

# ECE326

## PROGRAMMING LANGUAGES

### **Lecture 9 : Composition and Mixins**

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2019

# Fragile Base Class

- Changing base class can break derived classes
- Root cause
  - Implementation inheritance breaks encapsulation
    - Subclass depends on implementation of base class
- Solution?
  - Never change the base class
  - Author of base class must control all its subclasses
    - E.g. all subclasses are contained within same module
  - Don't use implementation inheritance

# New Interface Problem

```
/* Dictionary with key-value pair type (cstring, int) */
class Dict {
    /* some sort of hash table implementation */
public:
    /* both functions return true on success, false on fail */
    virtual bool set(const char * key, int value);
    bool get(const char * key, int & value);
};

/* Dictionary that only stores prime numbers */
struct PrimeDict : public Dict {
    virtual bool set(const char * key, int value) override {
        if (!is_prime_number(value)) return false;
        return Dict::set(key, value);
    }
};
```

# New Interface Problem

- What happens if we add a new interface?

```
class Dict {  
    ...  
public:  
    virtual bool set(const char * key, int value);  
    bool get(const char * key, int & value);  
  
    /* new interface */  
    int & operator[](const char * key);  
};
```

- We just opened a backdoor to bypass the prime number check made by PrimeDict::set

# Override Problem

```
class List { // list of integers
public:
    /* both functions return true on success, false on fail */
    virtual bool add(int value);
    virtual bool extend(const List & other);
};

/* Stores statistics on the numbers in the list */
struct StatsList : public List {
    int prime_counter = 0; // and other stats...
    virtual bool add(int value) override {
        if (is_prime_number(value)) prime_counter++;
        return List::add(value);
    }
    virtual bool extend(const List & other) override {
        for (iterator it = other.begin(); it != other.end(); it++)
            if (is_prime_number(*it)) prime_counter++;
        return List::extend(other);
    }
};
```

# Override Problem

- What if `List::extend` uses `List::add`?

```
bool List::extend(const List & other) {
    for (iterator it = other.begin(); it != other.end(); it++)
        if (!this->add(*it)) return false;
    return true;
}

bool StatsList::add(int value) {
    if (is_prime_number(value)) prime_counter++;
    return List::add(value);
}

bool StatsList::extend(const List & other) {
    for (iterator it = other.begin(); it != other.end(); it++)
        if (is_prime_number(*it)) prime_counter++;
    return List::extend(other);
}
```

# Override Problem

- What if List::extend uses List::add?

```
bool List::extend(const List & other) {  
    for (iterator it = other.begin(); it != other.end(); it++)  
        if (!this->add(*it)) return false;  
    return true;  
}  
  
bool StatsList::add(int value) {  
    if (is_prime_number(value)) prime_counter++;  
    return List::add(value);  
}  
  
bool StatsList::extend(const List & other) {  
    for (iterator it = other.begin(); it != other.end(); it++)  
        if (is_prime_number(*it)) prime_counter++;  
    return List::extend(other);  
}
```

prime counter  
is incremented  
twice!

# New Implementation Problem

```
class Album {
    std::vector<Photo> current, archived;
public:
    bool add(const Photo & photo);
    bool archive(const Photo & photo);
};

/* allows for removal of photos from album */
struct SecureAlbum : public Album {
    void remove(const Photo & photo) {
        current.erase(std::remove(current.begin(),
                                   current.end(), photo), current.end());
        archived.erase(std::remove(archived.begin(),
                                   archived.end(), photo), archived.end());
    }
};
```



# New Implementation Problem

- What if we made album track location and people?

```
using namespace std;
class Album {
    vector<Photo> current, archived;
    unordered_map<Person, vector<const Photo &>> people;
    unordered_map<Location, vector<const Photo &>> locations;
public:
    /* now a variable argument function */
    bool add(const Photo &, const Location * loc=nullptr,
            int num_people=0, ...);
};
```

- Now SecureAlbum will leave behind dangling references to deleted photos, eventually causing a crash!

# Solution

- Disallow inheritance
  - `final` keyword
    - Prevents class from being inherited
- Alternative: *composition*
  - Instead of subclassing, make it a *field* in your class
  - Instead of *is-a*, now a *has-a*
  - *Forward* methods to contained instance
  - Treats existing class as a *blackbox*
    - No more dependency of implementation
    - Enforces encapsulation

# New Interface

```
struct Dict {
    bool set(const char * key, int value);
    bool get(const char * key, int & value);
    int & operator[](const char * key);
};

class PrimeDict {
    Dict dict;
public:
    bool set(const char * key, int value) {
        if (!is_prime_number(value)) return false;
        return dict.set(key, value);
    }
    bool get(const char * key, int & value) {
        return dict.get(key, value);
    }
};
```

Adding new  
interface to Dict  
does not affect  
the interface of  
PrimeDict

Forwarding

# No Override

```
struct List {  
    bool add(int value);  
    bool extend(const List & other);  
};  
  
class StatsList {  
    List list;  
    int prime_counter = 0;  
public:  
    bool add(int value) {  
        if (is_prime_number(value)) prime_counter++;  
        return list.add(value);  
    }  
    bool extend(const List & other) {  
        for (iterator it = other.begin(); it != other.end(); it++)  
            if (is_prime_number(*it)) prime_counter++;  
        return list.extend(other);  
    }  
};
```


Avoid changing behaviour of base class when overriding methods by using composition instead.

Question: Is there a trade-off here?

