Programming Languages

Subclassing vs. Composition

WashU CSE 425s
Prof. Dennis Cosgrove
#11: Fri, Apr 17, 2020

S&Q: When Is Composition A Better Choice Over Inheritance?

what are some examples of when we SHOULD use the "internal instance variable " pattern instead of subclassing, and why is subclassing inappropriate in those cases?

What are some situations where using the last alternative to subclassing (put the "superclass" object as an instance variable of the "subclass") is better than using subclassing?

Composition and Delegation

```java
public class WrappedSet<E> implements Iterable<E> {
    private final ArrayList<E> delegate = new ArrayList<E>();
    public boolean add(E e) {
        if (delegate.contains(e)) {
            return false;
        } else {
            return delegate.add(e);
        }
    }
    public boolean contains(E e) {
        return delegate.contains(e);
    }
    public int size() {
        return delegate.size();
    }
    public boolean remove(E e) {
        return delegate.remove(e);
    }
    @Override
    public Iterator<E> iterator() {
        return delegate.iterator();
    }
}
```

```java
public class InheritedSet<E> extends ArrayList<E> {
    @Override
    public boolean add(E e) {
        if (contains(e)) {
            return false;
        } else {
            return super.add(e);
        }
    }
}
```

Composition and Delegation

```
public class WrappedSet<E> implements Iterable<E> {
    private final ArrayList<E> delegate = new ArrayList<E>();
    public boolean add(E e) {
        if (delegate.contains(e)) {
            return false;
        } else {
            return delegate.add(e);
        }
    }
    public boolean contains(E e) {
        return delegate.contains(e);
    }
    public int size() {
        return delegate.size();
    }
    public boolean remove(E e) {
        return delegate.remove(e);
    }
    @Override
    public Iterator<E> iterator() {
        return delegate.iterator();
    }
}
```

```
public class InheritedSet<E> extends ArrayList<E> {
    @Override
    public boolean add(E e) {
        if (contains(e)) {
            return false;
        } else {
            return super.add(e);
        }
    }
    @Override
    public void add(int index, E element) {
        if (contains(element)) {
            throw new UnsupportedOperationException();
        } else {
            super.add(index, element);
        }
    }
}
```

set(index,e)?!? Maybe I'll switch to LinkedList???

```java
public class WrappedSet<E> implements Iterable<E> {
    private final ArrayList<E> delegate = new ArrayList<E>();
    public boolean add(E e) {
        if (delegate.contains(e)) {
            return false;
        } else {
            return delegate.add(e);
        }
    }
    public boolean contains(E e) {
        return delegate.contains(e);
    }
    public int size() {
        return delegate.size();
    }
    public boolean remove(E e) {
        return delegate.remove(e);
    }
    @Override
    public Iterator<E> iterator() {
        return delegate.iterator();
    }
}
```

```java
public class InheritedSet<E> extends ArrayList<E> {
    @Override
    public boolean add(E e) {
        if (contains(e)) {
            return false;
        } else {
            return super.add(e);
        }
    }
    @Override
    public void add(int index, E element) {
        if (contains(element)) {
            throw new UnsupportedOperationException();
        } else {
            super.add(index, element);
        }
    }
    @Override
    public E set(int index, E element) {
        return super.set(index, element);  // TODO
    }
}
```

Imagine addAll's implementation changes

```
public class WrappedSet<E> implements Iterable<E> {
    private final ArrayList<E> delegate = new ArrayList<E>();
    public boolean add(E e) {
        if (delegate.contains(e)) {
            return false;
        } else {
            return delegate.add(e);
        }
    }
    public boolean contains(E e) {
        return delegate.contains(e);
    }
    public int size() {
        return delegate.size();
    }
    public boolean remove(E e) {
        return delegate.remove(e);
    }
    @Override
    public Iterator<E> iterator() {
        return delegate.iterator();
    }
}
```

```
public class InheritedSet<E> extends LinkedList<E> {
    @Override
    public boolean add(E e) {
        if (contains(e)) {
            return false;
        } else {
            return super.add(e);
        }
    }
    @Override
    public void add(int index, E element) {
        if (contains(element)) {
            throw new UnsupportedOperationException();
        } else {
            super.add(index, element);
        }
    }
    @Override
    public boolean addAll(Collection<? extends E> c) {
        return super.addAll(c); // TODO
    }
}
```

