so a wide top is a weak precondition and a wide bottom is a weak postcondition.

Superimposing the two funnels turns the rule into a containment statement:

Worked example: `One` and `Two`

The question. Does `Two` satisfy the substitution principle with respect to `One`?

```
class One {
    /**
     * @require row != null && col != null && 0 < val && val < 100
     * @ensure \result is the list of all of the positive
     *     integers n < val that appear in either row or col
     */
    public List<Integer> calc(int[] row, int[] col, int val) {
    }
}

class Two extends One {
    /**
     * @require row != null && col != null && 0 < val &&
     *       val < 100 && row only contains positive integers
     * @ensure \result is the list of all of the positive
     *     integers n < val that appear in either row and col
```

Assume that B `extends` A

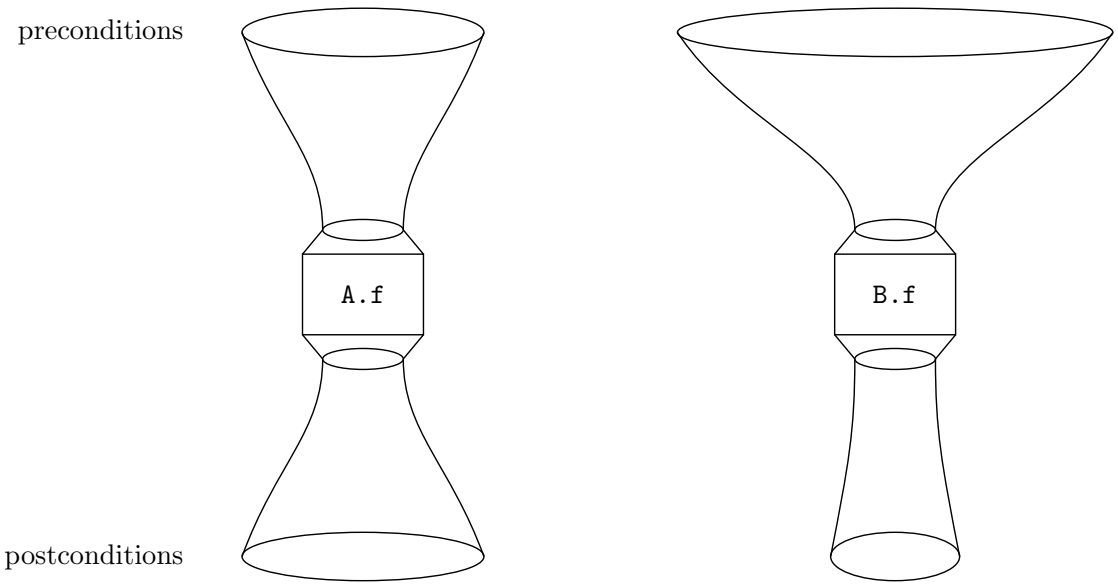


Figure 1: A method's contract drawn as a funnel. The cone above `A.f` is the set of inputs it accepts — its precondition; the cone below is the set of outcomes it may produce — its postcondition. A wider mouth is a weaker condition, so the legal override `B.f` has a wider top (weaker precondition) and a narrower bottom (stronger postcondition).

