S TEAM

Ana Antunes – *61025*

Joana Wang – *60225*

Nelson Matos – *60483*

Pedro Estróia – *60691*

Renato Viola – *60665*

gantt project

Additional Functionalities & Use Cases & Code Metrics Identification

**ADDITIONAL FUNCTIONALITIES**

**USER STORIES**

**Functionality 1: Task filter by duration, completion or date.**

As a user, I want to see the tasks filtered by a specific attribute, so that I may view the information I find relevant more easily.

**Functionality 2: Enabled editing multiple tasks’ general attributes simultaneously.**

As a user, I want to be able to modify the general properties of multiple tasks at once, so that I may save my time.

**USE CASES**

Diagram

Description automatically generated

**Use case: FilterTasks**

ID: 1

Description: The user filters the selected tasks by determining criteria (duration, progress, begin date, or end date), given the value.

Primary actors: User

Secondary actors: None

Preconditions:

1. The tasks are selected.

Main flow:

1. The use case starts when the user selects “Filter”.

2. The system asks the user for the filter criteria and its value.

3. The system applies color changes to tasks that satisfy the criteria.

Postconditions:

1. The selected tasks that satisfy the conditions were colored with a selected color.

Alternative flow: Cancel

Diagram

Description automatically generated

**Use case: SetTaskProperty**

ID: 2

Description: The user sets the properties of the selected tasks on the system.

Primary actors: User

Secondary actors: None

Preconditions: The selected tasks exist in the system.

Main flow:

1. The use case starts when the user selects “Task Properties”.

2. The system asks the user for the new properties.

3. The user enters the requested properties.

4. The system updates the task's properties.

Postconditions:

1. Task’s properties were updated.

Alternative flows: Cancel



ANA’S USE CASES

**HELP MENU**

**Use case: ViewSummary**

ID: 26

Description: The System shows a little summary of the app.

Primary actors: User

Secondary actors: None

Main flow:

1. The use case starts when the user selects *“Summary”.*

2. The system shows a resume of the main information of the app.

**Use case: ViewLicense**

ID: 27

Description: System shows the license of the app.

Primary actors: User

Secondary actors: None

Main flow:

1. The use case starts when the user selects *“License”.*

2. The system shows the license info of the app.

3. The user selects *“Ok”.*

Alternative flows:

3. The user selects *“Cancel”.*

**Use case: ViewLibraries**

ID: 28

Description: The System shows some useful libraries of the app.

Primary actors: User

Secondary actors: None

Main flow:

1. The use case starts when the user selects *“Libraries”*.

2. The system shows the libraries info of the app.

3. The user selects *“Ok”.*

Alternative flows:

3. The user selects *“Cancel”.*

**Use case: ViewLog**

ID: 29

Description: The System shows the Log info about the last modifications of the user

Primary actors: User

Secondary actors: None

Main flow:

1. The use case starts when the user selects *“View log”.*

2. The System shows log details, such as user directories, user classes, the date of the last modification of such class.

3. The user closes the text box.

Alternative flow:

None

**Use case: RecoverProject**

ID: 30

Description: To recover some useful data from project last versions.

Primary actors: User

Secondary actors: None

Main flow:

1. The use case starts when the user selects “Recover…”.

2. The System asks the user if he wants to restore.

3. The user confirm the recovery.

4. The System updates the project version.

Alternative flow 1:

2. The user jumps the version.

3. Go back to 2 of the Main flow.

Alternative flow 2:

(When there is only one last version of the Project, and the system updates automatically)

Point 2. and 3. of the “*main flow”* are jumped.

Alternative flow 3:

3. The user selects *“Cancel”.*



JOANA’S USE CASES

**TASK MENU**

**Use case: AddTask**

ID: 19

Description: The user adds a new task in the system task list.

Primary actors: User

Secondary actors: None

Preconditions: None

Main flow:

1. The use case starts when the user selects “New Task”.

2. The system asks the task manager to create and add a new task in the task list.

Postconditions:

1. A new task was added in the system.

**Use case: SetTaskProperty**

ID: 20

Description: The user sets the properties of the selected task on the system.

