**Liskov Substitution Principle**

Substitutability is a principle in object-oriented programming stating that,

In a computer program, if S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e. an object of type T may be substituted with any object of a subtype S) without altering any of the desirable properties of the program (correctness, task performed, etc.)

Liskov's notion of a behavioural subtype defines a notion of substitutability for objects; that is,

If S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program (e.g. correctness).

Liskov's principle imposes some standard requirements on signatures that have been adopted in newer OOP languages (usually at the level of classes rather than types):

In addition to the signature requirements, the subtype must meet a number of behavioural conditions. Behavioral subtyping (aka “IS-A” relationship):

1. Contravariance of method arguments in a subtype
2. Covariance of return types in a subtype
3. Preconditions cannot be strengthened in a subtype
4. Postconditions cannot be weakened in a subtype
5. Invariants of the super type must be preserved in a subtype
6. No new exceptions should be thrown by methods of the subtype, except where those exceptions are themselves subtypes of exceptions thrown by the methods of the supertype

**Precondition**

A precondition is a condition or predicate that must always be true just prior to the execution of some section of code or before an operation in a formal specification.

If a precondition is violated, the effect of the section of code becomes undefined and thus may or may not carry out its intended work. Security problems can arise due to incorrect preconditions.

**Postcondition**

A postcondition is a condition or predicate that must always be true just after the execution of some section of code or after an operation in a formal specification. Postconditions are sometimes tested using assertions within the code itself. Often, postconditions are simply included in the documentation of the affected section of code.

**Invariant**

In computer science, one can encounter invariants that can be relied upon to be true during the execution of a program, or during some portion of it. It is a logical assertion that is always held to be true during a certain phase of execution. For example, a loop invariant is a condition that is true at the beginning and end of every execution of a loop.

# Implementation Guidelines

1. No new exceptions should be thrown by methods of the subtype, except where those exceptions are themselves subtypes of exceptions thrown by the methods of the supertype
2. Clients should not know which specific subtype they are calling
3. New derived classes just extend without replacing the functionality of old classes

# Example

#include <iostream>

#include <string>

#include <exception>

class Employee {

int id;

std::string name;

public:

virtual ~Employee() {};

Employee(int ID, std::string Name) : id(ID), name(Name) {}

virtual double calculate\_bonus(double salary) = 0;

};

class PermanentEmployee : public Employee {

public:

virtual ~PermanentEmployee() {}

PermanentEmployee(int ID, std::string Name) : Employee(ID, Name){}

double calculate\_bonus(double salary) override {

return (salary \* 0.1);

}

};

class TemporaryEmployee : public Employee {

public:

virtual ~TemporaryEmployee() {}

TemporaryEmployee(int ID, std::string Name) : Employee(ID, Name){}

double calculate\_bonus(double salary) override {

return (salary \* 0.05);

}

};

class ContractEmployee : public Employee {

public:

virtual ~ContractEmployee() {}

ContractEmployee(int ID, std::string Name) : Employee(ID, Name){}

double calculate\_bonus(double salary) override {

throw std::runtime\_error("could not caculate bonus"); // new exception thrown in subtype

// voilation of implementation guide

}

};

int main() {

Employee \* emp = nullptr;

PermanentEmployee \* pemp = new PermanentEmployee(1, "Emp1");

TemporaryEmployee \* temp = new TemporaryEmployee(2, "Emp2");

ContractEmployee \* cemp = new ContractEmployee(3, "Emp3");

{

emp = pemp;

std::cout << "emp->calculate\_bonus(1000) " << emp->calculate\_bonus(1000) << '\n';

}

{

emp = temp;

std::cout << "emp->calculate\_bonus(1000) " << emp->calculate\_bonus(1000) << '\n';

}

{

emp = cemp;

try {

std::cout << "emp->calculate\_bonus(1000) " << emp->calculate\_bonus(1000) << '\n'; // throw exception

} catch(const std::exception & exp) {

std::cout << exp.what() << '\n';

}

}

delete pemp; pemp = nullptr;

delete cemp; cemp = nullptr;

delete temp; temp = nullptr;

return 0;

}

Output:

emp->calculate\_bonus(1000) 100

emp->calculate\_bonus(1000) 50

emp->calculate\_bonus(1000) could not caculate bonus

## Solution

**How to overcome this problem?**

Separate Interface for employee information and calculate bonus

#include <iostream>

#include <string>

#include <exception>

class EmployeeWithBonus {

public:

virtual ~EmployeeWithBonus() {}

virtual double calculate\_bonus(double salary) = 0;

};

class Employee {

int id;

std::string name;

public:

virtual ~Employee() {}

Employee(int id, std::string name) : id(id), name(name) {}

std::string getName() { return name; }

};

class PermanentEmployee : public Employee, public EmployeeWithBonus {

public:

virtual ~PermanentEmployee() {}

PermanentEmployee(int id, std::string name) : Employee(id, name){}

double calculate\_bonus(double salary) override {

return (salary \* 0.1);

}

};

class TemporaryEmployee : public Employee, public EmployeeWithBonus {

public:

virtual ~TemporaryEmployee() {}

TemporaryEmployee(int id, std::string name) : Employee(id, name){}

double calculate\_bonus(double salary) override {

return (salary \* 0.05);

}

};

class ContractEmployee : public Employee {

public:

virtual ~ContractEmployee() {}

ContractEmployee(int id, std::string name) : Employee(id, name){}

};

void displayBonus(EmployeeWithBonus \* emp, int salary) {

std::cout << "Bonus " << " is " << emp->calculate\_bonus(salary) << '\n';

return ;

}

int main() {

PermanentEmployee \* pemp = new PermanentEmployee(1, "Emp1");

TemporaryEmployee \* temp = new TemporaryEmployee(2, "Emp2");

ContractEmployee \* cemp = new ContractEmployee(3, "Emp3");

displayBonus(pemp, 5000);

displayBonus(temp, 4000);

displayBonus(cemp, 3000);

delete pemp; pemp = nullptr;

delete temp; temp = nullptr;

delete cemp; cemp = nullptr;

return 0;

}

Compilation Error

<source>: In function 'int main()':

<source>:63:18: error: cannot convert 'ContractEmployee\*' to 'EmployeeWithBonus\*'

63 | displayBonus(cemp, 3000);

| ^~~~

| |

| ContractEmployee\*

<source>:50:39: note: initializing argument 1 of 'void displayBonus(EmployeeWithBonus\*, int)'

50 | void displayBonus(EmployeeWithBonus \* emp, int salary) {

| ~~~~~~~~~~~~~~~~~~~~^~~

# References

<https://www.youtube.com/watch?v=gnKx1RW_2Rk&list=PL6n9fhu94yhXjG1w2blMXUzyDrZ_eyOme&index=6&t=182s>

<https://www.youtube.com/watch?v=Ntraj80qN2k&list=PLJFFAyzgew0-ZstdDTPgR0EN5ObxZYCZV&index=2&t=1848s>

<https://en.wikipedia.org/wiki/Liskov_substitution_principle>

<https://web.archive.org/web/20150905081111/http://www.objectmentor.com/resources/articles/lsp.pdf>

