MODE Assessment Report

Exam No: Y3838392

1 Metamodel

The following figure 1.1 presents the design of the metamodel implemented in my solution, which I have described using Emfatic. In this section I will explain the ideas behind the design and evaluate them.

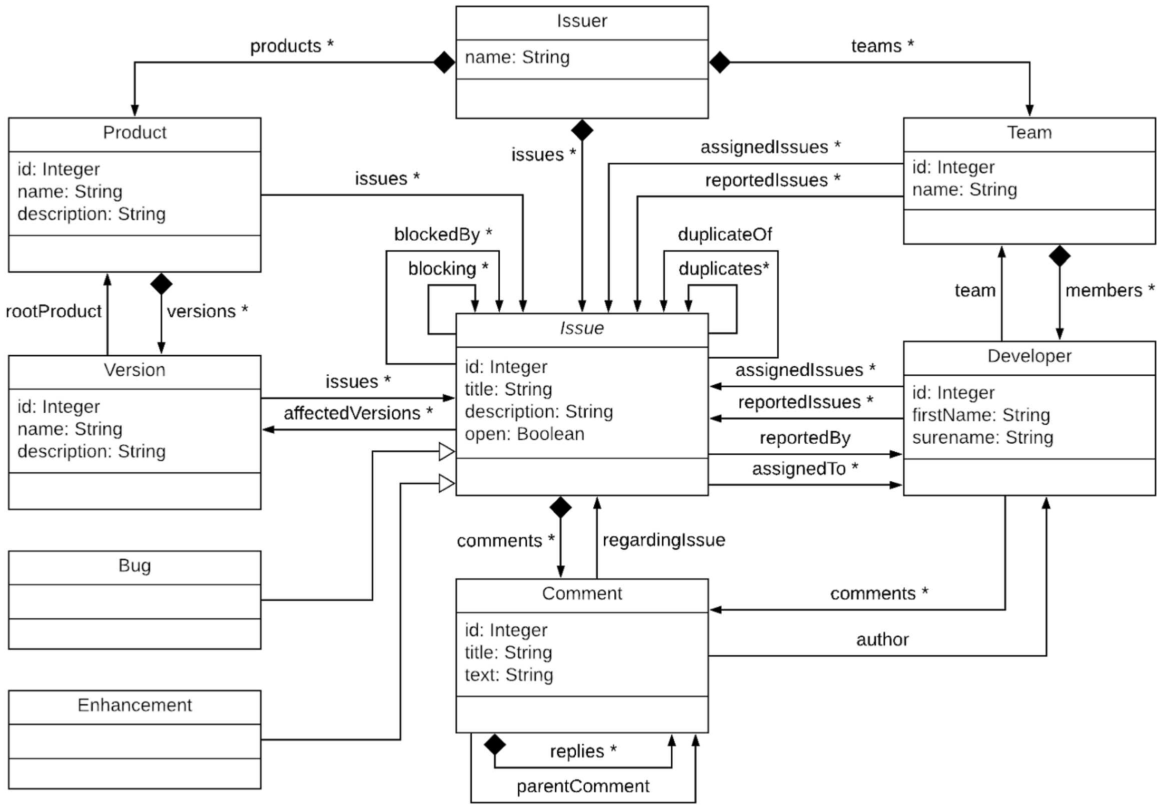
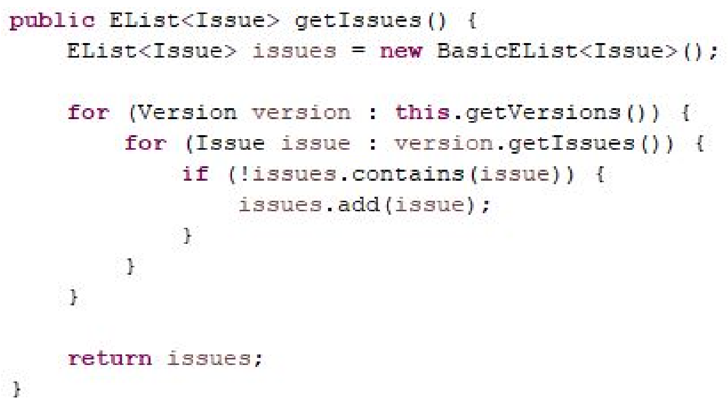


Figure 1.1 Metamodel design.