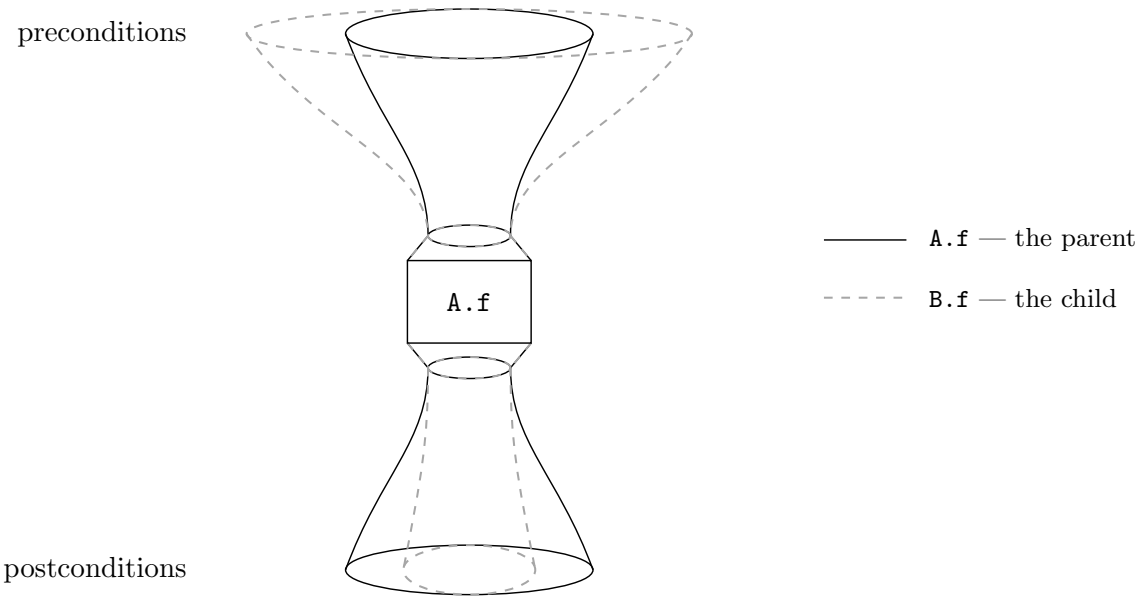


Figure 2: The same two funnels superimposed, the parent `A.f` solid and the child `B.f` dashed: the child's precondition mouth contains the parent's, and the parent's postcondition mouth contains the child's. The child should accept all inputs the parent accepts, and it should still guarantee outputs that satisfy everything the parent guarantees.

```
     */
    @Override
    public List<Integer> calc(int[] row, int[] col, int val) {
    }
}
```

The answer. Read `Two`'s `@ensure` as "either row **or** col" (identical to `One`'s), so that the only thing left differing is the precondition. The input sets each precondition admits are:

	Class One	Class Two
`row`	…, -1, 0, 1, 2, 3 …	1, 2, 3, …
`col`	…, -1, 0, 1, …	…, -1, 0, 1, …
`val`	1, 2, 3, 4, …, 99	1, 2, 3, 4, …, 99

`col` and `val` admit exactly the same values in both. `row` does not: `One` accepts any integers, `Two` accepts only positive ones. So there are inputs `One` would accept that `Two` rejects — `One`'s precondition does **not** imply `Two`'s. That is a **strengthened precondition**, and it breaks substitutability: client code written against `One` that happens to pass a `row` containing a negative number would break the moment a `Two` were substituted in.

Reading it off Figure 1: `Two.calc`'s top mouth is *narrower* than `One.calc`'s, which is exactly the shape the rule forbids.

A variant (homework). Remove `&& val < 100 && row only contains positive integers` from `Two`'s precondition and change its postcondition to the integers appearing in **BOTH** `row` and `col`, leaving:

```
class Two extends One {
    /**
     * @require row != null && col != null && 0 < val
     * @ensure \result is the list of all of the positive
     *     integers n < val that appear in BOTH row and col
     */
    @Override
    public List<Integer> calc(int[] row, int[] col, int val) {
    }
}
```

`val < 100` is gone as well, so `Two`'s precondition is now *weaker* than `One`'s, not equal to it. That is allowed, so the precondition side is settled. The question left is entirely about the postcondition: does *that* satisfy the substitution principle?

Working

`One` promises the *union* — the positive integers below `val` appearing in `row` **or** `col`. `Two` now promises the *intersection* — those appearing in **both**. Take `row = [1, 2]`, `col = [2, 3]`, `val = 10`: `One` must return `[1, 2, 3]`, `Two` returns `[2]`. Both postconditions pin the result down to a single list, and those lists differ, so `Two`'s postcondition does **not** imply `One`'s. Since the rule demands exactly that implication, it fails and LSP is violated. (The two are in fact incomparable — neither implies the other — which is still a failure, because "no weaker" means "implies".)

In funnel terms: `Two.calc`'s bottom mouth is not contained in `One.calc`'s; it juts outside it, and containment is what the rule demands. Client code that relied on `One`'s promise — say, looking up `3` in the returned list — silently gets a wrong answer rather than an exception, which is the fragility[11] SOLID exists to prevent.

This example is drilled again in week9-tutorial-substitution-principle-and-pre-post-conditions[12].

Interface Segregation Principle

> Many client specific interfaces are better than one general purpose interface.

ISP builds on **coupling** — the degree of interdependence between modules, classes or components — and its seven levels from `Content` (tightly coupled, "needy") down to `None` (loosely coupled, "independent"). The levels are from Week 6: see java-cohesion-and-coupling[13] for the levels and worked classifications, and week10-tutorial-coupling-and-cohesion[14] for the applied class.

