Single Responsibility Principle (SRP)

The **Single Responsibility Principle** states that a class should have **only one reason to change**. This means a class should have a **single, well-defined responsibility** and should not take on multiple concerns.

Real-World Analogy

- A **programmer** is responsible for writing code.
- If someone asks them to make coffee, it's outside their responsibility.
- Similarly, in software engineering, **each class should focus on its primary responsibility**.

Example: Journal Class

We create a `Journal` class to store diary entries.

Initial Implementation (Violation of SRP)

```
struct Journal
{
  string title;
  vector<string> entries;

  explicit Journal(const string& title) : title{title} {}

  void add(const string& entry)
  {
    static int count = 1;
    entries.push_back(boost::lexical_cast<string>(count++) + ": " + entry);
  }

  void save(const string& filename)  // ❌ Violates SRP
  {
    ofstream ofs(filename);
    for (auto& s : entries)
      ofs << s << endl;
  }
};
```

Problem:
The `Journal` class is responsible for both:

1. **Managing journal entries** (Adding, modifying, removing entries).
2. **Persisting data** (Saving to a file).

Persistence (saving to a file) is **a separate concern** and should be handled by another class.

Refactoring: Separation of Concerns

We move the **persistence logic** into a separate class `PersistenceManager`.

```
struct PersistenceManager
{
  static void save(const Journal& j, const string& filename)
  {
    ofstream ofs(filename);
    for (auto& s : j.entries)
      ofs << s << endl;
  }
};
```

Now, the `Journal` class **only** handles journal entries, and the `PersistenceManager` handles saving.

Usage

```
void main()
{
  Journal journal{"Dear Diary"};
  journal.add("I ate a bug");
  journal.add("I cried today");

  //journal.save("diary.txt");  // ❌ Not needed anymore

  PersistenceManager pm;
  pm.save(journal, "diary.txt");  // ✅ SRP-compliant
}
```

Why Follow SRP?

1. **Improved Maintainability** – If persistence logic changes (e.g., switching to a database), we only modify `PersistenceManager`.
2. **Better Code Reusability** – `PersistenceManager` can be used by other classes.
3. **Cleaner, Modular Design** – Each class does **one thing well**, making debugging and understanding easier.

Key Takeaways

- **A class should have a single responsibility.**
- **Separate concerns** (like persistence) into their own classes.
- **Easier to maintain and extend** when responsibilities are well-defined.