Primary actors: User

Secondary actors: None

Preconditions: The selected task exists in the system.

Main flow:

1. The use case starts when the user selects “Task Properties”.

2. The system asks the user for the new properties.

3. The user enters the requested properties.

4. The system asks the task mutator to update the task properties.

Postconditions:

1. Task properties was updated.

Alternative flows:

Cancel

**Use case: DeleteTask**

ID: 21

Description: The user deletes the selected tasks in the system.

Primary actors: User

Secondary actors: None

Preconditions: The selected tasks exist in the system.

Main flow:

1. The use case starts when the user selects “Remove Task”.

2. The system asks the task manager to delete selected tasks in the system.

Postconditions:

1. Selected tasks were deleted from the system tasks list.

Alternative flows:

No

Cancel

Icon

Description automatically generated

NELSON’S USE CASES

**PROJECT MENU**

**Use case: Load File**

Description: The User selects a file to load.

Primary actors: User.

Secondary actors: None.

Pre-conditions: None.

Main flow:

1. The use case starts when the user is presented with a window to choose a file.
2. The User chooses the desired file.

Post-conditions:

1. The selected file is loaded.

Alternative flows:

Cancel.

**Use case: Store File**

Description: The User selects a file to be stored.

Primary actors: User.

Secondary actors: None.

Pre-conditions: None.

Main flow:

1. The use case starts when the user is asked to specify a directory where the file most go.
2. The User chooses the desired directory.

Post-conditions:

1. The selected file is stored.

Alternative flows:

Cancel.

**Use case: Properties**

ID: 1

Description: User sets properties of the project.

Primary actors: User.

Secondary actors: None.

Pre-conditions: None.

Main flow:

1. The use case starts when the user selects the “Properties…” option available in the project menu.
2. The system prompts a new window with some general settings regarding the project.
3. The user chooses the new properties.

Post-conditions:

1. Properties of the project are changed.

Alternative flows:

Cancel.

**Use case: Create Project**

ID: 2

Description: The User creates a new project.

Primary actors: User.

Secondary actors: None.

Pre-conditions: None.

Main flow:

1. The use case starts when the user selects the “New…” option available in the project menu.
2. The system asks if the user wants to save previously created project.

3. The system displays a prompt with some fields for the user to fill.

4. User fills the required fields.

Post-conditions:

1. A new project file is created.

Alternative flows:

Cancel.

**Use case: Open Project**

ID: 3

Description: The User opens a project.

Primary actors: User.

Secondary actors: None.

Pre-conditions: Project most exist in the system.

Main flow:

1. The use case starts when the user selects the “Open…” option available in the project menu.
2. If there is a project opened
   1. The system shows a warning regarding the project and asks if the User wants to save it.

Include (Load File)

Post-conditions:

1. A new project file is opened.

Alternative flows:

Cancel.

**Use case: Recent Projects**

ID: 4

Description: The User selects a project to be loaded.

Primary actors: User.

Secondary actors: None.

Pre-conditions: None.

Main flow:

1. The use case starts when the user selects the “Recent Projects…” option available in the project menu.
2. The system shows a vertical box with labels containing the names of latest projects saved.
3. The user selects one of the saved projects.

Post-conditions:

1. The selected project is loaded.

Alternative flow: None.

**Use case: Save Project**

ID: 5

Description: The User saves the project.

Primary actors: User.

Secondary actors: None.

Pre-conditions: None.

Main flow:

1. The use case starts when the user selects the “Save…” option available in the project menu.

**Extension point**: Project does not exist in File System.

1. The project state is saved.

Post-conditions:

1. The project is saved/stored.

Alternative flow: None.

**Use case: Save Project As**

ID: 6

Description: The User saves the project with a certain name.

Primary actors: User.

Secondary actors: None.

Pre-conditions: None.

Main flow:

1. The use case starts when the user is asked to give the project a name.
2. The system displays a window containing all the available folders to store the file.
3. The user selects the desired folder.

