計算機科学第一

2012年度第9回

日付

もくじ

16: Favor composition over inheritance

16: 構成 > 継承

継承の利点

- 親クラスでの宣言の再利用
- 親クラスでのメソッド定義の再利用
- 親クラスの拡張
- 親クラスの型との包含関係

継承の利点

- 親クラスでの宣言の再利用
- 親クラスでのメソッド定義の再利用
- 親クラスの拡張
- 親クラスの型との包含関係

利点は多いが、使い方は要注意。

原則：**無闇に継承は使わない**

Alan Snyder: Encapsulation and inheritance in object-oriented programming languages (OOPSLA 1986)

- class Child extends Parent {
  void foo() { ... super.foo(); ... } }
- ParentとChildは別のプログラマが実装
- Parent.foo の実装変更 → Child.foo の挙動が（思いがけず）変化

Alan Snyder: Encapsulation and inheritance in object-oriented programming languages (OOPSLA 1986)

- class Child extends Parent {
  void foo() { ... super.foo(); ... } }

- ParentとChildは別のプログラマが実装
- Parent.foo の実装変更 → Child.foo の挙動が（思いがけず）変化
- 親クラスの実装が変ったら子クラスも変更しないといけない ← **情報隠蔽の破綻**

InstrumentedSet

- 集合を実装したクラス
- 追加された要素の総数を記録する

InstrumentedSetの実装例

- InstrumentedHashSet1
  - 継承を用いた素朴な実装 ← 動作せず
- InstrumentedHashSet2
  - やや強引な修正 (情報隠蔽の破綻)
- InstrumentedSet1 (Composition)
- InstrumentedSet2 (Wrapper Composition)

継承 vs 情報隠蔽の実際

- 親クラスの実装が変ったら子クラスも変更しないといけない ← **情報隠蔽の破綻**

- 具体例: Java 1.6 では動いていたのに、Java 1.7 では動かなくなった！

- Java 1.6 で開発してたのに、出荷する一ヶ月前になって Java 1.7 が発表された！

- 隣のチームがライブラリを新しくしたら、動かなくなった！

Composition

```
public class InstrumentedSet1<E> implements InstrumentedSet<E> {
        private Set<E> s;
        private int addCount = 0;
        public int getCount() { return addCount; }
        public InstrumentedSet1() { s = new HashSet<E>(); }

        public boolean add(E e) {
                addCount++;
                return s.add(e);
        }
        public boolean addAll(Collection<? extends E> c) {
                addCount += c.size();
                return s.addAll(c);
        }
        public void clear() { s.clear(); }
        public boolean contains(Object o) { return s.contains(o); }
        public boolean containsAll(Collection<?> c) { return s.containsAll(c); }
        public boolean isEmpty() { return s.isEmpty(); }
        public Iterator<E> iterator() { return s.iterator(); }
        public boolean remove(Object o) { return s.remove(o); }
        public boolean removeAll(Collection<?> c) { return s.removeAll(c); }
        public boolean retainAll(Collection<?> c) { return s.retainAll(c); }
        public int size() { return s.size(); }
        public Object[] toArray() { return s.toArray(); }
        public <T> T[] toArray(T[] a) { return s.toArray(a); }
}
```

Wrapper Composition

```
public class InstrumentedSet2<E> extends SetAdapter<E>
implements InstrumentedSet<E> {
        private int addCount = 0;
        public int getCount() { return addCount; }

        public InstrumentedSet2(Set<E> s) { super(s); }
        public InstrumentedSet2() { this(new TreeSet<E>()); }

        public boolean add(E e) {
                addCount++;
                return super.add(e);
        }
        public boolean addAll(Collection<? extends E> c) {
                addCount += c.size();
                return super.addAll(c);
        }
}
```

Wrapper compositionの利点

	Inheritance	Composition	Wrapper
情報隠蔽	×	○	○
再利用性	× HashSet	× HashSet	○ Set<E>

Wrapperクラスの作成手順

```
package lecture07.s16prefer_composition;

import java.util.Set;

public class SetAdapter<E> implements Set<E> {
    private Set<E> s;
    public SetAdapter(Set<E> s) { this.s = s; }
}
```

まずは決まりきったことを

書く、書く、書く

実装していないメソッドの雛形を追加

```
package lecture07.s16prefer_composition;

import java.util.Set;

public class SetAdapter<E> implements Set<E> {
    private S
    public Se
}
```

