Designing and Implementing Software Systems using User-defined Design Patterns

Mert Ozkaya[1] and Mehmet Alp Kose[2]
[1]*Department of Computer Engineering, Yeditepe University, Istanbul, Turkey*
[2]*Altinbas University, Institute of Graduate Studies, Istanbul, Turkey*

Keywords: Design Pattern Definition, Pattern-Centric Modeling, Code Generation, UML.

Abstract: Software design patterns are the design-level solutions for the commonly occurring problems in software development. Design patterns are applied in many industries where problems repeat with slight changes, and applying the same solution that is proven to be quality reduces the development time and maximises the software re-use. *DesPat* is a modeling toolset that offers a modeling notation set based on UML's class diagram for the users to design their software systems using the well-known 6 design patterns proposed by Gamma et al. (abstract factory, singleton, composite, observer, visitor, and facade). *DesPat* also supports the combinations of different pattern models for any software system, analysis of the pattern-centric models, and their automated generation into Java skeleton code. In this paper, we extend *DesPat* with a new toolset that enables users to define their own patterns. A pattern is defined with the types of components, component interfaces, and relationships (i.e., generalisation, dependency, realisation, and composition). Any pattern definitions can then be imported into the *DesPat* modeling toolset, through which one may specify software design models in accordance with the pattern definitions, check the models against the pattern rules, and transform their models in Java. We illustrate our extension with the gas station case-study.

1 INTRODUCTION

The notion of patterns has been initiated by Christopher Alexander in the seventies, who is a famous architect and considered the use of patterns for constructing buildings. According to Alexander, many buildings essentially include the applications of the same or similar solutions (i.e., patterns) and this always lead to the buildings with beautiful architectures (Alexander, 1979; Alexander et al., 1975). Alexander published a seminal book, discussing 253 different patterns in terms of the problems and solutions that can be applied on the buildings, constructions, and towns (Alexander et al., 1977). A pattern essentially addresses a problem that are come across many times in different constructions and proposes an effective solution that leads to the beautiful architecture. In the mid-nineties, patterns have been considered to be applied for the software development to maximise the quality of the software to be developed by reusing the tested and proven high-quality design decisions while minimising the development cost. Software design patterns have first been considered systematically by Gamma et al.'s seminal book (Gamma et al., 1994), which has been shown great interest by both the academia and industry. Gamma et al. essentially proposed 23 different software design patterns which are categorised into three groups: structural, creational, and behavioural patterns. The structural patterns are concerned with composing large software systems out of smaller and simpler system components. The creational patterns are concerned with controlling the creation of the components composing the software systems. The behavioural patterns are concerned with how the components composing software systems should behave and interact with each other.

While many software development frameworks have been existing today, they do not aid in using design patterns at code level. Also, as discussed in Section 2, many design approaches have been proposed, which do not however provide adequate level of support for defining patterns, specifying pattern-centric models, checking their conformances, and generating code. So, we initially proposed a modeling toolset called *DesPat* (Ozkaya. and Kose., 2021), which enables to design software systems using Gamma et al.'s six design patterns that are believed to be among the top-used design patterns in industry (Zhang and Bud-

Ozkaya, M. and Kose, M.
Designing and Implementing Software Systems using User-defined Design Patterns.
DOI: 10.5220/0010571404970504
In *Proceedings of the 16th International Conference on Software Technologies (ICSOFT 2021)*, pages 497-504
ISBN: 978-989-758-523-4

gen, 2013). These are the abstract factory, singleton, composition, facade, observer, and visitor patterns. *DesPat* offers a graphical notation set for each pattern type that is based on UML's class diagram (Rumbaugh et al., 2004). *DesPat* enables to specify for any software system the pattern-centric models that can be combined by re-using the same components across different models. *DesPat* is supported with a modeling editor that has been developed using the Metaedit+ meta-modeling tool (Kelly et al., 2013). With *DesPat*'s modeling editor, users may specify pattern-centric models, analyse them against the pattern rules, and produce Java skeleton code.