Include (Store File)

Post-conditions:

1. The file is stored in the File System.

Alternative flow:

Cancel.

**Use case: Import file**

ID: 7

Description: The User opens an existent file from local folders.

Primary actors: User.

Secondary actors: None.

Pre-conditions: The file most exist in the system.

Main flow:

1. The use case starts when the user selects the “Import…” option available in the project menu.
2. The system displays a menu with different kinds of import possibilities.
3. The user selects the desired options.

Include (Load File)

Post-conditions:

1. The file is imported.

Alternative flow: None.

**Use case: Export file**

ID: 8

Description: The User exports a project file to a folder.

Primary actors: User.

Secondary actors: None.

Pre-conditions: None.

Main flow:

1. The use case starts when the user selects the “Export…” option available in the project menu.
2. The system displays a window with export choices.
3. The user selects the wanted options.

Include (Store File)

Post-conditions:

1. The file is exported.

Alternative flows:

Cancel.

**Use case: Print**

ID: 9

Description: The User prints the project.

Primary actors: User

Secondary actors: None.

Pre-conditions: None.

Main flow:

1. The use case starts when the user selects the “Print…” option available in the project menu.
2. The system displays a window with print settings.
3. The User picks the desired options.

Include (Store File)

Post-conditions:

1. The project is printed.

Alternative flows:

Cancel.

**Use case: Print preview**

ID: 10

Description: The User visualizes how the expected output image should look like.

Primary actors: User.

Secondary actors: None.

Pre-conditions: None.

Main flow:

1. The use case starts when the User selects the “Print Preview…” option available in the project menu.
2. The system opens a new window displaying some settings (preview formats, …) and a preview image of the current project.

Post-conditions: None.

Alternative flows: None.



PEDRO’S USE CASES

**RESOURCE MENU**

**Use case: AddResource**

ID: 22

Description: The user adds a new human resource in the system resource list.

Primary actors: User

Secondary actors: None.

Preconditions: None.

Main flow:

1. The use case starts when the user selects “Add Resource”.

2. The system asks the user to enter the new resource properties, such as name, phone number, email, role, and resource payment rate.

3. The user enters the new resource properties.

4. The system creates and adds the new resource in the human resource list.

Postconditions:

1. The new resource was added in the human resource list.

Alternative flow:

Cancel

**Use case: SetResourceProperty**

ID: 23

Description: The user sets a selected resource properties in the system.

Primary actors: User

Secondary actors: None

Preconditions:

1. A resource is selected.

Main flow:

1. The use case starts when the user selects “Resource Properties”.

2. The system asks the user for resource new properties.

3. The user enters the resource new properties.

4. The system updates the resource properties.

Postconditions:

1. The new resource was updated.

Alternative flow:

Cancel

**Use case: DeleteResource**

ID: 24

Description: The user deletes the selected resources on the system.

Primary actors: User

Secondary actors: None

Preconditions:

1. There is at least one selected resource.

Main flow:

1. The use case starts when the user selects “Delete Resource”.

2. The system deletes the selected resources from the resource list.

Postconditions:

1. The resources were removed from resource list.

**Use case: SendEmail**

ID: 25

Description: The user sends email notification to a selected resource.

Primary actors: User

Secondary actors: None

Preconditions:

1. There is a selected resource.

Main flow:

1. The use case starts when the user selects “Send email message”.

2. The system opens a window to choose the email application or platform.

3. The user writes and sends the email.

Postconditions:

1. Email was sent.



RENATO’S USE CASES

**EDIT MENU**

**Use Case: Undo**

ID: 11

Description: The user undoes his latest action.

Main actor: User

Secondary actor: None

Pre-conditions: There exist actions in system.

Main flow:

1. The use case starts when the user selects ‘Undo’.
2. The system restores the state it was in before the user’s latest action.

Post conditions: System is in the state it was in before the user’s latest action.

