**COM00028M**

THE UNIVERSITY of York

**Degree Examinations 2013 – 2014**

## DEPARTMENT OF COMPUTER SCIENCE

## MSc. in Computing

# Java Programming Concepts (JAPC)

Open Assessment

Issued: *12:00 midday*, Wednesday 30 October 2013
Submission due by: *12:00 midday*, **Wednesday 8 January 2014**

All students should submit their answers electronically to the Department of Computer Science by 12:00 midday on Wednesday 8 January 2014, following the instructions enclosed. An assessment that has been handed in after this deadline will be marked initially as if it had been handed in on time, but the Board of Examiners will then normally apply a lateness penalty.

Your attention is drawn to the Guidelines on Mutual Assistance and Collaboration in the Student's Handbook.

**Any queries on this assessment should be addressed to Rob Alexander.**
**Email:** rob.alexander@york.ac.uk

No queries will be answered more than 6 weeks after hand-out (11 December 2013).

**Your examination number must be included at the top of each source file in your program and on the first page of your testing document. You must not identify yourself in any other way.**

# JAPC Open Assessment: "Study Planner Pro"

## 1 The Program

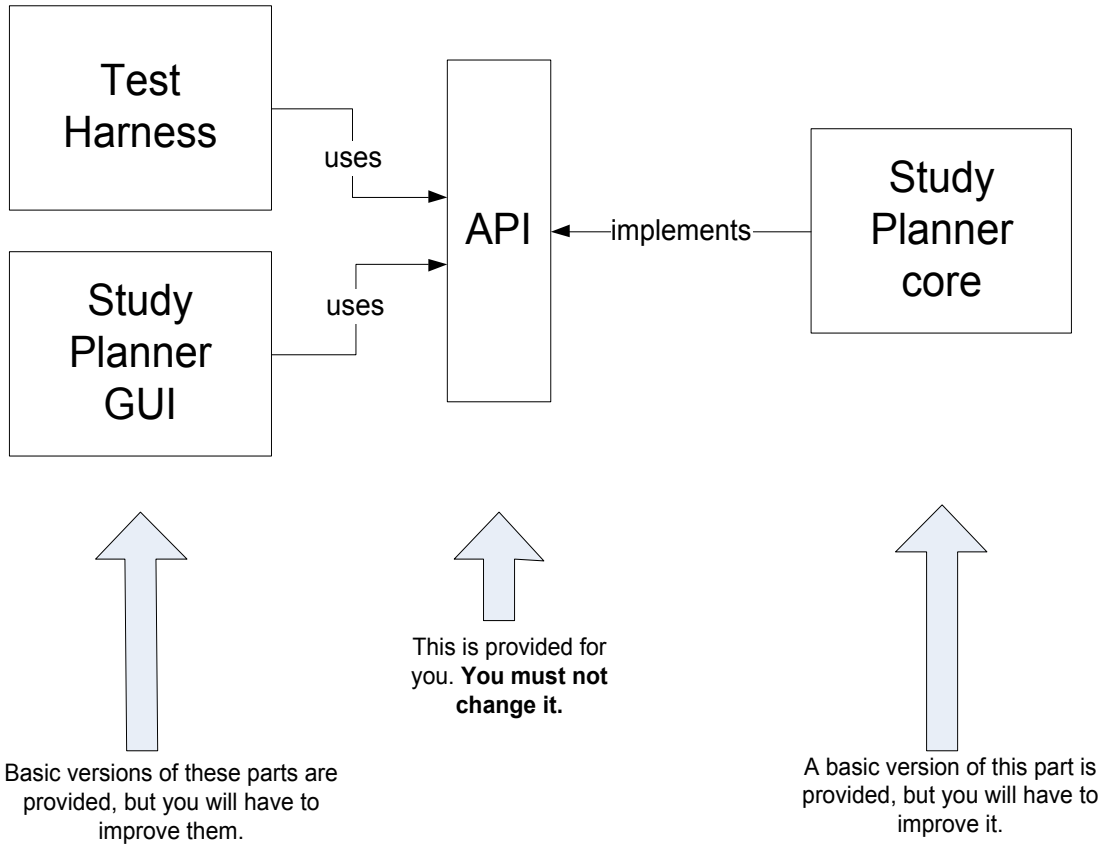
For this assessment, you need to implement a program that implements a study planner for university students. The program lets a user input a list of topics that they need to study, details of their calendar (including exam and essay deadlines), and preferences on when to start and stop studying each day and how long to study for in one go. It then generates a study schedule telling them when to study each topic and for how long.

