Goals Practice alternative approach to Observer pattern, using a separate distributor, and pull instead of push.

Preparation

Download links and other details can be found via the course web page:

`http://www.win.tue.nl/~wstomv/edu/sc-hse/`

1. In the book *Introduction to Programming Using Java* by David Eck, read §10.3.1.
2. In the (e)book *Programming in the Large with Design Patterns* by Eddie Burris, read Chapters 4 and 5, and the draft chapter about the Composite design pattern.
3. Review the slides of Lecture 10.
4. Read the handout *From Callbacks to Design Patterns*.

Assignments

1. Inspect the peach^3 feedback on your submission for *Simple Kakuro Helper*. You can resubmit to improve your submission. Aim for a maximal score. Rejected submissions did not score enough points, and must be improved to get them accepted by peach^3.

   Note that your score depends on

   - `KakuroCombinationGenerator`
   - `KakuroCombinationGeneratorTest`
   - `Intersector`

2. (Theory questions)

   (a) Why is it not a good idea to include unnecessary test cases?

   (b) What would be the consequence for testing of `generate(int, int)` in `KakuroCombinationGenerator`, when the postcondition would be weakened by dropping the requirement of lexicographic order? That is, when `generate` would be allowed to generate combinations in any order.

   Write your answers in the provided template file, and submit them to peach^3.

3. *Simple Kakuro Helper 2* You are requested to change the Simple Kakuro Helper as follows:

- Use a pull (by observer, from generator) strategy, rather than push (by generator, to observer).
- Factor the management of multiple observers out of the generator, to a separate composite observer.
- Reuse the code for the existing listeners through an appropriate adapter (via composition; why?).
- Optionally, improve performance of the generator, in case that `maxNumber` is larger than the default 9.

For this purpose, the provided skeleton code is extended with:

- **`interface`** `Generator<A>` for generators with a single observer, supporting pull of `A` objects
- **`interface`** `GeneratorObservers<A>` observing `Generator<A>`, without pushing
- **`class`** `CompositeGeneratorObserver<A>`
- **`class`** `PushPullAdapter`

Do *not* change these entities (instead, apply the adapter in client code):

- **`interface`** `GeneratorListener`
- **`class`** `Counter`
- **`class`** `Lister`
- **`class`** `Intersector`

Submit work to peach[3]. *Please, do not remove the* `//# ... TODO` *markers from given skeleton code.*

Hint to improve performance Consider the goal of generating all combinations with sum s, length n, smallest number k, and largest number m (`maxNumber`). A simple recursive approach just adds, one by one, numbers d with $k \leq d \leq m$ to the current combination, and recursively proceeds with sum $s - d$, length $n - 1$, and smallest number $d + 1$.

However, this may involve lots of unnecessary work. Using *Branch & Bound*, this can be avoided. What is the largest sum that can be made with n distinct positive numbers at most m? If that largest sum is still smaller than the desired sum s, then there are no such combinations and recursion can "bounded", even before looping over d. What is the smallest sum that can be made with n distinct numbers at least d? If that smallest sum is already too large, then recursion can also be "bounded", breaking the loop over d.

Deadlines: See peach[3]

