OPEN INDIVIDUAL ASSESSMENT

Author:
Y3507677

Module:
Model-Driven Engineering
(MODE)

June 13, 2019

Contents

I. Metamodel

While deriving a formal architecture from the specification, several options of how to implement the issue structure were investigated. The specification presents two issue types that must be implemented: bug, and enhancement request. The only behavioural difference between the two is that a "bug can block enhancement requests, but not the other way round" and while in many cases one may decide to use inheritance to implement this functionality, it is not believed that it is best suited in this instance. Thoughts on inheritance vs enums for the implementation:

- An issue would only be able to be marked as one type. It would not be possible to mark an issue as both a bug and something such as "good for newcomers" or "wontfix"; labels available on GitHub.
- Using inheritance increases the complexity of changing an issue's associated type. For example, if someone was to create an issue and mistakenly set the type to being a bug and they then want to change it to an enhancement request, then a new issue object must be created and all associated fields copied over.
- The issue type list is not going to change frequently and this best suits an enum to differentiate the issue type. It is still possible to add additional functionality to particular issue types through either the enum itself or though inheritance for that particular type. While the specification does only require two issue types to be implemented, it is important to think ahead when architecting a solution and knowing that enums allow for additional functionality to be implemented at a later date is important.
- An enum pattern allows for easy switching of types in the editor. For instance, once the editor has been generated it becomes very simple to implement the image retrieval for an Issue, as the issue type literal can be used to fetch the name of the image on the file system. If more issue types are required then it is as simple as adding the literal to the enum and adding the image to the file system.

Following the above considerations, the decision was made to use an enum strategy to differentiate between issue types in the application. enums will also be used to mark an issue as open or closed, or a version as in progress or complete. The reasoning being that in the future there may be an additional requirement to mark a version as on hold and if the application is built around Boolean logic then this minor change would become more complex. enum literals can also be converted in to friendly text descriptors for users to read. The implemented enums are as followers:

- **Version Status:** denotes the current status of a version. Values: IN_PROGRESS, COMPLETE. Possible additions could be: ON_HOLD, or CANCELLED.
- **Issue Status:** denotes the current status of an issue. Values: OPEN, CLOSED. Possible additions could be: CANCELLED, or REMOVED.
- **Issue Type:** allows for an issue to be marked as one of many types. The values assigned here have been inspired by GitHub's issue labels. Values: ENHANCEMENT, BUG, WONT_FIX, HELP_REQUIRED, DUPLICATE.

The root instance type of the metamodel is an `IssueTracker` and amongst a project name it contains teams, products, issues, and members. While an issue tracker can have no issues, it must contain at least one product, team and member. Products can contain multiple versions and an issue can be marked as being associated with multiple versions. This users a degree of flexibility in how they use the software. EVL constraints will be required to ensure that users only mark an issue with versions that correspond to the same product type.

Where there is a degree of ambiguity or flexibility in the specification is in the definition of a team. One could regard a team as being a group of members in a particular location or a number of members that hold the same role. As such, members can be associated with multiple teams and this allows for a member to be associated with both a 'developer' team and a 'remote workers' team, for example.

As a result of this, this makes the GMF diagram more cumbersome as a GMF compartment cannot be added to the team. To combat this, more effort has been made in the model-to-text transformation to ensure that users can easily view team and member information.

The specification mandates that the identifier of each issue must be unique, however, identifiers can also be added to other objects in the metamodel. As such, each class extends from an abstract class named `Identifiable`. This strategy was adopted from Hibernate ORM and would allow for persistence to be implemented easily at a later date, if required. While considerations were made towards each individual class having its own `id` property, this strategy easily allows for identifiers to be globally unique and for creation of them to be managed by one class. Teams are composed of members, which have references to issues that have been assigned to them, issues they've created and comments that they have created. These references make use of Emfatic opposites, which proved to be exceptionally useful in creating bidirectional links between classes. The member class also contains a `transient derived` field for the full name of the member. This is derived whenever the first name or last name changes and is useful for when information about the member needs to be presented.