The root element of the model is the Issuer class which can be given a name for personalization purposes. For example it can contain the name of the company using the issuer tool, which can then be used in templates. The Issuer stores three lists: one containing a collection of the products, one for developer teams and the last one for actual issues. All elements of any other type than Issuer have an ID which can be used to refer to it directly. There is no need for the Issuer to have an identifier as there can only be one root element.

The Product class objects can only be children of the Issuer element. A product can be given a name and a description. It also stores a list of versions as specified in the given scenario, where products and versions are distinct concepts.

The Version class objects can only be children of a Product element. A version can be given a name and a description. It also holds a reference to the main product and references to all the issues that are affecting the version. The *issues* list of the parent Product element is a readonly derived reference, which used later in the graphical editor, listing all the unique issues regarding any version of the product. Its Java implementation is shown in figure 1.2. It cannot be a concatenation of all the lists, as it is assumed that in issue can affect multiple versions, therefore a check must be made every time before adding an issue to the list in order to avoid multiple references to the same Issue.



```
public EList<Issue> getIssues() {
    EList<Issue> issues = new BasicEList<Issue>();

    for (Version version : this.getVersions()) {
        for (Issue issue : version.getIssues()) {
            if (!issues.contains(issue)) {
                issues.add(issue);
            }
        }
    }

    return issues;
}
```

Figure 1.2 Implementation of the *issue* reference of Product elements.

The Team class represents a group of people that are associated with it. Although the scenario did not specify the need for having multiple teams, this solution doesn't take away anything from the plain solution, as it allows having a singular team in the model and also giving the opportunity to extend the model by adding more teams. A team can be given a name and stores it's developers in a list called members. The *reportedIssues* and *assignedIssues* references are derived in an inherital manner, similar to the *issues* reference in the Product class.

The Developer class represents a person that can deal with the issues. Each developer can be given a first name and a surname. A developer also holds a list of referenced and assigned issues as well as references to the comments that the developer has produced.

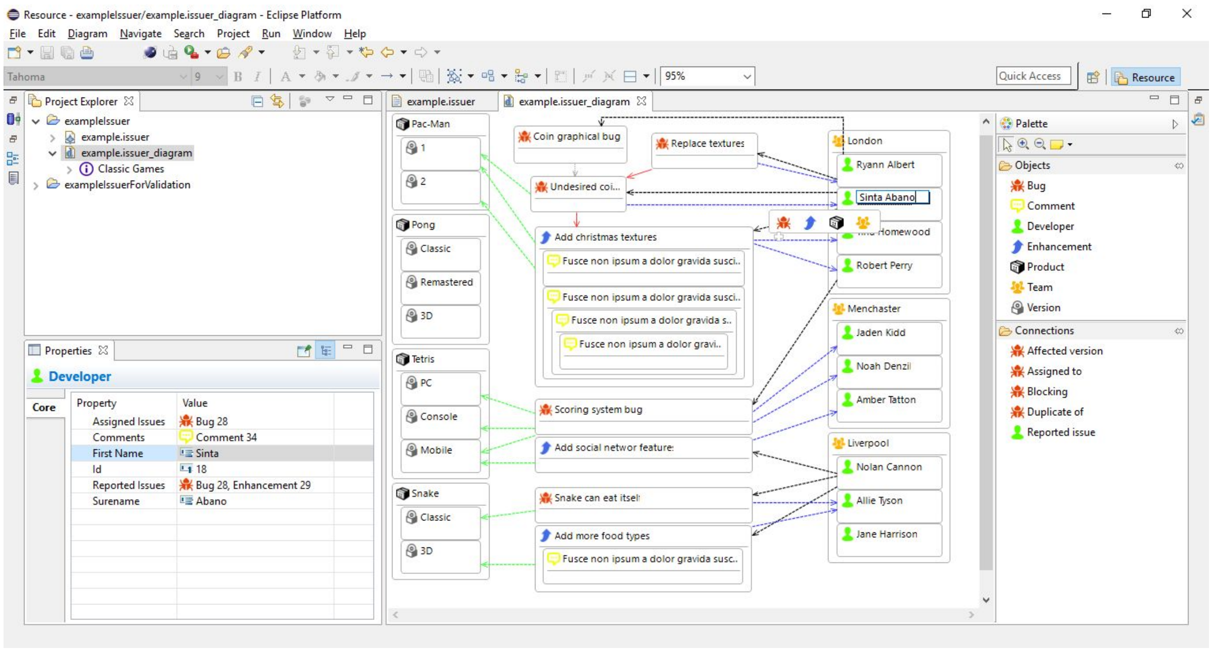
The objects of type Issue can only be children of the Issuer, which is the root component of the proposed metamodel. The other approach would be to implement issues as children of a product or a version, but this would make the requirement for an issue to be affecting multiple versions more complicated and also cause confusion as an issue could also be expected to be a child element of a Developer object. Therefore I have decided to keep the Issue class independent of any of these and supplement it with a range of references. An Issue object can be given a title, a description and a boolean value for the open state. The references to other types cover the list of affected versions, the developer that the issue was reported by, the list of developers that the issue is assigned to. The remaining references on the diagram (*blocking*, *blockedBy*, duplicates and *duplicateOf*) are pointing to other objects of type Issue. Issue objects also store the list of comments regarding it.