Add unimplemented methods
Make type 'SetAdapter' abstract
Rename in file (⌘2 R)
Rename in workspace (⌥⌘R)

13 methods to implement:
- java.util.Set.add()
- java.util.Set.addAll()
- java.util.Set.clear()
- java.util.Set.contains()
- java.util.Set.containsAll()
- java.util.Set.isEmpty()
- java.util.Set.iterator()
- java.util.Set.remove()
- java.util.Set.removeAll()
- java.util.Set.retainAll()
- java.util.Set.size()
- java.util.Set.toArray()

Press 'Tab' from proposal table or click for focus

エラーのサインをクリックすると、対処法が提示されるので適切なものを選択する。

この場合は “Add unimplemented methods” (未実装のメソッドの追加)。

自動生成されたメソッド群

```
public class SetAdapter<E> implements Set<E> {
    private Set<E> s;
    public SetAdapter(Set<E> s) { this.s = s; }

    @Override
    public boolean add(E e) {
        // TODO Auto-generated method stub
        return false;
    }
    @Override
    public boolean addAll(Collection<? extends E> c) {
        // TODO Auto-generated method stub
        return false;
    }
    @Override
    public void clear() {
        // TODO Auto-generated method stub
```

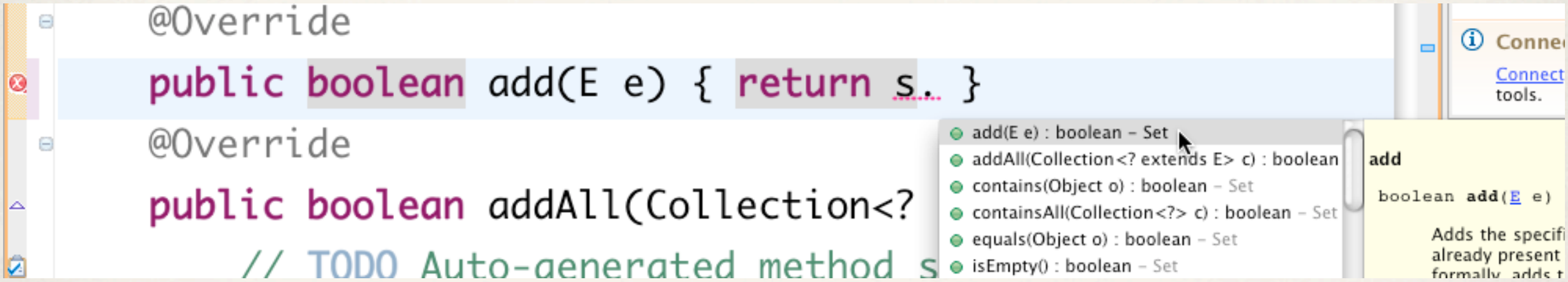
修正箇所を選択

```
    private Set<E> s;
    public SetAdapter(Set<E> s) { this.s = s; }

    @Override
    public boolean add(E e) {
        // TODO Auto-generated method stub
        return false;
    }
    @Override
    public boolean addAll(Collection<? extends E> c) {
        // TODO Auto-generated method stub
        return false;
    }
    @Override
```

矢印キーを使うのが吉（↑、↓が案外便利）

コードの一部を入力



候補を↑、↓で選択し、

Enter

※ void なメソッドは

return しないことに注意

セミコロンを追加

```
    @Override
    public boolean add(E e) { return s.add(e) }
    @Override
```

→→で括弧の右に移動して、

;を入力

一丁あがり！

```
@Override
public boolean add(E e) { return s.add(e); }
@Override
```

何度か練習すれば

SetAdapter を1-2分くらいで

作れるでしょう。

継承を利用すべきシーン

- 「子クラス *“is-a”* 親クラス」が成立する場合(IS-A関係)
- B は「常に、誰が考えても」Aなのか？

is-a関係は成立しているか？

- ~~そう見做せないこともある~~
- ~~そういう風に考える人もいる~~
- ~~プログラムの実装上はそう考えられる~~
- 誰が考えてもBはAでしょ
  - 犬 is-a 動物、 多角形 is-a 図形、
    二項演算 is-a 数式

不適切な継承をしている例も多い

- java.util: Stack extends Vector
- java.util: Properties extends Hashtable

Propertiesの問題

- Hashtableの API (get) は Properties のデフォルト値に未対応

  → PropertiesTest1

- Hashtable の API (put) は文字列以外の引数も受け付けてしまう。ほかにもいろいろ混乱が。。。

  → PropertiesTest2