The bridge from coupling to ISP is the `class R` example and its **risk of causing incidental dependencies**:

```
public class R {
    int x, y, z;
    char a, b, c;
}

public static void compute(R r) {
    return r.x * r.y;
}
```

[11] courses/csse2002/java-solid-principles.pdf
[12] courses/csse2002/week9-tutorial-substitution-principle-and-pre-post-conditions.pdf
[13] courses/csse2002/java-cohesion-and-coupling.pdf
[14] courses/csse2002/week10-tutorial-coupling-and-cohesion.pdf

```
public static void run() {
    R r = new R(1, 2, 3, 'a', 'b', 'c');
    compute(r);
}
```

`compute` is handed the whole of `R` but only reads two of its six fields (stamp coupling). ISP is that observation applied to a type's public surface:

- **Shrink interfaces to minimize incidental dependencies.**
- **Larger interfaces should be split into smaller ones**, to ensure that implementing classes only need to be concerned about the methods that are of interest to them.

The problem

The cost: when `Z` changes, clients `A`, `B`, and `C` may all be affected, even if each uses only a small part of `Z`, so they may need to be recompiled and redeployed.

The solution

`Z` is unchanged — it still has all five methods. What changed is that no client can *see* more of `Z` than it needs, so a change to `d()` can no longer force `A` or `B` to be recompiled.

Dependency Inversion Principle

> Depend upon Abstractions. Do not depend upon concretions.

Entities must depend on abstractions, not on concretions. The high-level module must not depend on the low-level module; both should depend on abstractions.

The cheapest version: program to the interface you already have

The first DIP example changes one type name, in three places. Before:

```
public static void main(String[] args) {
    ArrayList<Course> courses = new ArrayList<>();
    Student student = new Student(courses);
    double totalGradePoints = student.calculateTotalGradePoints();
}

class Student {
    private ArrayList<Course> courses;
```

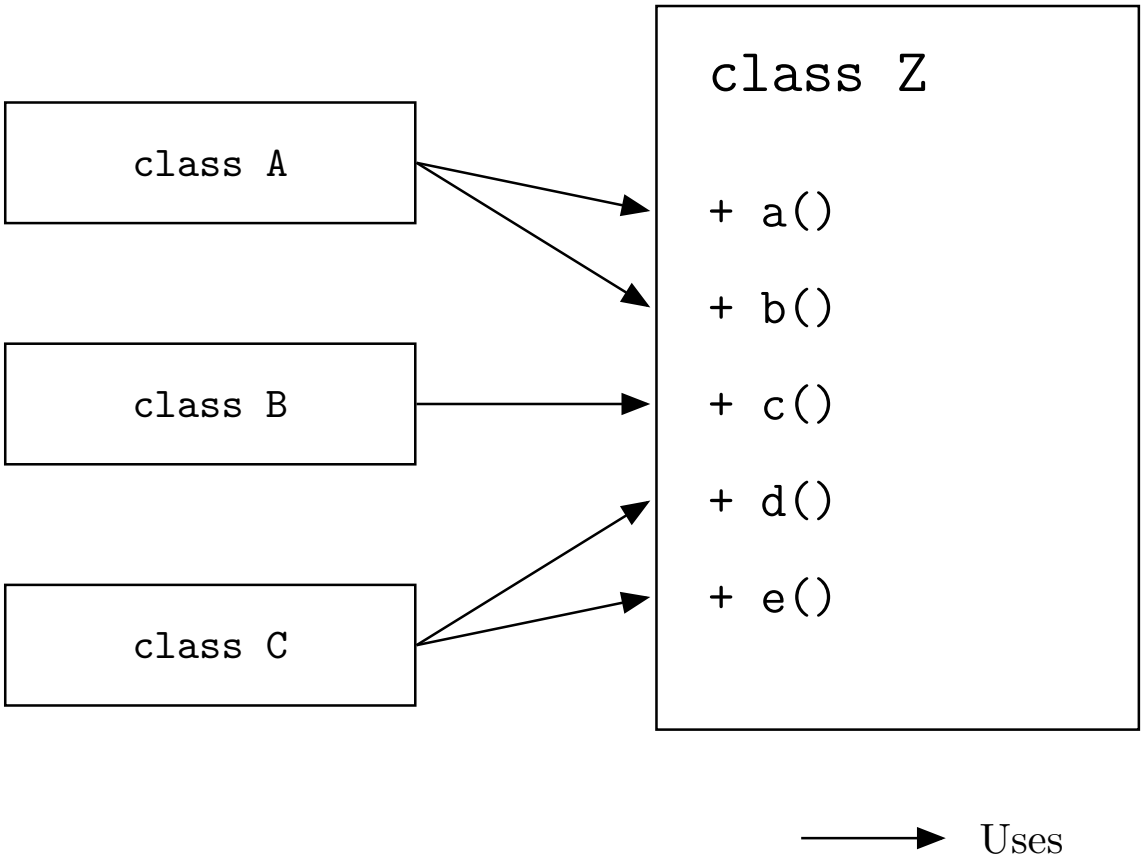


Figure 3: Despite only using a small subset of `Z`, each class inadvertently depends on all of `Z`. A node is a rectangle labelled with its Java keyword and name, with its members listed inside the same box as `+ a()` and no compartment rule. The solid `Uses` arrow points from the depender to the thing depended on, and ends on the member row it uses, not on the box edge.

