Project assignment

Software Engineering Course

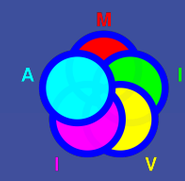
Master Degree in Computer Engineering

University of Salerno

2021/2022

Problem statement

- Develop an application implementing a scientific programmable calculator supporting complex numbers
- Starting point: the User Epics given at the end of this presentation



The task

- Groups of 4 students
- The groups define their user stories
  - Based on the provided User Epics
  - Teachers may add or change the user stories during the project
- The project will be performed using the Scrum process
- **TIME BOXING**: the program **does not need to be complete** at the end of the project!

Tools

- The groups must set up a Git repository for the project on GitHub
  - The repository must contain
    - Source code (obviously), including unit tests (based on JUnit)
    - A (short) document describing the software architecture
    - The product backlog
    - The sprint backlogs for each sprint
    - (At the end) A presentation of the project describing both the product and the process

Tools

- The team will use Trello (http://trello.com) as a platform for tracking the project activities (Sprint Task Board) and their completion
  - For each sprint, the tasks assigned to each member of the team and the time spent on them must be tracked

- Links:
  - https://blog.trello.com/how-to-scrum-and-trello-for-teams-at-work

The deadlines

- First delivery: Sun. November 21
  - Pre-game: Initial product backlog + 1st sprint planning and backlog + Description of software architecture (5 min. presentation per group)
- Second delivery: Sun. November 28
  - 1st Sprint Release + Sprint backlog + Updated prod. backlog + Project Burndown chart + Sprint review report + Sprint retrospective report
- Third delivery: Sun. December 5
  - 2nd Sprint Release + Sprint backlog + Updated prod. backlog + Project Burndown chart + Sprint review report + Sprint retrospective report
- Fourth delivery: Sun. December 12
  - 3rd Sprint Release + Sprint backlog + Updated prod. backlog + Project Burndown chart + Sprint review report + Sprint retrospective report + Final presentation (8 min. per group)

Classroom activity

- Two of the three weekly lectures will be used for short demonstrations of the software released on the previous week (during these demos you may be asked to show also the product/sprint backlogs, review/retrospective reports, burndown chart)

- A final demonstration/presentation will be given by each group in the week between Dec 13 and Dec 17

Artifacts: backlogs

- The product backlog and sprint backlogs should be kept or copied in a Word or Excel document for ease of inspection by the teachers
  - You may use a Google Docs/Sheets shared document, and have in your GitHub repo a file/document containing the links
- Don't delete completed items from the backlog; just move them to a different part of the doc

Artifacts: burndown chart

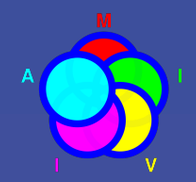
- You should make the burndown chart (or the "extended" burndown chart) for tracking the whole project (not the sprint burndown chart)

Artifacts: review report

- Keep it short (max 1 page)
- Essential information:
  - Story Points planned vs completed, Project Velocity
  - Short discussion of issues discovered implementing/examining the release (e.g. bugs, technical problems, technical debts)
  - Short discussion of issues with the Product Backlog (e.g. need to add/refine/modify some user stories)

Artifacts: retrospective report

- Keep it short (max 1 page)
- Follow the "starfish diagram"
  - Start / More of / Keep doing / Less of / Stop
- Keep it about the process (not the product)



Time organization

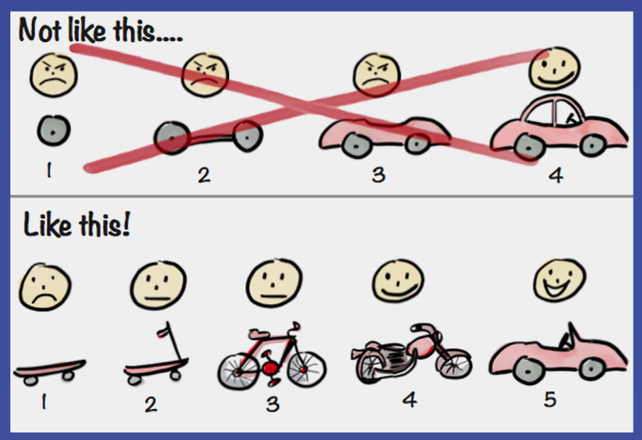
- Each member of the group is expected to work on the project for **10 hours** per week
- Better to have these hours divided between two or three days (agreed among the members of the team)

Time organization

- Pre-Game: the members of the team will work mostly together
  - Initial product backlog: the members of the team will fill the role of the Product Owner
    - Try to keep a shared vision of the product
  - Tools setup: GitHub repo, Trello, coding conventions, development platforms, external libraries (if needed)
  - Definition of Done
  - Design of the architecture of the program
  - First sprint planning (initial estimate of velocity, selected user stories)
  - If needed, get acquainted with the architecture and technology chosen (e.g. develop a simple "Hello world"-like program to check that everything is in place for compiling and running code following your architecture/technology/libraries etc.)