Products contain one or more versions within them and for the editor they are displayed using a GMF compartment. This results in them being displayed clearly to the user. Decisions were made as to whether or not an issue should have a reference to a product or a version and it was settled upon for an issue to have a version reference. What would be clearer, however, is for an issue to have a product that is associated with it and for the version list to filter based on the selection made. For simplicity, this was not implemented.

The issue class is the most complicated class in the metamodel. Containing multiple references to other issues to allow for dependencies, duplicates, and blocking actions to be implemented. These all use opposites for their implementation and contain distinguishing GMF links to appear differently on the diagram. Alongside this, issues also contain an attribute for their issue type which contains a multiplicity expression with a lower bound of one. Issues contain a comment value as they are associated with one another. Issues can also contain dependencies which could be use for sprints or similar. For example, an issue could be created which is of type 'enhancement' and name 'additional file types' and this has two dependencies which are support for .PNG and .SVG, figure 3 details an example of this.

Issues and comments feature a date created attribute to allow for correct tracking of creation. Comments contain a reference to the related issue, an author and comments. References to other comments allow for nested replies on issues and allows for a more natural discussion platform; similar to blogs or Reddit, for example. All comments on Issues and replies to comments are implemented with GMF compartments to allow for nesting to occur; figure 1 demonstrates this functionality on an issue as well as displaying the custom icons. Figure 2 also details some of the other features that have been described, such as the UUID generation for the issue. Figure 8 in Appendix A details a UML diagram from this metamodel.

II. Concrete Syntax and Editor

Following the generation of the GMF editor, several areas of the code were changed and these are denoted by `@generated NOT`. One of the key changes made is how the `IdentifiableImpl` class initialises the `uuid` field. This has been initialised to a random string universally unique identifier (UUID), which have an exceedingly low probability of resulting in a collision. However, if one did want to check if the UUID existed then this would be trivial to implement. As a result of the classes which require an identifier inheriting from this class, this is the only field that requires changing.

Both `CommentImpl` and `IssueImpl` contain a `date` field which are initialised to the instantiation date; this is different to that of the metamodel, which is `createdDate`. Upon an accessor requiring the `date`, the `createdDate` updates with the string variant of the created date. Inside the

VersionImpl an else is added to the product checking proxy for getProduct. This returns the container for the version, which is the product that it resides in. This is for convenience for the user so they do not need to manually set the product that the version is associated with in the diagram editor. Attempts were made to initialise all issue type arrays whenever an issue was created, adding an issue type of bug, however, there were issues with the approach. This is left as future work.

If an issue has at least one issue type assigned, then when the icon is loaded for it in the editor, the image returned is the corresponding image for the first enum literal. Otherwise, a default issue image is loaded. Where this is implemented (IssueItemProvider), this proved to be simple to implement. If inheritance was used to implement each of the desired issue types, then this would have been far more long-winded to implement and would have resulted in code duplication or another provider being implemented.

Each class in the metamodel contains its own GMF node with a corresponding label so users can easily differentiate between the nodes. For both simplicity and for a more visual overview of the relationships between the entities, EuGENia was the editor of choice. EuGENia makes it easier for creating an issue tracking system and would be more welcoming for less technical users of the system. When allowing for members to be part of multiple team, this removed the ability to use a GMF compartment in the diagram and as such, the diagrams could become quite cluttered. Opposites are used quite frequently in the metamodel and many GMF links exists in the model, a decision was made to remove links between issues and blockers in order to simplify the diagram slightly.