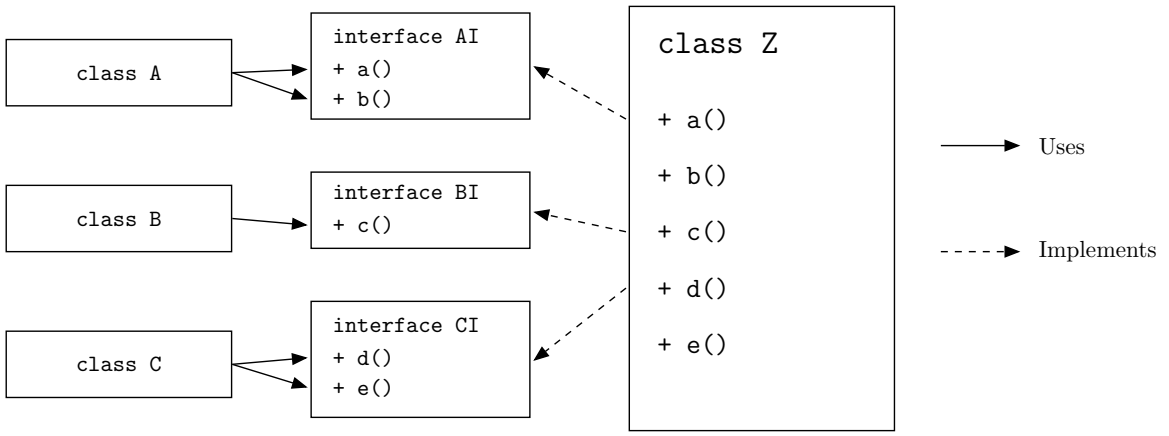
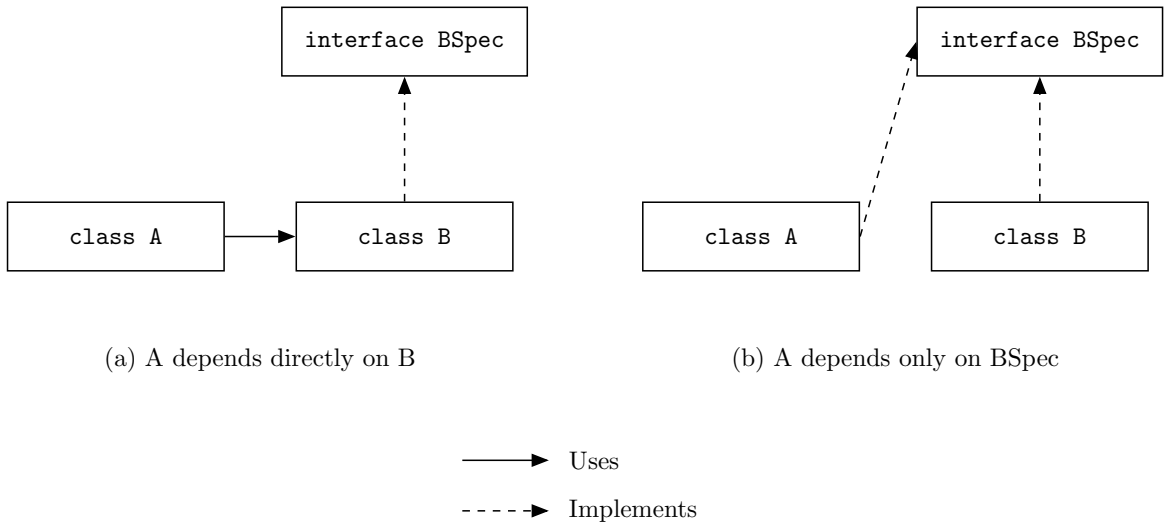


Figure 4: Create interfaces to minimize the incidental dependencies: `A` now depends only on `AI`, `B` only on `BI`, `C` only on `CI`, and `Z` implements all three. A solid line is Uses and a dashed line is Implements, both pointing from the depender to the thing depended on. Both have the same filled arrowhead, so only the line style tells them apart. The interfaces are named `AI`/`BI`/`CI` here and `BSpec` in the dependency inversion figure; the course does not settle on one naming scheme.