Bug and Enhancement classes inherit from the Issue class which is abstract in my solution. The other approach would be to use an enumeration and store the type of the issue as one of its values, but for the sake of clear concrete syntax I have chosen to implement them as separate subclasses.

An object of type object can be given a title and text content. We can also specify the author of the comment by passing a reference to a Developer object. The other approach would be to store comments in a separate container stored in the root component, and referencing it to the issues and developers, but I have come to a conclusion that a comment will never need to exist if there is no issue it regards to. A comment can also contain other comments in the replies list allowing for nested comments. Every comment has a reference to the parent object which can either be a *regardingIssue* or a *parentComment* reference. The regarding issue of a nested comment will have to be derived if needed.

2 Concrete syntax and editor

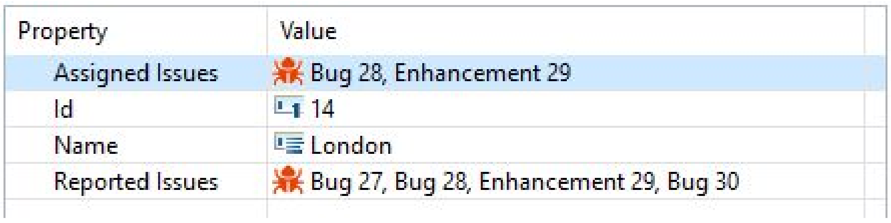
In my solution I have declared the concrete syntax using GMF rules[1], which can then be used to produce models using the graphical editor generated by EuGENia. The other possible approach instead of using the diagram editor was to work on the tree editor, which is also viable in this scenario. The reason for using the graphical diagram editor is the ease of referencing entities, as it features lines that link the desired object automatically, rather than choosing an item from a long drop down list which is the only option in the three editor. Also it can be used to visualize the dependencies unlike the tree editor, which only prints a list of the model's items. The only downside of this solution is the fact that the connections can become tangled over time, but the dropdown is still available in the properties window if needed. Figure 2.1 presents my final editor based on the defined abstract and concrete syntax, featuring an example issuer model diagram, which features all of the styling adjustments that I have implemented.



Property	Value
Assigned Issues	Bug 28
Comments	Comment 34
First Name	Sinta
Id	18
Reported Issues	Bug 28, Enhancement 29
Surename	Abano

Figure 2.1 Graphical model editor for Issuer models with an example model diagram.

The Product class is represented by a rectangular node, where its label is the product title. It can contain a list of Versions, which are represented in a similar manner. The list is represented as a GMF compartment, so that all Version nodes are the same size, making the model more consistent. Same principles apply to the Team and Developer classes, with the only differences being that the label of a team is denoted by its *name*, and the label of a developer is a concatenation of the *firstName* and *surname* properties. As mentioned in the previous section of the report, the Product and Team objects can derive the issues that are related to a particular product or team by checking the references of their individual versions/members. The figure 2.2 shows us the properties the team London from the example diagram shown in figure 2.1. As we can see, the *reportedIssues* and *assignedIssues* indeed show us a list of issues involving the four team members without any duplicates, even though one of the enhancements is assigned to both Tina and Robert.



Property	Value
Assigned Issues	Bug 28, Enhancement 29
Id	14
Name	London
Reported Issues	Bug 27, Bug 28, Enhancement 29, Bug 30

Figure 2.2 Properties of London object of type Team.

Bugs and Enhancements are represented as nodes with the title property as their labels. Thanks to the fact that they are two distinct types, they can be represented by two different icons. Both can contain a list of Comment entities inside a compartment.

