

IFML Level Universal Plugin Mechanism for Defining Custom Views on Tools' Internal Data

Audris KALNINS, Paulis BARZDINS, Edgars CELMS,
Janis BARZDINS, Agris ŠOSTAKS

Institute of Mathematics and Computer Science, University of Latvia, Raiņa bulvaris 29, Riga,
LV-1459, Latvia

{audris.kalnins, paulis.barzdins, edgars.celms, janis.barzdins,
agris.sostaks}@lumii.lv

ORCID 0000-0003-0907-7496, ORCID 0009-0006-7186-9776, ORCID 0000-0001-9608-3792,
ORCID 0009-0008-0040-5557, ORCID 0009-0003-5987-1644

Abstract. Software tool extension is a long-standing problem. The simplest kind of extension is to add, to an existing tool, new custom tabs that invoke extension programs for viewing the data in a novel way. However even this task is not simple and roughly speaking every advanced tool does this in its own way. The goal of this paper is to develop a unified method of adding new custom tabs for tools whose back-end is deployed on a data server, but the front-end is browser-based (HTML web pages). More precisely, adding custom tabs is considered, to define new views of data stored on the Data server. The instrument used in this paper will be the Interaction Flow Modeling Language (IFML) designed for building precise Platform Independent Models (PIMs) for the described systems. The proposed extension method will be explained on a Deep Learning Lifecycle Data Management (DL LDM) task: first build a simple DL LDM base tool which is already practically usable, then add a PLUS component which permits the end-user to define new views of back-end data in a relatively simple way (without using the internal implementation of the base tool).

Keywords: tool extension, plug-ins, IFML, OCL, deep learning lifecycle data management

List of abbreviations:

IFML	Interaction Flow Modeling Language
PIM	Platform Independent Model
OCL	Object Constraint Language
GUI	Graphical User Interface
DL LDM	Deep Learning Lifecycle Data Management

1. Introduction

Consider the following scenario: a software tool is “bought” (it will be referred to as the base tool), but the needs of the problem domain beg for an extension of the tool. How can this problem be solved? One option is to understand the implementation to such an extent that the buyer can add extensions directly, which would be done by deciphering

the existing implementation or studying an accompanying description if such exists. This option, however, is expensive and complicated.

A question arises of whether the tool couldn't be built to already facilitate easy addition of extensions. The simplest solution is to include a way of adding plug-ins to the base tool. However, a generic plug-in incorporation method does not currently exist; every advanced tool does this in its own way. One of the tools with the most advanced approach is Eclipse (McAffer et al., 2010, Budinsky et al., 2003). There they have good mechanisms in use e.g., OSGi for the Java world (WEB, d) and its implementation Equinox in the Eclipse environment (WEB, c), as well as Beck et al. (2003) and Shaver et al. (2003). But this method has deficiency where when choosing the mechanism to use, there are lots of characteristics to consider e.g., the way the system is accessed, execution strategy, evolution, security, etc. A good overview of the field of software extension methods is given in Klatt et al. (2008). Since then, there haven't been essential advancements in the area as a whole, only several domain-specific plugin mechanisms have been developed, e.g., Bühler et al. (2022).

The domain where tool specific plugin mechanisms have recently taken off is medical image processing: ImJoy, an open-source computational platform (Ouyang et al., 2019); OIPAV, an integrated software system for ophthalmic image processing (Zhang et al., 2019); AnatomySketch, an extensible open-source software platform for medical image analysis algorithm development (Zhuang et al., 2022). ImJoy platform offers a library of general use plugins for varied image analysis improvement where users can add their own plugins for specific diseases. OIPAV offers a linked set of image analysis plugins used for clinical diagnosis and treatment of ophthalmic diseases; this analysis differs significantly from other seemingly similar domains e.g., radiology. AnatomySketch in turn offers an extensible plugin library supporting collaboration between human experts and Artificial Intelligence (AI) algorithms for varied kinds of medical image analysis. This shows that the domain of medical image analysis has found a perspectival usefulness for extensible plugin platforms.

Another domain where tool extension is very important is Deep Learning Lifecycle Data Management (DL LDM). This area is characterised by existing tools falling into one of two pitfalls – lacking functionality or growing overly complex. Thus, the following problem becomes relevant: to define a relatively simple base tool (the specific tool looked at is named LDM Core Tool) that is easily understandable while having practical uses and have it contain a built-in plugin mechanism allowing end-users to obtain important new views on their DL LDM data. This problem is studied in authors' previous papers (Celms et al., 2020; Barzdins et al., 2022). The plugin mechanism described in Celms et al. (2020) was based on partially revealing the database of the tool to be extended. But feedback from practical applications revealed that this approach was inconvenient (too complicated) for end users. Barzdins et al. (2022) proposed a higher abstraction level – the database itself was not revealed, instead the extender of the tool could use an abstract data model (PIM level) in the form of a class diagram. Plugins were defined through Metamodel Specialisation (Kalnins et al., 2019). This made the extension process quite simple, but the implementation – significantly more complicated.

The following problem naturally appears: to develop a **universal plugin mechanism** which would be applicable not only to one specific tool but to a sufficiently broad class of tools. In addition, the proposed mechanism should support sufficiently important plugin types, the mechanism itself should be sufficiently easy to understand, it shouldn't

require completely opening the tool's internal structure, and it should have a relatively simple implementation.

The main result of the paper is to offer one possible solution to this problem for an important class of tools and an important type of plugin.

The tools considered are ones whose front-end is a browser-based GUI (HTML pages) but the back-end "lives" on a Data server and stores its data (further called Internal data of the tool) in a data store which can be a data base and/or a file system. This class of tools (further called Base tools) is very broad – it comprises of e-commerce, hospital systems, and amongst others the DL LDM tools mentioned above and discussed in Celms et al. (2020) and Barzdins et al. (2022). To support the development process of such systems, OMG has developed a special language called Interaction Flow Modeling Language (IFML) (WEB, a; Brambilla et al, 2015), which enables **precise** definition of interactions between the front-end and back-end for such systems. This means that systems of this kind can be **precisely** defined at the PIM level by means of the IFML. In the general MDA methodology (Kleppe et al., 2007) the PIM concept is slightly blurry – it is a class diagram not containing all the required info for generating the system code. However, in the case of IFML, it contains more – there is a Domain model (UML Class diagram) and an Interaction flow model. Together these models are sufficient for generating the system code. These models will further be called IFML PIM. See more on IFML in Section 2.1, which presents the basic elements of IFML, and Section 2.2, which explains our DL LDM base tool example (called LDMCoreTool) in IFML.

To formulate the main result of the paper more precisely some notations are introduced. When regarding a tool as an executable program the name of the tool will be given in bold, e.g., **Aaa**. But when regarding the IFML PIM model of the tool, the same name will be used only in the non-bold form, e.g., Aaa. Then IFML' will be a version of the standard IFML with some additional precisely defined actions (introduced later).

Now the main result of the paper can be formulated more precisely: a universal method is offered on how to convert any IFML PIM model Aaa into IFML' PIM model Bbb (that will be called AaaPLUS) which defines the tool **AaaPLUS** that enables a plugin mechanism on the basis of Aaa for defining custom views on the internal data of **Aaa**. From the technical point of view this means adding new custom tabs to the Domain classes of Aaa. A click on such a tab invokes the corresponding view program written by the tool Extender. This will be explained in detail in Section 3 on the DL LDM Core Tool. The implementation of model Aaa for a specific execution environment is ensured by tools such as WebRatio (WEB, b; Brambilla et al., 2015; Acerbis et al., 2015). The implementation of AaaPLUS requires an extension of WebRatio supporting the compilation of IFML' to the target environment. This problem will be discussed further in Section 4.

