The Principles You Need to Know

(and those you should forget)

David Sackstein

All rights reserved

Agenda

- Causes of complexity
- The principles you need to know
  - The Single Responsibility Principle
  - Encapsulation
  - Prefer Composition over Inheritance
  - The Open Closed Principle
  - System Organization and Refactoring
- What you should forget
  - Inheritance

Causes of Complexity

- Too many dependencies between components (coupling)
- Components that have too many responsibilities
- Components that are large
- Components with unclear usage contracts
- Components that expose their implementations
- Quick and dirty changes that introduce these problems over time

Addressing complexity

- Break large solutions into smaller components
- Each component should have:
  - one responsibility
  - a clear and simple interface
  - tests that use the interface
- Each component should use
  - only a small number of other components
  - the public interface of other components

Benefits of reduced complexity

- Testability
  - Components with a single responsibility and a clear contract can easily be held responsible.
- Extensibility
  - Abstractions are extension points. New implementations do not affect other components.
- Readability
  - Smaller, isolated functionality is easier to reason about

Benefits of reduced complexity

- Maintainability
  - Easier to read is easier to understand.
  - Easier to test is safer to change without regression.
- Reusability
  - Components with fewer dependencies carry less “baggage”.
  - Components with a clear contract are easier to “sell”.

The Principles You Need to Know

- The Single Responsibility Principle
- Encapsulation
- Prefer Composition over Inheritance
- The Open Closed Principle
- Immutability is your friend
- System Organization and Maintenance

The Single Responsibility Principle

- Classes and methods should have only one responsibility.
- Horizontally:
  - Implement one feature
- Vertically:
  - One level of abstraction
- Try to state the responsibility without using the word “and”.

All rights reserved

- Too long
- Too much detail

```cpp
void runProcess() {
    // Step 1: Connect to DB
    std::vector<RawDataRow> rawData;
    rawData.push_back({1, "Alice", 10.0});
    rawData.push_back({2, "Bob", 20.0});
    rawData.push_back({3, "Alice", 30.0});

    // Step 2: Join tables (simulate join by map)
    std::map<int, std::string> idToCategory;
    idToCategory[1] = "A";
    idToCategory[2] = "B";
    idToCategory[3] = "A";

    for (auto& row : rawData) {
        if (idToCategory.find(row.id) != idToCategory.end()) {
            row.name += "-" + idToCategory[row.id];
        }
    }

    // Step 3: Create DTOs
    std::vector<DTO> dtos;
    for (const auto& row : rawData) {
        DTO dto;
        dto.id = row.id;
        dto.name = row.name;
        dto.value = row.value;
        dtos.push_back(dto);
    }

    // Step 4: Analyze data - compute stats
```

- After extraction of methods, the method is clearer
- But there are mixed levels of abstraction

```cpp
void runProcess() {
    auto rawData = loadRawData();
    joinData(rawData);

    auto dtos = createDTOs(rawData);

    auto stats = analyzeData(dtos);

    bool errorOccurred = !readConfig("config.ini");

    if (errorOccurred) {
        sendNotification("Error reading config");
    } else {
        sendNotification("Process completed. Average value: " +
            std::to_string(stats.averageValue));
    }
}
```

- After further extraction runProcess does only one thing.
- It breaks down the process into 3 parts at the same level of abstraction

```
void runProcess() {
    auto dtos = loadData();
    auto stats = analyze(dtos);
    notify(stats);
}
```

The Single Responsibility Principle

- Each file, class, function, library should have one responsibility.
- One file should contain only one class
- Responsibilities can often be measured in size:
  - File < 200 lines
  - Function < 15 lines
  - Class header < 50 lines
  - Class source < 200 lines

The Single Responsibility Principle

- A good indication of SRP violations is a lack of cohesiveness.
- A cohesive class is one in which most fields are referenced by most methods.
- An island of associated fields and methods should really be a class by itself.
- Extract islands as classes.

All rights reserved

```cpp
class IncohesiveDevice {
private:
    std::string ipAddress;
    int port;

    float batteryLevel;
    bool isCharging;

    int brightness;
    int contrast;

public:
    IncohesiveDevice();

    void connect() const;
    void disconnect() const;

    void showBatteryStatus() const;
    void toggleCharging();

    void adjustBrightness(int delta);
    void adjustContrast(int delta);
};
```

Connectivity

Battery

Display

IncohesiveDevice has three responsibilities and three islands of fields and methods

```cpp
class NetworkManager {
    std::string ipAddress;
    int port;

public:
    NetworkManager();
    void connect() const;
    void disconnect() const;
};

class BatteryMonitor {
    float batteryLevel;
    bool isCharging;

public:
    BatteryMonitor();
    void showBatteryStatus() const;
    void toggleCharging();
};

class DisplayConfig {
    int brightness;
    int contrast;

public:
    DisplayConfig();
    void adjustBrightness(int delta);
    void adjustContrast(int delta);
};
```

- Extract classes
- Device is now a manager of objects, each with a single responsibility

```cpp
class Device {
    NetworkManager network;
    BatteryMonitor battery;
    DisplayConfig display;

public:
    NetworkManager& getNetworkManager();
    BatteryMonitor& getBatteryMonitor();
    DisplayConfig& getDisplayConfig();
};
```

Encapsulation

- The Three Pillars of Object-Oriented Design
  - Encapsulation
  - Polymorphism
  - Inheritance
- Which do we need and which should we avoid?
  - Encapsulation is the most important
  - Polymorphism helps achieve encapsulation
  - Inheritance (code reuse) undermines encapsulation

Two types of inheritance

- Polymorphism: Inheritance of contract
  - A commitment by the implementor and
  - A capability a consumer can depend on
- Code Reuse: Inheritance of implementation
  - Derived classes call methods of their base class
  - Template method: The base class also calls virtual methods of the derived class.