This program will be more complicated than it might first appear, because your program must follow certain rules so it can be used in two ways: either by a human (using a GUI), or by another program using an Application Programming Interface (API). You will have to implement three things:

- The planner itself, which implements a specified API
- A *test harness* that uses that API to test the behaviour of the simulation
- A basic GUI that allows an ordinary university student (who is not a programmer) to use the planner. This must use the API, and only the API, to interact with the planner code itself

I have provided basic versions of the above – your role is to extend them substantially according to the instructions in this document.

The following diagram outlines the architecture:



## 2 Work to Do

The aim of this assessment is to gain experience in the design and implementation of a small but tricky program using an object-oriented approach. You must design and implement a set of Java classes that implement the planner, the test harness and the GUI. While doing so, you must follow the following general requirements:

- All the code should be submitted as a Java Eclipse Project (JDK 1.7).
- You should also submit a short document justifying your choice of tests to implement.

and abide by the following restrictions:

- You may *not* use any third-party Java libraries (libraries that are not part of the core Java library supplied by Oracle with the JDK) as part of your program. The one exception is JUnit, which this assessment specifically requires
- You may *not* use any third-party code generators – your code must be hand-written – except for the basic autocomplete, refactoring and GUI editing features provided by Eclipse, Netbeans and other IDEs.

Beyond the above, detailed guidance on what is required of your submission is provided in the following section. Failure to adhere this specification will result in loss of marks.

# 3 Requirements

To help you plan your efforts, the requirements have been divided into three stages, which you should complete in order. Each is worth part of the marks, with the remainder being awarded for code quality and good choice of tests (see Section 5 for a mark breakdown). For clarity, I have also described a "Level 0" which represents the features provided by the code I have provided on the module web site.

## 3.1 Level 0

### 3.1.1 The API

The API for the planner is defined by the Java interfaces StudyPlannerInterface, CalendarEventInterface, StudyBlockInterface, StudyPlannerGUIInterface and TopicInterface, along with the concrete class StudyPlannerException. The interfaces themselves are complete (you need not, and must not, change them).

### 3.1.2 The Planner

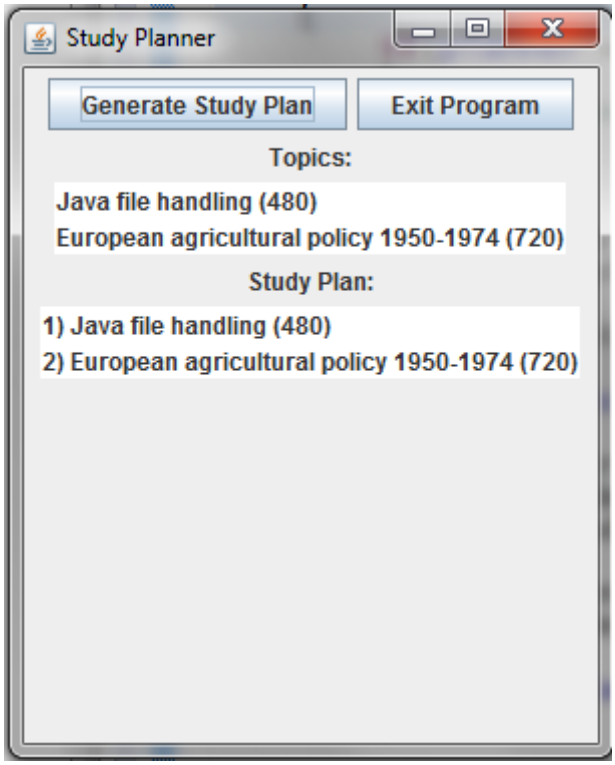
I have provided code that implements the following features:

- Topics may be added to the planner, and it maintains a list of these which can be retrieved
- When generateStudyPlan() is called, it will generate a study plan using a very dumb approach – it will propose that the student study each topic in turn for its full duration, without eating or sleeping. (While this is one approach to study, it is not a good one!)
- The most recent generated study plan can be retrieved via getStudyPlan()

NB there are no concrete classes for CalendarEventInterface. You will need to implement these yourself.

### 3.1.3 The GUI

The provided GUI displays a list of the topics held by the planner, and can generate and display a study plan.




### 3.1.4 The Test Harness

The provided test harness StudyPlannerBasicTest uses JUnit to run some very simple tests against the simulation, via the API.

You can use the provided tests for a basic form of *regression testing* – verifying that a change you just made hasn't broken something else. Run them often. The basic test set should run, successfully, on the final version of your code, with no modifications. If it doesn't run, or some tests fail, then something is wrong with your code.

