# Enactment of User Interface Development Methods in Software Life Cycles

**Iyad Khaddam, Hanaa Barakat, Jean Vanderdonckt**
Louvain School of Management, Université catholique de Louvain
Place des Doyens, 1 – B-1348 Louvain-le-Neuve, Belgium
{iyad.khaddam, hanaa.barakat, jean.vanderdonckt}@uclouvain.be

**ABSTRACT**
This paper presents UIDLC Manager, a software that provides user interface designers and developers with methodological guidance throughout user interface development life cycle. A methodologist firstly creates a dashboard model of the life cycle according to a corresponding meta-model in order to define a development path, decomposed into development tasks which structure the path into actions, and dependencies which serve as methodological milestones. A user interface designer or developer then enacts a previously defined development path by instantiating and interpreting a dashboard model while being provided with methodological guidance to conduct this development path. This guidance consists of steps, sub-steps, cheat sheets, and methodological actions. This approach is validated by applying it on nine classical user interface development life cycles, on two approaches for forward model-driven engineering of user interfaces based on a user interface description language, and on a linguistic approach for user interface software evolution.

**Author Keywords**
Dashboard model, dependency, development path, evolution, method enactment, method engineering, methodological guidance, model driven engineering, user interface description language, user interface development life cycle.

**ACM Classification Keywords**
• **Software and its engineering ~ Software implementation planning** • Software and its engineering ~ Software development methods • Software and its engineering ~ Software configuration management and version control systems • **Human-centred computing ~ HCI theory, concepts and models**.

## INTRODUCTION

When User Interface (UI) stakeholders such as designers, modellers, analysts, graphical designers, and developers are involved in a UI development life cycle (UIDLC), they often ask what they need to do when and how, therefore complaining about the lack of methodological guidance [6, 9, 25]. This is particular applicable to Model-Driven Engineering (MDE), where the sequence of development steps should be rigorously followed in order to guarantee the quality of the results, as opposed to flexible [14], open UIDLCs where the process can be initiated from different starting points [19].

MDE of UIs [19] explicitly relies on a structured transformation process, namely involving Model-to-Model transformation (M2M), Model-to-Code compilation (M2C) or Model-to-Code interpretation (M2I). UIDLC stakeholders do not easily perceive when some degree of freedom exists to allow alternative choices in the process [7] and when some degree of determinism constraints these choices. MDE is often considered as a straightforward process, where little or no degree of freedom is offered, even when multiple development paths are possible [11].

Facing the multiplicity of models, such as task, domain, abstract UI, concrete UI, context of use, in a particular development path, stakeholders in general, the designer in particular, are rarely provided with some guidance on when and how to produce such models [20]. The proliferation of models may even be considered as a hindrance to conduct the UIDLC in a realistic way.

When a particular step in the UIDLC should be executed, designers do not easily identify which software should be used for this purpose, especially when different pieces of software could support the same step, partially or totally [7]. When a particular software is selected, they often feel lost in identifying the right actions to execute in order to achieve the step required in the UIDLC [10].

The multiplicity of development paths conducted among or within various organizations, in particular software development companies [3], increases the feeling that executing an unsupported UIDLC requires extensive training to become effective and efficient. Typical development paths occur along the following lines [11]: forward engineering, reverse engineering, lateral engineering, cross-cutting, round-trip engineering [1], beautification [28], etc.

Although several standardization efforts (e.g., the international standard for describing the method of selecting, implementing and monitoring the software development life cycle is ISO 1220707) and official organizations promote the usage of process models in order to increase the productivity of the development life cycle and the quality of the resulting software, they do not often rely on an explicit definition and usage of a method in these process models.

The above observations suggest that MDE seems more driven by the software intended to support it, less by the models involved, and even less by a method that is explicitly defined to help UI stakeholders. Therefore, a UIDLC could rest on three methodological pillars: *models* that capture the various UI abstractions required

to produce a UI, a *method* that defines a methodological approach in order to proceed and ensure an appropriate UIDLC, and a *software* support that explicitly supports applying the method.

For this purpose, the remainder of this paper is structured as follows: Section 2 presents a characterization of these three pillars in order to report on some initial pioneering work conducted in the area of UI method engineering with the particular emphasis of methodological support. Section 3 introduces the dashboard model as a mean to define a method that may consist of one or many development paths by defining its semantics and syntax. Section 4 describes how a method could be enacted, i.e. how a development path can now be applied for a particular UI project by interpreting the dashboard model. Section 5 provides a qualitative analysis of the potential benefits of using this dashboard model for method engineering in the UIDLC. Section 6 discusses some avenues of this work and presents some conclusion.

## RELATED WORK

In general in computer science, a Software Development Life Cycle (SDLC) [29] is the structure imposed on the software development by a development method. Synonyms include software development and software process. Similarly, in the field of UI, a UI development life cycle (UIDLC) consists of the development path(s) defined by a UI development method in order to develop a UI (Figure 1). Representative examples of include: the Rational Unified Process (RUP) or the Microsoft Solution Framework (MSF). Each development path is recursively decomposed into a variety of development steps that take place during the development path. Each step uses one or several models (e.g., task, domain, and context) and may be supported by some software. All pieces of software, taken together support the development method.

For instance, the development path "Forward engineering" may be decomposed into a series of development steps: building a task model, building a domain model, building a context model, linking them, producing a UI model from these models, then generate code according to M2C. *Method engineering* [16, 21] is the field of defining such development methods so that a method is submitted to *method configuration* [17] when executed.

The meta-method Method for Method Configuration (MMC) [16] and the Computer-Aided Method Engineering (CAME) tool MC Sandbox have been developed to support method configuration. One integral part of the MMC is the method component construct as a way to achieve effective and efficient decomposition of a method into paths and paths into steps and sub-steps and explain the rationale that exist behind this decomposition. Method engineering has already been applied to various domains of computer science such as, but not limited to: information systems [17], collaborative applications [26], and complex systems [8, 13]. Typically, method engineering is based on a meta-model [18, 30] and could give rise to various adaptations, such as situational method engineering [16] and method engineering coupled to activity theory [21].