Polymorphism: Benefits

- Encapsulation: Users of the interface cannot see the implementation
- Programming by contract: Users of the interface can clearly see the contract
- Benefits of encapsulation:
  - Testability: Replacement of implementations using mocks.
  - Extensibility: Upgrading implementations without rebuilding the caller.

Polymorphism: Pitfalls?

- Encapsulation (!):
  - Users of the interface cannot see the implementation
  - This may make it difficult to fully understand the code, but not really
- It's a trade-off because:
  - Caller code should actually be easier to understand if the implementations are hidden
  - But there is such a thing as over abstraction

Inheritance for Code Reuse: Benefits

- Avoids code duplication by
  - Moving common code into the base class
  - Moves varying code into derived classes
  - It’s so easy! You can call the implementation without specifying the name of an object

Inheritance for Code Reuse: Pitfalls!

- It introduces coupling between the derived class and the base class:
  - Includes of the derived's header must also include the header of the base.
  - The size of the derived class depends on the size of the base. So, if the base changes you need to recompile. (No runtime compatibility)
  - May introduce cyclic dependencies, because the base class can call the derived class and vice versa
- Effectively the base and the derived classes are one big class split across multiple files
- This is a violation of the SRP

Inheritance for Code Reuse: Pitfalls!

- It breaks encapsulation
  - The derived class has access to all public members of the implementation class.
  - It’s like marking your implementation fields and methods public.
  - All members of all base classes are in the same namespace and can be confused without warning (the diamond problem)
- It limits testability of the derived class
  - The implementation in the base class cannot be replaced by a mock in a test
  - Coupled classes can only be tested together
- It limits reusability of the derived class
  - The derived class can only be reused if the implementation provided in the base class is desired

And it gets worse

- Often, inheritance is used to achieve both goals (polymorphism and code reuse)
  - A glorious violation of the Single Responsibility Principle
  - This complicates the class giving it multiple responsibilities.
  - The relationship between base and derived are not clear making it difficult to maintain
- Large inheritance hierarchies are difficult to maintain
  - When implementations change, inheritance relationships are required to change.

Summary so far

- Inheritance of Contract: Good
- Inheritance of Implementation: Bad
- Exception: Data classes can use inheritance because:
  - Data class expose all their data as public. No need for encapsulation
  - There are no methods in data classes, so no cycles
  - We don’t test data classes

Prefer Composition over Inheritance

- For code reuse – use composition
- Everywhere you use inheritance today you can use composition:
  - When derived class uses the base class
    - The base class becomes a member
  - When base class uses derived class
    - The derived class becomes a member

Base class becomes a member

```cpp
class FileProcessor : public Logger {
public:
    explicit FileProcessor(LogLevel level);

    void processFile(const std::string& filename);
};
```

```cpp
class FileProcessor {
public:
    explicit FileProcessor(ILogger&);

    void processFile(const std::string& filename) const;

private:
    ILogger& logger_;
};
```

Derived class becomes a member

```
class BaseLogger {
    LogLevel logLevel_;

    std::string to_string(LogLevel);

    virtual void writeMessage(const std::string& message) = 0;

public:
    explicit BaseLogger(LogLevel);
    virtual ~BaseLogger() = default;

    void log(LogLevel logLevel, const std::string& msg);
};
```

```
class FileLogger final : public BaseLogger {
    std::string fileName_;

    void writeMessage(const std::string& message) override;

public:
    FileLogger(LogLevel logLevel, const std::string& fileName);
};
```

Derived class becomes a member

```cpp
class Logger final {
    LogLevel logLevel_;
    IMessageWriter& writer_;

    std::string to_string(LogLevel);

public:
    Logger(LogLevel, IMessageWriter&);

    virtual ~Logger() = default;

    void log(LogLevel logLevel, const std::string& msg);
};
```

```cpp
class FileWriter final : public IMessageWriter {
    std::string fileName_;

public:
    explicit FileWriter(const std::string& fileName);

    void writeMessage(const std::string& message) override;
};
```

The Open Closed Principle

- A design should be Open for extension and Closed for modification.
- That is, adding new functionality should not require changing existing functionality.
- Avoid premature generalization:
  - Do <u>not</u> add features you don't need
  - Do <u>not</u> add empty interface methods to generalize
- But do use abstractions to simplify extending later.

All rights reserved

Immutability is your friend

- An immutable class cannot be modified
- This means that you only need to see its constructor to know its state.
- This reduces complexity and simplifies debugging
- Functional programming languages force immutability (the cost is copying).

Immutability in C++

- C++ is not strictly a functional programming language.
- It is an object-oriented language which typically uses mutation of state.
- Use const wherever possible.
- Dependencies, for instance should not be modified during the lifetime of an object.

System Organization

- Divide large projects into small components.
- Each component should have only one responsibility.
- Express the responsibility by a simple contract.
- Encapsulation: hide implementation details.
- Abstraction: allows interchangeability of components

Refactoring

- When adding new functionality, favor creating a new class (and file)
- When a function or class grows to big, extract methods and classes.
- Use (and add) tests to ensure there are no regressions.

Key Takeaways

The SRP	One class per file, should do one thing
The OCP	Avoid premature generalization
Encapsulation	Use visibility and interfaces
System Organization	Classes galore, divide files into folders
Inheritance	Don’t use it for code reuse. Composition!
Refactor, refactor	Get addicted. (You will need tests)
Decoupling	Coming up, with dependency injection!