End-User Composition of Graphical User Interfaces by Composite Pattern

Jean Vanderdonckt
Université catholique de Louvain, LouRIM Institute
Louvain-la-Neuve, Belgium
jean.vanderdonckt@uclouvain.be

Donatien Grolaux
ICHEC Brussels Management School and
Université catholique de Louvain, LouRIM Institute
Brussels/Louvain-la-Neuve, Belgium
donatien.grolaux@{ichec,uclouvain}.be

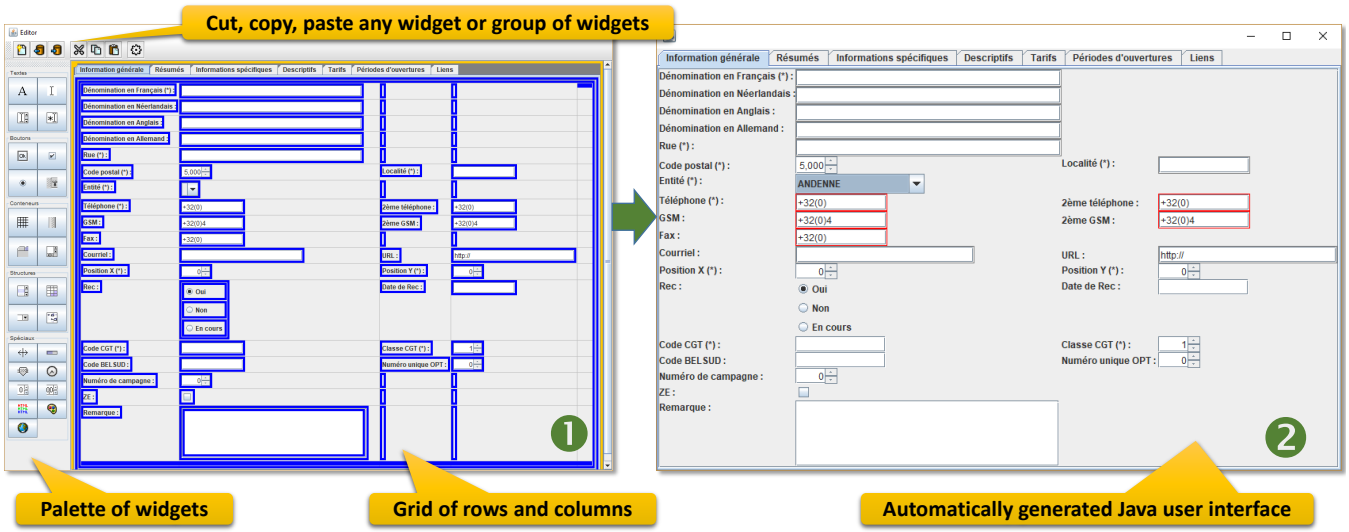

Figure 1: The two main ComPat steps: (1) add widgets from a palette and compose widgets, groups of widgets by cut, copy, paste to instantiate any pattern, (2) automatically generate a Java Swing graphical user interface.

ABSTRACT

We present ComPat, an open source visual editor enabling end users to compose graphical user interfaces based on the composite pattern, a common software engineering design pattern: any widget or group of widgets is treated the same way as a single instance of the same type of widget. ComPat exploits a grid of rows and columns, where each cell, regulated by layout constraints, is populated either by direct import of widgets from a palette or by pattern application. In order to compose graphical user interfaces, any portion could be cut, copied, pasted, and treated as a single object thanks to the composite pattern, thus facilitating reusability. Any portion becomes a pattern that can be applied either by direct instantiation or by rewriting. ComPat automatically generates a Java Swing graphical user interface corresponding to the composition and stores its definition in a User Interface Description Language based on a XML Schema.

EICS '19, June 18–21, 2019, Valencia, Spain

ACM ISBN 978-1-4503-6745-5/19/06.
https://doi.org/10.1145/3319499.3328236

CCS CONCEPTS

• **Applied computing** → **Image composition**; • **Software and its engineering** → **Graphical user interface languages**; Extensible Markup Language (XML); • **Human-centered computing** → *Graphical user interfaces*;

KEYWORDS

Composition of user interfaces; composite pattern; end-user programming; graphical constraints; UML.