In Human-Computer Interaction (HCI), we are not aware of any significant research and development on applying method engineering to the problem of engineering interactive systems, part from M2Flex [7], Sonata [14], and Symphony [13]. Several HCI development methods do exist and are well defined, such as a task-based development method [34], method-user-centred design [20], activity theory [21], but they are not expressed according to method engineering techniques, so they do not benefit from its potential advantages.

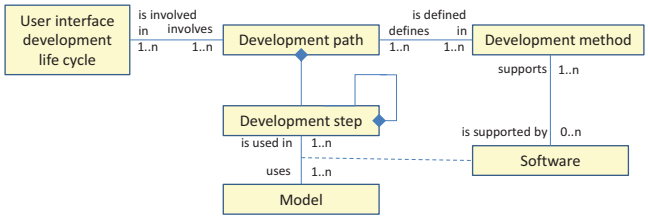


**Figure 1. Structure of a UI development life cycle.**

Probably the first one to address method engineering in HCI was the MIDAS (Managing Interface Design via Agendas/Scenarios) [25] environment. In this software, a methodologist was able to define a method by its different paths that could be followed and the steps required for achieving each path. MIDAS was able to show at any time when a method is executed, what are the different paths possible (e.g., design alternatives, criteria) by looking at design intentions stored in a library. MIDAS is tailored to the HUMANOÏD environment [25] and does not rely on a meta-model for defining a method and to execute. But it was a real methodological help.

User Interface Description Languages (UIDLs) [12] do not possess any methodological guidance based on method engineering because they mostly concentrate on the definition and the usage of their corresponding syntaxes and less on the definition of the method [3, 7].

TEALLACH [11] offers some method flexibility by enabling the designer to start from a task model, a domain model or a UI model and to then derive or link other elements related to each other. This flexibility is not method-oriented though. A more recent effort used Service Oriented Architecture (SOA) to define and enact a method [32], but there was no real software for achieving the method engineering. In conclusion, very few works exist on applying method engineering to HCI, but several existing work could benefit from it.

## A DASHBOARD META-MODEL FOR A METHOD

To adhere to method engineering principles, a meta-model [18] is defined that addresses its methodological concepts as outlined in Figure 2. The dashboard is based on a meta-model that allows the description of development steps via their decomposition in Tasks, Resources required in Tasks and Dependencies between Tasks. This Dashboard meta-model has been expressed using Ecore/Eclipse Modelling Framework (EMF) and implemented in the MOSKitt environment [33]. The main

entities, i.e. Task, Resource, Dependency and Action, are structured as follows.

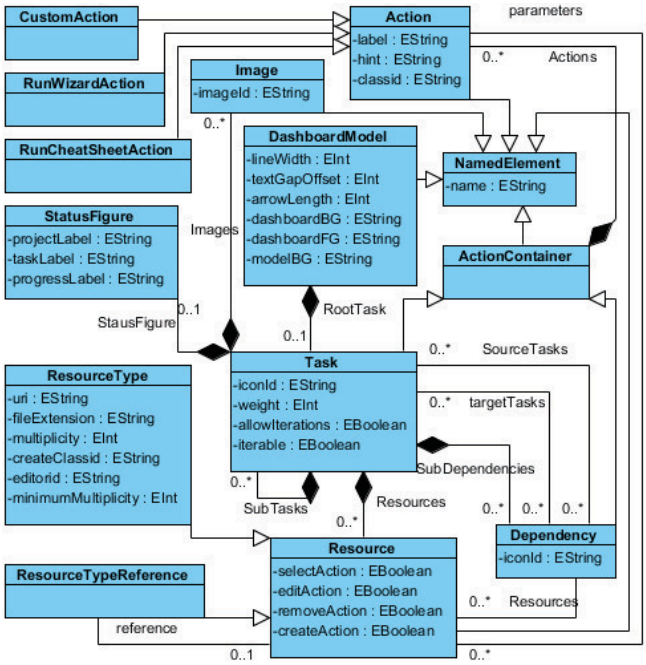


**Figure 2. The meta-model for a methodological dashboard.**

A **NamedElement** consists of a common ancestor for all metamodel elements. With the experience of the definition of several meta-models, we have found very useful to have a common ancestor element that all other elements in the meta-model inherit from. It simplifies several tasks in the following steps in the MDE approach, such as allowing to identify whether any given element belongs to this meta-model by checking its ancestry, and providing several properties we need in all elements, such as the 'name' property.

A **DashboardModel** represents a complete development path and at the same time is the root element of the meta-model. It holds the visual configuration to be used in the interpreter/enactment view.

A **Task** represents one development step of the development path. A Task is always bounded by Dependencies, except for the Tasks involving the first and last steps of the process. A Task can produce or consume zero or many Resources. As an ActionContainer, a Task can perform Actions on selected Resources.

A **Dependency** represents a milestone in the development path, which means that a series of development steps should be achieved before proceeding to the next development step. The Milestone is introduced as a straightforward mechanism for synchronizing different types of development steps, whatever their purpose is. Each Dependency is a step in the development path (Process) that forces the preceding Tasks to synchronize. A Dependency can require zero or more Resources from previous Tasks to be completed. As an ActionContainer, a Dependency can perform one or more Actions on selected Resources.

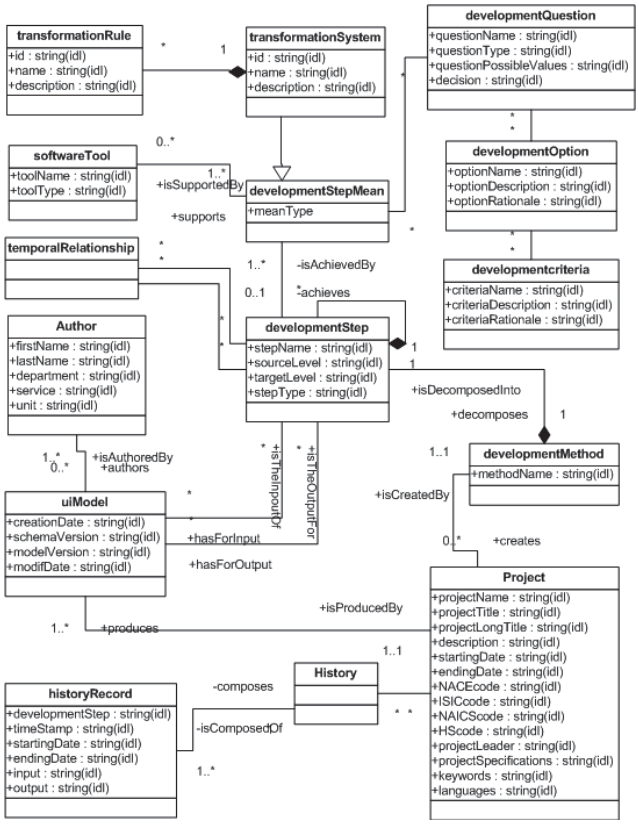


**Figure 3. The meta-model of UIDLC Manager.**

A **Resource** consists of a (im-)material entity, produced or consumed by a Task or a Dependency of this development path (Process): model definition files to meta-model..