Comments can contain list of other comment objects as well, which are ideally representing the replies. Comments can be nested, allowing for more complex discussions to be created.

In order to make the editor more intuitive for the users, I have manipulated the styling of the elements in various ways. First of all, I have added custom icons for all of the possible node entities, making it easier to destinguish its type, as the icons are semantically related to the type of object represented by the node. The icons were generated using the icondb.com[2] website, allowing me to choose the colours of the icons.

I have also considered changing the colours of the borders for each node type, but this made the models look less professional, so I went back to the previous design and decided to alter the default colours and styling of the links instead, where links represent the references. In order to cut off the redundant options in the editor I am only allowing for one of the references to be created using a line where there are two opposite references, for example *reportedIssues* for a Developer can be generated by linking two nodes in the diagram, but the *reportedBy* reference for an Issue object can only be generated by setting it in the property window. I have also purposely omitted the option to generate the *author* / *comments* references by using links as this would greatly increase the complexity of generated diagrams, hence reducing their readability. This leaves us with only five possible links that the user can put on the diagram:

- Affected version (green, dash style, originating from type issues, pointing at versions)
- Assigned to (blue, dash style, originating from issues, pointing at developers)
- Blocking (red, solid style, originating from issues, pointing at other issues)
- Duplicate of (grey, dash style, originating from issues, pointing at other issues)
- Reported issues (black, dash style, originating from developers, pointing at issues).

Changes performed in the editor affect both the diagram model and the actual skeleton model, which is based on the EMF file representing the issuer. This model can then be used for further model to model or model to text transformations.

3 Model Validation

I have implemented the validation using the Epsilon Validation Language. Unfortunately I did not manage to integrate my own validation rules inside the generated graphical editor, so I have settled for a simpler solution in the end. The validation can be done in another instance of Eclipse by using the EVL validation run configuration (Figure 3.1), which is given a model produced in the diagram editor as an input. For my validation rules I have assumed that all entities are given a value for all attributes.

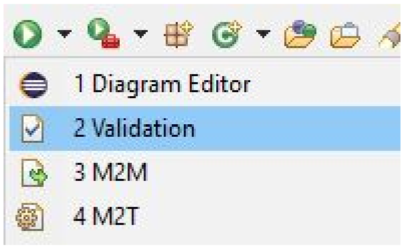


Figure 3.1 Run configurations used in the solution.

Every constraint has a message to inform the user about the discovered error and a proposed automatic fix if it is applicable. The messages usually affect issues, so in order to make their printing clearer in the code, I have implemented methods which print the most important properties of an issue and a list of issues. The following list is a documentation of all validation rules applied in my solution that cover the constraints given in the assessment scenario:

- uniqueId (Issue context)- this rule ensures that all the issues in a given model have unique *Id* values, by gathering all issues into one collection (*modelName!Issue.all()*) and selecting the issues that have the same id value as a particular issue. If the selection is not empty, that means the rule has been violated. I did not implement any quick fixes as it is hard to predict which issue needs to be changed. Message upon violation: “IssueError: issue Id must be unique. *issueString* COLLIDES WITH *issuesWithSameIdListString*”.
- noSelfDuplicate (Issue context)- this constraint looks if there exists a reference to itself in the duplicates list and offers the user to use “Remove the issue from its duplicates.” fix, which effectively removes the unwanted reference pointing to itself. Message upon violation: “IssueError: can not be a duplicate of itself. *issueString*”.
- noShortDescription (Issue context)- checks the description string length of each issue and prints out the following message if it is less than 10: “IssueError: description must be 10 characters or longer. *issueString*”.
- allAffectedVersionsHaveSameRootProduct (Issue context)- since issues can affect multiple versions, we must ensure that they originate from the same product, which is what this constraint is doing. If the any of the affected versions points to a different root product than the first version on the list, the user is presented with the following message: “IssueError: all versions affected by an issue must be of the same product. *issueString*”.
- noOpenBlockersForClosedIssues (Issue context)- this rule ensures that all of the *blockedBy* issue references of a closed issues are closed i.e. their *open* value is equal to *false.* The fix offered in my solution (“Close blocking issues which are open.”) goes through all issues

collected in the *openBlockers* list and sets the open value to *false*. Message upon violation: "IssueError: closed issues can not be blocked by an open issue. *issueString*".