Both have to build equals() and maybe toString()

```
public class WrappedSet<E> implements Iterable<E> {
    private final ArrayList<E> delegate = new ArrayList<E>();
    public boolean add(E e) {
        if (delegate.contains(e)) {
            return false;
        } else {
            return delegate.add(e);
        }
    }
    public boolean contains(E e) {
        return delegate.contains(e);
    }
    public int size() {
        return delegate.size();
    }
    public boolean remove(E e) {
        return delegate.remove(e);
    }
    @Override
    public Iterator<E> iterator() {
        return delegate.iterator();
    }
}
```

```
public class InheritedSet<E> extends LinkedList<E> {
    @Override
    public boolean add(E e) {
        if (contains(e)) {
            return false;
        } else {
            return super.add(e);
        }
    }
    @Override
    public void add(int index, E element) {
        if (contains(element)) {
            throw new UnsupportedOperationException();
        } else {
            super.add(index, element);
        }
    }
    @Override
    public boolean addAll(Collection<? extends E> c) {
        return super.addAll(c); // TODO
    }
}
```

What if ArrayList or LinkedList adds new methods?

```
public class WrappedSet<E> implements Iterable<E> {
    private final ArrayList<E> delegate = new ArrayList<E>();
    public boolean add(E e) {
        if (delegate.contains(e)) {
            return false;
        } else {
            return delegate.add(e);
        }
    }
    public boolean contains(E e) {
        return delegate.contains(e);
    }
    public int size() {
        return delegate.size();
    }
    public boolean remove(E e) {
        return delegate.remove(e);
    }
    @Override
    public Iterator<E> iterator() {
        return delegate.iterator();
    }
}
```

```
public class InheritedSet<E> extends LinkedList<E> {
    @Override
    public boolean add(E e) {
        if (contains(e)) {
            return false;
        } else {
            return super.add(e);
        }
    }
    @Override
    public void add(int index, E element) {
        if (contains(element)) {
            throw new UnsupportedOperationException();
        } else {
            super.add(index, element);
        }
    }
    @Override
    public boolean addAll(Collection<? extends E> c) {
        return super.addAll(c); // TODO
    }
}
```

java.util.Properties extends Hashtable

you literally wanted to constrain it from String to String and chose to extend something that allowed Object to Object?!?

java.util.Stack extends Vector

first, why is this not an interface?

second, why would methods other than push, pop, and peek be a good idea?

inheritance couples implementations together. are you prepared for that?
because, now we are stuck with this cruddy Stack ***FOREVER***

S&Q: When Is Composition A Better Choice Over Inheritance?

https://www.google.com/search?q=composition+vs+inheritance

some say composition is always better.
some say you should only inherit interfaces.

Ask Yourself: *Is Every B ***Really*** an A?*

me: *Is Every B ***Really*** an A?*

also me: *Yes.*

me: *You just want that sweet, sweet code reuse don't you? Seriously, is every B really an A?*

also me: *Well...*

me: *You are going to subclass, aren't you?*

Liskov Substitution Principle

if B is a subtype of A then
all B objects should be able to be used anywhere an A object is used

there should no surprises!

you cannot require more (preconditions: same or weaker)

you cannot deliver less (postconditions: same or stronger)

*** Should MutableList extend ImmutableList?

Should MutableList extend ImmutableList?

No. No it should not.

It breaks the contract of ImmutableList.

Anyone who is using an ImmutableList should reasonably be able to expect that it is immutable.

Imagine you memoize the length of an ImmutableList you are passed. Surprise! It's actually mutable!

Note: Kotlin seems to try to skirt this issue with List and MutableList

Is The Class You Would Like To Extend Designed To Be a Superclass?

- Is it abstract? (Dark side: if it is not final…)
- Does it provide specific instructions on how to extend it for maintainability?
  - AbstractMap<K,V> is built to support most functionality via overriding entrySet()
  - AbstractCollection<E> is built to support most functionality via overriding iterator() and size()
- Do you have complete control over the superclass?
- Are you prepared to co-evolve your class with the superclass for all time?

I Smell A Conspiracy Theory

Okay. So, Java made it so the class definition can also serve as a type. This is nice because it means I don't have to make an interface for every single class… but alright. Alright. You say I should make interfaces, so I make an interface to go with every darned class. Now you are telling me that I shouldn't use the sweetest bit of code reuse ever invented… subclassing. You just want functional programming to be better, don't you?!?

Inheritance Is The Ring of Power of Code Reuse

Granted, forwarding all methods can be annoying

Solution: design a language which makes delegation automatic.

However, there are very real constraints put on the flexibility of your system when you choose to subclass.

Further Reading On The Subject

http://web.mit.edu/6.005/www/fa14/classes/14-inheritance/

Effective Java