An **Action** represents an action to be performed by the user when enacting the process. An Action can range from launching a transformation to opening a cheatsheet to visiting a web page. An **ActionContainer** represents any element in the meta-model that can hold and perform Actions. A **CustomAction** represents a custom Action allows the methodologist to specify uncommon Actions with an external specification of the Action. A **RunWizardAction** expresses a specialized Action that runs the wizard specified by the hint parameter of the Action.

The UIDLC Manager is the software that implements the methodological dashboard whose meta-model is depicted in Figure 2. Figure 3 graphically depicts the meta-model of a project.

## METHOD DEFINITION AND ENACTMENT

In order to define a UIDLC based on one or many UI development paths (e.g., simplified, enhanced forward engineering, forward engineering with loops) as defined in Figure 1, methodologist has to create one Dashboard model based on the meta-model outlined in Figure 2. A Dashboard model therefore represents the definition of a particular development path, but may also contain several

development paths in one model thanks to the concept of milestone. A milestone consists of synchronization points between tasks (e.g., development steps) involved in a development path and is attached to a *synchronization condition*. Such a condition governs the contribution of each task to the milestone (AND, OR, XOR, NOT, *n* iterations). Once the synchronization condition is satisfied, the milestone is considered to be achieved and the development path can proceed to the next development step.

**Definition**

Figure 4 depicts in Moskitt how a Dashboard model is created for the development path "Forward Engineering" that consists of the following development steps (that are represented as tasks to achieve to complete the development step) [32]:

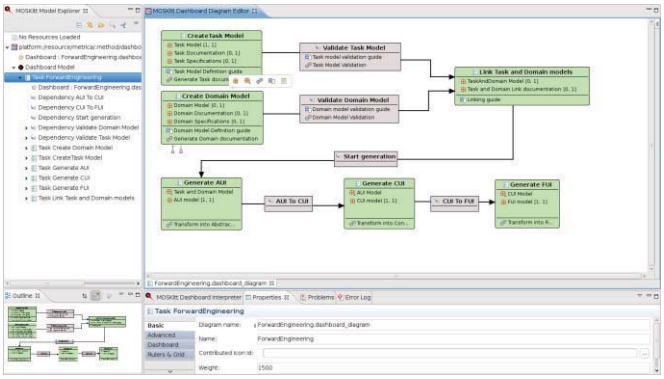


**Figure 4. The Dashboard model for the "Forward engineering" development path.**

*1. Create Task Model*. This task is aimed at creating a task model that is compliant with the task meta-model, whatever the task meta-model would be. This task has three resources:

1. One and only one task model that will result from this task.
2. An optional document containing a documentation of the task modelled.
3. An optional set of task formal specifications.

A "task model definition guide" is a cheatsheet provided for giving methodological guidance on how to define a task model. Figure 4 details some potential development steps and sub-steps for this purpose in a cheatsheet. A *cheatsheet* is hereby referred to as a methodological panel that is provided from the methodologist to the method applier with any rules, heuristics, principles, algorithms, or guidelines that are helpful for achieving the associated task (here, creating a task model that is correct, complete, and consistent). An action "Generate Task Documentation" is added in order to specify a task model would ultimately result from it. The tool allows passing parameters to customize the generation.

*2. Validate Task Model*. Once the task model has been created, its validity with respect to its corresponding task meta-model is checked by means of Eclipse model checking techniques. Therefore, only one action is triggered: "Validate Task Model". Note that this task serves as a milestone: the method applier cannot proceed with the next tasks if the synchronization condition is not satisfied.

*3. Create Domain Model*. This task is aimed at creating a domain model that is compliant with the task meta-model, whatever the task meta-model would be. It contains three resources, one cheatsheet and one action that are similar to those introduced for the task model.

*4. Validate Domain Model*. Once the domain model has been created, its validity with respect to its corresponding domain meta-model is checked by Eclipse model checking.

*5. Link Task and Domain models*. This task is aimed at establishing a link from the nodes of a task model to the appropriate nodes of a domain model thanks to the set of mappings accepted between these two models (e.g., a task observes a domain class, a task supports input/output of a set of attributes taken from different classes, a task triggers a method belonging to a class). Note that there is a dependency between this task and the two previous ones in order to ensure that the linking will be applied on two syntactically valid task and domain models.

*6. Milestone: start the Abstract UI generation*. When the task model has been linked to a domain model, we have all the elements in order to initiate a generation of an Abstract UI [15]. Again, this serves as a milestone.

*7. Generate AUI*. This task is aimed at (semi-) automatically generating an Abstract UI (AUI). For this purpose, an input resource "Task and domain models linked" (coming from the previous milestone) will result into an output resource "AUI model" by means of the action "Transform into AUI". This action is related to a set of transformation rules that are automatically applied to the input resource in order to obtain the output resource. Only one set of transformations is defined, but several alternative sets of transformation rules could be considered, thus leaving the control to the method applier by selecting at run-time which set to apply. Furthermore, this action is related to a transformation step (here, a M2M), but it could also be attached to an external algorithm that is programmed in a software. When all these alternatives coexist, a cheatsheet could be added to help the method applier in selecting an appropriate technique for ensuring this action (e.g., a transformation or an external algorithm) and parameters that are associated to this action.

