Interface Segregation Principle (ISP)

Have you ever implemented an interface… only to realize you had to write empty methods just to make the compiler happy?

Or updated a shared interface… and suddenly, multiple unrelated classes started breaking?

If yes, you've probably encountered a violation of one of the most misunderstood design principles in software engineering: The Interface Segregation Principle (ISP).

Let's understand it with a real-world example—and see why this principle helps you build cleaner, more focused code.

Imagine you're building a media player app that supports different types of media:

Audio files (MP3, WAV)
Video files (MP4, AVI)
You might start with what feels like a convenient design: a single, unified interface that handles everything.

Python

```python
class MediaPlayer(ABC):
    @abstractmethod
    def play_audio(self, audio_file):
        pass

    @abstractmethod
    def stop_audio(self):
        pass

    @abstractmethod
    def adjust_audio_volume(self, volume):
        pass

    @abstractmethod
    def play_video(self, video_file):
        pass

    @abstractmethod
    def stop_video(self):
        pass

    @abstractmethod
    def adjust_video_brightness(self, brightness):
        pass

    @abstractmethod
    def display_subtitles(self, subtitle_file):
        pass
```

At first, it seems efficient. One interface, all capabilities. But as your app grows, problems start to show.

Let’s say you want to create a pure audio player—a class that should only handle sound:

Python

```
class AudioOnlyPlayer(MediaPlayer):
    def play_audio(self, audio_file):
        print(f"Playing audio file: {audio_file}")

    def stop_audio(self):
        print("Audio stopped.")

    def adjust_audio_volume(self, volume):
        print(f"Audio volume set to: {volume}")

    # □ Unwanted methods
    def play_video(self, video_file):
        raise NotImplementedError("Not supported.")

    def stop_video(self):
        raise NotImplementedError("Not supported.")

    def adjust_video_brightness(self, brightness):
        raise NotImplementedError("Not supported.")

    def display_subtitles(self, subtitle_file):
        raise NotImplementedError("Not supported.")
```

Yikes.

Even though AudioOnlyPlayer only needs audio methods, it’s forced to implement unrelated video functionality. You either throw exceptions or write empty methods. Neither is ideal.

What’s Wrong With This?
Interface Pollution
The MediaPlayer interface is doing too much. It combines multiple unrelated responsibilities:

Audio playback
Video playback
Subtitle handling
Brightness control
This violates the Interface Segregation Principle (ISP).

Fragile Code
Now, imagine you add a new method to the interface, like enablePictureInPicture(). Suddenly, all existing implementations—audio-only, video-only, or otherwise—must update.

This tight coupling slows you down and increases the risk of bugs.

Violates Liskov Substitution
A client may expect any MediaPlayer to support video, but passing in an AudioOnlyPlayer will crash the program with an UnsupportedOperationException.

That's a clear Liskov Substitution Principle (LSP) violation.

Enter: The Interface Segregation Principle (ISP)
Clients should not be forced to depend on methods they do not use.

In simpler terms: Keep your interfaces focused. Each interface should represent a specific capability or behavior. If a class doesn’t need a method, it shouldn’t be forced to implement it.

This is especially important in larger codebases with evolving requirements.

Why Does ISP Matter?
Increased Cohesion, Reduced Coupling: Interfaces become highly focused. AudioOnlyPlayer only knows about audio methods. VideoPlayer (if it only played video without sound) would only know about video methods. This minimizes unnecessary dependencies.
Improved Flexibility & Reusability: Smaller, role-specific interfaces are easier for classes to implement correctly. You can combine capabilities as needed (like a full video player implementing both audio and video interfaces).
Better Code Readability & Maintainability: It's much clearer what a class can and cannot do. When the MediaPlayer interface was fat, a developer looking at an AudioOnlyPlayer might be misled. With ISP, the implemented interfaces clearly state its capabilities.

Enhanced Testability: When testing a client that uses, say, an IAudioPlayer interface, you only need to mock the audio-specific methods, not a whole slew of unrelated video methods.
Avoids "Interface Pollution" and LSP Violations: Classes aren't forced to implement methods they don't need, drastically reducing the likelihood of UnsupportedOperationExceptions and making subtypes more reliably substitutable for the interfaces they claim to implement.

Applying ISP
Time to apply ISP and break down our MediaPlayer interface into more logical, focused pieces.

Step 1: Define Smaller, Cohesive Interfaces
Instead of one bloated MediaPlayer interface, we'll create multiple focused ones:

Python

```python
class AudioPlayerControls(ABC):
    @abstractmethod
    def play_audio(self, audio_file):
        pass

    @abstractmethod
    def stop_audio(self):
        pass

    @abstractmethod
    def adjust_audio_volume(self, volume):
        pass


# Video-only capabilities
class VideoPlayerControls(ABC):
    @abstractmethod
    def play_video(self, video_file):
        pass

    @abstractmethod
    def stop_video(self):
        pass

    @abstractmethod
    def adjust_video_brightness(self, brightness):
        pass

    @abstractmethod
    def display_subtitles(self, subtitle_file):
        pass
```
Step 2: Classes Implement Only the Interfaces They Need
Now our specific player classes can implement only the relevant interfaces.