## 3.2 Some Important Requirements on the Use of Interfaces

The changes I have asked for in this assessment do not require you to make any changes to the provided interfaces (e.g. StudyPlannerInterface). **I will run my own automated test programs against your planner. If you do not implement the API exactly as required (using exactly the interfaces I have provided), I will not be able to do this and you will lose a large number of marks.**

Similarly, your planner must only interact with your GUI class through the provided interface (StudyPlannerGUIInterface). If your planner class calls additional methods on the GUI class, you may lose a large number of marks. You can ensure that you obey this rule by only having a StudyPlannerGUIInterface variable within your planner class, and not having any StudyPlannerGUI variables or method parameters.

(Rationale for the above: you may wish to imagine that someone else has been commissioned to write an alternative version of the GUI using a GUI library other than Swing, perhaps for the Android mobile phone platform. The aim is for the two GUIs to be interchangeable.)

For similar reasons, if you change the *name* of a concrete implementation of an interface (e.g. StudyPlanner is the concrete implementation of StudyPlannerInterface), or you change the class

description so that it no longer implements the interface, the tests will not run and you may lose a large number of marks.

You will, of course, have to change other details of the implementation classes: change the bodies of the inherited methods, and add new helper methods, create new classes that your implementation classes then refer to, etc. As long as the interfaces remain intact, this will not cause any problems. How you implement the purely internal details of the planner is up to you, as long as the interfaces are maintained.

## 3.3 Level 1

You should extend the provided implementation to add the following features. In all cases, any API methods you implement should be consistent with their descriptions in the corresponding Java interface.

*Note:* these requirements are numbered, but this is primarily for ease of reference (e.g. when asking questions about them), not as a suggested order of implementation. They may not be listed in the most efficient order in which to implement them. Similarly, if one requirement refers to "all other features" or "all state" then this includes all features and state introduced by later-numbered requirements.

1. Throw a StudyPlannerException if client code tries to generate a study plan when there are no topics
2. Throw a StudyPlannerException is client code tries to create two topics with the same name
3. Implement deleteTopic() from StudyPlannerInterface
4. Add some basic GUI support for managing topics:
    a. Provide input boxes and a button for creating a new topic (including the length of study required)
    b. Provide a button for deleting a topic selected in the topic list
    c. Whenever a topic is created or deleted, the GUI should be updated
5. Implement the version of generateStudyPlan() that takes a starting time and generates a study plan starting from that time
6. Generate a study plan using a simple algorithm to assign study – using a fixed block length of 60 minutes, study one block of each topic in turn, in the order in which the topics were added to the planner using addTopic(), then repeat until all topics have been fully studied
    a. Make sure that generateStudyPlan() can be called repeatedly, e.g. after changing break and block lengths, and each time produce a complete plan containing all needed study
7. Configure standard study block length and break length
    a. Default should be 60 minutes and 0 minutes, respectively
    b. If break length is set to zero, there should be no gap between the end of one study block and the start of the next
    c. If break length is set to more than zero, there should be exactly that much gap between the end of one study block and the start of the next
8. Change it so that if there's not enough study time left in a topic for a full block, schedule a shorter one

   a. Don't introduce any extra breaks when doing this – the standard break (if any is set) should start as soon as this short session ends, and be of normal length.
9. Add a box to the GUI that displays the current study plan as text, showing both study periods and any breaks:

```
12:00 History of Postmodernism in Art
13:00 (break)
13:15 Methods for Comparative Literature
14:15 (break)
…
```

You should create a new JUnit test set for the StudyPlannerInterface part of the API to test the changes you have made. This should consist of (only) your *three* most interesting and important tests (although I strongly advise that you create a large number of tests for your own purposes, you should *not* submit these as part of your assessment – only submit your best tests).

You must use version 4 of jUnit to organise your tests, and you should supply your tests as part of your submission, ready to run. **Tests that were performed by hand (not automated), or that do not have source code provided, will not count for any marks!**

Once you have the above features implemented, working and tested, you should proceed to Level 2.

## 3.4 Level 2

1. Implement start and end times for daily study, such that the planner will not plan any study before the daily start time or after the daily end time.
   a. Make the default 9AM to 5PM every day
   b. Implement the setDailyStartStudyTime() and setDailyEndStudyTime() methods to allow the start and end times to be set by client code
   c. Throw a StudyPlannerException on any call of those methods if the start time is later than the end time, or if the resulting study day is too short to fit in at least one block of the currently-set length (***not*** including the break after the block)
2. Change the study planning algorithm so that if there is not enough time left in the day for a whole study block, it creates a short study block in the same way as it does when there is not enough time left in a topic.
3. Enforce a minimum block length of 10 minutes
   a. If client code tries to set a standard block length below this, throw a StudyPlannerException
   b. When planning, if a block would be below minimum length because there is not enough time left in the current study day, start the block at the start of the next day.
   c. When planning, if a block would be below minimum length because there is not enough required study time left in the topic, make the block the minimum length anyway i.e. make the student study a few more minutes on the topic than would normally be required.
4. Support arbitrary events added to the timetable using addCalendarEvent()