However, *DesPat* in its current form supports a pre-defined list of patterns, which prevents users from defining their own solution domain and specifying models in that domain. So, in this paper, we propose an extension to *DesPat* for enabling users to define new patterns, specify pattern-centric models, and generate software code from their models. We developed a tool extension that provides users with an easy-to-use graphical user interface (GUI) for defining patterns in terms of component and interface types and their relationships. The extension tool transforms the pattern definitions into a meta-model in the context of *DesPat* that can be accepted in the Metaedit+ meta-modeling environment. The transformed meta-model herein not only includes the mapping into the *DesPat* notation set but also the rule definitions and translation algorithms in Java. So, upon importing a pattern definition into Metaedit+, one may obtain a modeling editor in Metaedit+ automatically, use the editor for specifying pattern-centric software models, checking the models for the pattern rules, and producing Java skeleton code automatically.

2 RELATED WORK

The literature includes several software modeling and design languages that can be used for designing software systems and performing useful operations (e.g., formal verification and code generation) before implementation. UML, for instance, has been considered as one of the top-used general-purpose design languages (Ozkaya, 2018b; Malavolta et al., 2012), which is supported with several tools and can be extensible via its extension mechanism (Ozkaya, 2019; Ozkaya, 2018a). While UML and its derivatives may be used for specifying pattern-centric software models, none of them enables pattern-specific operations such as checking their pattern conformance and generating pattern-specific code. Moreover, combining patterns (e.g., the system components playing different roles in different pattern models) and defining new patterns and using those definitions for designing software systems are not inside their scope either. Many other attempts, e.g., (Kim, 2015; Mak et al., 2004; Mapelsden et al., 2002; Hedin, 1997; Nicholson et al., 2009; Taibi and Ling, 2003; Mikkonen, 1998; Saeki, 2000), have been made towards the pattern definitions and specifications. Those approaches are essentially concerned with defining and using patterns for software design, combining patterns, checking the pattern conformances, verifying code against patterns, and formally verifying the pattern models. However, many of those approaches do not provide an up-to-date tool support that is available for use and thus remain as the proof-of-concept only. Also, those that support formal verification for the analysis of models require steep learning curve for many practitioners as a process algebra based notation set is imposed (Malavolta et al., 2012). Besides, the existing approaches require precise syntax and semantics to be used for defining patterns. While this is important for processing the definitions, novice practitioners find precise notations hard to use for quickly experimenting with defining patterns. Our approach discussed in this paper provides a graphical user interface that enables users to simply define the component/interface types and their relationships via some dialog boxes, lists, and menus without having to learn any pattern definition notations. We also enable the transformation of pattern definitions into precise meta-model.

3 OVERVIEW OF *DesPat*

DesPat offers a graphical modeling notation set for each design pattern supported, depicted in Figure 1. *DesPat*'s notation set is inspired from UML's class diagram (Rumbaugh et al., 2004), which many practitioners are familiar with (Ozkaya, 2018b; Malavolta et al., 2012). *DesPat*'s notation set for each pattern consists of a set of elements and their relationships. An element may be a component or an interface, where the former represents the composing unit of a software system and the latter represents the points of interaction for the components. Concerning the relationships, the interface realisation relationship is for specifying for a component the interface(s) that the component realises and each consists of the operations provided in the component's environment. The generalisation relationship is for specifying the commonalities among different components. The dependency relationship is for specifying which component may send/receive message to/from which other components. Note that a component may be dependent