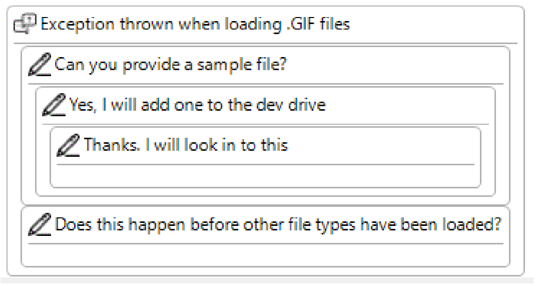
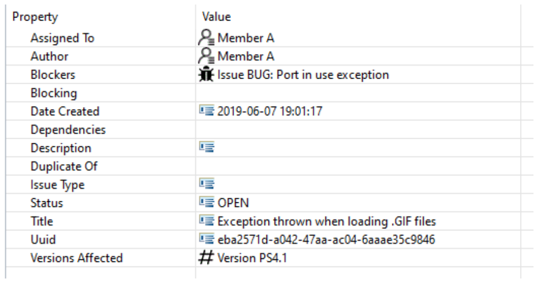

Exception thrown when loading .GIF files
Can you provide a sample file?
Yes, I will add one to the dev drive
Thanks. I will look in to this
Does this happen before other file types have been loaded?

Fig. 1: Nested comments in the editor



Property	Value
Assigned To	Member A
Author	Member A
Blockers	Issue BUG: Port in use exception
Blocking	
Date Created	2019-06-07 19:01:17
Dependencies	
Description	
Duplicate Of	
Issue Type	
Status	OPEN
Title	Exception thrown when loading .GIF files
Uuid	eba2571d-a042-47aa-ac04-6aaae35c9846
Versions Affected	Version PS4.1

Fig. 2: Properties of an issue

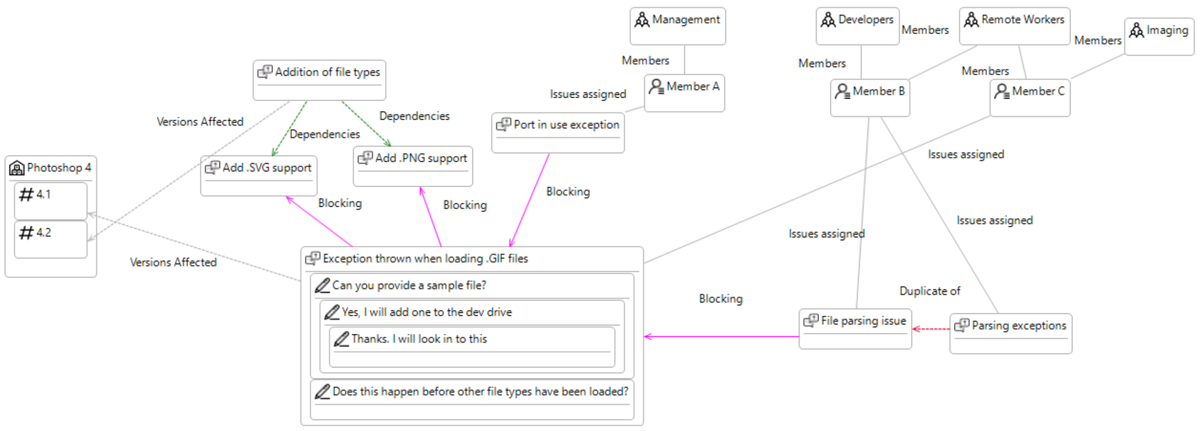
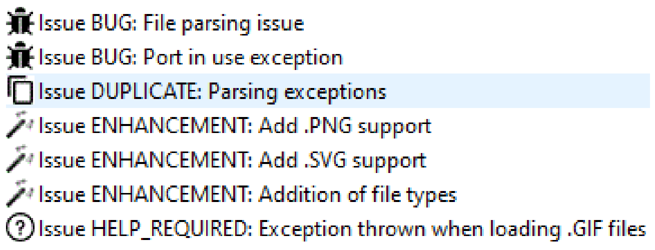


Fig. 3: Editor diagram

GMF link colours have been kept to a colour gradient of severity. Duplicates are red, blockers are magenta and dependencies are green. This makes the links slightly easier to understand. Considerations were made as to whether or not to keep labels in the diagram, however, it is my belief that this should be down to the user to decide.