    a. Events have a name, a type (exam, essay hand-in, or “other”), a start time, and duration.
        i. The CalendarEventInterface does not support setting or checking the type of an event, so you will need to handle this some other way.
    a. Change the study planning algorithm so that no study is scheduled during these events
        i. When enforcing minimum block size, events should be treated in the same way as the end of the study day i.e. if a study block would be below minimum length because it there is not enough time left before the next calendar event then it should be moved to after the event.
    b. If addCalendarEvent() is called for an event that overlaps an existing calendar event, throw a StudyPlannerException
    d. When a new calendar event is added, notify the GUI
    e. The minimum block size does *not* apply to calendar events – they can be of any length.
5. Add a feature to the GUI that lets the user add a calendar event at a specified date and time
6. Update the study plan listing in the GUI to include calendar events
7. Ensure that breaks after study are only scheduled in spaces that would be eligible for study time
    a. If an study block ends with an event or the end of a study day, don’t schedule a break at all
    b. Truncate any breaks that would overlap with the start of an event or the end of the study day (e.g. if there are only 7 minutes left at the end of the day and a break of duration 10m is due, *don’t* schedule a 3m break at the start of the next day – just forget about the extra 3 minutes)
8. Implement setTargetEvent() from TopicInterface in your topic class
    a. This is only valid for exam and essay calendar events, not for any other type – if client code tries to supply another type of event, it should throw a StudyPlannerException
    b. Implement getTarget() in TopicInterface for your topic class
9. Add a feature to your GUI that allows the user to set the target of a topic
10. Implement saveData() and loadData() for your StudyPlanner class.
    a. After returning from loadData() the externally-visible state of the planner should be exactly the same as it was prior to calling saveData()
        i. E.g. after calling loadData() to load the state from a file, client code should be able to call getStudyPlan() without first calling generateStudyPlan() and get exactly the same study plan that they would have got from getStudyPlan() had they called it before the state was saved.
        ii. saveData() might have been called on the same planner object some time earlier, or a different planner object (which may no longer exist).
    b. Use a file format of your own devising
    c. References to TopicInterface objects etc held by the client do *not* need to be valid after data is loaded from a file – you can assume that the client will call getTopics() etc again to get new references.

    d. If loadData() is given an empty or invalid stream, it should throw a StudyPlannerException
11. Add buttons the GUI that allowing saving and loading the planner's data – both should present a standard file selection dialog box.
12. Double-check that the planner ignores seconds and milliseconds in all cases – if e.g. a study planner start time is supplied that contains nonzero seconds or milliseconds, round it down to the nearest minute
13. Review the exception-handling behaviour of your GUI. In general, if the planner throws an exception the GUI should display an informative dialog box, wait for the user to click "ok", then return to normal operation.
14. Check over the JavaDoc in the various Interfaces I supplied, and make sure you've fully implemented the API described there.

As with Level 1, you should create a new JUnit test set for the StudyPlannerInterface part of the API to test the changes you have made. This should include (only) your *three* most interesting tests.

## 3.5 Level 3

1. Add a calendar display to the GUI that is similar to mainstream calendar products such as Google Calendar
    a. The calendar should clearly show the current study plan and all calendar events, with days arranged horizontally and time vertically (in the manner of Google Calendar's "week view")
    b. All blocks and events should be shown as boxes covering a period of time – the size of events in the calendar (and gaps between events) should be proportional to their duration.
        i. At least a week should be visible at one time
    c. Show the current daily start and end times as horizontal lines
    d. A scroll bar should be provided allowing the user to move freely back and forth through time
    e. For study blocks, the topic should be shown on each calendar item; for calendar events, the type and name of each event should be shown.
        i. Font size should be fixed, and appropriately chosen to be readable on screen
        ii. A consequence of the above is that the text on each block and event will sometimes be truncated to fit in the box. For a box where that has happened (only), when the user hovers the mouse pointer over it a tooltip should appear giving the full version of the text.
2. The user should be able to click on an empty space in the calendar display (as implemented for the item above) and create a new calendar event starting at that time.
    a. The user should be prompted for the event name, duration and type
3. When the user clicks on an exam or essay event, any study blocks for topics which target that event should be highlighted by changing the background colour of their boxes.
    a. When the user clicks again elsewhere in the calendar display, the blocks should revert to their normal colour
4. If the user right-clicks on a study block on the calendar, a popup menu should appear that offers a "Start Timer" item. If this is selected, a new window should appear with an

animated countdown timer. The timer should start at the duration of the study block and count down to zero. Once it reaches zero, a dialog box should appear which is modal to the main planner GUI window – the user must click on its single button ("Ok") in order to continue using the planner in any way.

