SOLID Design Principles

Instructor: Mehroze Khan

SOLID Principles

- Good software systems begin with clean code
- The SOLID principles tell us how to **arrange our functions and data structures** into classes, and how those classes should be **interconnected**
- The goal of the principles is the creation of mid-level software structures that:
  - Tolerate change
  - Are easy to understand
  - Are the basis of components that can be used in many software systems
- The term “mid-level” refers to the fact that these principles are applied by programmers working at the module level.

SOLID Principles

- **SRP:** Single Responsibility Principle
- **OCP:** Open-Closed Principle
- **LSP:** Liskov Substitution Principle
- **ISP:** Interface Segregation Principle
- **DIP:** Dependency Inversion Principle

SRP-Single Responsibility Principle

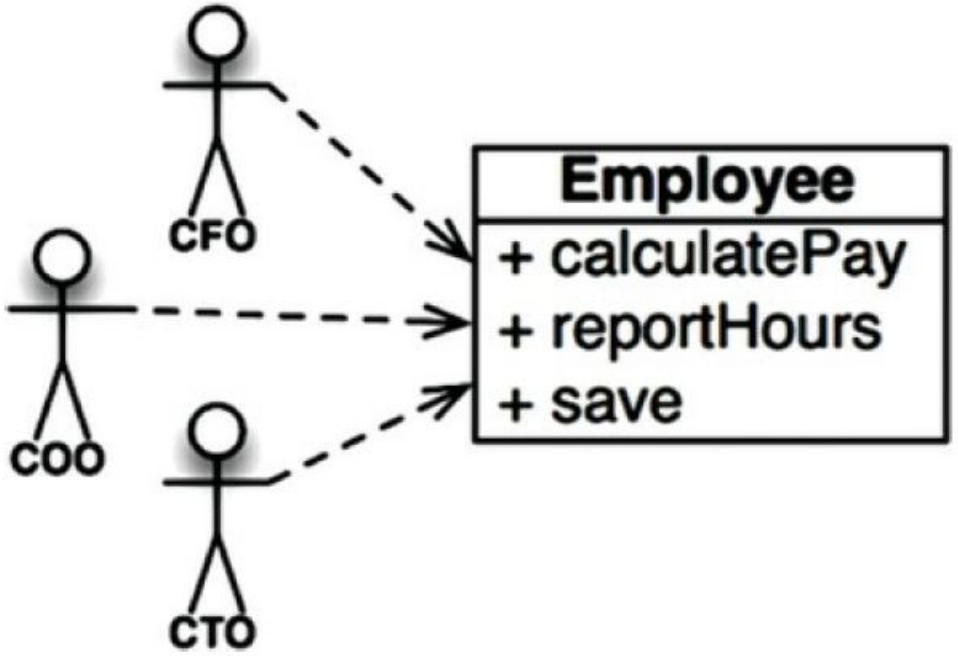
- The Single Responsibility Principle states that **a class should do one thing and therefore it should have only a single reason to change**.
- A module should have a reason to change by only **one actor.**
- A module should be responsible to **one and only one actor.**
- Only one potential change (database logic, logging logic, and so on.) in the software's specification should be able to affect the specification of the class.

Why use SRP?

- Many different teams can work on the same project and edit the same class for different reasons, this could lead to **incompatible modules**.
- It makes **version control easier**. For example, say we have a persistence class that handles database operations, and we see a change in that file in the GitHub commits. By following the SRP, we will know that it is related to storage or database-related stuff.
- **Merge conflicts** are another example. They appear when different teams change the same file. But if the SRP is followed, fewer conflicts will appear – files will have a single reason to change, and conflicts that do exist will be easier to resolve.