Alternative flows: None

**Use Case: Redo**

ID: 12

Description: The user redoes his latest undone action.

Main actor: User

Secondary actor: None

Pre-conditions: There exist actions in the system that have been previously undone.

Main flow:

1. The use case starts when the user selects ‘Redo’.
2. The system restores the state it was in before the user’s latest action was undone.

Post conditions: System is in the state it was in before the user’s latest action was undone.

Alternative flows: None

**Use Case: Refresh**

ID: 13

Description: The user refreshes the application.

Main actor: User

Secondary actor: None

Pre-conditions: None

Main flow:

1. The use case starts when the user selects ‘Refresh’.
2. The system refreshes its user interface.

Post conditions: System’s user interface is updated.

Alternative flows: None

**Use Case: Search**

ID: 14

Description: The user searches for an item.

Main actor: User

Secondary actor: None

Pre-conditions: None

Main flow:

1. The use case starts when the user selects ‘Search’.
2. The user inputs the name of what it is searching for.
3. The system displays suggestions for what the user is looking for.
4. The user chooses an item suggestion.
   1. Task selected:
      1. The system displays the Gantt chart.
      2. The system marks the task as selected.
   2. Resource selected:
      1. The system displays the Resource chart.
      2. The system marks the resource as selected.

Post conditions: Searched item is selected.

Alternative flows: None

**Use Case: Cut**

ID: 15

Description: The user cuts one or more items.

Main actor: User

Secondary actor: None

Pre-conditions: One or more items must be selected.

Main flow:

1. The use case starts when the user selects ‘Cut’.
2. The system stores the selected items for posterior use.
3. The system removes the selected items from the list of items being displayed.

Post conditions: Selected items are stored and hidden.

Alternative flows: None

**Use Case: Copy**

ID: 16

Description: The user copies one or more items.

Main actor: User

Secondary actor: None

Pre-conditions: One or more items must be selected.

Main flow:

1. The use case starts when the user selects ‘Copy’.
2. The system stores the selected items for posterior use.

Post conditions: Selected items are stored.

Alternative flows: None

**Use Case: Paste**

ID: 17

Description: The user pastes one or more items.

Main actor: User

Secondary actor: None

Pre-conditions: One or more items must have been cut or copied previously.

Main flow:

1. The use case starts when the user selects ‘Paste’.
2. The system adds the stored items to the list of items on display.

Post conditions: Stored items are displayed.

Alternative flows: None

**Use Case: Settings**

ID: 18

Description: The user accesses the application’s settings.

Main actor: User

Secondary actor: None

Pre-conditions: None

Main flow:

1. The use case starts when the user selects ‘Settings’.
2. The system displays a window with various tabs.
3. The user tampers with the settings.
4. The user confirms his changes.

Post conditions: Settings are changed.

Alternative flows: Cancel



ANA’S CODE METRICS

**Martin Packaging Metrics**

**Abstractness(A)**

Abstractness in this context, is the number of abstract classes in the package to the number of all classes.

This metric help developers to measure the degree of abstraction of the package and it has a range of [0,1]. A=0 indicates a completely concrete package, A=1 indicates a completely abstract package.

Analyzing “**A**” metrics with excel Box Plots:

As we can see on the Abstractness Box Plot(blue) in excel, we have a low average of abstraction (0,25), and we only have one bigger package abstraction (1.0), as we can see on the top point of the Abstractness Box Plot, that is “net.sourceforge.ganttproject.gui.options.model” package, with 3 abstract classes/interfaces.

Also, we have many packages without any abstract class/interface, such as “net.sourceforge.ganttproject.chart.item”, to avoid the complexity, and turn the code more efficient, “CharItem” class should be an abstract class.

We can conclude that this project uses a low level of abstraction, the developers should increase to a near number 1 number as possible (as the example given upper), by hiding some implementation details that are not useful right now and simplify code structure.

**Afferent Couplings(Ca)**

The Afferent Couplings packaging metric represents the number of classes of other packages that depend to this package.