2. Research background

2.1. IFML Standard: Short overview

The complete IFML standard contains many different features; in this section the ones used in this paper will be briefly explained.

From the viewpoint of this paper the IFML model consists of two kinds of diagrams – Domain diagrams and Interaction Flow diagrams. Domain diagrams are UML class diagrams describing the data model of the system and possibly containing OCL assertions as well. Interaction Flow diagrams describe the User Interface structure of the system and possible user interactions with the system. The top-level elements of Interaction Flow diagrams are View Containers (windows, pages, etc.), they define the general structure of the system GUI by named top level rectangles. A View Container can contain other nested View Containers. View containers in turn contain View Components visualised as rounded rectangles (lists, forms, details ...). View Components characterize the function of the given element from the user point of view – what content they display or what data can be entered by the user. View Components contain View Component parts, e.g., a Form contains Simple fields for specific value entering or Selection fields for value selection from a list.

There are also Events – the user interaction “commands” that guide the general flow of interaction. Events are visualized as small circles in View elements, possibly with a graphical symbol inside. Typical events are Selection from a list (shown as ) and Form Submit (shown as ). There are also Events without a symbol inside, e.g., menu or button click; their meaning can be distinguished by accompanying text. The effect of an Event is represented by an Interaction flow, a line which connects the event to the View Container to be activated next in the result of the Event. An interaction flow can be a navigation flow which simply invokes the next View container to be shown, but it can also transfer parameter values from source element to the target one, this is shown by a Parameter binding construct – a grey parallelogram explaining the value transfer.

Finally, there are Action symbols (hexagons similar to ones used in UML Activity diagrams) with texts inside describing the action to be performed. Action symbols can serve as a source or target of an Interaction flow.

All the basic graphical elements of Interaction diagrams described so far can have a set of Stereotypes defined in the IFML standard. The stereotypes specialize the meaning of a graphical element; stereotype names are taken from the traditional GUI design practice. It is assumed that in any implementation of a GUI system consistent with this IFML diagram the stereotyped element will have the corresponding functionality. Thus, a View container can have stereotypes <<Window>>, <<Modal>>, <<Modeless>>, or <<Menu>> (also <<Page>>, used in Brambilla et al. (2015)).

A Menu contains named items each starting an Interaction flow leading to the corresponding target View Container to be activated. In addition, there are “specialized stereotypes” with more specialized semantics – [H] for Home page of a dialog system, [D] for Default page – a page appearing by default in a parent container when this parent container (Window) is opened. A View element can have stereotypes <<List>>, <<Form>>, or <<Details>>. A reminder that a new Modal page forbids access to all other existing pages, but a Modeless one permits such access. In addition, there can be qualifiers [XOR] (from eXclusive OR) specifying that the given View Container (Window or page) can simultaneously show only one of its contained Containers. There are more stereotypes in IFML, but mentioned are only those used in the examples here. Action stereotypes won't be used, but precise definition of semantics for some actions with assigned names will be given. These action names are shown in red.

To sum up, an IFML model of a system is sufficient for generating a system implementation code – the functioning of the system can be described with sufficient

precision. In particular, the HTML5 code for web pages of the system can be generated quite adequately.

The IFML standard also permits extending the element stereotype list with user defined stereotypes; adding a new stereotype requires adding support for it both in the IFML diagram editor and in the system code generation. There already exists certain tool support for IFML based system development. The main tool to mention here is WebRatio (WEB, b; Brambilla et al., 2015; Acerbis et al., 2015) which is developed by the original IFML team and therefore supports all standard features of IFML. To a degree IFML support is also included in the EnterpriseArchitect tool (WEB, c). The tool support for IFML based is rapidly extending, our position on using tools is explained in Section 4 (Discussion and Conclusion).

2.2. Tool example in IFML

For an example Base tool, a Deep Learning Life Cycle Data Management (DL LDM) tool is chosen, named **LDM Core Tool**, as presented in Barzdins et al. (2022). Fig. 1 presents the general structure of this tool.

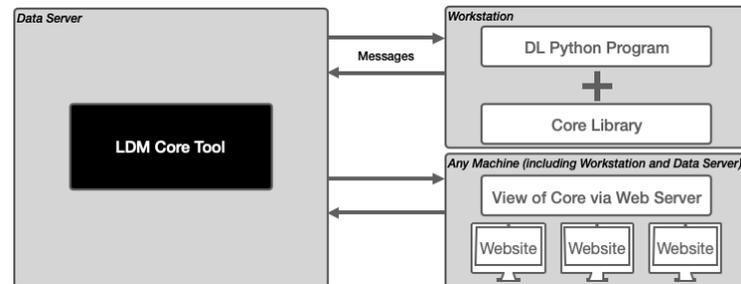


Figure 1. General structure of the LDM Core Tool (Barzdins et al., 2022).

Part of the tool is the interaction between workstations and the data server (a library, named Core Library, for sending and receiving data); this will not be touched within the paper. The other part is the web access to the data server as a GUI, which will be extended.

Figures 2 and 3 define this LDM Core Tool in IFML. As mentioned, in IFML a precise PIM level model is defined. Fig. 2 shows the Domain model of the tool – it is a UML class diagram showing data that the tool operates with. In this and the following figures no association role names are shown. Assumed by default is that the association role name leading to a class is equal to the class name started in lower case. For example, the association leading from the Run class to the Project class will have role names “run” and “project” respectively.

Fig. 3 presents the functional elements of the Interaction Flow model. The two models together describe the functionality of the LDM Core Tool’s web page – its Home page and all contained visual elements which can be accessed by the user of the tool in order to interact with the tool. For example, the elements with stereotype <<List>> show the relevant instances of the relevant Domain model class specified by the <<DataBinding>> clause. This specification can be further restricted by an OCL expression. The instances are shown as a table, where rows contain the

<<VisualizationAttributes>> of the instance. The stereotype <<ALL>> for VisualizationAttributes means that all attributes of the relevant class instance are shown in a table row. Fig. 3 doesn't show in detail how the interaction between the Data Server and Workstation occurs – it is performed by using functions from the Core Library (explained in Barzdins et al. (2022)). Some purely technical activities such as creating a new project instance are also not shown in this IFML diagram.

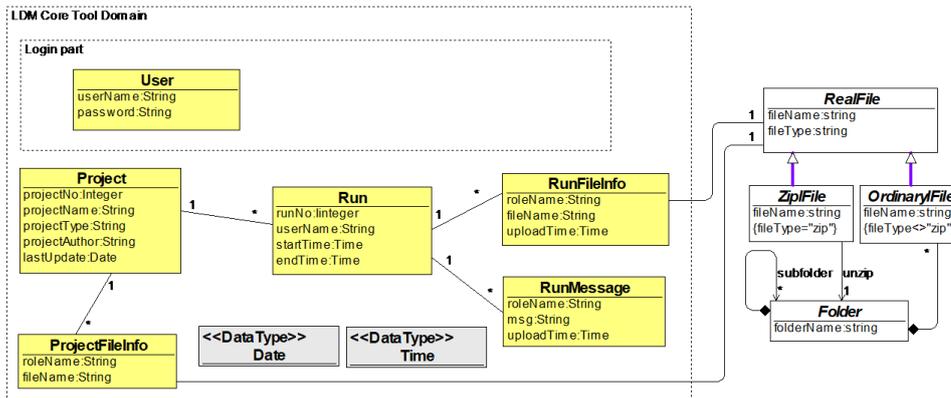


Figure 2. LDM Core Tool Domain.