While issue lists do display a custom icon, to further aid the user, the first issue type literal is prepended to the issue title when the title is requested from the issue provided; figure 4 demonstrates this functionality. This could further be improved through formatting the literal in a friendlier fashion. Custom icons were retrieved from [1].



Issue BUG: File parsing issue
Issue BUG: Port in use exception
Issue DUPLICATE: Parsing exceptions
Issue ENHANCEMENT: Add .PNG support
Issue ENHANCEMENT: Add .SVG support
Issue ENHANCEMENT: Addition of file types
Issue HELP_REQUIRED: Exception thrown when loading .GIF files

Fig. 4: Issue list. Demonstrating issue type icons and custom text

The metamodel design decisions proved to be effective when implementing additional functionality for the editor and the choice of using a diagrammatic editor also. While creating test models and diagrams, the workflow felt effective and familiar. While inheritance would have also been an effective implementation, it would have taken more time to implement and created an additional overhead in maintenance. Diagrams did offer visual feedback on how the system could be improved through issue dependencies being moved to their own class and extending from issue to tidy up the diagram slightly; though, this is left as future work.

III. Model Validation

String-based critiques and constrains all have guards that ensure that the object is defined to ensure that no errors are encountered when validating an object. To aid the user in presenting information in a consistent fashion, critiques have been implemented that prompt the user if they have not populated certain fields, such as the project's name, or if a name does not begin with a capital letter. Fixes for errors, such as a name not being capitalised, are presented to the user in the editor to allow for quick resolutions. The use of enums to determine an issue's type has allowed for an effective implementation of certain constraints in the EVL and proved to be a correct design decision early on.

Context	**Critique/Constraint**	**Name**	**Description**
Issue Tracker	Critique	ProjectName	Ensures that the project name is defined correctly.
Issue	Constraint	DescriptionLength	Ensures that an issue's description is at least ten characters in length.
Issue	Constraint	ProductsAllSameType	Checks whether or not all of the versions that are assigned to an issue are of the same product type. This is a pitfall of having a version associated with a product and not a product. However, a product is a parent of an issue. An alternative implementation would be to return a set of versions associated with a product that is referenced from an issue.
Issue	Constraint	DuplicateNotSelf	Since the user can assign a duplicate issue to an issue this constraint checks that the duplicate is not itself. The fix is to remove the duplicate issue.
Issue	Constraint	NotConflictingType	Prevents an issue from being marked as both a bug and an enhancement since this does not make sense. If an enhancement would come out of a bug fix then this should be marked as a separate issue.
Issue	Critique	CloseDuplicates	Recommends to the user that duplicate issues have their issue type set to duplicate and that they be closed.
Issue	Constraint	DetectBlockingCycle	Detects blocking cycles between issues and informs the user that they exist.
Issue	Constraint	NoOpenBlockers	Prevents issues from being closed that currently has open blockers.
Issue	Constraint	NotBlockingBugs	Prevents enhancement requests from being closed when they have open bugs that are blocking the issue.
Issue	Constraint	HasVersion	Ensures that every issue has a version assigned to it.
Team	Critique	TeamName	Ensures that the team name is defined correctly.
Member	Critique	FirstNameCasing	Prompts the user to check if the member's first name casing is correct.
Member	Critique	LastNameCasing	Prompts the user to check if the member's last name casing is correct.
Comment	Critique	MinLength	Prompts the user to populate the comment body.
Version	Constraint	OpenIssues	Prevents a version from being marked as complete when it still has open issues.
Version	Constraint	HasProduct	Ensures that a version has an associated product.

TABLE I: Discussion of each of the EVL constraints

Critiques such as name casing being correct and string lengths have been implemented to instil consistency in the application by the users. Issues with a description of less than ten characters may be useless to other users and constraints such as these will help to alleviate issues such as this.