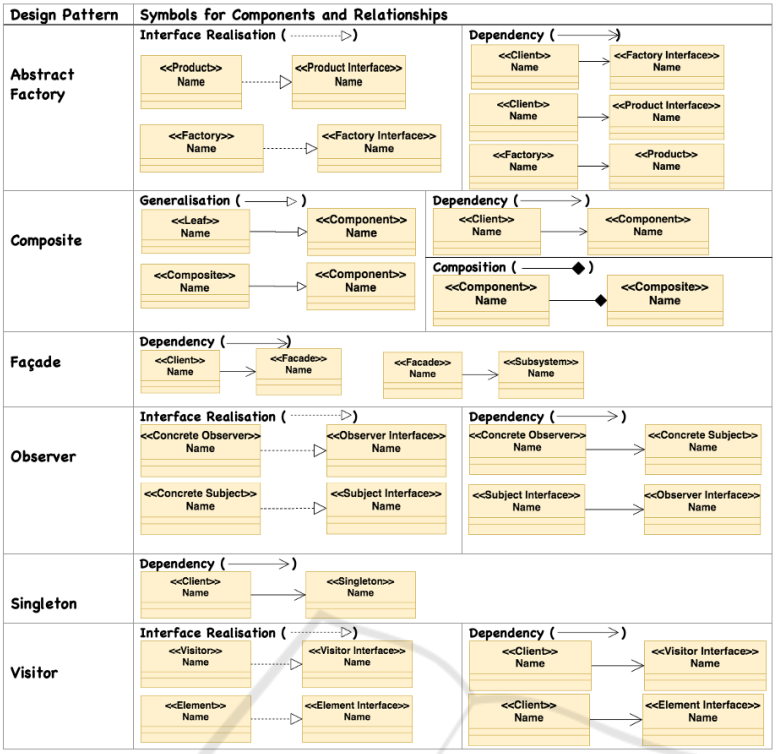

Figure 1: *DesPat*'s notation set.

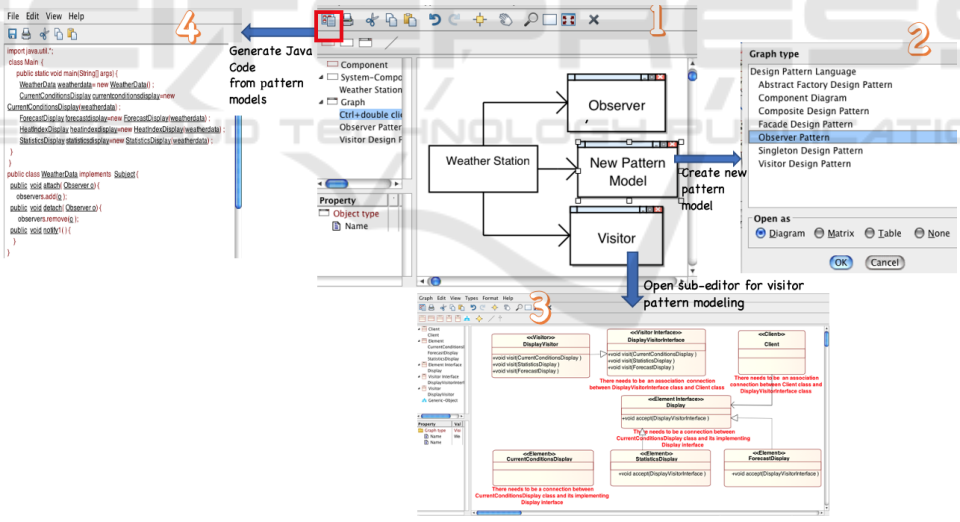

Figure 2: *DesPat*'s modeling editor.

on an interface of some component. The composition relationship is for specifying the whole-part relationship between components where one component is composed of some other component(s).

Figure 2 depicts *DesPat*'s modeling toolset[1]. So, users firstly specify for any software system under design a boxes&lines diagram as depicted in the editor (1), where a component representing the system is linked with a set of boxes each representing a different pattern model. Users may right-click on a pattern model box, which opens up a new dialog box as depicted in (2) for choosing one of the pattern types supported by *DesPat*. This prompts a new sub-editor as depicted in (3) where users specify the pattern models using *DesPat*'s notation set for the corresponding pattern. Lastly, users may click on the icon de-

[1] *DesPat*'s web-site: https://sites.google.com/view/despat

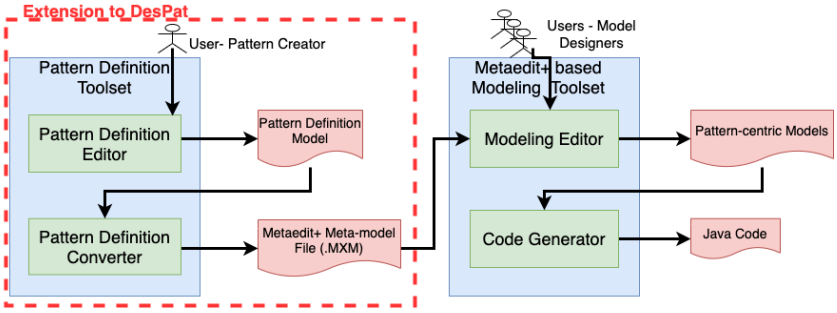

Figure 3: *DesPat*'s extended tool architecture.