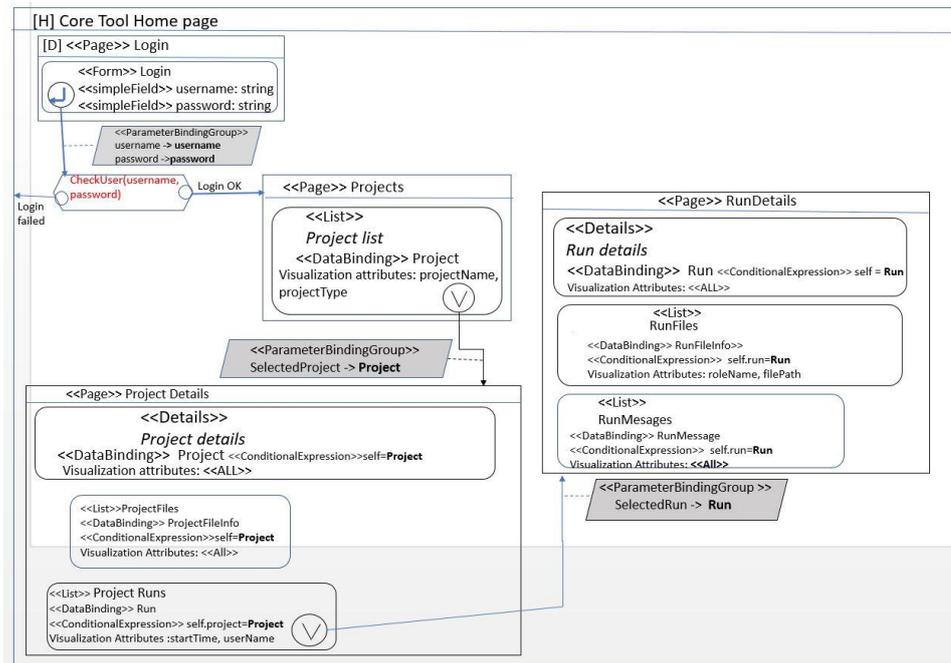


Figure 3. LDM Core Tool Home page in IFML.

In addition, the tool has the User Login page to allow only registered users to work with the tool (see Fig. 3). To support this feature the Domain model contains the User class with typical User attributes – username and password. The creation of User instances isn't shown in the IFML diagram in Fig. 3. Typically, the registration process and User instance creation is performed by a special user – Administrator.

A user can click on a row in the table displayed by a List element, then the corresponding class instance is selected as a parameter to be passed via an Interaction link (an arrowed line) to another visual element. The grey parallelogram attached to the link refines which element part in the target visual element receives the parameter value. All the described facilities are part of the standard semantics of IFML. In this paper some simple extensions to the elements of IFML are used to make the diagrams more expressive:

1. in the parameter passing construct the receiving parameter name is made bold,
2. the IFML elements related to Tool extension are shown with bold frames.

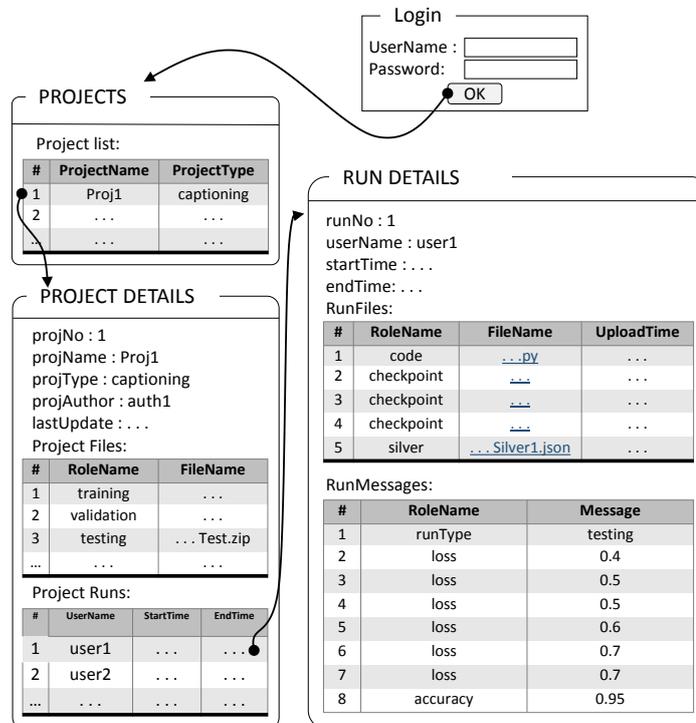


Figure 4. LDM Core Tool front-end web view.

Fig. 4 shows one possible end-user view of the LDM Core Tool which displays the Domain model instance shown in Fig. 5.

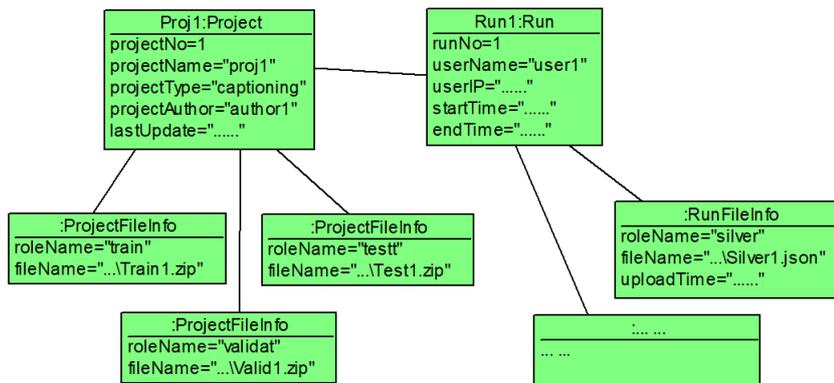


Figure 5. LDM Core Tool Domain instances.

3. Research results

3.1. Custom Tab panel

The first step on the way to Extension definition is defining places in the tool interface description in IFML where plugin invocations will be positioned, these places will be named Custom Tabs. Recall from the Introduction that the goal is to define these Tabs for arbitrary Domain classes. From the GUI point of view this means that for any instance of the considered Domain class appearing in GUI, Tabs defined for this class should be shown (instance is the context for the given Tab), these Tabs typically appear in a group typically named Tab panel (the group is shown in a horizontal row) in the class instance image.

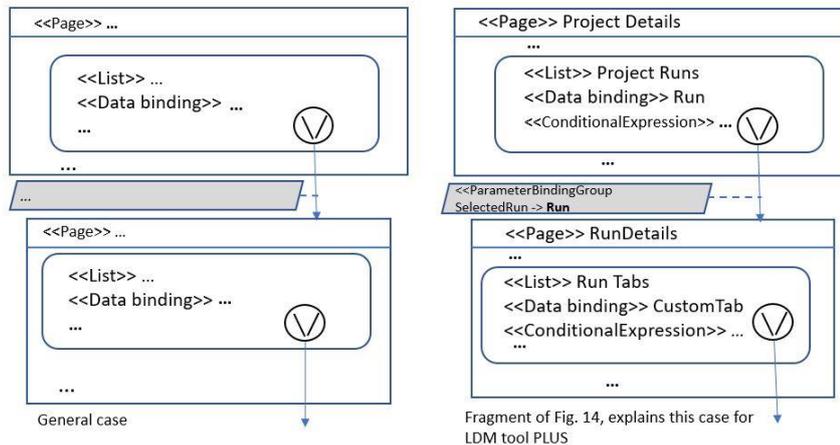


Figure 6. Class and Tab selection.