While attempts were made to fully integrate all the EVL constraints using the EVL GMF integration tutorial [2], success was not found. When attempting to bind the constraints to the editor, XML schema is not found and so the class and marker type cannot be added. The manifest submitted with the TrackIt project contains what efforts were made. Some validation warning do appear and these are shown in figure 5. Figure 6 details some of the messages that occur after running the EVL file.

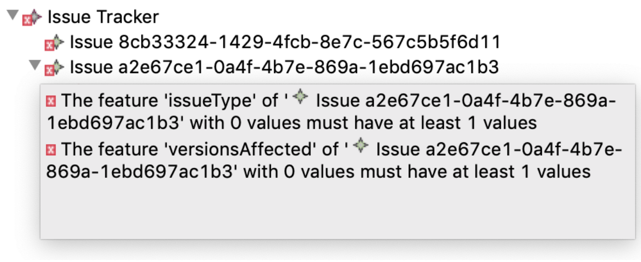
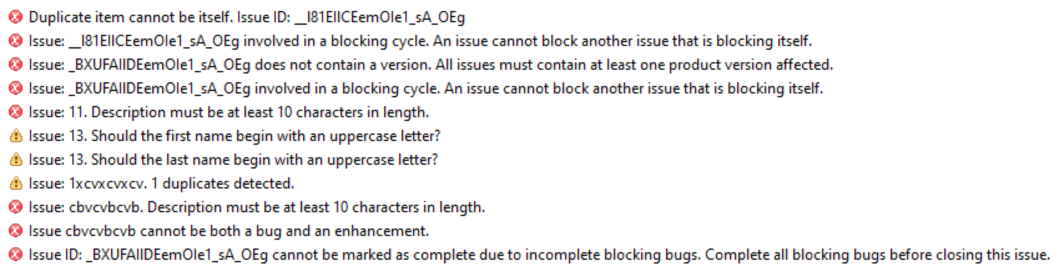

Issue Tracker
Issue 8cb33324-1429-4fcb-8e7c-567c5b5f6d11
Issue a2e67ce1-0a4f-4b7e-869a-1ebd697ac1b3
The feature 'issueType' of ' Issue a2e67ce1-0a4f-4b7e-869a-1ebd697ac1b3' with 0 values must have at least 1 values
The feature 'versionsAffected' of ' Issue a2e67ce1-0a4f-4b7e-869a-1ebd697ac1b3' with 0 values must have at least 1 values

Fig. 5: Live validation warnings in the editor



```
Duplicate item cannot be itself. Issue ID: __I81EIICEemOle1_sA_OEg
Issue: __I81EIICEemOle1_sA_OEg involved in a blocking cycle. An issue cannot block another issue that is blocking itself.
Issue: _BXUFAIIDEemOle1_sA_OEg does not contain a version. All issues must contain at least one product version affected.
Issue: _BXUFAIIDEemOle1_sA_OEg involved in a blocking cycle. An issue cannot block another issue that is blocking itself.
Issue: 11. Description must be at least 10 characters in length.
Issue: 13. Should the first name begin with an uppercase letter?
Issue: 13. Should the last name begin with an uppercase letter?
Issue: 1xcvxcvxcv. 1 duplicates detected.
Issue: cbvcvbcvb. Description must be at least 10 characters in length.
Issue cbvcvbcvb cannot be both a bug and an enhancement.
Issue ID: _BXUFAIIDEemOle1_sA_OEg cannot be marked as complete due to incomplete blocking bugs. Complete all blocking bugs before closing this issue.
```

Fig. 6: EVL messages

IV. Transformations

Before a transformation is run, it is important that validation is run to ensure that all the required fields have been populated and the model conforms to the required standard. Issues were encountered where if the UUID field of an object was not populated then the transformation would fail. Both the model-to-text and model-to-model transformations are best run from within the editor to ensure that the UUID fields, amongst others, populate correctly and do not cause any issues.