ACM Reference Format:
Jean Vanderdonckt and Donatien Grolaux. 2019. End-User Composition of Graphical User Interfaces by Composite Pattern. In *ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '19), June 18–21, 2019, Valencia, Spain.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3319499.3328236

1 INTRODUCTION

Visual editors for Graphical User Interfaces (GUIs) primarily share a similar editing metaphor [19, 25]: widgets are dragged from a palette and dropped onto a canvas to design the layout, or at least the part of the layout that can specified graphically. Supported widgets properties are then edited through forms. To complete the portion that is unsupported by the visual editor, developers have to engage the resulting GUI resource files and application code into event-based programming. This process is not adequate for end users, who are neither GUI experts nor skilled developers. However, end users do have some interaction experience that may be exploited to compose parts or wholes of existing GUIs into another, newly composed, GUI. Indeed, end users have already seen and experienced many GUIs, and they typically identify GUI fragments that they would like to reuse, usually from different interactive applications. For instance, they would like to compose a fragment they saw in one application, with a group of widgets they used in another application [7], perhaps with a style that is consistent to their own style guide. Visual editors may be used by end users, but are limited to the visual portion, without enabling them to specify GUIs further, and certainly not for composing fragments as described since GUI fragments are not logically defined for composition. With their experience, end users certainly benefit from a thorough comprehension of GUIs they use on a daily basis and for what they would like to see designed for them. Thus, why not empower users with a visual editor that enables them to compose GUI fragments by direct manipulation of widgets and their properties? For this purpose, this paper provides three contributions:

- ComPat (Fig. 1), a Java Swing visual editor for enabling end users to compose graphical user interfaces based on the composite pattern.
- An explanation of how ComPat implements the composite pattern for user interface composition.
- A definition of a User Interface Description Language tailored for graphical user interfaces composition based on a XML Schema Definition.

2 RELATED WORK

The Sushi User Interface Management System (UIMS) [21] is probably the first editor for GUI composition by observing that many tools help users creating new GUIs by compositing them out of smaller pieces. They introduce Workspaces, a model for compositing together various editors to manipulate sets of widgets and their properties which communicate in terms of a selected set and the properties possessed by widgets in that set. Many other tools have been introduced and tested for composing GUIs, such as [2–5, 10, 11, 14, 15, 17, 20, 23]. A convenient way to compare them is to classify

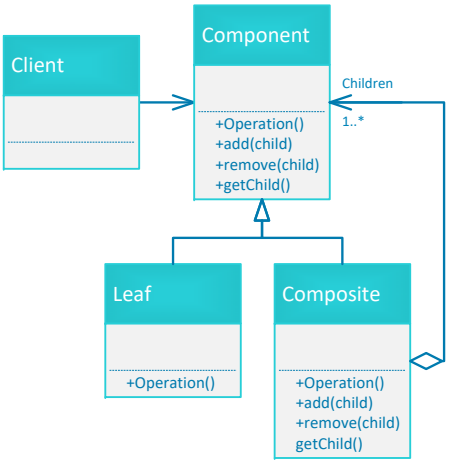

Figure 2: The original Composite Pattern.

them according to their level of the Cameleon Reference Framework (CRF) [6] they are working (*i.e.*, task and concepts, abstract user interface, concrete user interface, and final user interface), which initiated a design space [22]. Another survey [16] preferred to rely on the composition unit to compare various approaches. Comparing the capabilities of all these tools represents a significant amount of work that is beyond the scope of this paper. We refer to the literature for further details. For instance, Occello et al. [20] conducted some experiments to compare compositions produced by Model-Driven Engineering (MDE). Composability of widgets and their associated services is based on composition operators defined according to standard CRF abstraction levels [10], which demonstrates that composition can span over multiple levels of abstraction. The composition could also consider different interaction modalities, not just the graphical modality [17]. In most of these cases, composition is achieved by applying different types of composition operators, formally expressed in different languages (*e.g.*, web services, set theory [17], automated planning [11], feature model). Instead of relying on composition operators, ComPat supports composition by populating a grid with widgets or groups of widgets, which are considered as patterns. Patterns' definitions remain transparent for end users: since any widget or group of widgets is recreated by cut, copy, paste operations, no underlying pattern data base is maintained.