The IFML standard implies many limitations – the only possibility is to use a List for such goal. Tab panel does not appear as a View Element in IFML, therefore in IFML defined GUI you can't simply click on a tab. In addition, the class name appears only in the DataBinding construct linking a Domain element to interface element. Fig. 6 shows how to at first select a Domain Class instance (here a Run instance used in Fig. 14) and then a Tab associated to it.

3.2. Extension definition

Extension definition will be started with the description of the 4 actors participating in the extension process: Base tool PLUS Developer, Customer who has “bought” the Base tool PLUS with three sub-actors: Administrator, Configurator, and ordinary End-user. Their roles correspondingly are Administrator, Configurator, and Enduser. Administrator registers other users and their roles (in the tool data store, built as instances of User class in the Domain metamodel). The Configurator will perform the extension definition.

The proposed extension mechanism has two parts. The first is an addition of the PLUS component to the tool itself by its Developer. This then allows the second part – Customer adding extensions to the tool through the PLUS component.

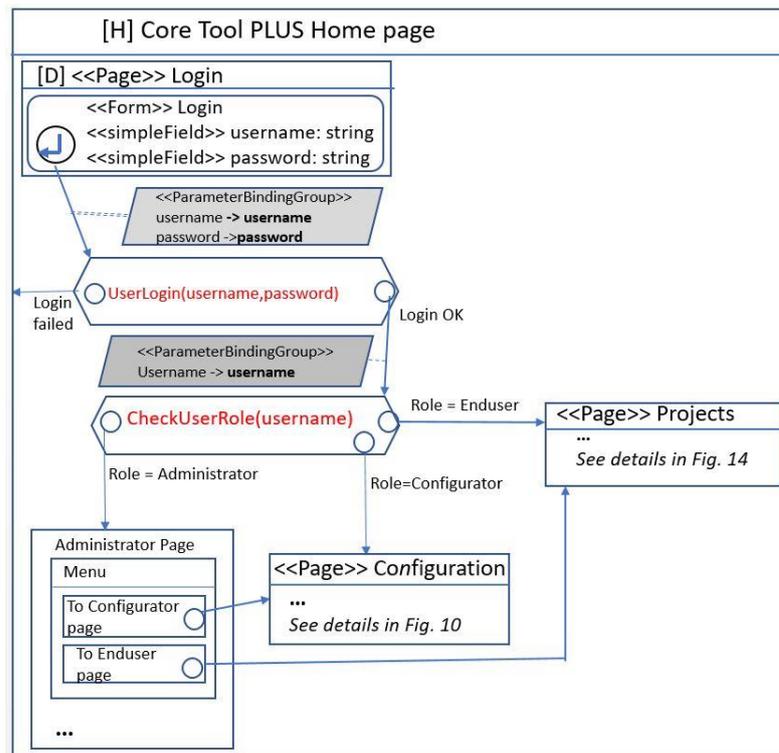


Figure 7. User Login part in the Core Tool PLUS.

The Developer includes in the PLUS component also the extended functionality of the User Login fragment (see Fig.7). There User roles are checked as well (roles can have values Enduser, Configurator, and Administrator). When a user with role Configurator logs in he is forwarded to the Configuration page where the real tool extension is being performed. It is assumed that the Configurator can also act as an Enduser if he wants to check the configuration result; this is possible via the corresponding menu. The Administrator page is shown only schematically, the User registration is not shown. The Administrator can act as a Configurator or Enduser if this is needed.

The second type of extension is performed by the Customer, or more precisely, the sub-actor with the role Configurator who has access to the Domain model of base tool PLUS in the form of class diagram but not to any details of its implementation. In other words, when the base tool is “bought” it is supplemented by its Domain model, but the implementation itself remains a “black box”. This Domain model is sometimes also called the Logical Metamodel (LMM) of the tool.

This two-part extension mechanism amounts to the main goal of this paper – to enable the Configurator to define new Custom Tabs for Domain class instances. From the semantics point of view, they are like context menu (WEB, e) items. Thus, they are context sensitive – the reaction to a click on them is dependent on the instance of the Domain class itself to which the tab is assigned. A special OCL expression (named `corrOCLExpr` in models) defines whether the reaction is enabled for the given class instance. The reaction upon a click on the tab is the invocation of the corresponding extension program written by the extender (here named “View program”) with parameter values computed by base tool PLUS from the relevant attribute values. The possibilities of a “real” definition of Custom Tabs using the IFML environment were already discussed 3.1.

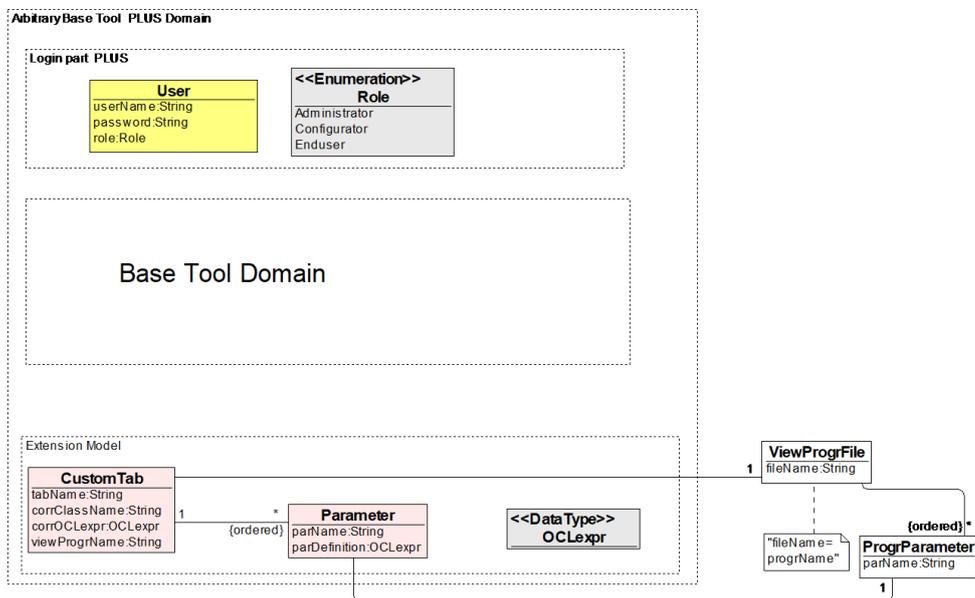


Figure 8. Arbitrary base tool Domain with an Extension Model (called Base Tool PLUS Domain).

Formally the result of an extension appearing in the Domain model is shown in Fig. 8. In this figure an arbitrary base tool is considered. If the LDM Core Tool (see Fig. 2) specifically is considered, then the LDM Core Tool PLUS Domain model is obtained (Fig. 9). Most typically the invoked View program will generate a new HTML page whose content is dynamically modified using the supplied parameter values. The parameters to be passed to this program are specified using relevant OCL expressions. Typically, the parameter values are file names on the Data Server, any such file can be used as a parameter, the file selection is specified by an OCL expression. If needed new files can be formed using OCL expressions, e.g., for an Accuracy graph visualization. This graph should show how the result Accuracy changes with each Run, the graph argument values are just consecutive integers, and the Accuracy values are obtained from the corresponding messages stored on Data Server. If needed, parameters with primitive types can be used as well.