- noBlockingCycles (Issue context)- to check if there exist any direct or indirect blockings, the issue runs the *isBlocking(rootIssue)* method, passing itself as the *rootIssue* parameter. This method is recursive and it runs itself on every issue reverenced in the *blocking* list with the same parameter, returning *false* if the *blocking* list is empty and *true* if the *rootIssue* exists in it. The results of the individual issues running the method are then changed to a single boolean value using the or operation, returning the appropriate result for the initial issue. As the result the following message will be printed for each of the issues participating in a blocking cycle: "IssueError: blocking cycle involving *issueString*".
- noBlockingBugObjects (Enhancement context)- this rule ensures that none of the issues blocked by an object of type Enhancement are of type Bug. The rule selects any bugs referenced by an enhancement and stores them in the *blockedBugs* sequence. It the sequence is not empty it prints out the following message: "IssueEnhancmentError: enhancements can not block bugs. issueString IS BLOCKING BUGS *blockedBugsListString*".

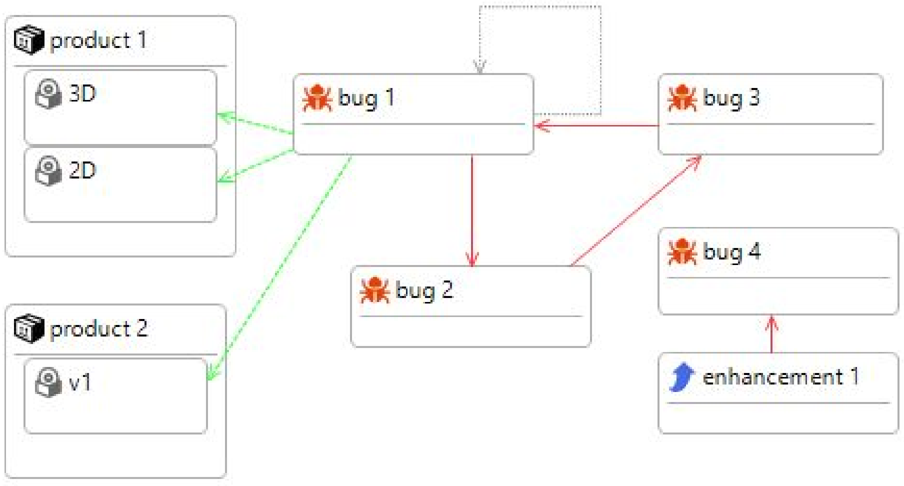
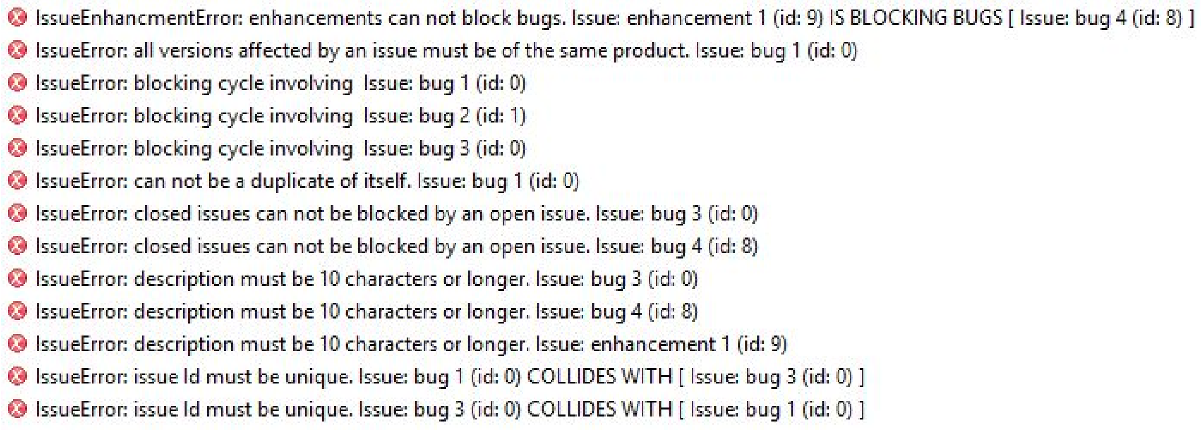