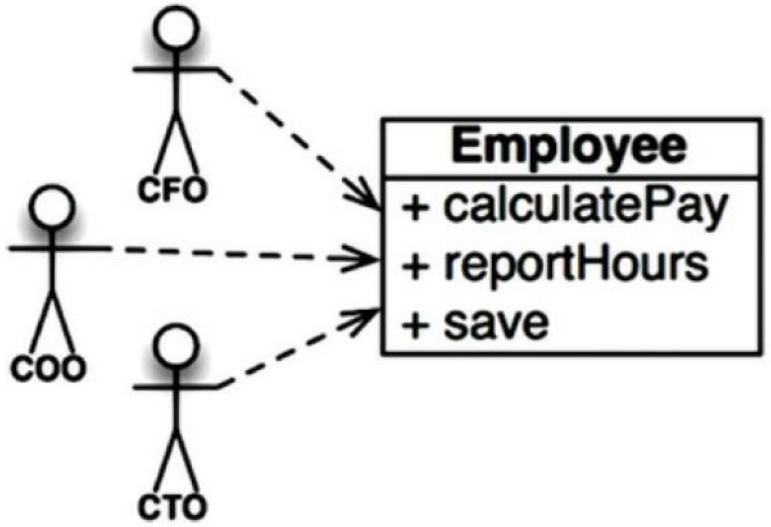
Employee Class from Payroll Application

- Employee class from a payroll application has three methods: calculatePay(), reportHours(), and save()



Employee Class from Payroll Application

- This class violates the SRP because those three methods are responsible to three very different actors.
  - The **calculatePay()** method is specified by the **accounting department**, which reports to the **CFO**.
  - The **reportHours()** method is specified and used by the **human resources department**, which reports to the **COO**.
  - The **save()** method is specified by the **database administrators** (DBAs), who report to the **CTO**.



```cpp
class Employee {
public:
    string name;
    int hoursWorked;
    double hourlyRate;

    // Constructor
    Employee(string n, int h, double r) {
        name = n;
        hoursWorked = h;
        hourlyRate = r;
    }

    // Function to calculate pay
    double calculatePay() {
        return hoursWorked * hourlyRate;
    }

    // Function to report hours worked
    void reportHours() {
        cout << name << " worked " << hoursWorked << " hours." << endl;
    }
```

```
    // Function to save employee data (to a file, database, etc.)
    void save() {
        cout << "Saving employee data..." << endl;
        // Simulate saving process
    }
};

int main() {
    // Example usage
    Employee emp("John Doe", 40, 25.0);
    cout << "Pay: $" << emp.calculatePay() << endl;
    emp.reportHours();
    emp.save();

    return 0;
}
```

Employee Class from Payroll Application

- By putting the source code for these three methods into a single Employee class, the **developers have coupled each of these actors to the others**.
- This coupling can cause the actions of the CFO's team to affect something that the COO's team depends on.

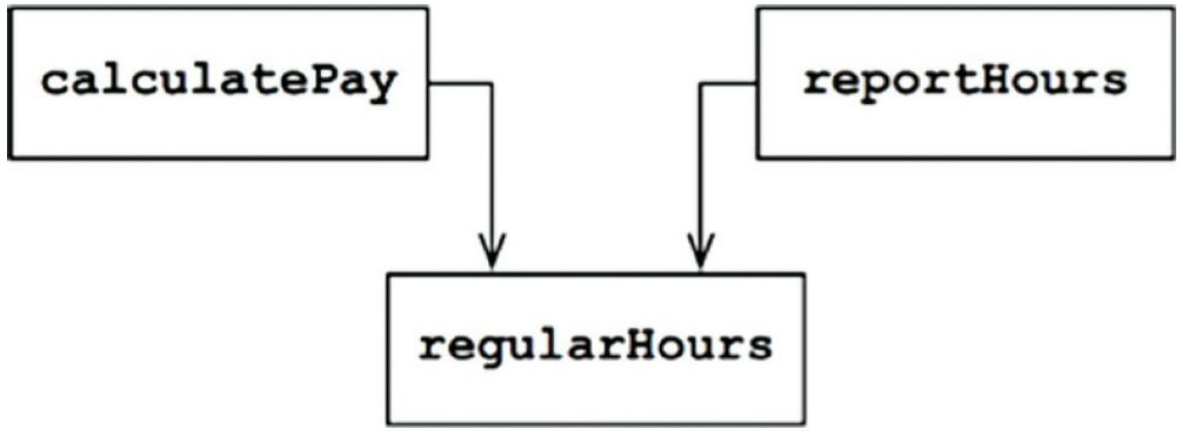
```
// Violating SRP
class User {
public:
    void saveToDatabase() {
        // Code to save user to database
    }
    void sendEmail() {
        // Code to send email
    }
};

// Following SRP
class User {
public:
    void save() {
        // Code to save user to database
    }
};

class EmailService {
public:
    void sendEmail(User user) {
        // Code to send email
    }
};
```