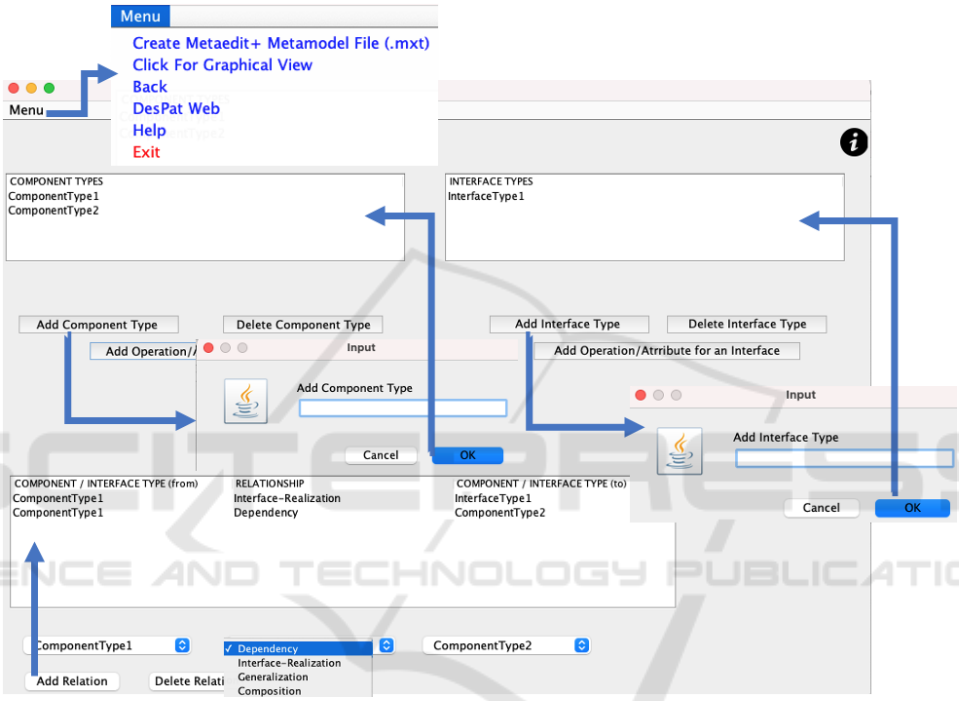

Figure 4: Defining a design pattern via the pattern definition editor.

picted in (4) to generate a Java implementation from the pattern model(s) of the software system under design. Note that in (3), the pattern model includes red-colored texts that are the warning messages which are generated automatically at modeling time when the pattern models violate the pattern rules.

4 DEFINING DESIGN PATTERNS

As depicted in Figure 3, *DesPat*'s modeling toolset has been extended with a *pattern definition toolset*, whose goal is to let users define new patterns that can be used via *DesPat*'s modeling toolset for specifying software design models in accordance with the pattern rules and produce Java implementation automatically.

The pattern definition toolset includes an editor and converter, which have been developed in Java and are each supported with a graphical user interface (GUI). The toolset can be downloaded as a standalone *jar* file via *DesPat*'s project web-site[1].

4.1 Pattern Definition Editor

The pattern definition editor is accessible via a GUI that offers a menu for the following activities: *(i)* opening an existing pattern definition, *(ii)* creating a pattern definition, *(iii)* browsing *DesPat*'s webpage, and *(iv)* opening the user manual document.