Figure 3.2 Example of an invalid model.



```
IssueEnhancmentError: enhancements can not block bugs. Issue: enhancement 1 (id: 9) IS BLOCKING BUGS [ Issue: bug 4 (id: 8) ]
IssueError: all versions affected by an issue must be of the same product. Issue: bug 1 (id: 0)
IssueError: blocking cycle involving  Issue: bug 1 (id: 0)
IssueError: blocking cycle involving  Issue: bug 2 (id: 1)
IssueError: blocking cycle involving  Issue: bug 3 (id: 0)
IssueError: can not be a duplicate of itself. Issue: bug 1 (id: 0)
IssueError: closed issues can not be blocked by an open issue. Issue: bug 3 (id: 0)
IssueError: closed issues can not be blocked by an open issue. Issue: bug 4 (id: 8)
IssueError: description must be 10 characters or longer. Issue: bug 3 (id: 0)
IssueError: description must be 10 characters or longer. Issue: bug 4 (id: 8)
IssueError: description must be 10 characters or longer. Issue: enhancement 1 (id: 9)
IssueError: issue Id must be unique. Issue: bug 1 (id: 0) COLLIDES WITH [ Issue: bug 3 (id: 0) ]
IssueError: issue Id must be unique. Issue: bug 3 (id: 0) COLLIDES WITH [ Issue: bug 1 (id: 0) ]
```

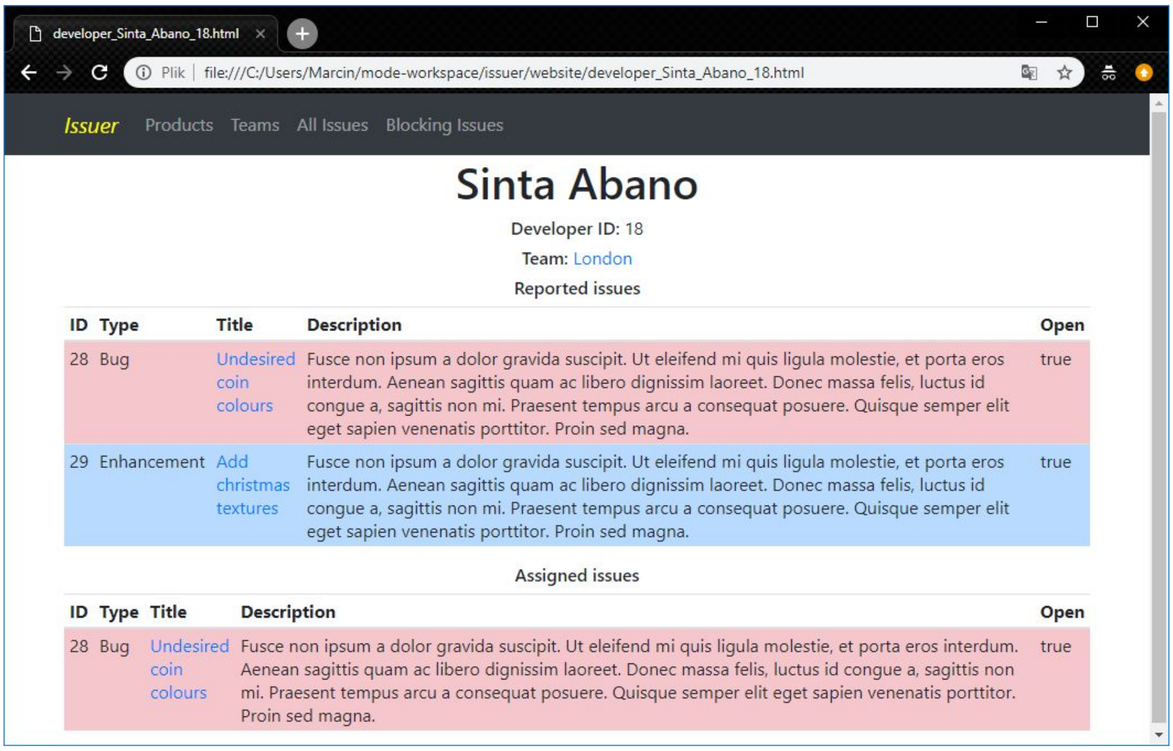
Figure 3.3 Errors discovered in the example model by the EVL program.

The same error messages can be seen when we run the EVL program and set the input model to the example issuer model called *'exampleForValidation.issuer'* stored in the diagram editor folder of my submission.