Symptom 1: Accidental Duplication

- Suppose that the CFO's team decides that the way non-overtime hours are calculated needs to be tweaked. In contrast, the COO's team in HR does not want that tweak because they use non-overtime hours for a different purpose.
- A developer is tasked to make the change and sees the convenient regularHours() function called by the calculatePay() method. Unfortunately, that developer does not notice that the function is also called by the reportHours() function.
- The developer makes the required change and carefully tests it. The CFO's team validates that the new function works as desired, and the system is deployed.
- Of course, the COO's team doesn't know that this is happening. The HR personnel continue to use the reports generated by the reportHours() function—but now they contain incorrect numbers.

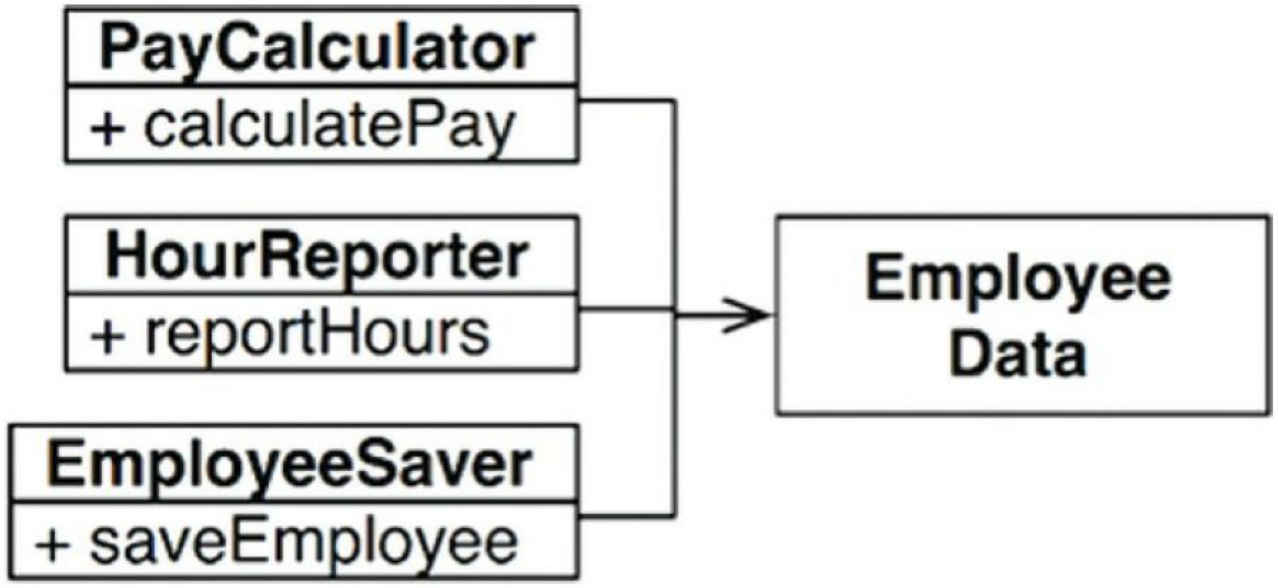


Symptom 2: Merges

- Suppose that the CTO's team of DBAs decides that there should be a simple schema change to the Employee table of the database.
- Suppose also that the COO's team of HR clerks decides that they need a change in the format of the hours report.
- Two different developers, possibly from two different teams, check out the Employee class and begin to make changes. Unfortunately, their changes collide. The result is a merge.
- In our example, the merge puts both the CTO and the COO at risk. It's not inconceivable that the CFO could be affected as well.
- Once again, the way to avoid this problem is to separate code that supports different actors.