Upon choosing to create a new pattern definition, a GUI that is depicted in Figure 4 is displayed, through which users define the component and interface types and relationships between the component and interface types. We consider four types of relationships that are supported by *DesPat* and discussed

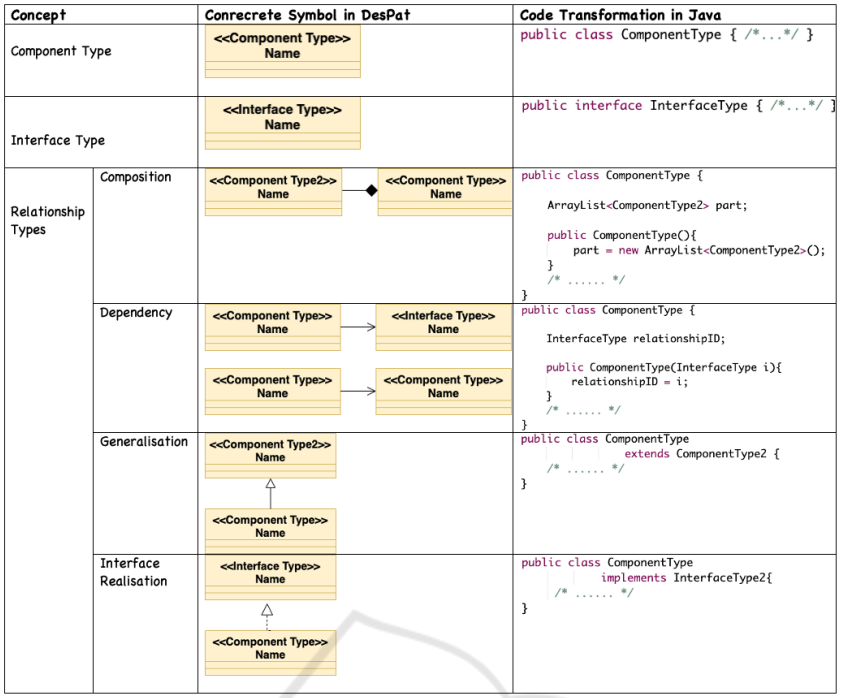

Figure 5: Mapping pattern definition concepts with *DesPat*'s meta-model (concrete symbols and Java transformation).

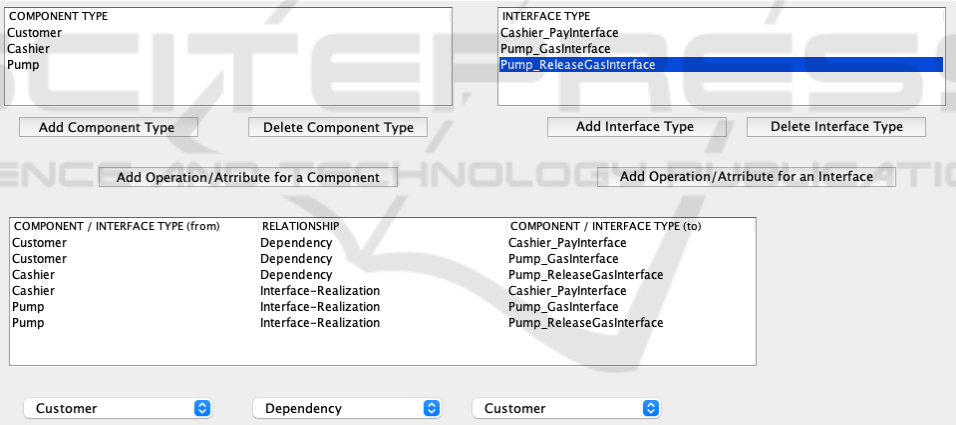

Figure 6: The gas station pattern definition via the pattern definition editor.

in Section 3, which are interface realisation, composition, generalisation, and dependency. So, to use any pattern defined, a software system needs to be structured in terms of those components and interfaces in a way that meets the relationship constraints. For any component/interface type, one may also define the attributes and operations. Whenever a component/interface type is selected via the GUI and the corresponding button (e.g., *add operation*) is clicked in Figure 4, a new GUI appears for defining the operations/attributes. While operations are defined in terms of their return type, name, and parameter lists, attributes are defined in terms of name and default value.