ModernAudioPlayer (Audio-only)

Python

```
class ModernAudioPlayer(AudioPlayerControls):
    def play_audio(self, audio_file):
        print(f"ModernAudioPlayer: Playing audio - {audio_file}")

    def stop_audio(self):
        print("ModernAudioPlayer: Audio stopped.")

    def adjust_audio_volume(self, volume):
        print(f"ModernAudioPlayer: Volume set to {volume}")
```
SilentVideoPlayer (Video-only)

Python

```
class SilentVideoPlayer(VideoPlayerControls):
    def play_video(self, video_file):
        print(f"SilentVideoPlayer: Playing video - {video_file}")

    def stop_video(self):
        print("SilentVideoPlayer: Video stopped.")

    def adjust_video_brightness(self, brightness):
        print(f"SilentVideoPlayer: Brightness set to {brightness}")

    def display_subtitles(self, subtitle_file):
        print(f"SilentVideoPlayer: Subtitles from {subtitle_file}")
```
What if we need a player that handles both? It implements both interfaces!

ComprehensiveMediaPlayer (Both audio + video)

Python

```python
class ComprehensiveMediaPlayer(AudioPlayerControls, VideoPlayerControls):
    def play_audio(self, audio_file):
        print(f"ComprehensiveMediaPlayer: Playing audio - {audio_file}")

    def stop_audio(self):
        print("ComprehensiveMediaPlayer: Audio stopped.")

    def adjust_audio_volume(self, volume):
        print(f"ComprehensiveMediaPlayer: Audio volume set to {volume}")

    def play_video(self, video_file):
        print(f"ComprehensiveMediaPlayer: Playing video - {video_file}")

    def stop_video(self):
        print("ComprehensiveMediaPlayer: Video stopped.")

    def adjust_video_brightness(self, brightness):
        print(f"ComprehensiveMediaPlayer: Brightness set to {brightness}")

    def display_subtitles(self, subtitle_file):
        print(f"ComprehensiveMediaPlayer: Subtitles from {subtitle_file}")
```

Common Pitfalls While Applying ISP

Even with the right intentions, it's easy to misuse ISP if you're not careful. Here are some common traps to avoid:

1. Over-Segregation (a.k.a. "Interface-itis")

The mistake: Creating a separate interface for every single method — like `Playable`, `Stoppable`, `AdjustableVolume`, etc.

Why it's a problem: You end up with **too many tiny interfaces** that are hard to manage and understand. It's just as bad as having one big, bloated interface.

What to do instead: Group related methods by **logical roles or capabilities**.For example:

- `playAudio()`, `stopAudio()`, and `adjustAudioVolume()` naturally belong together in an `AudioPlayer` interface.

2. Not Thinking from the Client's Perspective

The mistake: Designing interfaces based only on how implementers work — not how clients use them.

Why it's a problem:ISP is really about **making life easier for the client** — not the implementer.

Fix: Design your interfaces by looking at **what the client actually needs** to do — and nothing more.

3. Lack of Cohesion

The mistake: Creating interfaces that aren't tightly related — mixing unrelated methods together.

Why it's a problem: Low cohesion makes interfaces confusing and hard to reason about.

Fix: Make sure every method in an interface relates to **a single, well-defined responsibility**.

Think of your interface as a **role —** would it make sense for all these actions to be part of that role?

Common Questions About ISP

How do I know how small my interfaces should be?

There's no strict number of methods or "one-size-fits-all" guideline. The best rule of thumb is: **design interfaces based on client needs**.

Ask yourself:

- Are all methods in the interface used by every implementing class?
- Are different clients interested in different capabilities?

If yes, it's a strong signal that the interface should be split.

Think in terms of **roles** or **capabilities**—interfaces should represent a cohesive set of behaviors that make sense together from the client's perspective.

Won't creating lots of small interfaces just add more files and complexity?

At first glance, yes—it might feel like you're adding more moving parts.

But this is **intentional structure**, not clutter. Over time, it pays off by:

- Making your code easier to understand
- Reducing coupling between unrelated components
- Preventing unnecessary dependencies

Instead of trying to comprehend one giant interface with 15 methods, you now deal with **clear, focused contracts**. It's a shift from **accidental complexity** to **intentional design**.

Should I apply ISP only to new code, or is it worth refactoring old code too?

You should definitely apply ISP when writing new code.

For existing code, refactoring is worth it when you notice any of the following:

- Frequent use of `UnsupportedOperationException`
- Classes implementing methods they don't use

- Interface changes breaking many unrelated classes
- Confusion about which methods clients can safely call

Start with the interfaces causing the most pain. Focus on the ones that are bloated, unstable, or widely misused.

Can a class implement multiple small interfaces?

Absolutely—and that's one of the key benefits of ISP.

A class can fulfill **multiple roles** by implementing several small, targeted interfaces. This gives you incredible flexibility and composability.

For example, an `AudioPlayer` might implement:

- `LoadableMedia`
- `PlaybackControls`
- `VolumeControl`
- `AudioFeatures`

Each interface is simple and focused, and the class only opts into the behaviors it supports.

How does ISP relate to the Liskov Substitution Principle (LSP)?

ISP and LSP are closely aligned.

- **ISP ensures** that interfaces are minimal and relevant.
- **LSP ensures** that implementations of those interfaces behave correctly and predictably.

When interfaces are too broad (violating ISP), classes are often forced to implement methods they don't support. This commonly leads to LSP violations like throwing `UnsupportedOperationException` where the client expects normal behavior.

By applying ISP, you make LSP easier to follow because each interface becomes a clean, reliable contract that implementers can fulfill completely and correctly