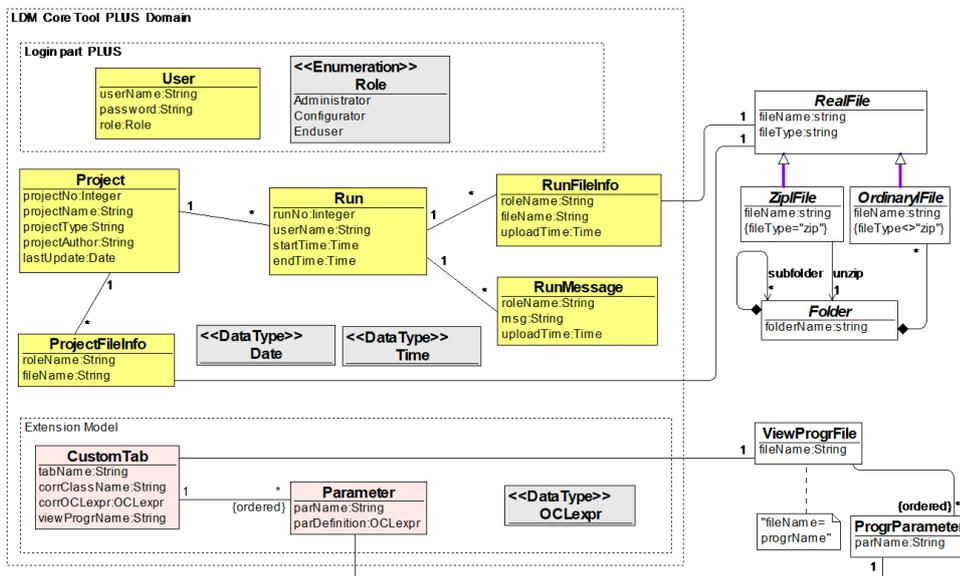


Figure 9. LDM Core Tool PLUS Domain.

To sum up, defining a specific extension means:

- 1) *Define the configuration* (concrete instances for classes CustomTab and Parameter). You can see how it is done in our approach in Fig. 10 which shows the Extension Configuration page in IFML. This is for any base tool.
- 2) *Build the View program* which in our case is a program generating the HTML page to be displayed when the user clicks the corresponding Tab. This program also uses all the supplied parameter values for modifying dynamic parts of the page. Thus, this page can display data passed via the specified parameters. In addition, during the HTML page generation the program can invoke functions

contained in a specified standard program library or a library provided by the extender, but we don't go into details on where these libraries are located.

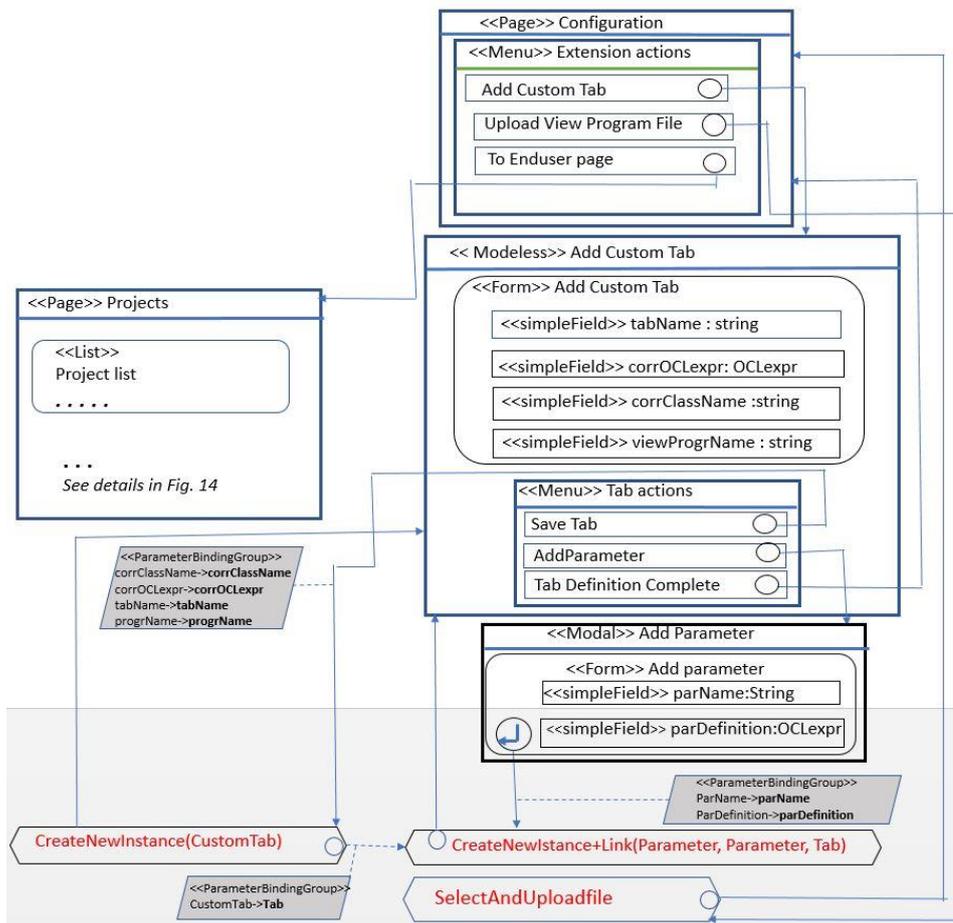


Figure 10. Extension Configuration page in IFML.

It should be noted that Action semantics in standard IFML are left quite open – it can be any kind of action as it is in UML activity diagrams. But here the action is to be characterized as precisely as possible. Action stereotypes aren't used here but these Actions are given specific names which explain the action semantics in a natural way, in diagrams these names are shown in red colour. The most used such Action name here is CreateNewInstance which creates a new instance of the class specified by the parameter of the Action. In addition, here this Action is used in a complicated scenario – typically a new Tab instance must be added, and several Parameter instances related to the new Tab instance. To create several new Tabs, this process can be repeated in a loop. If the new class instance (here the Parameter instance) must be accompanied by a new link instance, the combined CreateNewInstance+Link Action is used. The first parameter of

the combined Action is the class to be instantiated (here Parameter), the next two parameters specify the link start and end instances (here Parameter and Tab). Note the dashed arrow linking both Actions in Fig. 10; this in IFML means that only parameters are passed when the first Action completes but control is not passed to the second Action. Another Action used in the example is SelectAndUploadFile which requires several actions to be performed by the user: first select the required file via File Explorer, then press the Open button and then another button – Upload. The result is the selected file uploaded to the tool’s internal memory (on the data server). The file here must be an executable form of the View program.

The web view of the Extension Configuration page is shown in Fig. 11. Note that the first user action here performed after they have entered the attribute values for a new Custom Tab should be to save this Tab (it will be saved as a new instance of the corresponding Domain class). Then they can one by one add several Parameters for this Tab (and their links to the Tab instance will also be created). In this web view corresponding to the IFML diagram fragments of the tool GUI are shown but not the tool actions performed in background.