Opening an existing pattern definition is performed via another GUI (similar to Figure 4) that enables to choose one of the existing patterns and edit/delete the existing component/interface type(s) and relationship type(s) of the chosen pattern definition.

For any pattern definition opened/created, one may click from the menu of the corresponding GUI (see "Menu" in Figure 4) to export the pattern definition into a Metaedit+ meta-model file as discussed in the next sub-section. Also, one may click from menu to display the graphical visualisation of the pattern definition, which opens up a UML class diagram that describe the component and interface types and their

relationships. So, this may help in understanding and communicating the pattern definitions.

Lastly, any patterns created are stored persistently on the user machine using the SQLite library[2], through which the patterns may be accessed and edited later via the "open" activity.

4.2 Pattern Definition Converter

The pattern definition converter works as integrated with the pattern definition editor. Indeed, one may run the converter by clicking from the menu of the open/create GUIs, as discussed above. Given any pattern definition created via the editor, the converter produces a meta-model definition for the Metaedit+ environment. The produced meta-model is in the *.mxt* XML file format, which can be imported in Metaedit+. The meta-model here includes *(i)* the mapping of the concepts and relationships defined in the pattern into the concrete symbols in the *DesPat* notation set introduced in Section 3, *(ii)* the definitions of the warnings that are raised upon the violations of the pattern relationships, and *(iii)* the code-generator definition for producing Java skeleton code from any given model that satisfies the pattern definition. Figure 5 essentially shows how the pattern definition concepts are mapped in *DesPat* and transformed in Java.

Upon importing the produced meta-model file in Metaedit+, a Metaedit+-based modeling editor is obtained automatically for specifying software design models in accordance with the pattern definition. The modeling editor herein enables for checking models against the pattern definitions at modeling time and raise any violations (e.g., missing relationships missing concepts) with precise warning messages. The modeling editor is also integrated with a code generator that is produced from the Java mapping embedded inside the meta-model file. The code generator herein can be used for generating a Java skeleton code from any model specified with the editor.

5 CASE-STUDY: GAS STATIONS

To illustrate *DesPat*'s extension, we consider defining a pattern for the gas station systems (Naumovich et al., 1997). Any gas station systems may be represented with different configurations, while each system is expected to have a varying number of instances of the same types of components that interact with each other under certain protocols. So, the pattern definition imposes three types of components – customer, cashier, and pump. Each customer makes a payment request to a cashier and then makes a gas request to a pump. Upon receiving the payment request, the cashier may make a release-gas request to pump for that customer. The pump may then accept the customer's gas request.

5.1 Meta-model Definition

Figure 6 shows the gas station pattern defined via the pattern definition editor. Therein, the customer, cashier, and pump component types and also a set of interface types are defined. As specified for the relationships, components interact with each other via interfaces. So, cashier realises a customer-interface which is used by the customers to send payment requests to cashiers. Pump realises a cashier-interface which is used by the cashiers to request pump to release gas for the customers who already made the payment. Pump also realises a customer-interface which is used by the customers to request gas after payment.

We also defined some operations for the interface types. Cashier's cashier-interface includes the *pay* operation for making payment, pump's customer-interface includes the *gas* operation for receiving gas request, and pump's cashier-interface includes the *releaseGas* operation for receiving gas-release request.

5.2 Model Specifications

We used the pattern converter to obtain a modeling editor in Metaedit+ for our pattern definition. Using the modeling editor, we specified the gas station model shown in Figure 7, which consists of two customers, one cashier, and one pump that interact with each other in accordance with the pattern relationship rules. Note that the interface operations depicted are essentially the operations defined in the pattern and thus come with the interface specifications by default.

In Figure 8, we added a new customer component to the same configuration (i.e., *cust3*), and the modeling editor indicates warning messages (shown with red-colored texts). The warning herein is due to that the dependency relationships defined in the pattern are not satisfied with *cust3*. Indeed, *cust3* needs to be related to the *CashierInterface* and *PumpInterface_Cust*, as defined in the pattern in Figure 6.