(a) A depends directly on B

(b) A depends only on BSpec

Figure 5: Inverting a dependency. In (a) `A` depends directly on the concrete `B`, so changes in the implementation can propagate to `A`. In (b) `A` depends only on `BSpec`, so changes inside `B` are less likely to affect `A` as long as `BSpec` stays the same. The edge from `A` to `BSpec` in (b) is drawn *dashed*, the `Implements` style, even though `A` merely uses `BSpec`.

```
    public Student(ArrayList<Course> courses) { this.courses = courses; }

    public double calculateTotalGradePoints() {
        return totalGradePoints;
    }
}
```

After: every `ArrayList` in a *declared type* position becomes `List`, while `new ArrayList<>()` is left alone. The constructor call is the one place a concrete class has to be named.

```
public static void main(String[] args) {
    List<Course> courses = new ArrayList<>();
    Student student = new Student(courses);
    double totalGradePoints = student.calculateTotalGradePoints();
}

class Student {
    private List<Course> courses;
    public Student(List<Course> courses) { this.courses = courses; }

    public double calculateTotalGradePoints() {
        return totalGradePoints;
    }
}
```

See java-collections-framework[15] for the `List`/`ArrayList` split this relies on, and java-abstraction[16].

Dependency Injection

> Dependency Injection (DI) is a design pattern that implements the Dependency Inversion Principle (DIP), by injecting dependencies into a class instead of creating them internally, promoting loose coupling.

In the `Student` code above, the `Student` object receives its dependency through the constructor. The dependency is created in `main` and handed in — `Student` never calls `new` on it.

The full example runs the same move on a service that creates its own collaborator:

[15] courses/csse2002/java-collections-framework.pdf
[16] courses/csse2002/java-abstraction.pdf

```
class AlertService {
    public void sendAlert(String msg) {
        EmailSender sender = new EmailSender();
        sender.send(msg);
    }
}
```

- `AlertService` is tightly coupled to `EmailSender`.
- `AlertService` can only send emails.
- `AlertService` creates its own dependency.

Introducing the abstraction and injecting it fixes all three:

```
interface MessageSender {
    void send(String msg);
}

class EmailSender implements MessageSender {
    @Override
    public void send(String msg) {
        System.out.println("Sending EMAIL: " + msg);
    }
}

class SmsSender implements MessageSender {
    @Override
    public void send(String msg) {
        System.out.println("Sending SMS: " + msg);
    }
}

class AlertService {
    private MessageSender sender;

    public AlertService(MessageSender sender) {
        this.sender = sender;
    }

    public void sendAlert(String msg) {
        sender.send(msg);
    }
}
```

- `AlertService` no longer depends directly on `EmailSender`.
- It depends on the interface `MessageSender`.
- The actual sender is injected from outside.

Stability, exposure, and the "proper" form of DIP (creating an abstraction where none exists yet) are worked through in java-solid-principles[17] and drilled in week11-tutorial-dependency-inversion[18].

Summary and key takeaways

The takeaways for the whole block, not only this lecture:

- Test Driven Development — java-testing[19]
- Refactoring — java-refactoring[20]
- Coupling and Cohesion — java-cohesion-and-coupling[21]
- SOLID Principles — java-solid-principles[22]
  - **S**ingle Responsibility
  - **O**pen-closed Principle
  - **L**iskov Substitution Principle
  - **I**nterface Segregation Principle
  - **D**ependency Inversion

Further reading

- *Design Principles and Design Patterns* by Robert C Martin
- https://www.freecodecamp.org/news/solid-principles-explained-in-plain-english/
- https://www.baeldung.com/solid-principles
- https://youtu.be/kF7rQmSRlq0

Applied class

No separate applied-class topic on SOLID this week — see week7-tutorial-black-box-and-glass-box-testing[23] for this week's applied class (black-box/glass-box testing, a continuation of java-testing[24] from Week 5).

[17] courses/csse2002/java-solid-principles.pdf
[18] courses/csse2002/week11-tutorial-dependency-inversion.pdf
[19] courses/csse2002/java-testing.pdf
[20] courses/csse2002/java-refactoring.pdf
[21] courses/csse2002/java-cohesion-and-coupling.pdf
[22] courses/csse2002/java-solid-principles.pdf
[23] courses/csse2002/week7-tutorial-black-box-and-glass-box-testing.pdf
[24] courses/csse2002/java-testing.pdf