*8. Milestone "AUI to CUI".* This milestone serves as a synchronization point for initiating the next development step through the task required for this purpose [1].

*9. Generate CUI.* This task is similar to the "Generate AUI" except that a CUI is produced instead of an AUI, but with parameters that govern the CUI generation.

*10. Milestone "CUI to FUI".* This milestone serves for initiating the last step and corresponds to a transformation [1].

*11. Generate FUI*. This task is aimed at transforming the CUI resulting from the previous task into code of the Final UI (FUI) by means of M2C transformation. Again, we may want to specify here that the transformation could be achieved by code generation or by interpretation of the CUI model produced. In the first case, a code generator is executed while a FUI interpreter renders the CUI into a FUI in the second case. Again, one default interpreter could be specified or the method applier can pick another one from a list of potential interpreters or rendering engines.

### Enactment

Once one or several development paths of a UI development method have been defined in a dashboard model, the method can be *enacted* [3,6] by instantiating the dashboard model. This instantiation results into a run-time representation of the Dashboard (Figure 7) that depicts the progression of tasks already achieved, future and pending tasks, all with their associated resources. For instance, if a task requires to output resources to be created, this task will only be considered finished when the corresponding actions will have been able to produce the required resources. The method enactment is then under the responsibility of the person who is in charge of applying the method defined, e.g. an analyst, a designer. In the next section, we review potential benefits brought by the MDA approach under the light of this dashboard approach.

## THE PRISM UIDLC

The prism UIDLC is different from common. It is based on a linguistic perspective to the development of the GUI. It mainly addresses the integration between HCI and Software Engineering in the development of a software product with usable UIs, with focus on the evolution of the software.

The linguistic perspective to the GUI development considers the interaction between the human and the machine as a communication text that is written differently than the human language, based on Nielsen's virtual protocol [27]. This GUI text is analysed linguistically to identify what is exchanged on each linguistic level; what are the semantics, syntactical rules, lexemes and alphabets used. It re-arranges GUI concepts on these linguistic levels and defines communication interfaces between them, in order to realize (refine with more details) concepts from upper levels on lower levels. More details on this linguistic perspective can be found in [22,23].

The linguistic perspective defines 6 levels for the development of the GUI. These levels are presented in the table 1, with description of concepts on each level, in addition to defining the communication interface between levels. The first level is "goal and task", which should be separated into two levels: "goal" and "task". But because task analysis cannot make this separation, we merge both levels into one. This merge is less confusing to the HCI community who is familiar with task analysis.

### An example on the linguistic UI development

UI development from the linguistic perspective is iteratively refined. At first, we identify task input elements: input elements that modify a task state. A task can pass through several states like: created, offered, started, completed, suspended, destroyed, and erred. More on task modelling from this perspective (a linguistic task model) is in [23]. State transitions define required input elements on the UI, for the task.

| World | Level | Artifacts | GUI Key Concepts | Communication interface | |
|---|---|---|---|---|---|
| | | | | Realize from upper level | Define for lower level |
| Conceptual | Goal & Task | Goals, Tasks | | - | -Define goals and real objects |
| | | | UI Elements | goals | -Define tasks and relations amongst<br>-Define task input elements. |
| | Semantic | Detailed functions: System, Input, Output | | -Realize tasks by defining needed detailed functions. | -define input and output elements |
| Perceptual | Syntax-time | Time containers: groups of Navigation elements | Navigation | -Realize distribution of UI elements on time by defining time containers. | -Define time containers<br>-Define navigation elements. |
| | Syntax-space | Space containers: Placement rules. | Placement | -Realize placement of UI elements in time containers on the screen. | -Define space containers<br>- Place UI elements on space containers |
| | Widgets | GUI widgets | GUI Widgets | -concretize UI elements by mapping with appropriate widgets. | -Select appropriate concrete widgets for UI elements |
| Physical | Widgets Properties | Properties of GUI widgets | | -Realize widgets attributes. | -set attributes of widgets<br>-define nothing for lower level |

**Table 1. The linguistic perspective to UI development: levels, concepts, activities and the communication interface.**

Take the example of a GUI for registration to a conference. The end user needs to fill registration information and then pay the fees. Registration information include the user's personal information, registration type (regular, student or discounted fees), and additional information if exists, and billing information. The goal of the user from using the GUI is: Register for a conference. This *goal* is further refined at the *task* level by performing two tasks: "Fill registration information" and "Pay conference fees". The task level should identify task input elements, which are in this case, input elements each completes the related task. Figure 5 presents the "Finalize Order" task input element that completes the first task. On the *semantic* level (Figure 5 refines only the task "Fill registration information"), we refine tasks by defining necessary detailed functions to carry out these

tasks. Detailed functions define needed input and output elements for the performance of the task. They also define the interface to communicate with software modules.

On the *syntax-time*, we define a correct distribution on time (that respects environment constraints): when each group of UI elements will appear on the screen. We may have two styles: (1) Display UI elements at the same time. (2) Define navigation as in Figure 6.

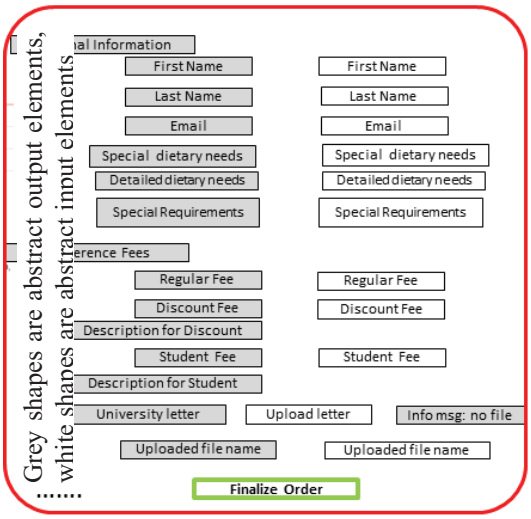


**Figure 5. Outcome on the semantic level. "Finalize Order" is a task input element.**

The *syntax-space* (Figure 7) refines only the syntax-time style 1 for concision: place elements on screen= vertical placement, horizontal, or any other form of placement. Figure 7 depicts only the output of this level for the first time container: Personal Info. according to syntax-time style 2 in Figure 6.