4 Model-to-text transformation

In order to generate a hyperlinked website based on a model, I have written an EGX program together with a set of EGL templates. The templates can be combined together by using imports, allowing me to maintain a similar design throughout the whole website. Every template contains references to online sources that are necessary to use Bootstrap[3]. Bootstrap is a free, open-source front-end CSS framework providing my solution with multiple classes that can be used to manipulate the styling and topology of the HTML elements more easily. The other possible approach would be to propose my own CSS stylesheet, which would be more specific to my solution, but since the structure of the website is more important than the design for this assessment, I have decided to use a complete framework.

Another common element included in every single page is a navigation bar imported from a navbar.egl file, which allows the user to easily reach the most important pages of the website. The navigation bar and the common styling features such as the fonts can be seen on figure 4.1, which is a screenshot of an example developer page.


developer_Sinta_Abano_18.html
Plik
file:///C:/Users/Marcin/mode-workspace/issuer/website/developer_Sinta_Abano_18.html
Issuer
Products
Teams
All Issues
Blocking Issues

Sinta Abano

Developer ID: 18

Team: London

Reported issues

ID	Type	Title	Description	Open
28	Bug	Undesired coin colours	Fusce non ipsum a dolor gravida suscipit. Ut eleifend mi quis ligula molestie, et porta eros interdum. Aenean sagittis quam ac libero dignissim laoreet. Donec massa felis, luctus id congue a, sagittis non mi. Praesent tempus arcu a consequat posuere. Quisque semper elit eget sapien venenatis porttitor. Proin sed magna.	true
29	Enhancement	Add christmas textures	Fusce non ipsum a dolor gravida suscipit. Ut eleifend mi quis ligula molestie, et porta eros interdum. Aenean sagittis quam ac libero dignissim laoreet. Donec massa felis, luctus id congue a, sagittis non mi. Praesent tempus arcu a consequat posuere. Quisque semper elit eget sapien venenatis porttitor. Proin sed magna.	true

Assigned issues

ID	Type	Title	Description	Open
28	Bug	Undesired coin colours	Fusce non ipsum a dolor gravida suscipit. Ut eleifend mi quis ligula molestie, et porta eros interdum. Aenean sagittis quam ac libero dignissim laoreet. Donec massa felis, luctus id congue a, sagittis non mi. Praesent tempus arcu a consequat posuere. Quisque semper elit eget sapien venenatis porttitor. Proin sed magna.	true

Figure 4.1 An example developer page produced by EOL.

The user can access any of the generated pages by using hyperlinks. Every text element which represents a name or a title of another entity form the the model is actually a hyperlink leading to a more detailed page about that object. For example on figure 4.1 the users can click any of the options from the navigation bar, but they can also reach the team page of the developer and an issue page regarding any of the issues related to the developer by clicking on appropriate hyperlinks. They are generated by operations shared across all other templates which return a valid HTML hyperlink to an object of interest. I have written a *getHyperlink()* method for the necessary types from my metamodel returning different attributes for the inner text of the hyperlink depending on the type. Another type of method imported from the *common-operations.egl* template is *getBootstrapClass()* allowing to apply different colors to elements representing issues depending on their type and value of the open flag, which makes the page more readable.

In order to keep the URLs of the generated pages organised, every URL regarding an entity from a model includes the objects type followed by its name/title and ID. This way every URL is unique and informs the user about the topic of the page in a short form.

My solution covers all the requirements given in the assessment scenario with use of a set of 9 different templates listed below:

- products.egl (transforms Issuer)- produces a web page containing a listing of products from a given model as a table. The table presents the product ID, the name of the product and the number of product versions together with a list of their names as hyperlinks.
- teams.egl (transforms Issuer)- produces a web page containing a listing of teams from the given model as a table. The table presents the team ID, the name of the team and the number of members together with a list of their names as hyperlinks.
- all-issues.egl (transforms Issuer)- displays two tables listing key information about all the issues from the input model, one showing open issues and the other showing closed. The rows have different colors depending on the issue type and the open flag value.
- blocking-issues.egl (transforms Issuer)- lists issues which are blocking at least one issues in a similar manner to all-issues.egl, with an additional attribute "total issues blocked", which is the sum of direct and indirect blockings derived for each issue.
- team.egl (transforms Team)- produces a web page displaying all attributes of a team together with tables for issues reported by and assigned to any of the team members.
- developer.egl (transforms Developer)- displays all attributes of a developer together with tables for reported and assigned issues. For the comments I have decided to only provide the number of comments rather than their actual content and related issue. This allows to quickly evaluate the developers contribution without overflowing the page with unnecessary details.
- product.egl (transforms Product)- produces a web page displaying all attributes of a product object together with tables for open and closed issues affecting any of the product versions.
- version.egl (transforms Version)- displays open and closed issues affecting the version object in two separate tables.
- issue.egl (transforms Bug and Enhancement)- displays all the attributes of an issue object. Some attributes are hidden if empty for example if the *blocking* list is empty, the "blocking" and "total issues blocked" labels will not be displayed at all. The issue page also displays all the regarding issue in together with their replies in a nested manner. The font size is reduced slightly for each comment generation in the tree.

