OSD Industrial practice team project

Deadline: 4pm, Friday 11th December (Submission on Keats)

October 20, 2015

- Teams are listed on Keats.
- Your team has the task to develop a scheduling system for agile development. This takes as input descriptions of development tasks, and details of the skills and cost of available developers, and generates a schedule assigning developers to tasks.
- You should write the software using UML-RSDS (the tool and manual are available at: www.dcs.kcl.ac.uk/staff/kcl/uml2web/) to specify the class diagram and use cases and to generate an executable system.
- An initial metamodel file for the problem is provided (in the output/mm.txt file on Keats), as are test cases (in.txt, in1.txt, in2.txt, test100.txt, test200.txt, test500.txt).
- Teams should be organised to have a leader – responsible for overall planning/coordination. Divide teams into members/subteams with responsibility for research/testing/documentation/training or transformation coding.
- An agile approach using exploratory and evolutionary prototyping is recommended.
- At tutorials you will have access to technical advisors experienced in UML-RSDS.
- Deliverable of project: stand-alone Java program which can read files of staff and task data, and produce correct schedules.
- Report: include final class diagram and operation specifications. Describe the development and management approach taken, the project process (who did what, why), results of testing, and efficiency evaluation.
- Marking: 30% for class diagram and constraints; 30% for implementation; 20% report quality and organisation; 20% for effective group organisation and project management.

1 Problem Statement: Resource Scheduling

Specify the following system as a class diagram and use cases in UML-RSDS:

The system is intended to do release planning for an agile development process. The development or modification work to be done is divided into a number of *Story* objects, which have a $storyId : String$ unique key. Each story has an ordered list *subtasks* of $Task$ objects, which define particular work tasks. Tasks have a unique $taskId : String$ key, and an Integer *duration*. A task may depend on other tasks (which must be completed before it is started). A task has a set, *needs*, of *Skill* objects which represent skills needed to carry out the task. In turn, a *Skill* has a unique $skillId : String$. An entity type $Staff$ represents staff, and has a unique $staffId : String$, and an Integer *costDay*. A set, *has*, of skills is associated to each staff object. Finally, the task schedule for an iteration is represented by a class *Schedule*, with an attribute $totalCost : Integer$, and an ordered list *assignment* of *Assignment* objects, where each *Assignment* has associated staff and task objects.

The required system operations are:

- *allocateStaff*: for each unallocated task t, all of whose *dependsOn* tasks have already been allocated, find an available (unallocated) staff member who has all the skills required by t, and assign the task to the cheapest such staff member, s. Create a new assignment for t and s, and add this to the schedule.

- *calculateCost*: add up the products $s.costDay * t.duration$ for the assignments of the schedule and add this to the *totalCost* of the schedule.

- *displaySchedule*: print the list of assignments, with information of the $staffId$, *costDay*, $taskId$, *duration* for each assignment.

Evaluate your solution on several test cases of planning problems (the files in.txt, in1.txt, in2.txt in *output.zip* on Keats). Does your approach always find the schedule with lowest total cost? How efficient is your solution on large problems (test100.txt, test200.txt, test500.txt)? Identify ways to improve your solution.

Add a $duration : Integer$ attribute to *Schedule*, and compute this as part of the *calculateCost* operation. [hint: define a recursive operation $totalDuration() : Integer$ of $Task$ which calculates the duration of the task together with those it depends on]

The system only calculates a schedule for a single iteration: the iteration is complete when all possible allocations to available staff have been made. Define a use case *nextIteration* to continue the schedule with a further iteration. No new classes or attributes are needed, and the use case itself has a single, very simple, constraint on *Schedule* or on *Assignment*.

2 Instructions for using UML-RSDS

- Download *umlrsds.jar* and place this in a directory which has a writable subdirectory called *output* (the *output* directory containing the test datasets is on Keats, in compressed form: copy this and uncompress it to form your own *output* directory).

  Start UML-RSDS by the command

  ```
  java -jar umlrsds.jar
  ```

  from the terminal command line. Class diagrams (eg., from output/mm.txt) can be loaded using the *Load data* option on the *File* menu.

- Updated versions of the system can be saved (to output/mm.txt) by the option Save on the File menu, and loaded by the option Recent. Regular saving is recommended, as the UML-RSDS interface sometimes 'freezes' in the K4U.13/14 environment.

- To generate code from a UML-RSDS specification, select option *Design* from the *Synthesis* menu, then option *Generate Java* 4.

- Java code files GUI.java and Controller.java are written to the *output* subdirectory, and can be compiled and run as usual using javac and java on GUI.java. The "load model" command of GUI loads a model from in.txt, and "save model" saves it to out.txt. (You will need to rename in1.txt to in.txt in order to process it, likewise for other input datasets).

- The test data sets are provided, in files *in.txt*, *in1.txt*, *in2.txt*, *test100.txt*, *test200.txt*, *test500.txt*

- You should evaluate your solution in terms of its correctness and efficiency: how long it takes to execute on the provided datasets.

Only the following OCL operators are needed: $<:$, $sortedBy$, $exists$, $select$, $first$, $size$, $isDeleted$, $:$, $display$, $collect$, sum, max, and the usual logical and comparitor operators.