3 APPLYING THE COMPOSITE PATTERN

In Software Engineering, the *composite pattern* is a partitioning design pattern which treats a group of objects in the same way as a single instance of the same type of object. It is one of the Gang-of-Four Design patterns [8] aimed at composing objects into a tree to represent part-whole hierarchies. This pattern joins two benefits into one solution: a part-whole hierarchy is represented in a way that clients can treat part and whole objects uniformly in a tree structure.

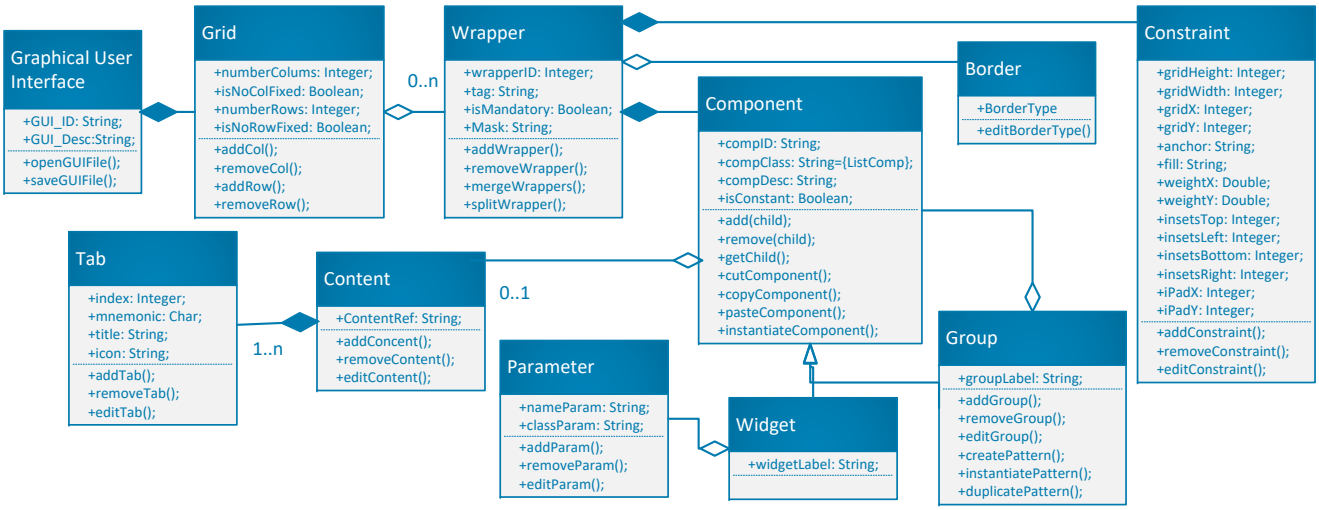

Figure 3: The Composite Pattern applied to composing graphical user interfaces.

This composite pattern could be implemented in two ways depending on how child-related operations (*e.g.*, adding a child to a container) are ensured:

(1) *Design for uniformity*: child-related operations are defined in the main `Component` interface, thus treating `Leaf` and `Composite` objects uniformly. But type safety is lost because clients can perform child-related operations on `Leaf` objects, thus provoking potential inconsistency.
(2) *Design for type safety*: child-related operations are defined only in the `Composite` class, thus preserving its application scope. `Clients` must treat `Leaf` and `Composite` objects differently. Type safety is preserved since clients cannot trigger child-related operations, thus preserving consistency.

We found this composite pattern particularly appropriate for composing GUIs for the same reasons: (1) a GUI is typically stored as a hierarchy of individual widgets (the `Leafs`) and groups of widgets (the `Groups`), which can then be decomposed recursively into other groups, (2) a group of widgets is defined as a new GUI design pattern that can be instantiated later on, thus fostering reusability and composition, and (3) design for uniformity is preferable for GUIs. The same design pattern has also been used for composing classifiers in meta-heuristics [26] and other domains.

