WS14: Software Design – Exercise

Paolo G. Giarrusso

Organization

- Divide in groups of 3 people
  - for today
  - for the later exercises
- Feel free to tune these groups before registering
- Before submitting your solutions, register these groups according to instructions on the website: http://ps-mr.github.io/

Learning goal today

- Understand why software design is interesting, why it is complex, why it matters.

Software design is complex

- Different goals
  - Maintainability
  - Extensibility
  - Reusability
  - (Asymptotic) performance

Why do we care?

- When designing a library, we design (part of) the language that client code will speak
- Our choices affect how easy/hard it is to design programs with certain properties
- Related to programming language design, a topic we won't tackle)

API design & performance

```
void loopOver(List l) {
  for (int i = 0; i < l.size(); i++) {
    Object element = l.get(i);
    //Use element
  }
}
```

- What’s the expected performance?
- What’s the performance if `l` is a `LinkedList`?

Software design is complex

- Different goals, again:
  - Maintainability
  - Extensibility
  - Reusability
  - (Asymptotic) performance

Software design is complex

- Non-goal: perfect design
  - In many cases, different solutions involve tradeoffs
  - Picking what matters and what does not is often a matter of heated debate
  - IOW: non-goal: picking what you care about
- Goal: being able to reason
- Analyzing the tradeoffs can be more objective
- If you know your goals, you can pick better
  - But should you reuse a general solution (which might fit badly) or use a different design?

Case study: collection libraries

Some possible subgoals:

- Share client code over all sequence data structures?
- Share code over all collections
  - Is "Collection" a good abstraction to program against?
  - A Set is a Collection…
  - But s.add(el); s.remove(el); behaves differently!

Possible collections

- Mutable/immutable
- Sequence/Set/Map/Bag
- Different implementations of the same abstract type
- Eager or lazy
- Synchronized or not

Possible collections, #2

Collections or not?

- String
- BitVectors
- InputStreams

Possible operations

- Add/remove elements
  - Mutable/immutable variants
- Traversing a collection
  - What's a good pointer into the collection?
    - An index
    - Part of the collection
- Transforming a collection (map, filter)
  - What should be the type of the resulting collection?

Scenario 1: no reuse

- Suppose that you had unrelated classes for
  - TreeSets
  - HashSets
  - TreeMaps
  - HashMaps
  - Arrays
  - ArrayLists
  - LinkedLists
  - Strings
- Name two problematic scenarios
- Sketch some abstractions

Home exercises (1)

(1) Try to refine your designs according to the discussion.

- (1a) Try to design a map operation, taking a collection of elements, an element transformer and returning a new collection of transformed elements. Which collection type should it return for a given collection? If map had to return different types for different collections, could you share code across implementations?

Home exercises (2)

- (1b) Should we have shared methods for adding and removing elements for all collections?
    - Can we have a shared interface for adding and removing elements covering also `Map` (that is, dictionaries)? After discussing tradeoffs, compare with the choices made by a standard collection library (such as the Java one or the STL one).
    - What are the guarantees these common methods such provide? After `coll.add(el); coll.remove(el);` should `coll` be the same as before? What if `coll` is a `Set`?

Home exercises (3)

(2) Pick a subset of an existing collection library (like the Java one) and analyze its design according to the criteria we described

- Non-goal: cover the whole library
- You might refer to existing design documents for the library you pick (which you should quote). For Java, one such document is: http://docs.oracle.com/javase/7/docs/technotes/guides/collections/index.html