

Exercise session 4: Inheritance anomalies and aspect-oriented programming

1. Inheritance anomalies

1.1. Which types of inheritance anomaly are predominant in Java? Hint: think of the buffer example(s) given during the class and the "common" form of concurrent Java classes:

```
public class xxxx {  
    public synchronized ret1 method1(arg11, arg12, ...) {  
        ...  
        while (!....) wait();  
        ...  
        notifyAll();  
        ...  
    }  
  
    public synchronized ret2 method2(arg21, arg22, ...) {  
        ...  
        while (!....) wait();  
        ...  
        notifyAll();  
        ...  
    }  
    ...  
}
```

1.2. Classify the following approaches, according to their proneness to the three anomalies outlined in the class:

- Bodies (active objects, cf. Synchronous Java)
- Monitors (standard Java)
- Behaviors (cf. class)
- Guards (cf. class)

2. Aspect-oriented programming with AspectJ

(Use <http://eclipse.org/aspectj/doc/progguide/index.html>.)

Suppose an interface `ListModel` which describes methods for adding one or multiple items:

```
interface ListModel {
    void add(ListItem o);
    void add(Collection o);
}
```

2.1. Lists are synchronized with observers. Outline an aspect which notifies observers *once* about changes, i.e., either (1) the addition of an element (`add(List)`), or (2) the addition of several elements (`add(Collection)`):

```
aspect ObservableListModel{
    // some pointcut //
    ...
    // some advice called upon pointcut for informing observers //
    ... :
    { // inform each observer: sufficient as pseudo-code here // };
}
```

2.2. What happens if `add(Collection)` uses `add(ListItem)`? Propose a solution which works whether `add(Collection)` is implemented that way or not (hint: use `cflow`).

Suppose now we extend `ListModel` as follows:

```
interface ListModelExt extends ListModel {
    void add(ListItem[] li);
}
```

2.3. Outline an extended aspect for dealing with this case as well (hint: aspects can be abstract or concrete; abstract aspects can be extended; pointcuts can be overridden; there is no such thing as `super.xxx`).

2.4. Reflect on how to support modularity of aspects. How can aspects evolve incrementally in parallel to types/classes of the application without rewriting?