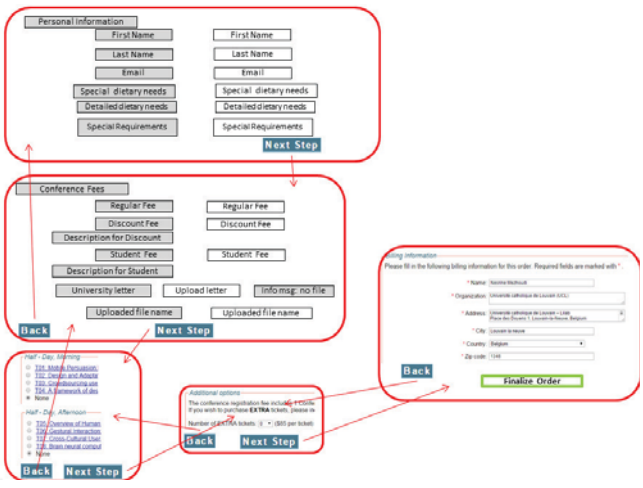

**Figure 6. Distribute on time containers and define navigation elements for parts of the GUI.**

On the *widget level* (Figure 8), map UI elements to concrete GUI widgets. Finally, on widget Properties level (Figure 9): Setting properties of widgets to get the final GUI. On the final GUI, we note that every element is related to the level of abstraction that defines it. The "next step" button in Figure9 is defined at the syntax-time level, while other input elements (text boxes) are defined at the semantic level. The reader can foresee that the button "Finalize Order" that is defined at the task level will appear on a screen at a later step (Figure 6). Please notice that upper levels may impose constraints on the choices on lower levels (like selection of widgets).

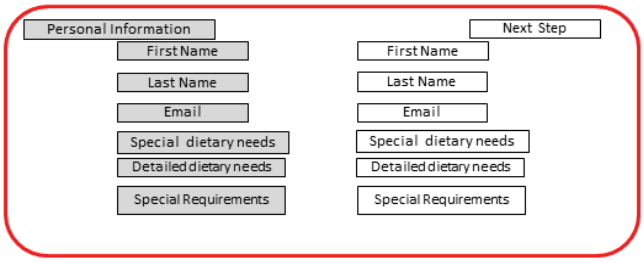


**Figure 7. Placement of elements on the screen.**

### Prism-DLC

Prism is the development life cycle that aligns the UI development (from the linguistic perspective) and the software development. The main difference from other DLCs activities is the use of a classification step to analyse and classify UI requirements in order to determine the level(s) of enactment.

The linguistic perspective allows perceiving the UI since the analysis phase. This allows defining the term "UI requirements": *any modification(s) on the UI is based on a UI requirement*. UI requirements may issue from usability (like adapt to the user's culture), software design decisions (software modules and interaction capabilities), and software detailed design modules (like allow the user to interact with a function in the system). UI requirements impact on the UI might be decomposed on several levels, which is grouped in a UI batch.

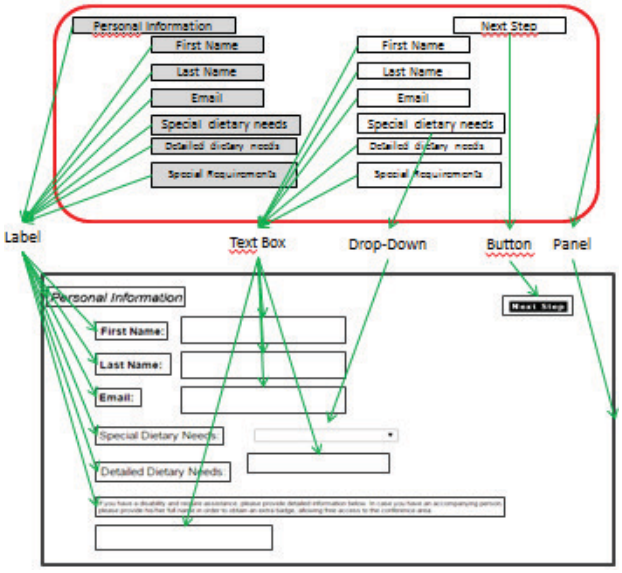


**Figure 8. Mapping UI elements with concrete GUI widgets.**

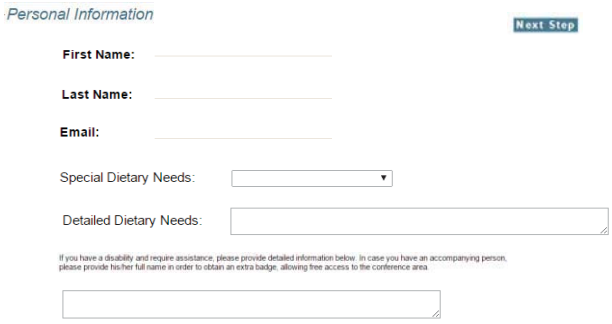


**Figure 9. Setting widgets properties and the final GUI.**

A UI patch then is the impact of a UI requirement on different levels. Different development paths can be enacted in Prism. It can be used in a real software development to align UI and software developments, or to create a UI prototype to elicit usability requirements. In this paper, we explain the first development path as it is the more interesting one. Other development paths can be figured out by the reader.

Prism does not impose any constraint on the software DLC to integrate the UI development in. Anyway, in order to explain how integration and development is performed in Prism, we show integration with general development phases: analysis, design, detailed design, coding and testing. The Prism DLC is graphically depicted in Figure 10.

*The software leads the UI development*: This approach may be preferred by software engineers. The system development starts with the analysis activity of the system. When the analysis is completed, requirements pass through the classification phase to create the UI patch. This UI patch allows creating the task model from the software analysis.

The UI patch created after the software analysis may impacts other linguistic levels. Note that software requirements are expressed at different levels of details. A user may express very detailed requirements like the preference for a specific theme of colours. The classification activity identifies UI-related aspects in every requirement and maps them to the appropriate linguistic level.

While developing the task level, usability shortcoming in analysis might be identified. The feedback loop from the task level to the analysis phase, not only ensures that usability requirements are gathered, it also assess consistency between the task model and the system analysis.