A. *Model-to-Text Transformation*

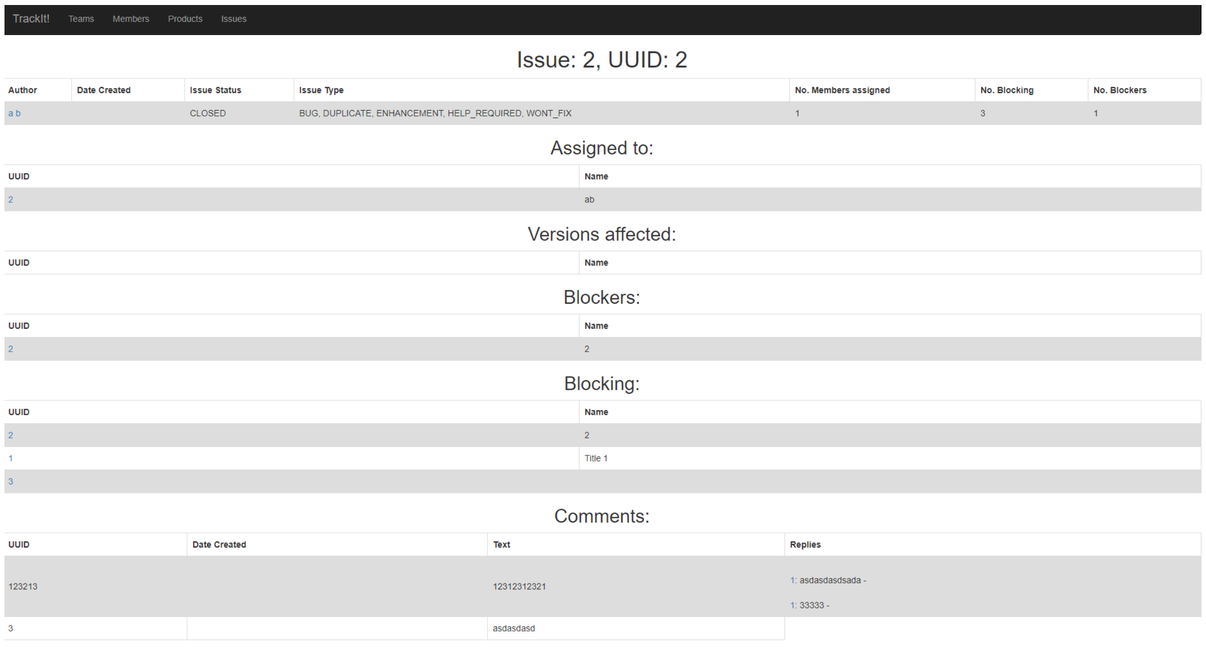
The user of the model-to-text transformation is expected to have access to the internet in order for remote scripts to be fetched during the loading of the HTML page. The transformations have been constructed using Bootstrap [3] and allows for a more familiar layout to be built in terms of styling. Without this, the pages would be style-less and unpleasant to look at. Figure 7 displays the output of a transformation for an issue. Transformations have been created for every aspect of the metamodel and all UUIDs/authors are hyperlinks to a page which displays more information on the object. Versions also feature a progress bar which displays a percentage of all the issues that have been closed for the corresponding issue.

Every .EGL file contains an import for a utilities class which contains a number of common operations across each of the transformations. These operations create strings, such as hyperlinks for issues or team members. Alongside this, there are two files which are loaded and processed which are a navigation bar and CSS for the grids that are used in various places. The navigation bar fetches CSS and JavaScript files from the Bootstrap CDN. Since the EGL files are presenting data to the user, where data does not exist the files instead state to the user that no data is present instead of displaying an empty grid. Below is a discussion of each of the files.