# No Override

```
struct List {  
    bool add(int value);  
    bool extend(const List & other);  
};  
  
class StatsList {  
    List list;  
    int prime_counter = 0;  
public:  
    bool add(int value) {  
        if (is_prime_number(value)) prime_counter++;  
        return list.add(value);  
    }  
    bool extend(const List & other) {  
        for (iterator it = other.begin(); it != other.end(); it++)  
            if (is_prime_number(*it)) prime_counter++;  
        return list.extend(other);  
    }  
};
```

Answer: the extend function will no longer accept StatsList as an argument because it is not a subclass of List.



# Drawback

- Forwarding requires re-implementation
  - Can be tedious for large number of methods
  - In contrast, subclassing allows for immediate code reuse
- Solutions
  - Automatic forwarding (delegation)
  - Type embedding
  - Mixin
  - Traits
  - Protocol (in Java, called Interface)


# Delegation

- Kotlin

- Google's preferred language for Android app development
- Statically-typed language that runs on JVM

```
interface Base {  
    fun print()  
}  
  
class BaseImpl(val x: Int) : Base {  
    override fun print() { print(x) }  
}  
  
class Derived(b: Base) : Base by b  
  
fun main() {  
    val b = BaseImpl(10)  
    Derived(b).print()  
}
```

Automatically forwards  
this method call to  
b.print(), where b is an  
instance of BaseImpl



# Delegation

- Can also override delegation

```
interface Base {  
    fun print()  
}  
  
class BaseImpl(val x: Int) : Base {  
    override fun print() { print(x) }  
}  
  
class Derived(b: Base) : Base by b {  
    override fun print() { print("abc") }  
}  
  
fun main() {  
    val b = BaseImpl(10)  
    Derived(b).print()  
}
```

Prints "abc" instead





# Delegation

- Java Delegate Annotation
  - Allows for even more fine-grained control over delegation

```
public class DelegationExample {  
  
    // only allow add and remove to be forwarded  
    private interface SimpleCollection {  
        boolean add(String item);  
        boolean remove(Object item);  
    }  
  
    @Delegate(types=SimpleCollection.class)  
    private final Collection<String> collection =  
        new ArrayList<String>();  
}
```

# Delegation

- Java Delegate Annotation
  - Can exclude methods from being forwarded

```
public class DelegationExample {  
  
    private interface Add {  
        boolean add(String item);  
        boolean addAll(Collection<? extends String> x);  
    }  
  
    // do NOT forward add and addAll  
    @Delegate(excludes=Add.class)  
    private final Collection<String> collection =  
        new ArrayList<String>();  
}
```

# Type Embedding

- Allows for composition with automatic forwarding
- Introduced in Go
  - Does not have (implementation) inheritance
  - Uses composition for code reuse

```
type Job struct {  
    command string    // go syntax is IDENT TYPE  
    *log.Logger        // embedded types have no identifier,  
}                    // just the type: * log.Logger  
  
job := &Job{"take out trash", log.New(os.Stderr, "", log.LUTC)}  
  
// job has all methods of log.Logger, including Println  
Job.Println("working on it now...")
```

# Mixin

- Code reuse without becoming the parent class
  - Inclusion rather than inheritance
- Provides functionality for another class
- Can contain states (e.g. have member variables)
- Frequently same as implementation inheritance
  - Convention: do not use mixin as an abstract base type
    - `isinstance(obj, mixin)` is semantically meaningless
- Also implemented using metaprogramming techniques

# Mixin

- Python Comparable

# Comparable Mixin, you must supply `__lt__` to enable these

```
class Comparable:
```

```
    def __eq__(self, other):
```

```
        return not (self < other) and not (other < self)
```

```
    def __ge__(self, other):
```

```
        return not (self < other)
```

```
    def __ne__(self, other):
```

```
        return self < other or other < self
```

```
    def __le__(self, other):
```


```
        return self < other or not (other < self)
```

```
    def __gt__(self, other):
```

```
        return not (self < other) and other < self
```

# Mixin

```
class Student(Person, Comparable):  
    def __init__(self, name, score):  
        Person.__init__(self, name)  
        self.score = score  
  
    def __lt__(self, other):  
        return self.score < other.score
```



Receives all the other  
comparison operators  
just by implementing  
the less than operator

```
>> a = Student("Alice", 50)  
>> b = Student("Bob", 60)  
>> a == b, a != b, a >= b, a > b, a <= b, a < b  
(False, True, False, False, True, True)  
>> c = Student("Clive", 50)  
>> a == c, a != c, a >= c, a > c, a <= c, a < c  
(True, False, True, False, True, False)
```

# Mixin, Traits, and Protocol

- Provides functionality without subtyping

	Mixin	Trait	Protocol
Allows for states (member variables)	Yes	No	No
Interface only (no method definition)	No	No	Yes
Code Reuse	Yes	Yes	No
Composability	Inheritance	Commutative	Commutative

- The Comparable example is also a trait
  - It contained no states (no member variables were used)

# Conclusion

- Use inheritance only for interface (i.e. subtyping)
- Use composition and other techniques for code reuse
  - Avoid fragile software, difficult to maintain and extend
- Dependency Inversion Principle
  - High level modules should not depend on low level modules.
    - Both should depend on interfaces.
  - Interface should not depend on implementation.
    - Implementation should depend on interface.
  - Can swap out any component without behaviour change
    - Performance change is possible, and improvement is preferred