After the analysis is completed, the UI is fixed. This version of the UI might be communicated with the user as a premature version of what is expressing in requirements. Later modifications on the UI should not affect this version, which we call: the analysis-UI version. Modifications to this version should be communicated/approved with the user first.

The design phase starts with immediate feedback from the UI. As the task level is fixed, the design should implement each task appropriately: mapping to the domain model (which is part of the software design phase) and identify required UI elements. Design decisions (as UI requirements) are also linguistically classified to identify their impact on the UI. A UI patch is also created to express this impact. Note that the feedback loop from the semantic level to the design level is present to ensure that the UI and the design are consistent.

If the UI patch at the design phase contains implications on the task level, this means a shortcoming in the requirements. Tasks were not identified properly. If the software life cycle can handle such incompleteness, an alert can be triggered.

At the detailed design phase, the same repeat as with the design phase. After completing the detailed design, the semantic level is fixed. The UI for carrying out tasks is completely defined. No further modifications can be performed on the semantic level without repeating the design and detailed design phases. This version of the UI is called the *design-UI*.

In parallel with the implementation phase, the UI can be refined on the navigation, placement, widgets selection and stylistics on the last level. This gives the UI design the freedom to manipulate these aspects with the guarantee that any implemented design is compatible with the semantic and the task levels. Both activities are synchronized to start the testing activity.

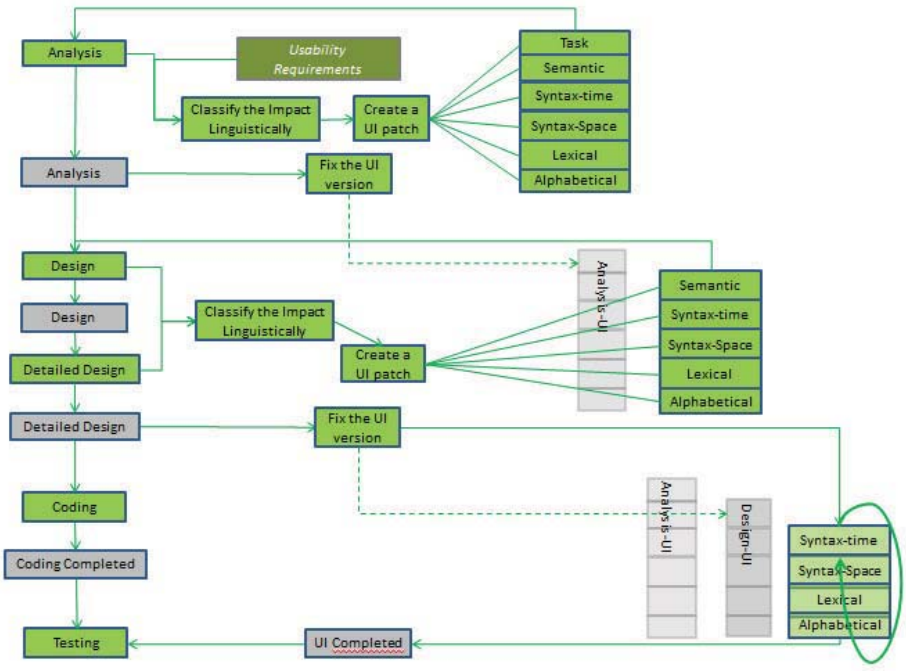


**Figure 10. The Prism DLC.**

Testing can be decomposed into two activities (not depicted in Figure 10). Validation is to assess the implementation conforms to the specification (the design), and Verification to verify that the product satisfies user's requirements. Note that validation testing can be done on the design-UI version and verification can be done on the analysis-UI version. UI Validation testing is to compare the design-UI version with the final UI version on the navigation design, placement, widgets and stylistics. Functionality is guaranteed. Verification might be possibly enacted before fixing the design-UI.

## EXAMPLES ON OTHER UIDLC

The purpose of this section is to demonstrate that the dashboard model is independent of any method, any meta-model and any User Interface Description Language (UIDL). It could be used for defining any UIDLC, any method that supports UIDLC (such as [4, 5, 24] to name a few), any meta-model of a model involved in such a UIDLC, and any UIDL (see [6, 19] for some representative examples). The only requirement is that each model should be explicitly linked to its corresponding meta-model in order to check its validity and conformity with respect to the meta-model as it is typically the case in MDE. Transformations gathered in transformation steps [1] should satisfy the same requirement, unless they are executed outside the Eclipse platform. The advantage of this approach is that all models and transformations between are defined by their corresponding meta-models in Eclipse, but forces to define them beforehand.

We evaluated UIDLC Manager on several UIDLCs. Table 1 contains the list of evaluated DLCs with a comparative analysis from the method engineering point of view. The table shows the number of development steps in each method, the number of check points and the number of connections among development steps. These DLCs differ in the coverage of development phases an in the distribution of activities on each development phase. In order to illustrate this difference, we project activities in each DLC on the generic development phases, defined as: Requirements Analysis (R), Design (D), Detailed Design (DD), Coding (C), Testing (T) and Maintenance (M). The result is shown in the right-most column in Table 1. Due to space constraints in this paper, we only illustrate the modeling of V-Cycle DLC using our tool in Fig 11. We also illustrate the projection of V-Cycle on the generic development phases in Figure 12. For the other UIDLC that have been realised with UIDLC Manager, the reader can visit the web-page: https://sites.google.com/site/user interfacedevelopmentcycles/uidlcmanager. Screenshots are given of respective UIDLCs that are typically found in HCI.

| SDLC | Dev. steps | Check points | Connections | Distribution measures |
|---|---|---|---|---|
| Collin's Circle [4] | 7 | 7 | 17 | R D DD C T M<br>2 5 3 1 1 0 |
| Curtis & Hefley's [5] | 24 | 24 | 61 | A D DD C T M<br>6 3 3 3 6 3 |
| Nabla [24] | 25 | 27 | 68 | A D DD C T M<br>6 5 2 2 8 1 |
| O Cycle [31] | 6 | 6 | 14 | A D DD C T M<br>1 2 0 1 1 1 |
| Spiral Model [2] | 22 | 22 | 43 | A D DD C T M<br>9 6? 6? 4 4 ? |
| Star Model [15] | 6 | 6 | 15 | A D DD C T M<br>2 1 0 1 1 0 |
| V Cycle [8] | 8 | 8 | 19 | A D DD C T M<br>1 2 1 1 3 1 |
| Waterfall [29] | 8 | 8 | 22 | A D DD C T M<br>1 1 1 1 1 1 |
| Prism-SDLC | Max= 26 | | | A D DD C T M<br>10 8 9 5 1 1 |