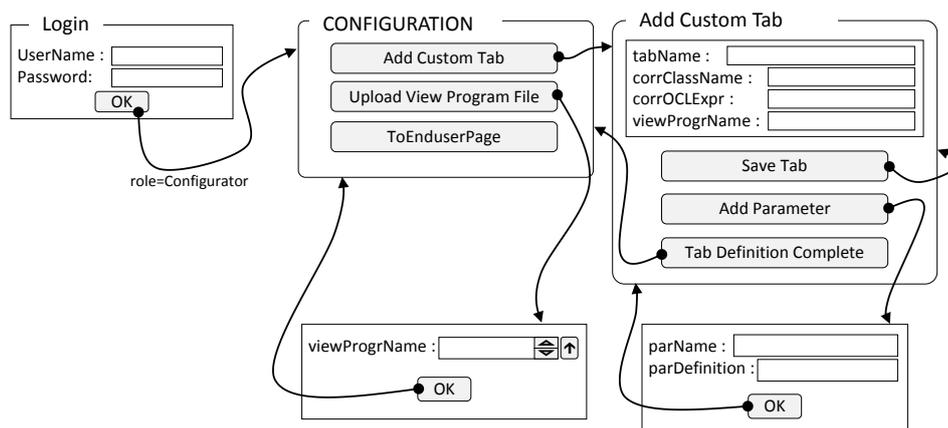


Figure 11. LDM Core Tool PLUS Extension definition web view.

3.3. Extension in work

Now returning to the specific case of LDM Core Tool PLUS. Fig. 12 and 13 show one possible extension, consisting of a new Tab instance named “GoldSilver” and its two parameters; this Tab will be applied to the Run class. Fig. 13 also shows some LDM Core Tool Domain instances. In our explanation these two separate figures are used to emphasize the fact that Extension instances are created using the extension mechanism built in LDM Core Tool PLUS (see Fig. 10), but the LDM Core Tool Domain instances are created and modified by the LDM Core Tool itself when the user is accessing its front-end from the Workstation (see Fig. 1), but this action also involves the related LDM Core Tool backend on the Data Server. This is possible since only browser-based

tools are considered which gives us the possibility to interact with the Data Server as well as view and modify data on it. Certainly, LDM Core Tool PLUS should support all these mentioned features, but this is not very complicated.

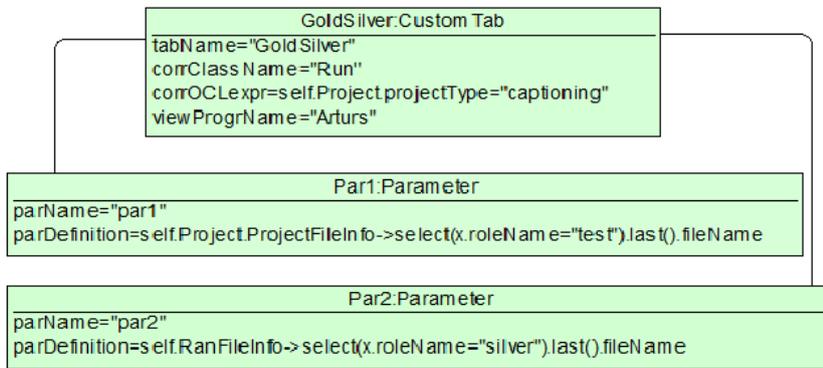


Figure 12. Extension example.

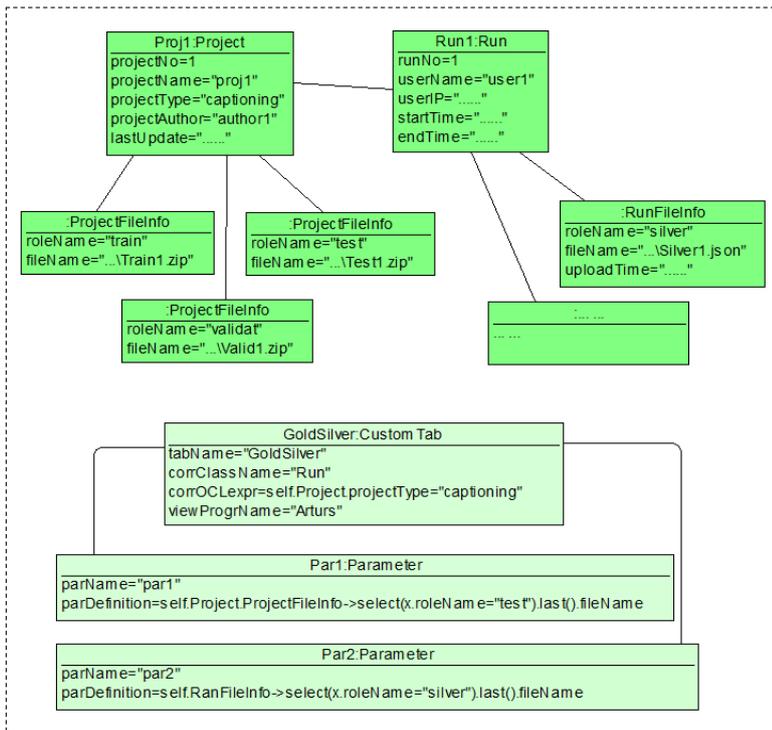


Figure 13. Instance of LDM Core Tool PLUS Domain.

Now the end-user view of our example is shown via an IFML diagram in Fig. 14. As a web page it is shown in Fig. 15. In it one specific extension example is assumed. But the IFML diagram in Fig. 14 is a general view on our LDM Core Tool Extension. This

diagram cannot formally specify the Extension execution result – it can be any new view on example data. Therefore Fig. 14 shows a “cloud” icon to show the results of execution – it can be any new page or even a new element in an existing page. But the actions necessary to supply parameters for the Extension program invocation are presented precisely – parameter values are passed as an ordered list. For all this two more custom IFML Actions are used. The first is EvaluateAllParameters which evaluates OCL expressions for parameters linked to the given Tab instance and produces an ordered list of these values. The second is InvokeProgram which means invoking a program with the name specified by progrName parameter on the value list prepared by the first action. Note that Fig. 14 differs from Fig. 4 only by content of elements shown in bold frames. And the content in bold frames in fact is universal – it is valid for any base tool, only the used class and tab names will be different. The user Login fragment was already shown in Fig. 7 therefore it is not repeated in Fig. 14.

One more comment on Fig. 14. Previously it was asserted that the described Custom Tab concept is close in semantics to the Context menu. The most adequate way to show this feature would be via a dynamic menu. But IFML has no such construct. The closest way to show this in IFML is via a List of Tabs. The selection of a Tab instance from the List is semantically close to a click on the Tab. Therefore in Fig. 14 Custom Tabs are shown as elements of a List from which a Tab instance is selected. In Fig. 14 it is shown that a selection of a Tab from the List leads to a pair of Actions, which show in detail the invocation of the relevant View program as defined by the tool configurator.