Analyzing “**Ca**” metrics with excel Box Plots:

The higher value is on “net.sourceforge.ganttproject.task”(2559), how we can see on the top point of “Afferent Couplings Box Plot”(yellow).

This package has many interfaces/abstract classes, so it’s normal to have a substantial quantity of “Ca”, but “2559”? “Yes, that’s the problem”. So, if we want to replace something on a class, we have breaking changes repeatedly to lots of code, in this case, to all the classes from other packages (1-2559 except abstract classes/interfaces) that use the replaced class from other package.

**Efferent Couplings(Ce)**

The Efferent Couplings packaging metric represents the number of classes of this package that depend to other package classes.

Commonly, excessive values are inconvenient because they can cause problems with the maintenance and code development.

Analyzing “**Ce**” metrics with excel Box Plots:

The classes in “net.sourceforge.ganttproject” have a very high “Ce” value(how we can see on the top point of “Efferent Coupling Box Plot” (yellow)). This is because the classes are not well structured, this package has more than 50 classes/interfaces, instead, those should be divided and reorganized into different packages (e.g. “Resource..” classes inside “resource” package, already created, and/or delete unused classes like “ResourceTreeImageGenerator”).

There are many other classes in the same situation, and the average (202) is very high, that causes instability of a package. As “Ca”, if we want to change one class, this will produce a numerous external classes modification to the package.

**Distance from the Main Sequence(D)**

This metric is used to measure the balance between stability and abstractness (The closer D is to 0, better (means that “A” and “I” are near)).

Analyzing “**D**” metrics with Bar Chart excel:

There are some packages, with stability and no abstraction, and the opposite, those sides represent the higher values of “D” (maximum unbalance) (e.g. “net.sourceforge.ganttproject.filter” package has less abstraction, than stability and “net.sourceforge.ganttproject.gui.options” package has less stability than abstaction).

We conclude, the packages of the project are with low stability, and low abstraction, as we can observe on the excel Bar Chart(D).

**Instability(I)**

This metric is used to measure the relative susceptibility of class to changes. According to the definition instability is the ration of outgoing dependencies to all package dependencies and it accepts value from 0 to 1. Better values are near to 1, that means this package can be easily changed.

So, having many classes of other packages depending on this package, and not depending on almost any package class, give stability to the package, so high “Ce” numbers and low “Ca” values is the ideal pattern.

Analyzing “**I**” metrics with excel Box Plots:

As we can see on “Instability Box Plot” (gray), the average of stability is 0,50, so that means the project have many package classes depending on others, not s sustainable structure.

An example of that is “biz.ganttproject.core.option” package(I=0,0), a not structured package. This happens by the number of classes/interfaces inside, so because of that, many classes from other packages are depending on this package, and that induces code instability.

**CODE SMELLS RELATION to 1 phase identification:**

As we are talking about packaging metrics, sometimes there isn’t a direct relation between class Code Smells, but we can still analyze some potential correlation between them.

* Data Class – Unnecessary data classes in a package can make “Ca” higher and “Ce” even higher, and consequently Instability(I) problems because this “data classes”, that we don’t need (Ce), are depending on another ones, and other ones in this one (Ca).
* Duplicated Code – Having less abstraction(A) increases the Duplicated Code in package classes and the same inversely.
* Message Chains – The “message chain” classes of a package are dependent on navigation along all the class structure, so it can increase “Ce” and “I”.



JOANA’S CODE METRICS

**Complexity metrics**

The complexity metric, also known as cyclomatic complexity is used to indicate the program's complexity. This may be applied to several parts of a program, such as individual methods, classes, packages, modules, or even the whole project. The complexity also greatly impacts the cost of maintaining a program. And then, the main goal of this metric is to modularize a more maintainable and testable program.

The metric is based on a quantitative measure of the number of linearly independent paths through a program's [source code](https://en.wikipedia.org/wiki/Source_code). It’s computed using the control-flow graph of the program. The nodes of the graph correspond to basic blocks of program commands. The directed edge connected between two nodes indicates that the second one will be immediately executed after the first command. Another way to measure the program complexity is by counting the line of codes (LOC).