**Table 2. A comparison between different DLCs.**

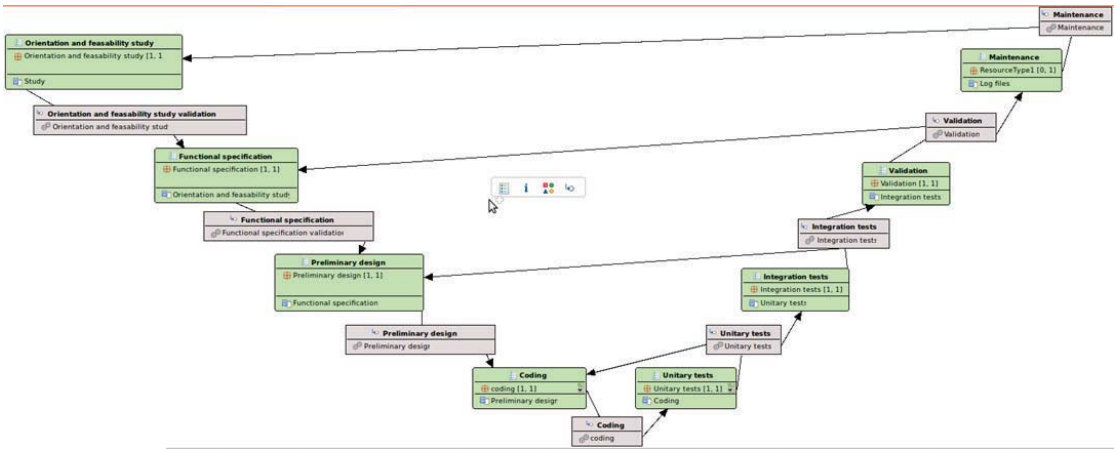


**Figure 11. The V-Cycle using UIDLC Manager.**

## CONCLUSION

In this paper, we presented the dashboard model as a way to support the method engineering of a user interface development life cycle. For this purpose, we first defined what such a development life cycle is and how to structure it according to the principles of method engineering [3, 16, 17].

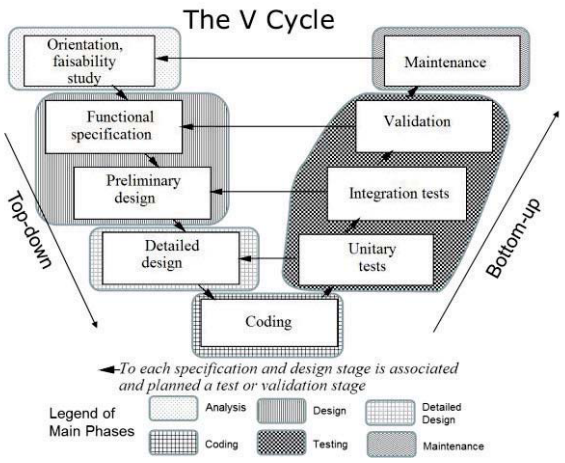


**Figure. 12 Distribution of the V-Cycle activities on generic development phases.**

This development life cycle is then expressed in terms of the following concepts: one or several development steps are defined in one single dashboard in order to create one development method, a development (sub-)step becomes a task to be achieved in the dashboard, the models involved in a development step become resources to be created and consumed by a task in the dashboard, the software required to manipulate these models become associated to resources via their associated file extension and/or from a list of potential software (e.g., model editor, model validator, model checker, transformation engine). The next step of this research will consider the forthcoming ISO 24744 standard on method engineering [2] that defines a set of concepts that support the definition and the enactment of a method based on well-defined concepts along with a graphical notation that combines structural aspects (e.g., how a task is decomposed into sub-tasks) and temporal aspects (e.g., how tasks are related to each other through dependencies and constraints).