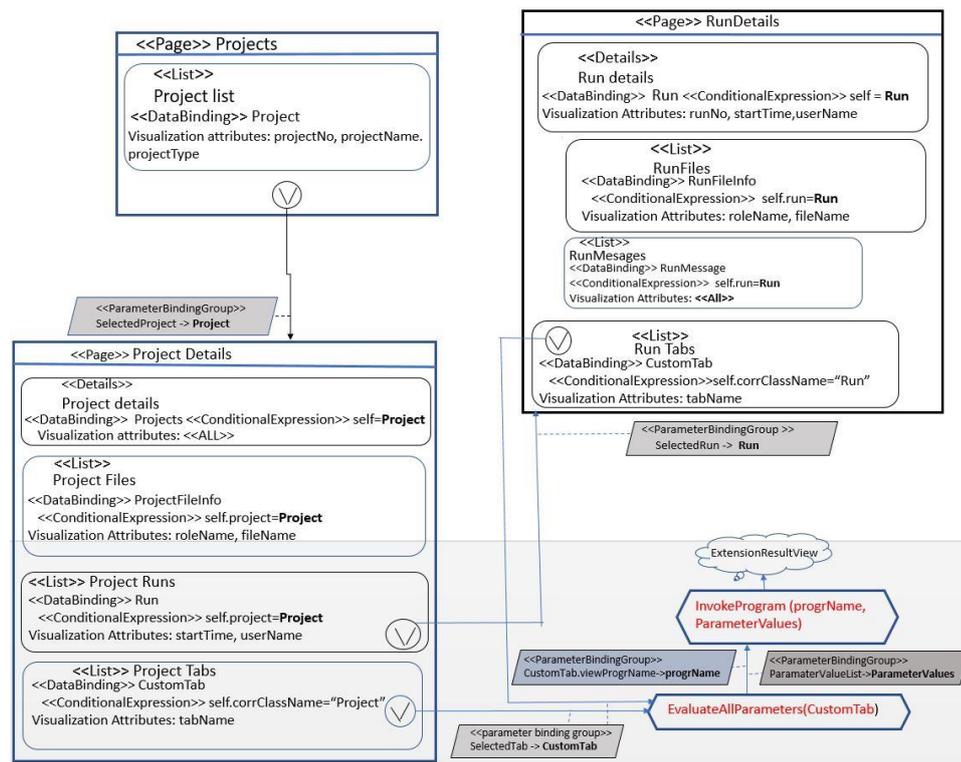


Figure 14. LDM Core Tool PLUS user view in IFML.

Now to return to a LDM Core Tool PLUS execution example. Assume that Proj1 is chosen and then Run1 from the Runs related to it. In that case the LDM Core Tool PLUS system must find the Tabs related to the Run class and show these Tab symbols (clickable areas) in the image of Run1 in the frontend view. More precisely, the system must scan all CustomTab instances and find those where `corrClassName="Run"`. In our case (see Fig. 13) there is only one such tab – “GoldSilver”. Further, a click on this tab invokes the program “Arturs” with two parameters – `par1` and `par2`, with values `par1=self.project.projectFileInfo->select(x[x.roleName="test"]).last().fileName` (in our case, `par1 = ..\Test1.zip`) and `par2=self.runFileInfo->select(x.roleName="silver").last().fileName` (in our case, `par2= ..\Silver1.json`).

“Self” refers to the selected relevant class instance (here it is Run1). The functionality of LDM Core Tool PLUS (explained on this example but applies to the general case) is defined by the IFML diagram in Fig. 14. This figure offers the selection

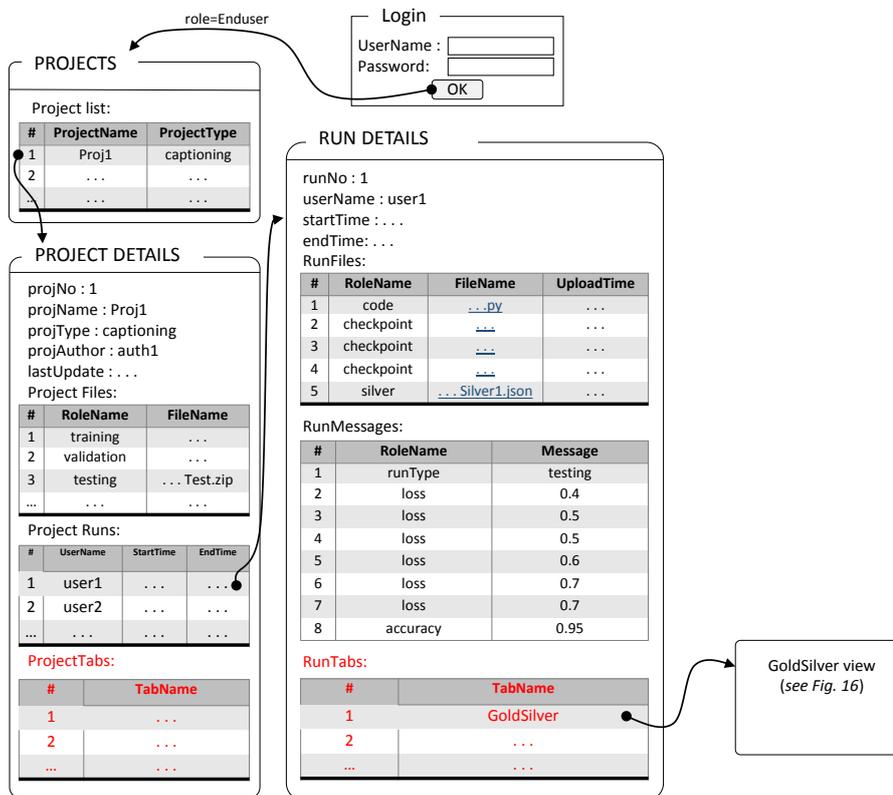


Figure 15. LDM Core Tool PLUS front-end web view.

of a specific Tab instance for the chosen Domain Class instance via a List view element offering all Tabs related to this class (the selection is constrained via a relevant OCL

expression). The Tool user then selects one of these Tab instances. In Fig. 14, the actions after selection are precisely those that were explained informally in the example; Fig. 14 simply offers the precise formal definition in IFML of these actions for the general case. In turn Fig. 15 shows the web view of the chosen instances of the involved classes as shown in the tool's front-end as a result of the Tab selection defined in Fig. 14. Fig. 16 shows the resulting visual view of invoking the program specified in this tab (in our example it is program "Arturs").

IMAGE	GOLD	SILVER
	three men fishing	a man is riding a boat
	Riga city skyline	a city filled with lots of buildings
	cars standing in traffic	a car is parked on a street
...

Figure 16. GoldSilver view.

Thus, the offered extension mechanism is completely and formally described with IFML diagrams. Hence the compilation of these IFML diagrams to a real execution environment is not a very complicated task (Brambilla et al., 2015; Acerbis et al., 2015; WEB, c). One step, namely the evaluation of OCL expression is not sufficiently precisely defined in these documents, though IFML's formal definition also contains similar OCL expressions in two examples there. There are several real ways to perform this evaluation – one is to use Eclipse OCL, the other is to restrict the used OCL expressions to such that can be expressed as a Select construct in relational DB (see Akehurst et al., 2001) where both kinds of expressions are compared). This guarantees that the used IFML constructs can really be compiled to an execution environment. Some future research may be needed to select the simplest solution for the last issue. Another issue for compilation is our special named Actions, our comments on this are given in the next Section.

4. Discussion and Conclusion

This paper describes a tool extension mechanism on the IFML level, restricted to adding new context sensitive tab symbols to an existing tool. More precisely, the paper describes a universal method of how a tool defined in IFML can be converted into a "wiser" tool for which an end-user himself can add extensions in the form of custom tabs without needing any information on the internal implementation of this "wise" tool. This possibility is ensured by the fact that in IFML the system front-end is defined at a higher abstraction level where Domain model classes can be more directly linked to the corresponding visual elements in the tool's front-end. For this purpose, the IFML

construct `<<DataBinding>> ClassName` was used (see Fig. 3, for example `<<DataBinding>> Project, <<DataBinding>> Run`). This allows the visual places for Custom Tabs to be defined at a precision typical to the IFML level.

If we were to descend to a lower level of abstraction (namely the HTML5 level), problems would appear of how to visually define extensions of such kind.

