

SCOOP project: SANTA STRIKES BACK

Hand-out: 14 June

Due: 19 July

The goal of the project is to design and implement (in SCOOP) two small applications that represent interesting synchronisation scenarios. No GUI is required (hooray!) – the applications can use either standard text output (console) or a text log file. The main focus is on the concurrency issues and the way you handle them with SCOOP.

You need to submit the code of your applications (as a zip file) and the project report (send both by e-mail to concur-course@se.inf.ethz.ch) and present the application before July 19.

1. Santa Claus

The solution for this nice little problem requires a mixture of non-trivial synchronisation patterns. It has already served as a homework for one of the exercise sessions. We have tried to tackle the problem using the synchronisation mechanisms of Java. The experience gathered during that exercise will help you solve the problem in SCOOP. Here is the problem statement:

Santa lives in his little house in Jyëvväskjølgrø; his favourite activity is to have a nap. So he sleeps all the time, except for rare moments when he is awoken by either a group of elves or a group of reindeer. There are 10 elves who live in the area and work in a toy workshop. They produce toys and, once in a while, they run out of ideas and need to ask Santa for help. They can only wake up Santa if they form a group of three. There are also nine reindeer that live in the stable nearby; Santa uses them to deliver toys to kids. The reindeer are as lazy as Santa himself, so for the most of the day they just sleep. When they want to work, they need to form a group of nine (that is all the reindeer must join) and wake up Santa. If they manage to do it, Santa does the delivery round and then goes back to sleep (so do the reindeer). It is important to note that in a situation when there is a group of three elves and the group of nine reindeer trying to wake up Santa at the same time, it is the reindeer that get the priority.

You have to implement the scenario in SCOOP. I would expect your application to be based on classes SANTA, REINDEER, ELF plus any additional classes that implement useful abstractions. Use inheritance whenever it is necessary – remember that abstraction is beautiful and there is nothing uglier than an application with several classes that look almost the same and contain the same code.

2. Active objects

The goal of this particular task is to find an elegant and efficient way to simulate active objects in SCOOP. Although SCOOP does not follow the active object approach, we should be able to implement scenarios that require the use of rendez-vous synchronisation. We will use our beloved Santa once again:

Santa has finally decided to go with the wind of change and computerise his business. Before going online, he wants to try out locally a simple client-server approach to optimise his consultancy activities. From now on, the elves who want to consult Santa are not required to form a group anymore – they can simply send an individual request to Santa’s server and wait until Santa accepts to meet them. After issuing a request the elf is blocked until Santa has accepted his request and met him. After the consultation the elf should work on his own for a while, then play with his kids (each elf has three kids; the kids are always ready to play with their dad, except if they are currently eating). Important: elves never sleep, they are always active!

Unlike the elves, Santa sleeps most of the time. Nevertheless, he wakes up (by himself) once in a while and checks for requests from the elves. If there are any pending requests, he accepts a request and consults the elf who issued the request. Santa does not need to accept all pending requests – in fact, we should be able to parametrise Santa to accept n requests in a row. After satisfying a given number of request Santa goes back to sleep. A very important thing: the old Santa is able to process just one request at a time!

Santa has optimised the business and he only does consulting. The delivery business has been outsourced to UPS, hence there are no reindeer involved anymore. They were useless anyway, weren’t they?

Your task is to implement the scenario in SCOOP. Obviously, the client-server scenario described above calls for a rendez-vous-style synchronisation. You need to implement active objects that represent Santa, elves, and elves’ kids. Make sure that the schedules (bodies) of the active objects specify correctly the subsequent activities.

For Santa, the activities are: sleep, accept some pending request, process the request (i.e. meet the elf), and so on. Remember that accepting a request is a blocking activity.

For an elf, the activities are: issue a request to Santa, work a bit on your own, play with your kids. Remember that after issuing a request the elf is blocked until Santa accepts the request. Also, the elf has to play with all his kids (not necessarily at the same time) before issuing the next request to Santa.

For a kid, the schedule is very simple: eat, play with your dad, eat, play with your dad, and so on.

Once again, make sure that you use inheritance properly to achieve the necessary level of abstraction. Try to provide a general class that implements an active object and then specialised classes for Santa, elves, and kids. Try to parametrise Santa in such a way that he accepts n pending requests before going back to sleep.

Sleeping, elves' own work, eating, and playing can be represented with simple routines *nap*, *work*, *eat*, and *play*. The details are irrelevant – it is enough to output a corresponding message, e.g. “Santa is sleeping”.

The main focus is on the correct implementation of the rendez-vous concept. You need to use some nice trick here since SCOOP does not allow an object to execute any features on another object that is currently busy (i.e. locked by someone else).

3. Assessment

The final mark will depend on the following criteria:

- correct implementation: the applications do what they are supposed to do.
- quality of the accompanying project report.

Questions to be answered in the project report:

- 3.1. What synchronisation patterns (producer-consumer, reader-writer, etc.) have you used in each application?
- 3.2. How have you implemented each pattern? Give a short description of the algorithm you have used and include code snippets.
- 3.3. What are the main problems with the implementation of these synchronisation patterns in SCOOP?
- 3.4. Can you think of additional mechanisms or abstractions that SCOOP should offer?
- 3.5. As a programmer, what is your feeling about SCOOP? Does it allow you to program efficiently and express what you want to express? How would you compare SCOOP with other mechanisms (say Java multithreading with monitors) in terms of expressivity? Usability and “programmer-friendliness”? When answering that question, please try to consider all important elements, e.g. ease of code reuse, etc.
- 3.6. What is your opinion about the `scoop2scoopli` tool? What would you add or improve?

4. Tools

You will use the **EiffelStudio 5.5** IDE, together with the **scoop2scoopli** pre-processor.

5. Schedule

- June 7 introduction to `scoop2scoopli`, examples
- June 14 SCOOP examples, project Q&A
- June 21 SCOOP examples, project Q&A
- July 19 deadline for project submission

6. Support

Apart from the exercise sessions, you will have the possibility to ask questions during the office hours every Thursday from 14:00 to 16:00 (RZ J3). You can also ask questions by e-mail (scoop-support@se.inf.ethz.ch). If you need an individual meeting with me on a different day, just send a request to scoop-support@se.inf.ethz.ch.