a. While a timer is running, the study planner and its GUI should behave completely normally.
b. If the user starts a second timer while the first timer is running, the first timer should be cancelled and the second timer should be started in its place.
    i. The user should be prompted by a dialog box warning that this will happen, and giving them a choice of whether to continue.
    ii. As with the timer completion window, this dialog box should be modal to the main GUI window

As with the previous levels, you should create a new test set test the changes you have made. As the level 3 requirements are primarily concerned with the GUI, these tests should be descriptions of user actions rather than automated jUnit tests. This should include (only) your *three* most interesting tests, giving you a total of 9 tests to submit.

Warning – All of your automated JUnit tests from levels 1 and 2 should work correctly on the final version of your code.

## 3.6 File Formats

All files should be kept in the default working directory for the program. This is normally the Eclipse project directory.

There is no specific file format for saving the planner data in – you must define your own.

## 3.7 A Few Hints

- Unless otherwise specified, it should be okay for client to perform operations on an empty planner (one with no topics or events) unless they refer to a specific topic or event that doesn't exist. For example, asking for the list of topics when there are no topics in the planner should *not* cause a crash or throw an exception – your code should merely return an empty list.
- For this assessment, you should calculate all times at a resolution of minutes; in cases where you do acquire a time expressed with seconds or milliseconds (e.g. as part of the no-argument version of generateStudyPlan()) you should *round down* to the nearest minute.
- When defining your file format, you would be well-advised to make it easily human readable, as you may spend quite a lot of time starting at your saved data wondering why it won't load back properly!

# 4 The Testing Document

Your testing document should describe the 9 tests that you created and selected as you created your program. It should be no more than 2 A4 pages in an 11pt font, and the format of your test description should be as the example below.

| Test Name | Test Goal | Expected Outcome |
|---|---|---|
| PlannerLevel4Test.testDetectsTooMuchStudy() | Verify that if there is more study to do before an exam or essay, the system will warn the user. | If a topic with 400 hours of study is targeted at an exam tomorrow, and a plan generated, the warning list will contain a single “not enough study time” warning. |
| PlannerLevel4Test.testAllowsWritingTimeForEssays() | ... | ... |
| ... | ... | ... |

## 4.1 What is an 'interesting' test?

I have used the word 'interesting' earlier in this document, when talking about what tests you should perform and submit. By an “interesting test" I mean a test that:

- Tests something new that you've added – something from Level 1-3, not the barebones functionality of Level 0
- Tells you something about the program which wouldn't be obvious from a moderately careful reading of the program (i.e. it tests program logic of some complexity)
- Could reveal errors of several kinds in several places (i.e. it exercises many lines of code, ideally spread over several methods or indeed classes)

# 5 Marking

**Mark breakdown:**

| Requirement | Marks (out of 50) | Total (out of 50) |
|---|---|---|
| Level 0 functionality is still present, and interfaces are correct | Zero for success; penalty of up to 5 marks for failure | n/a |
| Level 1 requirements are correctly implemented | 10 | 10 |
| Level 2 requirements are correctly implemented | 15 | 25 |
| Level 3 requirements are correctly implemented | 15 | 40 |
| Suitable tests are provided and documented | 5 | 45 |
| The program is well-designed and robust (this includes object-oriented design, a clear programming style, and good handling of errors) | 5 | 50 |

# 6 Submission

The submission should be made electronically, using the department's electronic submission system.

1. You should submit a .**ZIP** file (not any other form of archive file) which contains:
    a. The Eclipse project files for your application, making sure that the source files (.java), all the class files (.class) are present
        - Include your exam number in the name of the zip file
    b. Your testing document, as a **PDF** or **Word** file
    c. A brief readme.txt that explains how to run the GUI version of the your code
2. Your programs should be already compiled and ready to run using JDK 1.7, and must not use any third-party libraries (other than JUnit)
3. **Add your examination number in the documentation comments of all source files, apart from those supplied by me that you have not changed in any way (i.e. the interfaces)**
4. **Do NOT write your name or user name or email in any part of the program or the testing document**

See http://www.cs.york.ac.uk/submit/ for details of electronic submission. In addition to the above, you must conform to the submission requirements set out on that web page.