This metric evaluates the class complexity on three parameters:

- **Average operation complexity** (OCavg).

- **Maximum operation complexity** (OCmax).

- **Weighted Methods per Class** (WMC): this measures the sum of complexities of methods defined in a class.

As we can see on the collected class metrics diagram, there´s an extreme WCM value in the class “net.sourceforge.ganttproject.task.TaskManagerImpl”. The indicated value is 173.00 when the class average WCM value is 13.61. That´s a trouble spot, showing that there´s a class with many high-complexity methods with too many lines of code. It, therefore, represents the complexity of a class as a whole and then causes a “large class” code smell. The class will take on more and more responsibilities, making it get larger and larger.

Another trouble spot is on class “GanttOptions.GanttXMLOptionsParser”, which has an extreme OCmax and OCavg value compared with both average values. That means the class has the most complex method in the program codebase. This method is responsible for more tasks than it should be, turning the method very large and complex, which can cause code smells such as long methods. It makes the code hard to maintain and hard to read.

**Bibliografia:**

<https://en.wikipedia.org/wiki/Cyclomatic_complexity>

<https://www.geeksforgeeks.org/complexity-metrics/>

<https://thevaluable.dev/complexity-metrics-software/>



NELSON’S CODE SMELLS

**LINES OF CODE**

Purpose of usage:

The chosen metric is a common basis when it comes to estimate either the size or the amount of time/effort that was required to produce a certain software project. This metric can give us an overall idea of the complexity of some of the classes/methods presented in the project. Usually there are big classes/methods which are commonly undesirable as they often times lead to misconceptions and are quite laborious to read. Despite some of the classes/methods being large by default, there are others that can be refactored or even reduced (removing non-functional lines of code). With this metric we can constantly measure the length of classes/methods and carefully plan countermeasures when certain threshold is met.

Looking at the *class metrics table* we can see that the average of the LOC far exceeds the average of the CLOC and JLOC which indicates that there was a moderate amount of documentation produced. After further analyzing the previously mentioned tablewe can state the number of CLOC and JLOC presented in some of the classes are unbalanced, in example, the “*GanttProject”* class is a very dense and complex class that should’ve been more documented. We have to make sure that there is a balance between its documentation and its intricacies.

Notes: LOC -> Lines of Code, JLOC -> Javadoc Lines of Code and CLOC -> Commented Lines of Code.

As stated before, this metric gives us a rough idea of the complexity and code size of the project. Let’s illustrate with some examples/cases:

* If, for instance, we take 2 projects, let’s call them A and B, respectively. The first one being about 10 thousand LOC and the second around 15 thousand LOC, we aren’t able to say much about their relative complexity. On other hand, if we compare project A to a project C that has 300 thousand LOC, in general, the second one is more complex.
* Another thing that we need to consider is the level/experience of the software developer, a more experienced developer tends to write less LOC compared to a less experienced one. This can lead to different values shown in the *class metrics* table.

Taking in consideration the 1st Phase of the Gantt Project we can recall a couple of code smells that are vehemently related to this metric.

Both of the following code smells, “Long Method” and “Reminder” contribute evenly to this metric as they, respectively, increase the number of LOC and CLOC.



PEDRO’S CODE METRICS

**Class dependency metric**

This metric shows the number of dependencies related to classes, as it is possible to know information about how many classes/interfaces depend on a certain class, how many modules is each class dependent on, and some more information.

With this metric, we can see several values:

* Number of cyclic dependencies (***Cyclic***) – number of modules where the dependency goes the two ways regarding each class, e.g. if we are evaluating a class A, the *Cyclic* calculates the number of classes that depend on A and that A is dependent on, simultaneously.
* Number of dependencies (***Dcy***) – number of classes which each class directly depends on.
* Number of transitive dependencies (***Dcy\****) – number of classes which each class directly or indirectly depends on.
* Number of dependents (***Dpt***) – number of classes that directly depend on each class.
* Number of transitive dependents (***Dpt\****) – number of classes that directly or indirectly depend on each class.
* Number of package dependencies (***PDcy***) – number of packages which each class directly depends on.
* Number of dependent packages (***PDpt***) – number of packages that directly depend on each class.