- **navbar.egl**. The navigation bar displays the four associations present in the issue tracker and contains hyperlinks to each section: teams, members, products and issues. Making it easy for the user to navigate to their desired section.
- **issue.egl**: transforms each issue in to a HTML file and presents information regarding each section of the object. The top displays key information about the issue. Since issues can have a number of issue types associated with them, the enum list is converted into a string and each literal is displayed. Parent-most comments are displayed as their own rows in a grid and replies are flattened and displayed in the right-most column of the row.
- **issues.egl**: displays a list of all open, closed and blocking issues. Blocking issues are sorted by the number of issues they are blocking in a descending order.
- **products.egl**: details all products in the model and all the versions associated with each product. In addition to this, the total number of versions that a product has is also detailed.
- **product.egl**: details all open and closed issues pertaining to a product. The top of the page details key statistics about the product as well as detailing the total progress through all the issues associated with all versions associated with it.

- **version.egl**: similar to product.egl, with the exception of having a version status associated with it.
- **members.egl**: similar to products.egl except detailing information about all members and the teams that they are associated with.
- **member.egl**: details all issues assigned and created by a given user.
- **teams.egl**: similar to products.egl except detailing information about all teams and their members.
- **team.egl**: details all information about a given team with key information at the top. Issue statistics per-user are displayed which details the number of issues each user has assigned to them and the total number of issues that they have created.

An example model is contained within the project workspace that can be run again to regenerate the HTML files. Otherwise, the target HTML exists in the TrackIt/egl/target directory.



Issue: 2, UUID: 2

Author	Date Created	Issue Status	Issue Type	No. Members assigned	No. Blocking	No. Blockers
a b		CLOSED	BUG, DUPLICATE, ENHANCEMENT, HELP_REQUIRED, WONT_FIX	1	3	1

Assigned to:

UUID	Name
2	ab

Versions affected:

UUID	Name

Blockers:

UUID	Name
2	2

Blocking:

UUID	Name
2	2
1	Title 1
3	

Comments:

UUID	Date Created	Text	Replies
123213		12312312321	1: asdasdasdsada - 1: 33333 -
3		asdasdasd	

Fig. 7: Model-to-text transformation output for an issue

Due to generating raw HTML, time constraints and limitations of EGL, it was not possible to generate full HTML nested comments for issues. Instead, the parent-most comment is detailed in a row and then replies are flattened and displayed in the right-most cell of the table. While a full nested comment structure would suit well to a problem like this, the comments are displayed.

B. Model-to-Model Transformation

The model-to-model transformation removes closed issues and adds the resulting set to an output model. As a result of this transformation, any products, teams and versions are also removed. This acts as a filter throughout its transformation process and was quite simple to implement given the metamodel structure and no further changes were required to the metamodel in order to accomplish this transformation. To perform the transformation, the Epsilon Transformation Language was used.

The core of the transformation is in the `IssueTracker` rule which filters sets and the result is assigned to the output of the transformation. This uses a number of operations which are defined in the file and serve to create more readable code. Once all of the transformations are complete, the resulting issue set iterated over and any issues that had blockers or blocking issues that don't exist in the resulting set are removed. As a result of the metamodel's architecture, many of the transformations could be performed in one line and this has resulted in a clean transformation. The data structures that are used in the model are all `OrderedSets` and so there was no requirement to check if duplicates had been returned when performing certain operations; though, this was a concern when performing multiple selects on various objects.

The use of enums in the metamodel has did not present any issues when developing the transformation nor were any potential issues spotted. If Boolean logic was used to denote the status of an issue then and another state was added then it would have impacted on the transformation.