Fig. 3 represents how GUIs are modelled for composition in ComPat by applying the composite pattern (Fig. 2). Any `Graphical User Interface` is associated to a `Grid`, which represents the `GridBagLayout`[1] layout manager that aligns wrappers by maintaining rectangular grid of cells. It requires two attributes indicating the number of columns and rows. Two Boolean attributes, *i.e.*, `isNoColFixed` and `isNoRowFixed` indicate whether the grid number of columns or rows is fixed or not. A `Grid` contains zero, one or many `Wrappers` characterizing how columns and rows are formatted, such as occupying one or more cells. A `Wrapper` basically contains instances of the `Component`, which is the class where the composite pattern is applied, thus resulting into simple `Widgets` and `Groups` of widgets treated in the same way. The `Constraint` class represents a `GridBagConstraint`[2] with their meta data and information for positioning the `wrapper`, such as coordinates inside the `Grid`. In particular, the `Anchor` attributes specifies directions towards which the cell is justified: Center, North, Northeast, East, Southeast, South, Southwest, West, and Northwest. The double attributes `weightX` and `weightY` define the horizontal weight and vertical weight. The `fill` attribute indicates how the component's wrapper will be resized to fill its wrapper based on four values: none, horizontal, vertical, or both. The internal and external padding have their four respective insets.

The `Component` captures all the widgets inside the display area by representing them as a list of valid widgets types along with their parameters. The `Parameter` element is made for specifying all the properties corresponding to a `Widget` class via a name (*e.g.*, ToolTip, Icon, Text, Mnemonic, Selected) and a class indicating the type of the parameter value (*e.g.*, `java.lang.Boolean`, `java.lang.String`). A `Content` element is present as a `Wrapper` child only if the `Component` inside this display area is a container, *i.e.*, widgets that can contain other widgets like split pane, grid, scroll pane, toolbar and tabbed pane. The `Tab` element represents one tab of a tabbed pane container. A `Tab` could hold a title, a mnemonic character (*e.g.*, 'A', 'B', '1'), and an icon taken from a palette. Finally, the `Border` element is

[1]See https://docs.oracle.com/javase/7/docs/api/java/awt/GridBagLayout.html

[2]See https://docs.oracle.com/javase/7/docs/api/java/awt/GridBagConstraints.html

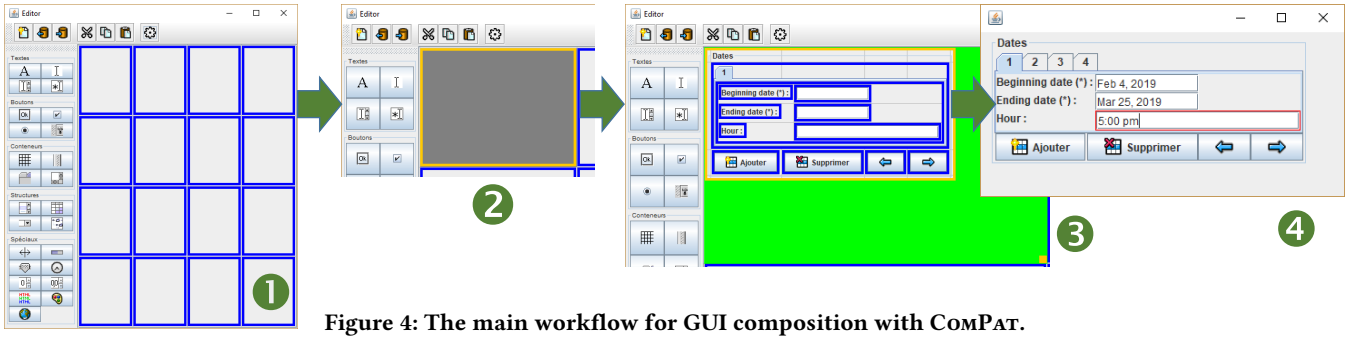

Figure 4: The main workflow for GUI composition with ComPat.

present in a `Wrapper` if its component has a border, whose type could be: `BevelBorder`, `TitledBorder`, `EmptyBorder` or `EtchedBorder`. The `Content` is the title of the border in the case of `TitleBorder`. The UML Class Diagram of Fig. 3 has been transformed into a XML Schema Definition (XSD) in order to check the conformity of any XML file based on this format. This User Interface Description Language (UIDL) is minimalist for this purpose since it only focuses on widgets and groups of widgets for GUI composition. Consequently, the CRF level corresponding to this UIDL is the Final User Interface for Java Swing. However, the structure may accommodate GUIs for other environments thanks to the generic `Widget` and `Group` of `Widgets` with their properties as parameters. Every created GUI can be edited, saved, the resulting output file is saved into a XML-compliant file. This file keeps the memory of the GUI structure and the configurations of each widget.