Generally, a high number of class dependencies means more coupling, which is the degree each module depends on the others. It is used to measure the strength of associations between different modules, so the higher the coupling is the more complex the system tends to be. This can lead to several problems, making the project much more difficult to debug and to eventually extend it, by adding more functionalities, for instance. If the calculated dependency numbers of a class is too high in comparison to the average, that possibly means that the coupling is also elevated, which can translate into an unnecessary increase of complexity in the overall project.

Looking at the retrieved statistics from the IntelliJ plugin, there are a few values that seem to be above the average.

The ***Cyclic*** of each class fluctuates between two values, 0 and 515, and the second value seems a bit above the normal range, which can mean that there are too many cycles. This two-way dependency can be an issue in a way that it makes the code harder to understand and extend, as well as debug.

The ***Dcy*** had only one value that was a little extreme, in the GanttProject class (246), being this a possible troublespot. This can be explained by the fact that this is the main frame of the project, and it’s necessary that this class communicates with several other classes, so it’s understandable that the GanttProject has more dependencies than the remaining classes.

***Dcy\**** and ***Dpt\**** has a value that is shown many times, 867 and 560, respectively. This value is not particularly low, which can be explained by the fact that many classes communicate with many classes indirectly, being that the difference between transitive and normal dependency. This is a possible troublespot, as the level of coupling is too high, and that may difficult the understanding of the code.

The ***Dpt*** is significantly above the others in three classes: GanttLanguage(116) , GPAction(97) and GPLogger(81). GanttLanguage is the class responsible for the language of the app, dealing with some formats of dates, which are abundantly used throughout the project, mainly regarding tasks, being an explanation for this high value. GPAction is the class responsible for the actions (such as *ok* and *cancel*) associated with events. Lots of other classes take care of the consequences of these actions, so this can be a possible reason. *GPLogger* is responsible for the ‘Log’ in the ‘Help’ tab, which shows errors, warnings, etc., so it makes sense to have many classes dependent on it, since the things shown in the log are done in several different classes.

The ***PDcy*** is relatively high in GanttProject possibly because, as we said in ***Dcy***, this is the main frame of the project, so it depends on a lot of classes, from different packages, as it takes care of lots of different information.

The ***PDpt*** has high values in GanttLanguage(31) and GPLogger(27). The explanation is similar to the one described in the ***Dpt*** paragraph.

There are a couple of code smells referred in the first phase that may impact these statistics: the data class, which is a class with information only (getters), may cause more unnecessary coupling; message chain, in which many classes depend on each other to call many methods from different classes successively, increasing the number of dependencies.

In conclusion, many of these values are extremely big, and they correspond mostly to complex classes that take care of important information used in most of the project, such as GanttProject. However, some values are enormous in almost every class, and by that we can conclude the project has an insanely big level of coupling, resulting in many code understanding and extending problems, which were strongly felt during the code study part of the project asssignment.

**Bibliography:**

SHARMA, Shweta et al./ International Journal of Computer Science & Engineering Technology (IJCSET)

https://www.techtarget.com/searchapparchitecture/tip/The-basics-of-software-coupling-metrics-and-concepts

SHAIKH, Mohsin, International Journal of Advanced Computer Science and Applications, Vol. 8, No. 10, 2017



RENATO’S CODE METRICS

**CHIDAMBER AND KEMERER METRICS**

The Chidamber and Kemerer suite of metrics was designed to measure the characteristics of object-oriented systems, by quantifying attributes deemed valuable in analyzing the difficulty and complexity of implementating changes during code maintenance, but also the effects that could derive from such alterations. Every metric is calculated for each class.