## REFERENCES


1. Aquino, N., Vanderdonckt, J., and Pastor, O. Transformation Templates: Adding Flexibility to Model-Driven Engineering of User Interfaces. In *Proc. of 25th ACM Symposium on Applied Computing SAC'2010* (Sierre, March 22-26, 2010). ACM Press, New York, (2010), 1195–1202.
2. Boehm, B. A Spiral Model of Software Development and Enhancement. *IEEE Computer 21*, 5, (1988), 61-72.
3. Brinkkemper, S. Method engineering: engineering of information systems development methods and tools. *Information and Software Technology 38*, 4, (1996), 275–280.
4. Collins, D. *Designing object-oriented user interfaces*. The Benjamin/Cummings Publishing Company, Inc, Redwood City, 1995.
5. Curtis, B. and Hefley, B. A WIMP no more, the maturing of user interface engineering. *Interactions 1*, 1, (1994), 22–34.
6. Céret, E., Dupuis-Chessa, S., Calvary, G., Front, A. and Rieu, D. A taxonomy of design methods process models. *Information and Software Technology 55*, 5, (2013), 795–821.
7. Céret, E., Dupuy-Chessa, S. and Calvary, G. M2FLEX: A process metamodel for flexibility at runtime. In *Proc. of IEEE Conf. on Research Challenges in Information Systems RCIS'2013*. IEEE Press, Piscataway, (2013), 1–12.
8. Forsberg, K. and Mooz, H. System engineering overview. In *Software Requirements Engineering*, R.H. Thayer and M.Dorfman (eds.), Second edition. IEEE Press, Piscataway, (1997), 44–72.
9. Gonzalez-Perez, C. and Henderson-Sellers, B. A work product pool approach to methodology specification and enactment. *Journal of Systems and Software 81*, 8, (2008), 1288–1305.
10. Göransson, B., Gulliksen, J., and Boivie, I. The usability design process - integrating user-centered systems design in the software development process. *Software Process: Improvement and Practice 8*, 2, (2003), 111–131.
11. Griffiths, T., Barclay, P.J., Paton, N.W., McKirdy, J., Kennedy, J.B., Gray, P.D., Cooper, R., Goble, C.A., and Pinheiro da Silva, P. Teallach: a model-based user interface development environment for object databases. *Interacting with Computers 14*, 1, (2001), 31–68.
12. Guerrero García, J., González Calleros, J.M., Vanderdonckt, J., and Muñoz Arteaga, J. A theoretical survey of user interface description languages: preliminary results. In *Proc. of LA-Web/CLIHC'2009* (Merida, November 9-11, 2009). IEEE Press, Piscataway, (2009), 36–43.
13. Godet-Bar, G., Rieu, D., Dupuy-Chessa, S., and Juras, D. Interactional objects: HCI concerns in the analysis phase of the Symphony Method. In *Proc. of ICEIS'2007*, 5, (2007), 37–44.
14. Godet-Bar, G., Dupuy-Chessa, S., and Rieu, D. Sonata: Flexible connections between interaction and business spaces. *Journal of Systems and Software 85*, 5, (2012), 1105–1118.
15. Hartson, H.R. and Hix, D. Towards empirically derived methodologies and tools for human-computer interface development. *International Journal of Man-Machine Studies 31*, 4, (1989), 477–494.
16. Henderson-Sellers, B. and Ralyté, J. Situational method engineering: state-of-the-art review. *Journal of Universal Computer Science 16,* 3, (2010), 424–478.
17. Hug, Ch., Front, A., Rieu, D., and Henderson-Sellers, B. A method to build information systems engineering process metamodels. *Journal of Systems and Software 82*, 10, (2009), 1730–1742.
18. Jeusfeld, M.A., Jarke, M., and Mylopoulos, J. *Metamodeling for Method Engineering*. The MIT Press, New York, 2009.

19. Pérez-Medina, J.-L., Dupuy-Chessa, S., and Front, A. A Survey of Model Driven Engineering Tools for User Interface Design. In *Proc. of Int. Workshop on Task Models and Diagrams for User Interface Design TAMODIA 2007*. Lecture Notes in Computer Science, vol. 4849. Springer, Berlin, (2007), 84–97.
20. Karlsson, F. and Ågerfalkb, P.J. Method-User-Centred Method Configuration. In *Proc. of Situational Requirements Engineering Processes SREP'2005* (Paris, August 29-30, 2005). Ralyté, J., Ågerfalk, P.J., Kraiem, N. (Eds.), 2005.
21. Karlsson, F. and Wistrand, K. February. Combining method engineering with activity theory: theoretical grounding of the method component concept. *European Journal of Information Systems 15*, 1, (2006), 82–90.
22. Khaddam, I., Mezhoudi, N., and Vanderdonckt, J. Towards a linguistic modeling of graphical user interfaces: eliciting modeling requirements. In *Proc. of 3rd Int. Conf. on Control, Engineering & Information Technology CEIT'2015* (Tlemcen, 25-27 May 2015). IEEE Press, Piscataway, (2015), 1–7.
23. Khaddam, I., Mezhoudi, N., and Vanderdonckt, J. Towards Task-Based Linguistic Modeling for designing GUIs. In *Proc. of 27ème conference francophone sur l'Interaction Homme-Machine IHM'2015* (Toulouse, 27-30 October 2015). ACM Press, NY, Article #17, 2015.
24. Kolski, C. A "call for answers" around the proposition of an HCI-enriched model. *Software Engineering Notes 3*, 23, (1998), 93–96.
25. Luo, P. A human-computer collaboration paradigm for bridging design conceptualization and implementation. In *Proc. of DSV-IS'94* (Carrara, June 8-10, 1994). Focus on Computer Graphics Series. Springer Vienna, (1994), 129–147.
26. Molina, A.I., Redondo, M.A. and Ortega, M. A methodological approach for user interface development of collaborative applications: A case study. *Science of Computer Programming 74*, 9, (2009), 754–776.
27. Nielsen, J.A. Virtual protocol model for computer-human interaction. *International Journal of Man-Machine Studies 24*, 3, (1986), 301–312.
28. Pederiva, I., Vanderdonckt, J., España, S., Panach, I., and Pastor, O. The beautification process in model-driven engineering of user interfaces. In *Proc. of 11th IFIP TC 13 Int. Conf. on Human-Computer Interaction INTERACT'2007* (Rio de Janeiro, September 10-14, 2007). Lecture Notes in Computer Science, 4662. Springer, Berlin, (2007), 409–422.
29. Pressman, R.S. *Software engineering, a practitioner's approach*. Fifth Edition. McGraw Hill, 2001.
30. Sauer, S. Applying meta-modeling for the definition of model-driven development methods of advanced user interfaces. *In Proc. of Model-Driven Development of Advanced User Interfaces MDDAUI'2007*. H. Hussmann, G. Meixner, D. Zuehlke (Eds.). Studies in Computational Intelligence, 340, (2007), 67–86.
31. Scapin, D., Leulier, C., Vanderdonckt, J., Bastien, Ch., Farenc, Ch., Palanque, Ph., and Bastide, R. Towards automated testing of web usability guidelines. In *Proc. of 6th Conf. on human factors & the web HFWeb'2000* (Austin, 19 June 2000). Ph. Kortum & E. Kudzinger (Eds.). University of Texas, Austin, 2000.
32. Sousa, K., Mendonça, H., and Vanderdonckt, J. Towards method engineering of model-driven user interface development. In *Proc. of Int. Workshop on Task Models and Diagrams for User Interface Design TAMODIA 2007*. Lecture Notes in Computer Science, 4849. Springer, Berlin, (2007), 112–125.
33. The Moskitt Environment, ProDevelop, Valencia, 2015. Accessible at http://www.moskitt.org/eng/proyecto-moskitt/
34. Wurdel, M., Sinnig, D., and Forbrig, P. Task-Based Development Methodology for Collaborative Environments. In *Proc. of EIS'2008*. Lecture Notes in Computer Science, vol. 5247. Springer, Berlin, (2008), 118–125.