References

[1] “Flaticon, the largest database of free vector icons.” [Online]. Available: https://www.flaticon.com/

[2] [Online]. Available: https://www.eclipse.org/epsilon/doc/articles/evl-gmf-integration/

[3] M. Otto and J. Thornton, “Bootstrap.” [Online]. Available: https://getbootstrap.com/

Appendix A
UML Diagram

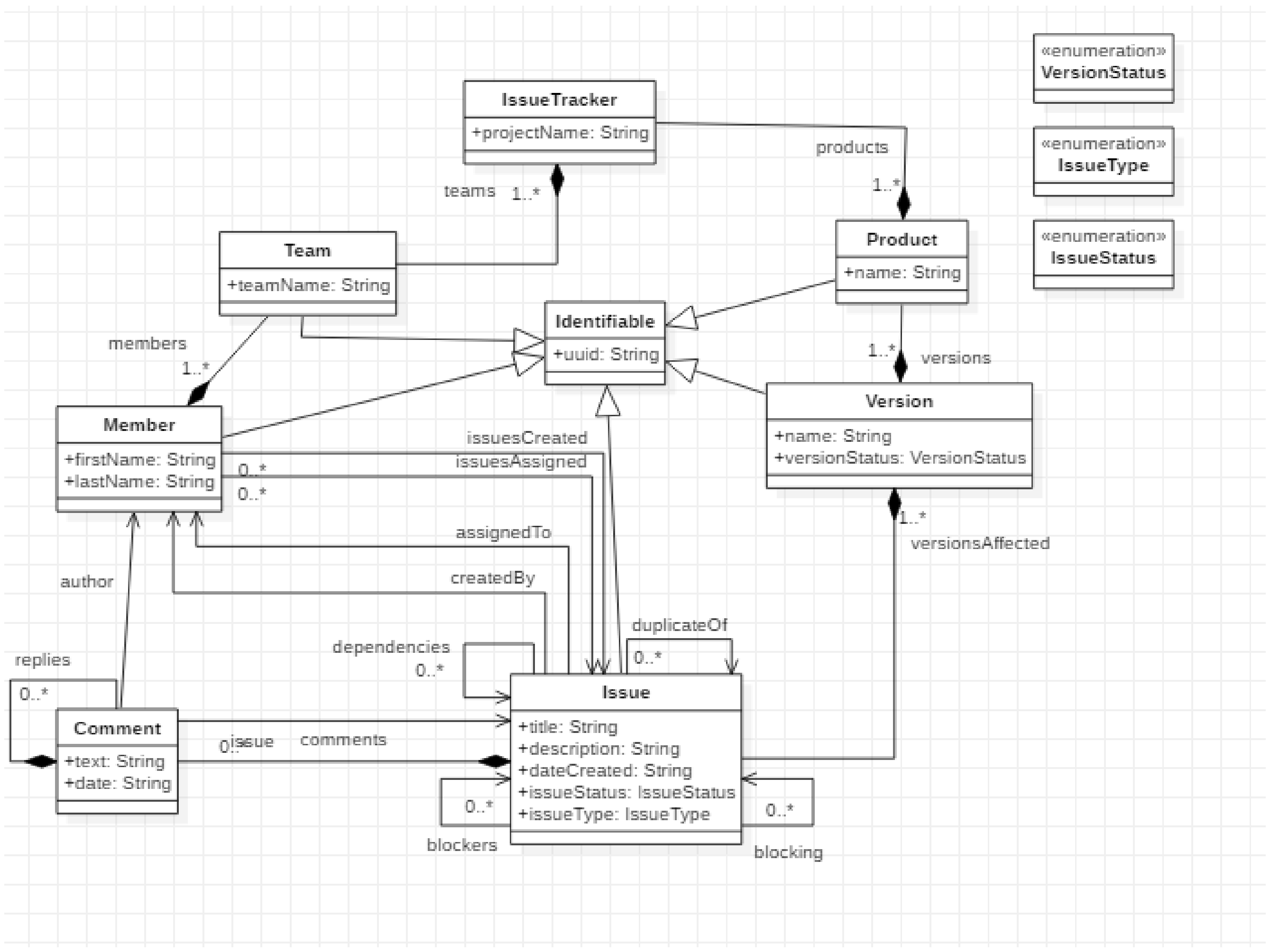


Fig. 8: UML diagram

End of Examination