The metrics which comprise the studied suite are as follows:

* **Coupling Between Objects (*CBO*)** – Number of classes the particular class is coupled with. A high degree of coupling might prove to be detrimental, as it harms modularity and lessens reuse. With more coupling, other parts of the design become increasingly sensitive to change, thus becoming more prone to faults and difficulting maintenance.
* **Depth of Inheritance Tree (*DIT*)** – Maximum inheritance path from the class to its root. Deep inheritance trees indicate higher design complexity, as the deeper a class is in the hierarchy, the more complex it is likely to be, since it inherits more methods and variables. On a positive note, due to method inheritance, deeper trees promote reuse. However, a higher DIT has been found to increase the density of bugs and decrease quality, as deep class hierarchies are complex to develop.
* **Lack of Cohesion in Methods (*LCOM*)** – Measures how well the methods of a class are related to each other. It’s important to note that this metric has received a fair deal of criticism. In spite of its shortcomings, a higher LCOM generally indicates inappropriate design and high complexity, as well as a higher likelihood of errors, suggesting the class should be split into smaller classes.
* **Number of children (*NOC*)** - Number of immediate sub-classes derived from a class. A higher NOC may indicate: a high reuse of the base class; that the base class may require further testing; an improper abstraction of the parent class; misuse of sub-classing. Possibly due to high reuse, high NOC has been found to indicate fewer faults.
* **Response For a Class (*RFC*)** - Number of different methods that can potentially be executed by the class in response to a message. In other words, a function of the number of methods in the class and the number of different methods called within the methods of the class. A high RFC generally indicates more bugs and less quality, as classes with a high RFC are more complex and tougher to understand. Testing and debugging might prove quite troublesome as well.
* **Weighted Methods per Class (*WMC*)** – Number of methods defined in the class. The more methods a class has, the more likely it is to be more application specific, and thus less reusable. Derived classes also tend to be more impacted by changes in the base class, since they inherit a great number of methods. High WMC generally indicates more faults and could mean that the class could be restructured into smaller classes.

POTENTIAL TROUBLE IN THE CODEBASE

Upon analyzing the values of this set of metrics in the GanttProject codebase, there are several things to note. But the most interesting cases to look at are the ones with the most extreme values.

Some classes do present metrics with interesting values, potentially indicating something worth looking into.

But none were remotely as noteworthy as the *GanttProject* class, with its metrics taking values of {**CBO:** 118; **DIT:** 7; **LCOM:** 9; **NOC:** 0; **RFC:** 365; **WMC:** 156}.

As we can see, this class has very high CBO, RFC and WMC, which indicates that it is very complex, lengthy and heavily dependent on other classes.

This doesn’t immediately equate to faulty code though, as it presents a fairly low LCOM value, suggesting that its code is rather coherent. The high value for DIT does imply high complexity, but it also indicates reusability.

It’s also relevant to note that this class is used as sort of system class, which is inherently bound to make it somewhat more complex and larger than other classes.

At a first glance, it wouldn’t seem hard to infer that this class is very likely to be problematic. But with further context, we really can’t be so sure.

RELATED CODE SMELLS

The values of these metrics are able to reflect the existence of certain smells within the codebase.

2 of these smells were identified by the S-Team during phase 1 of the project. These are as follows:

* Message Chain – The code has long message chains, where you call to get an object back, where you call another object back, call an object on it again, and so on… This sort of programming practice affects the value of the CBO and RFC metrics directly.
* Long parameters list – A method who receives many arguments can be difficult to use and understand. The more arguments a method receives, the more methods it invokes, thus increasing the value of the CBO and RFC metrics.

BIBLIOGRAPHY:

[*https://www.aivosto.com/project/help/pm-oo-ck.html*](https://www.aivosto.com/project/help/pm-oo-ck.html)

[*https://www.hindawi.com/journals/isrn/2013/276105/*](https://www.hindawi.com/journals/isrn/2013/276105/)

[*https://www.aivosto.com/project/help/pm-oo-cohesion.html#LCOM1*](https://www.aivosto.com/project/help/pm-oo-cohesion.html#LCOM1)