4 COMPAT SYSTEM WALKTHROUGH

Main Worflow

Fig. 4 graphically depicts the main workflow for creating, editing, and composing a GUI in ComPat. Based on the applied composite pattern (Fig. 3), ComPat relies on a grid structured into rows and columns (① in Fig. 4) that can be resized, moved, merged and split depending on the constraints imposed to the GUI layout. To determine the layout, a constraint-based approach similar to [18] has been developed: each row or column can be subject to application of local and global constraints to align their contents on any of the eight traditional compass directions plus the center: from North-East to South-West. These eight constraints are then automatically propagated to cells belonging to the row or column of interest, but can be edited further (⑤ in Fig. 5). Constraint-based approaches have been proved fruitful not only for expressing layout relationships, such as alignment, proportion, size, but also for using constraint solvers to compute admissible solutions to the Constraint Problem [13].

Once a cell is selected (② in Fig. 4), it welcomes any of the 25 widgets supported so far in ComPat and classified into five groups: four basic widgets (*i.e.*, label, single line edit field, multiple lines edit field, optionally with an input mask, and password field), four buttons (*i.e.*, push button, check box, radio button, toolbar), four containers (*i.e.*, grid, divider, tabbed dialog box, and bidirectional scrolling area), four input/output widgets (*i.e.*, list box, table, drop down list/combo box, and tree), and nine special widgets (*i.e.*, separator, progress bar, line with cursor or spin button, clock for date and hour, integer input field, real input field, HTML, multimedia object, and URL field). Once cells are filled in with their respective widgets, they can be grouped to form a higher-level widget in a container (③). At this stage, any widget, group of widgets, or group of groups can be defined as a new pattern that can be reused in the same project or in another. Thanks to the composite design pattern, this process is recursive: any group can be decomposed recursively into an arbitrary amount of sub-groups. Gridlines of the layout can be moved, resized, redefined to properly align widgets or groups of widgets according to the desired layout. All grouping relationships are visually materialized by blue containers and internally stored based on the composite pattern introduced in the previous section. For instance, ③ highlights a newly defined pattern in yellow which consists of a tabbed dialog box, which in turn contains several widgets. Since it is a tabbed dialog box, several push buttons were added to move across tabs: `Add` to add a new instance of the tab, `Delete` to remove a previously created tab, and <= and => to move across tabs. At any time, by clicking on the "Gears" icon in the toolbar, ComPat automatically generates a GUI in Java Swing corresponding to the specifications (④): this GUI can be immediately operated for rapid prototyping. Data can be input depending on the widgets selected and the behavior of widgets can be explored for End-User Development (EUD). Simple data domains can be attached to appropriate widgets, such as a list of countries to a list box, a table of zip codes.

Editing Properties of Composition

Fig. 5 represents complementary pathways to further edit properties of the 25 supported widgets. Most properties are editable in direct manipulation: the specification of each property directly changes the appearance of a widget. For instance, changing the layout of a widget, adding an alternate text in a tool tip, adding an icon, defining a short and a long