Solution

- Most obvious way to solve the problem is to separate the data from the functions.
- The three classes share access to EmployeeData, which is a simple data structure with no methods.
- Each class holds only the source code necessary for its particular function.
- The three classes are not allowed to know about each other. Thus, any accidental duplication is avoided.



```cpp
// Class holding employee data
class EmployeeData {
public:
    string name;
    int hoursWorked;
    double hourlyRate;

    // Constructor
    EmployeeData(string n, int h, double r) {
        name = n;
        hoursWorked = h;
        hourlyRate = r;
    }
};

// Class responsible for calculating pay
class PayCalculator {
public:
    double calculatePay(const EmployeeData& employee) {
        return employee.hoursWorked * employee.hourlyRate;
    }
};
```

```cpp
// Class responsible for reporting hours worked
class HourReporter {
public:
    void reportHours(const EmployeeData& employee) {
        cout << employee.name << " worked " << employee.hoursWorked << " hours." << endl;
    }
};

// Class responsible for saving employee data
class EmployeeSaver {
public:
    void save(const EmployeeData& employee) {
        cout << "Saving employee data for " << employee.name << "..." << endl;
        // Simulate saving process
    }
};
```

Bookstore Invoice Program

```
// Book class definition
class Book {
public:
    string name;
    int price;

    // Constructor
    Book(string n, int p) {
        name = n;
        price = p;
    }
};
```

```cpp
// Invoice class definition
class Invoice {
private:
    Book book;
    int quantity;
    double discountRate;
    double taxRate;

public:
    // Constructor
    Invoice(const Book& b, int q, double d, double t) {
        book = b;
        quantity = q;
        discountRate = d;
        taxRate = t;
    }

    // Calculate total amount
    double calculateTotal() const {
        double discountedPrice = book.price * (1 - discountRate);
        return discountedPrice * quantity * (1 + taxRate);
    }
```

```
    // Print invoice details
    void printInvoice() const {
        cout << quantity << " x " << book.name << " " << book.price << "$" << endl;
        cout << "Total: " << calculateTotal() << "$" << endl;
    }

    // Save invoice to file
    void saveToFile(const string& filename) const {
        ofstream outFile(filename);
        if (outFile) {
            outFile << quantity << " x " << book.name << " " << book.price << "$" << endl;
            outFile << "Total: " << calculateTotal() << "$" << endl;
        } else {
            cout << "Unable to open file." << endl;
        }
    }
};
```

- Invoice Class contains some fields about invoicing and 3 methods:

 calculateTotal method, which calculates the total price

 printInvoice method, that should print the invoice to console

 saveToFile method, responsible for writing the invoice to a file

- What is wrong with this class design?

Violations of SRP

- The first violation is the **printInvoice** method, which contains our printing logic. The SRP states that our class should only have a single reason to change, and that reason should be a change in the invoice calculation for our class.
- But in this architecture, if we wanted to change the printing format, we would need to change the class. This is why we should not have printing logic mixed with business logic in the same class.
- There is another method that violates the SRP in our class: the **saveToFile** method. It is also an extremely common mistake to mix persistence logic with business logic.

How to Fix?

- We can create new classes for our printing and persistence logic so we will no longer need to modify the invoice class for those purposes.

How to Fix?

```cpp
// Invoice class definition
class Invoice {
public:
    Book book;
    int quantity;
    double discountRate;
    double taxRate;

    // Constructor
    Invoice(const Book& b, int q, double d, double t) {
        book = b;
        quantity = q;
        discountRate = d;
        taxRate = t;
    }

    // Calculate total amount
    double calculateTotal() const {
        double discountedPrice = book.price * (1 - discountRate);
        return discountedPrice * quantity * (1 + taxRate);
    }
};
```

How to Fix?

```cpp
// Class responsible for printing invoice details
class InvoicePrinter {
public:
    void print(const Invoice& invoice) const {
        cout << invoice.quantity << " x " << invoice.book.name
             << " " << invoice.book.price << "$" << endl;
        cout << "Total: " << invoice.calculateTotal() << "$" << endl;
    }
};

// Class responsible for saving invoice to a file
class InvoicePersistence {
public:
    void saveToFile(const Invoice& invoice, const string& filename) const {
        ofstream outFile(filename);
        if (outFile) {
            outFile << invoice.quantity << " x " << invoice.book.name
                    << " " << invoice.book.price << "$" << endl;
            outFile << "Total: " << invoice.calculateTotal() << "$" << endl;
        } else {
            cout << "Unable to open file." << endl;
        }
    }
};
```

References

- Clean Architecture: A Craftsman's Guide to Software Structure and Design, 1st Edition, Robert C. Martin, Pearson, 2017.