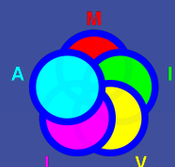
Time organization

- At the beginning of each sprint
  - If necessary, Product Backlog grooming: less than 1 hour, together
    - Try to address the issues discovered during the previous sprint review
  - Sprint planning: 1.5 hours, together
    - Choose user stories for current sprint
    - Define and assign tasks (requires some design)
    - When dividing a user story into tasks, ensure that there are "intermediate checkpoints" where, even if the story is not completed, the code compiles and passes its tests



Time organization

- At the beginning of each workday within the sprint
  - Daily meeting: 5 minutes, together
    - What you did last day, what you are going to do today, problems/obstacles you have encountered
  - Refine and discuss design decisions: 5-30 minutes, subsets of the team
    - Discuss decisions with other members whose work is going to be impacted
    - Agree on the interfaces between the different parts



Time organization

- ✦ At the end of each workday within the sprint
  - Take the time needed to integrate the code that is ready with the rest of the release (you can do this more often than daily)

Time organization

- At the end of each each sprint
  - Code review: 30 minutes, individually
    - Examine the code produced by other team members, to spot problems in design or coding (see the checklists later in this presentation)
    - Better to agree on who reviews what, so as to avoid duplicate work
    - Not necessary to have a 100% coverage, but try to have a fair sampling (for instance, ensure that work by a team member is reviewed by a different colleague at each sprint)
    - Problems found can be discussed in the Sprint Review

Time organization

- ✦ At the end of each each sprint
  - Sprint Review: 40 minutes, together
  - Sprint Retrospective: 20 minutes, together

Evaluation criteria

- The rating of each student will be based on the individual tasks performed by the student (as documented in each iteration) and on the overall evaluation of the project
  - Individual tasks: 50%
  - Overal project evaluation: 50%

- Each member of the group will be asked at the end to provide (confidentially) an estimate of the relative effort contributed by each other member
  - No "free riders"!

Evaluation criteria

- Appearance and usability: 10%
- Quality of the design: 30%
- Quality of the coding: 15%
- Quality of the tests: 15%
- Compliance with the Scrum process: 30%
- Effort: a multiplicative coefficient that scales all of the above
  - You are expected to work **10 hours/week**, and divide the tasks among the team members!

Design checklist

- Are the packages/classes designed to provide a set of highly related services (high cohesion)?
- Are unnecessary dependencies avoided (low coupling)?
- Are abstraction mechanisms used to improve reusability and/or break dependency chains?
- Are design patterns used when appropriate?
- Are duplications/repetitions in the design avoided? (Don't Repeat Yourself principle)

Coding checklist

- Are naming conventions followed consistently? Are names chosen so as to be easily understandable?
- Is code formatting (e.g. indentation) consistent and readable?
- Are "hard coded" constants/"magic" numbers in the code avoided?
- Is the code adequately commented? (Comments should describe the interfaces, and why you are doing things a certain way – not be redundant)
- Are duplications/repetitions in the code avoided? (Don't Repeat Yourself principle)
- Is the code structure readable? (e.g. split too complex operations into several methods)

Testing checklist

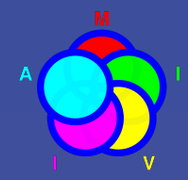
- Are the unit tests automated?
- Do the tests cover the public part of each class? (at least for business-logic classes)
- Do the test cases cover adequately the input space (e.g. including limit values, special conditions etc.)

User Epics

WARNING!

- The following User Epics are in order of priority (most important first)

- You are not expected to complete ALL the User Epics within the project duration
  - You must complete as much as you can WITHOUT compromising on the quality



Basic operation of the calculator

The operation of the calculator revolves around a stack data structure. The stack contains complex numbers (of course real numbers are supported as a special case of complex numbers). The user can either input a number (using the Cartesian notation, e.g. "7.2+4.9j"; the imaginary part can be omitted, writing "42" instead of "42+0j"), that is pushed onto the stack, or the name of an operation, that is executed taking its operands from the stack and pushing its result onto the stack. The user interface should show the top locations of the stack (at least 12 elements) and should have a text area for entering the user input. The calculator should support at least the following operations: "+" (addition), "-" (subtraction), "*" (multiplication), "/" (division), "sqrt" (square root), "+-" (invert sign).