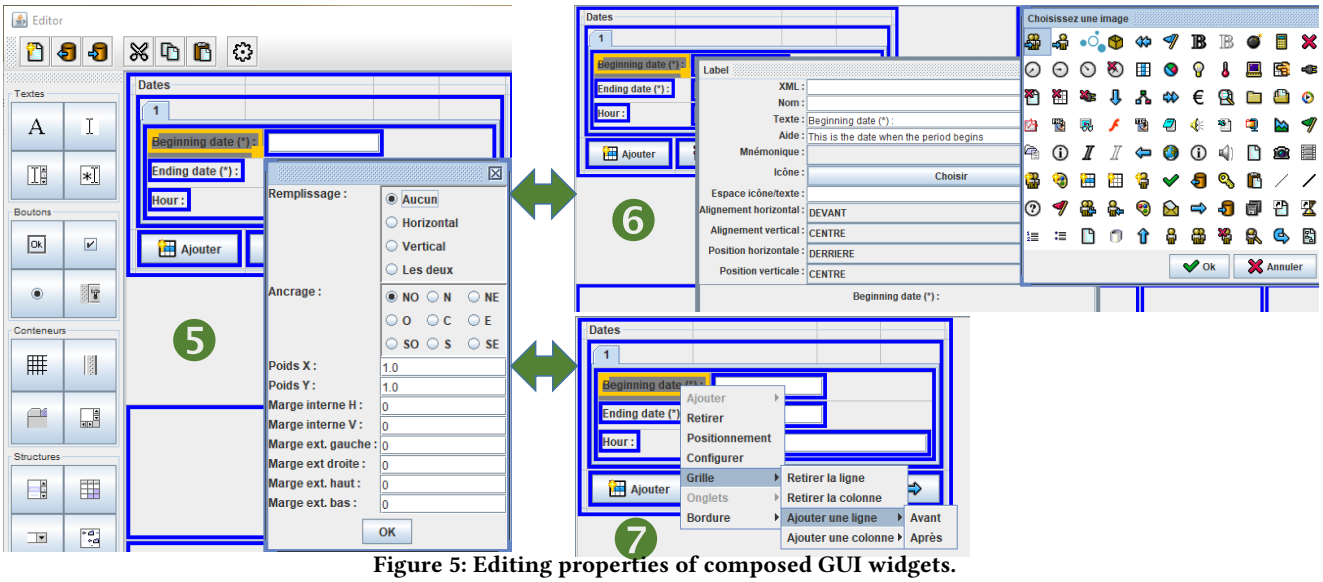

Figure 5: Editing properties of composed GUI widgets.

label for various resolutions are immediately propagated to the rendering of each widget. For each widget, three sets of properties can be edited by direct manipulation:

(1) *Cell local layout properties*, such as how the widget should fill the cell (*i.e.* none, horizontally, vertically or both), the nine aforementioned layout constraints which are first inherited from the container but that can be further edited at the local level, logical positions, and margins (see ⑤ in Fig. 5)

(2) *Widget general properties*, such as the short and long labels, the help text to be displayed in the tooltip, the keyboard shortcut, an optional icon extracted from a set of icons, positioning of data inside the widget (see ⑥). The bottom part of ⑥ in Fig. 5 provides a real-time rendering of the widget based on all properties edited so that the end user can immediately have feedback on how the widget will look like and to check its conformance with respect to requirements. This capability is ensured thanks to the reflexive programming mechanism implemented in ComPat. In general, *reflection* [24] is the ability of a computer program to examine, introspect, and modify its own structure and behavior at runtime. Here, Java reflection [9] has been exploited to ensure this mechanism and enable real-time rendering of every widget. For instance, "Beginning date (*)" is displayed with its final location, size, and proportion, with other properties like shortcut. Since the field is required, a "(*)" marker is added.

(3) *Pattern global layout properties* govern the layout constraints inside a pattern formed by a group of widgets.

Other properties are included for XML support and handling. For instance, a XML identifier can be assigned to any widget to further refer to it without any ambiguity.

Composition by Copy/Paste

The end user can copy any widget (*e.g.*, an edit field with its labels), group of widgets (*e.g.*, a date with an associated calendar), or group of groups (*e.g.*, a complete address group), then paste them into any other cell and compose by direct manipulation. At any time, a GUI is automatically generated based on the properties specified. For example, the group of widgets that was previously defined as a pattern (in ③ of Fig. 4) was copied and pasted into another cell. Any group of widgets defined as a pattern an be instantiated in two ways:

(1) *By direct instantiation*: the pattern is simply pasted into another cell and its original definition is simply transferred while preserving its definition. Modifying this instantiation is therefore not possible, thus preventing the end user to endanger the GUI consistency.