Most probably it will be quite difficult to define a universal extension mechanism at such a low level. However, the extension method proposed in this paper evidently can be used as an idea (but not a formal mechanism) for systems at the HTML5 level as well. It is because usually for any such system it is clear for the end-user what front-end elements correspond to what Domain model elements. Therefore, we can hope that systems in HTML5 can also be transformed into “wiser” HTML5 systems which would permit the end-user to add Custom tabs without using internal implementation and by defining them in a way similar to the one described before. Future research will focus on a more detailed solution for this issue.

Another aspect vital for the usability of the proposed approach is the tool support. As it was mentioned in section 2.1, the main tool for implementing IFML is WebRatio (WEB, b; Brambilla et al., 2015; Acerbis et al., 2015). It supports all standard features of IFML for model building and for compiling the created models to executable code. WebRatio is built as a plugin for the Eclipse environment (Budinsky et al., 2003), therefore it can rely on all supplied features there including Java code generation and support program libraries. Models in Eclipse Modeling Framework (EMF) in fact are a usable UML class diagram subset with class operations included. The newest version 9 of WebRatio (WEB f) already supports Low-Code Development style with many automated features of generating code from IFML diagrams, for example generating code for creating and updating IFML Domain class instances in a database. But a more general support for IFML Actions requires some extension of WebRatio by additional plugins. Thus, for example, a Pattern based development is discussed in Rodriguez-Echeverria et al. (2019). There typical patterns for CRUD (Create Read Update Delete) are discussed with partially automatic generation of IFML interaction diagrams for functionality related to CRUD operations. This is performed by a special CRUD plugin to WebRatio. In addition this plugin generates code for Create, Read, Update, Delete, Link, and Unlink actions. Rashid et al (2021) discusses how WebRatio by means of a set of plugins can be extended to a platform for drawing geographical maps needed for development of geographical information systems (GIS).

All this shows that that an efficient implementation (including code generation) of the tool extension for the DL LDM area discussed in this paper would require a specific plugin for WebRatio. It seems that the resources required for the development of this plugin would not be very large, especially considering that our team has significant experience in the use of the Eclipse framework. Thus, this task would be one of our next steps.

Acknowledgements

The research was partially supported by ERDF project 1.1.1.1/18/A/045, Latvian Council of Science project lzp-2021/1-0479 and research organization base financing at the IMCS UL.

References

- Acerbis, R., Bongio, A., Brambilla, M., Butti, S. (2015). Model-Driven Development Based on OMG's IFML with WebRatio Web and Mobile Platform. In: Engineering the Web in the Big Data Era. ICWE 2015. Lecture Notes in Computer Science, Vol. 9114. Springer. https://doi.org/10.1007/978-3-319-19890-3_39
- Akehurst, D.H., Bordbar, B. (2001). On Querying UML Data Models with OCL. In: <<UML>> 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools. UML 2001. Lecture Notes in Computer Science, Vol. 2185. Springer. https://doi.org/10.1007/3-540-45441-1_8
- Barzdins, P., Kalnins, A., Celms, E., Barzdins, J., Sprogis, A., Grasmanis, M., Rikacovs, S. (2022). Metamodel Specialisation based Tool Extension. Baltic Journal of Modern Computing, Vol. 10, No. 1, pp. 17-35.
- Beck, K., Gamma, E. (2003). Contributing to Eclipse. Addison-Wesley.
- Brambilla, M., Fraternali, P. (2015). Interaction Flow Modeling Language: Model-Driven Ui Engineering of Web and Mobile Apps with Ifml. Morgan Kaufmann.
- Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T. J. (2003). Eclipse Modeling Framework: A Developer's Guide. Addison-Wesley.
- Bühler, F., Barzen, J., Harzenetter, L., Leymann, F., Wundrack, P. (2022). Combining the Best of Two Worlds: Microservices and Micro Frontends as Basis for a New Plugin Architecture. In: Barzen, J., Leymann, F., Dustdar, S. (eds) Service-Oriented Computing. SummerSOC 2022. Communications in Computer and Information Science, Vol 1603. Springer. https://doi.org/10.1007/978-3-031-18304-1_1
- Celms, E., Barzdins, J., Kalnins, A., Barzdins, P., Sprogis, A., Grasmanis, M., Rikacovs, S. (2020). DSL approach to Deep learning Lifecycle data management. Baltic Journal of Modern Computing, Vol. 8, No. 4, pp. 597-617.
- Kalnins, A., Barzdins, J. (2019). Metamodel specialisation for graphical language support. Software and Systems Modeling Journal. Vol. 18, No. 3, pp. 1699-1735.
- Klatt, B., Krogmann K. (2008). Software Extension Mechanisms. Proceedings of WCOP'08, Vol. 2008-12, 20 pp.
- Kleppe, A. G., Bast, W., Warmer, J. (2007). MDA explained: The model driven architecture: practice and promise. Addison-Wesley.
- McAffer, J., Lemieux, J. M., Aniszczyk, C. (2010). Eclipse Rich Client Platform, 2nd ed. Addison-Wesley.
- Ouyang, W., Mueller, F., Hjelmare, M., Lundberg E., Zimmer C. (2019). ImJoy: an open-source computational platform for the deep learning. Nature Methods, Vol. 16, pp. 1199-1200.
- Rashid, M., Rasheed, Y., Azam, F., Anwar, M. (2021). Extension of Interaction Flow Modeling Language for Geographical Information Systems. ACM Digital Library, Proceedings of ICSCA 2021, pp. 186-192.
- Rodriguez-Echeverria, R., Preciado, J., Rubio-Largo, A., Conejero, J., Prieto, A. (2019). A Pattern-Based Development Approach for Interaction Flow Modeling Language. Hindawi Scientific Programming, Vol. 2019, 15 pp.
- Shahin, G., Zamani, B. (2021). Extending Interaction Flow Modeling Language as a Profile for Form-making Systems. IEEE Xplore, Proceedings of IKT 2021.
- Shaver M., Ang, M. (2003). Eclipse Platform Technical Overview. Technical Report, Object Technology International. <http://eclipse.org>
- Zhang, L., Xiang, D., Jin, C., Shi, F., Yu, K., Chen, X. (2019). OIPAV: an Integrated Software System for Ophthalmic Image Processing, Analysis, and Visualization. Journal of Digital Imaging, Vol. 32, pp.183–197.
- Zhuang, M., Chen, Z., Wang, H., Tang, H., He, J., Qin, B., Yang Y., Jin, X., Yu, M., Jin, B., Li, T., Kettunen, L. (2022). AnatomySketch: An Extensible Open- Source Software Platform for Medical Image Analysis Algorithm Development. Journal of Digital Imaging, Vol. 35, pp. 1623-1633.

- WEB (a). Interaction Flow Modeling Language, Version 1.0. OMG document /2015-02-05.
<http://www.omg.org/spec/IFML/1.0/>
- WEB (b). WebRatio. <https://webratio.com>
- WEB (c). Equinox: The eclipse foundation. The Community for Open Innovation and Collaboration. <https://www.eclipse.org/equinox>
- WEB (d). OSGi core release 8. OSGi Core 8.
<http://docs.osgi.org/specification/osgi.core/8.0.0/toc.html>
- WEB (e). Context menu. https://en.wikipedia.org/wiki/Context_menu/
- WEB (f). The Low-Code Development Platform for Digital Transformation.
<https://www.webratio.com/site/content/en/home>

Received April 21, 2023, revised June 10, 2023, accepted June 13, 2023