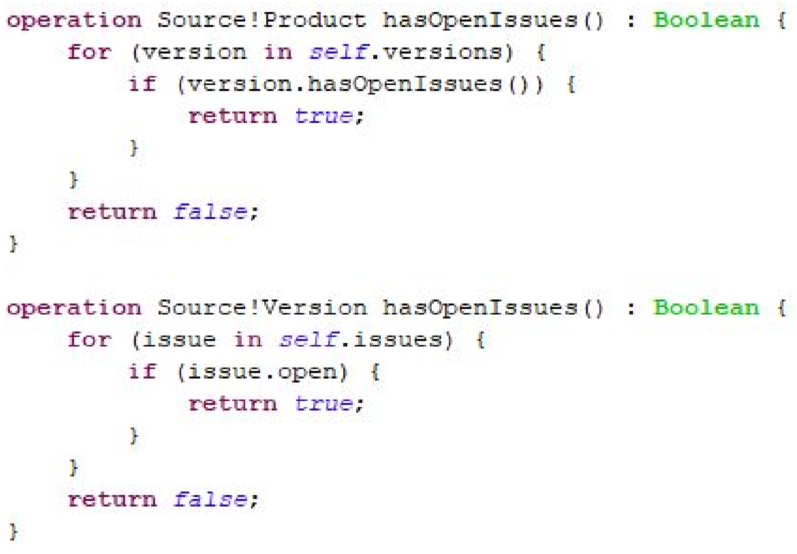
For testing purposes I have used the model proposed in section 2 of the report as the input model. The generated website is located in the "website" folder inside the "issuer" project provided in my submission.

5 Model-to-model transformation

In order to manipulate a model and construct another one based on it, I have used the Epsilon Transformation Language. In the given scenario there was no need to change the types of the entities from the source model as it is using the same metamodel as the target model. It required the ETL program to select only the data that is needed for the simplified version of the model excluding all elements that are unrelated to open issues.

The ETL code for the solution has only one general transformation rule and a set of functions that are called inside it to keep the code readable. The rule is called *OnlyOpenIssues* and it transforms an Issuer object (which is the root of the metamodel) from the source into a new instance of Issuer stored in the target model. The name attribute is carried over to the new instance, but the lists containing issues, products and teams are filtered out so that only the elements related to open issues are left.

For the actual issue list it is easy to just select the open issues and copy their comments, but for teams and products we need to look at the references of their child elements to see if they have anything to do with the issues we are interested in. To make the checking easier I have written a function *hasOpenIssues()* for each of the following types: Product, Version, Team and Developer. They return true if the element of interest is related to an open issue in anyway. The functions can be called from within another as presented on the figure 5.1 where *version.hasOneOpenIssue()* is called in the function concerning the Product type.



```
operation Source!Product hasOpenIssues() : Boolean {
    for (version in self.versions) {
        if (version.hasOpenIssues()) {
            return true;
        }
    }
    return false;
}

operation Source!Version hasOpenIssues() : Boolean {
    for (issue in self.issues) {
        if (issue.open) {
            return true;
        }
    }
    return false;
}
```

Figure 5.1

Once the selection of of the parent elements is done, then the versions and developers are filtered out using the same principals. This results in some products and teams being copied to the generated model but their number of versions/developers might be reduced. For example if one of two two versions is affected by an open issue and the other is not, the team will be present in the simplified model, but it will only have one version. If none of the versions is related to an open issue, then the product element will be omitted as well.

To present the result of my solution, I have included a simplified model which has been generated using ETL given the model from figure 2.1 as the source. I have also generated a website of the target model which is stored in the *‘websiteAfterM2M’* folder.

Bibliography

[1] EuGENia GMF tutorial https://www.eclipse.org/epsilon/doc/articles/eugenia-gmf-tutorial
[2] IconDB https://www.iconsdb.com
[3] Bootstrap https://getbootstrap.com/