(2) *By rewriting*: the pattern is also instantiated and pasted into another cell, but the end user can rewrite its definition in the same way the original pattern was created. In this way, a new pattern can be created and later used as any other pattern.

In both cases, there is no need to re-edit cell or group properties since they will be inherited logically from their initial definition. Since these constraints are logically expressed, without making any reference to any particular system of coordinates, the new constraints are simply re-computed, re-evaluated and solved again. At any time, the newly composed GUI can lead to an automatic generation of a Java GUI to be used for prototyping, discussion, validation, or any other form of testing.

5 CONCLUSION

We presented ComPat, an open source Java Swing visual editor along with patterns examples and a demonstration.

This editor is aimed at supporting end users in creating, editing, and composing GUIs by cut, copy, and paste. The composition is transparent for the end users since no external representation of a pattern exists. Each created GUI can be stored with into a XML file conforming to the corresponding UIDL. In contrast to other UIDLs, such as extensive ones covering multiple levels of the CRF [6], this minimalist UIDL only keeps the structure of widgets with their properties.

Consequently, ComPat relies on a very simple composition mechanism: any widget, group of widgets, or group of groups, recursively defined, could initiate a composition, but this composition is mainly conducted with two basic operations: union and difference. As opposed to ComposiXML [17] which offers a set of 17 composition operations, ComPat focuses the composition efforts only on those two operations, which largely contributes to its simplicity appreciated by end users in the context of End-User Development (EUD). As opposed to Placid [10] which offers a composition ensured by an automatic planner, ComPat relies on the responsibility of the person who is composing, this leaving freedom, but no checking. ComPat automatically generates code for a Java Swing GUI, thus supporting rapid GUI prototyping, but not its code incorporation into the rest of the development. Consequently, any post-generation code modification is lost after re-generation: when the GUI model changes, the generated code changes accordingly, but does not keep track of any prior manual modification. Several solutions grouped under the umbrella of Round-Trip Engineering (RTE) [12] address this problem, such as Transformation templates [1] applied to any XML-based GUI description. The ComPat project is accessible through the GitHub directory https://github.com/jeanvdd/ComPat/.

ACKNOWLEDGEMENTS

The authors would like to thank Benoît Hambucken (DefiMedia, Contraste) for developing ComPat.

REFERENCES

[1] Nathalie Aquino, Jean Vanderdonckt, and Oscar Pastor. 2010. Transformation Templates: Adding Flexibility to Model-driven Engineering of User Interfaces. In *Proc. of SAC '10*. ACM, New York, 1195–1202.

[2] Carmelo Ardito, M. Francesca Costabile, Giuseppe Desolda, Rosa Lanzilotti, Maristella Matera, and Matteo Picozzi. 2014. Visual Composition of Data Sources by End Users. In *Proc. of AVI '14*. 257–260.

[3] Ch. Brel, Ph. Renevier-Gonin, A. Giboin, M. Riveill, and A.-. Dery. 2014. Reusing and Combining UI, Task and Software Component Models to Compose New Applications. In *Proc. of BCS-HCI '14*.

[4] Ch. Brel, Ph. Renevier-Gonin, Audrey Occello, Anne-Marie Pinna-Dery, C. Faron-Zucker, and M. Riveill. 2010. Application Composition Driven by UI Composition. In *Proc. of HCSE' 10*. 198–205.

[5] Ch. Brel, Ph. Renevier-Gonin, A.-M. Pinna-Dery, and M. Riveill. 2012. Application and UI Composition Using a Component-Based Description and Annotations. In *Proc. of SEAA '12*. IEEE Computer Society, Washington, DC, USA, 204–207.

[6] Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Laurent Bouillon, and Jean Vanderdonckt. 2003. A Unifying Reference Framework for multi-target user interfaces. *Interacting with Computers* 15, 3 (2003), 289–308. https://doi.org/10.1016/S0953-5438(03)00010-9

[7] A. Delgado, A. Estepa, J.A. Troyano, and R. Estepa. 2016. Reusing UI elements with Model-Based User Interface Development. *International Journal of Human-Computer Studies* 86 (2016), 48 – 62.