For instance, if the user enters the numbers "5" and "9" and then the operations "-" and "sqrt", the stack will contain just the number "0+2j" ("-" takes "5" and "9" from the stack and pushes their difference "-4", then "sqrt" takes "-4" from the stack and pushes its square root "0+2j").

Stack manipulation commands

The calculator includes the following operations for manipulating the stack: "clear" that removes all the elements; "drop" that removes the last element (i.e. the top); "dup" that pushes a copy of the last element; "swap" that exchanges the last two elements; "over" that pushes a copy of the second last element.

Examples:

STACK BEFORE (bottom to top)	OPERATION	STACK AFTER (bottom to top)
$Z_1\ Z_2\ Z_3 \ldots Z_{k-2}\ Z_{k-1}\ Z_k$	clear	(empty)
$Z_1\ Z_2\ Z_3 \ldots Z_{k-2}\ Z_{k-1}\ Z_k$	drop	$Z_1\ Z_2\ Z_3 \ldots Z_{k-2}\ Z_{k-1}$
$Z_1\ Z_2\ Z_3 \ldots Z_{k-2}\ Z_{k-1}\ Z_k$	dup	$Z_1\ Z_2\ Z_3 \ldots Z_{k-2}\ Z_{k-1}\ Z_k\ Z_k$
$Z_1\ Z_2\ Z_3 \ldots Z_{k-2}\ Z_{k-1}\ Z_k$	swap	$Z_1\ Z_2\ Z_3 \ldots Z_{k-2}\ Z_k\ Z_{k-1}$
$Z_1\ Z_2\ Z_3 \ldots Z_{k-2}\ Z_{k-1}\ Z_k$	over	$Z_1\ Z_2\ Z_3 \ldots Z_{k-2}\ Z_{k-1}\ Z_k\ Z_{k-1}$

Variables

The calculator supports 26 variables, named with the letters from "a" to "z". For each variable "x", the operation ">x" takes the top element from the stack and saves it into the variable "x". The operation "<x" pushes the value of the variable "x" onto the stack. The operation "+x" takes the top element from the stack and adds it to the value of the variable "x" (storing the result of the addition into "x"). The operation "-x" takes the top element from the stack and subtracts it from the value of the variable "x" (storing the result of the subtraction into "x").

User-defined operations

The user can define a new operation by specifying a name and a sequence of operations (including the push of numbers). When the user-defined operation is invoked, all the operations in the sequence are executed in order. The definition of a user-defined operation may contain other user-defined operations. The user can delete a user-defined operation. The user can modify the definition of a user-defined operation. The user can save to a file the existing user-defined operations, and reload them from a file, even in a different usage session.

Example: the user can define an operation "hypotenuse", that takes from the stack the values of the two catheti and pushes the value of the hypothenuse. The definition of this operation could be the sequence:

```
dup * swap dup * + sqrt
```

Save/Restore variables

The calculator includes the following operations that operate on the variables: "save", which saves a copy of all the 26 variables on a "variable stack" (distinct from the stack used for the operands by the operations); "restore", which restores for all variables the last values that were saved on the "variable stack" (removing them from that stack). It is possible to perform several times the "save" operation, preserving several sets of variable values. This mechanism can be used to temporarily modify a variable within a user-defined operation, without making this modification visible outside of the execution of the user-defined operation.

For example, the "hypothenuse" operation (that takes the two catheti from the stack) could be defined with the following sequence :

```
save >b >a <a <a * <b <b * + sqrt restore
```

Another example: the following sequence can be used to define a "solve2degree" operation for solving second degree equations; at the beginning the stack must contain the 3 coefficients of the equation a, b and c (pushed in this order); at the end the stack will contain the two solutions:

```
save >c >b >a <b <b * 4 <a <c * * - sqrt >d
<b +- <d – 2 <a * /    <b +- <d + 2 <a * / restore
```

Transcendental functions

The calculator includes the following operations, taking complex numbers as operands (from the stack) and producing (possibly) complex results (pushed onto the stack): "mod" (modulus/magnitude); "arg" (argument/phase); "pow" (power); "exp" (exponential); "log" (natural logarithm); "sin" (sine); "cos" (cosine); "tan" (tangent); "asin" (arc sine); "acos" (arc cosine); "atan" (arc tangent).