[8] Ralph Johnson Erich Gamma, Richard Helm and John Vlissides. 1995. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[9] Ira R. Forman and Nate Forman. 2005. *Java Reflection in Action.* Manning Publications Co., Shelter Island, NY, USA.

[10] Yoann Gabillon. 2015. User Interface Composition: Operators and Composability Checking. In *Proc. of IHM '15*. ACM, New York, USA, Article 13. https://doi.org/10.1145/2820619.2820632

[11] Yoann Gabillon, Gaelle Calvary, and Humbert Fiorino. 2014. PLACID: A Planner for Dynamically Composing User Interface Services. In *Proc. of IHM '14*. ACM, New York, USA, 123–129.

[12] Thomas Hettel, Michael Lawley, and Kerry Raymond. 2008. Model Synchronisation: Definitions for Round-Trip Engineering. In *Theory and Practice of Model Transformations*, A. Vallecillo, J. Gray, and A. Pierantonio (Eds.). Springer.

[13] Noreen Jamil. 2013. Constraint Solvers for User Interface Layout. *IJWA* 5, 3 (2013), 136–146. http://www.dline.info/ijwa/fulltext/v5n3/4.pdf

[14] Cédric Joffroy, Benjamin Caramel, Anne-Marie Dery-Pinna, and Michel Riveill. 2011. When the Functional Composition Drives the User Interfaces Composition: Process and Formalization. In *Proc. of EICS '11*. ACM, New York, NY, USA, 207–216.

[15] Effie Karuzaki and Anthony Savidis. 2015. Yeti: Yet Another Automatic Interface Composer. In *Proc. of EICS '15*. ACM, New York, 12–21.

[16] Kung-Kiu Lau and Tauseef Rana. 2010. A Taxonomy of Software Composition Mechanisms. In *Proc. of SEAA '10*. IEEE Computer Society, Washington, DC, USA, 102–110. https://doi.org/10.1109/SEAA.2010.36

[17] S. Lepreux, A. Hariri, J. Rouillard, D. Tabary, J.-Cl. Tarby, and Ch. Kolski. 2007. Towards Multimodal User Interfaces Composition Based on UsiXML and MBD Principles. In *HCI Intelligent Multimodal Interaction Environments*, Julie A. Jacko (Ed.). Springer, Berlin, 134–143.

[18] C. Lutteroth and G. Weber. 2008. Modular Specification of GUI Layout Using Constraints. In *Proc. of ASWEC '08*. 300–309. https://doi.org/10.1109/ASWEC.2008.4483218

[19] B. Michotte and J. Vanderdonckt. 2008. GrafiXML, a Multi-target User Interface Builder Based on UsiXML. In *Fourth International Conference on Autonomic and Autonomous Systems (ICAS'08)*. 15–22. https://doi.org/10.1109/ICAS.2008.29

[20] A. Occello, C. Joffroy, and A.-M. Dery-Pinna. 2010. Experiments in Model Driven Composition of User Interfaces. In *Proc. of DAIS'10*. Springer, Berlin, 98–111.

[21] Dan R. Olsen, Jr., Thomas G. McNeill, and David C. Mitchell. 1992. Workspaces: An Architecture for Editing Collections of Objects. In *Proc. of CHI '92*. ACM, New York, NY, USA, 267–272.

[22] Fabio Paternò, Carmen Santoro, and Lucio Davide Spano. 2011. *A Design Space for User Interface Composition.* Springer, Berlin, 43–65.

[23] Anthony Savidis and Constantine Stephanidis. 2010. Software Refactoring Process for Adaptive User-interface Composition. In *Proc. of EICS '10*. ACM, New York, NY, USA, 19–28.

[24] Brian Cantwell Smith. 1984. Reflection and Semantics in LISP. In *Proc. of POPL '84*. ACM, New York, USA, 23–35.

[25] L. A. Valaer and R. G. Babb. 1997. Choosing a user interface development tool. *IEEE Software* 14, 4 (July 1997), 29–39.

[26] John Woodward, Jerry Swan, and Simon Martin. 2014. The 'Composite' Design Pattern in Metaheuristics. In *Proc. of GECCO Comp '14*. New York, NY, USA, 1439–1444.