5.3 Model Transformation

Having specified the gas station model depicted in Figure 7 in accordance with the pattern definition given in Figure 6, we used the code generation tool

[2]SQLite: https://sqlite.org/

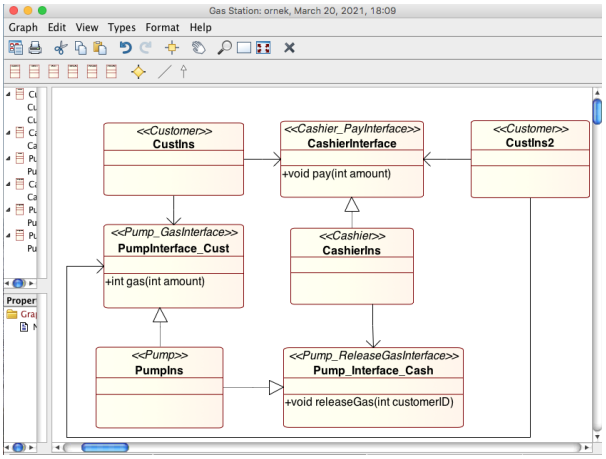

Figure 7: The gas station specification in Metaedit+.

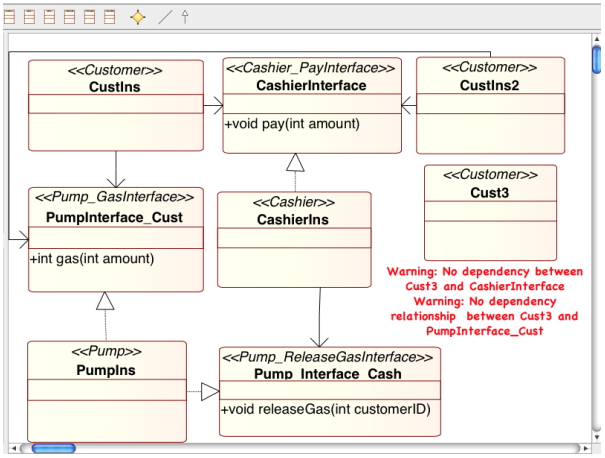

Figure 8: Detecting pattern rule violation at modeling time.

```
1  public class CustIns{
2      CashierInterface var1;
3      PumpInterface_Cust var2;
4      public CustIns(CashierInterface arg1,
5                     PumpInterface_Cust arg2){
6          var1 = arg1;
7          var2 = arg2;
8      }
9  }
10 public class CustIns2{
11     CashierInterface var1;
12     PumpInterface_Cust var2;
13     public CustIns(CashierInterface arg1,
14                    PumpInterface_Cust arg2){
15         var1 = arg1;
16         var2 = arg2;
17     }
18 }
19 public class CashierIns implements
20                     CashierInterface{
21     PumpInterface_Cash var1;
22     public CashierIns(PumpInterface_Cash arg){
23         var1 = arg;
24     }
25 }
26 public class PumpIns implements
27                PumpInterface_Cust, PumpInterface_Cash{
28
29 }
30 interface CashierInterface{
31     void pay(int amount);
32 }
33 interface PumpInterface_Cust{
34     int gas(int amount);
35 }
36 interface PumpInterface_Cash{
37     void releaseGas(int customerID);
38 }
```

Customer Definition

Customer Definition

Cashier Definition

Pump Definition

Interface Definitions

Figure 9: The Java skeleton code transformed from the gas station specification in Metaedit+.

integrated with the modeling editor and obtained the Java skeleton code in Figure 9. So, the Java skeleton code includes a separate class definition for each component and an interface definition for each interface specified in the gas station model. Note that the definition of a class that depends on some other class/interface includes a variable for storing a reference to the dependent class/interface. Indeed, customer includes variables for sending method-calls to the cashier and pump components over their interfaces and cashier includes a variable for the pump interface. Also, the interface definitions include the method definitions as specified in the (meta-)model.

ACKNOWLEDGEMENT

This work was supported by a project of the Scientific and Technological Research Council of Turkey (TUBITAK) under grant 120E144.

6 CONCLUSIONS

DesPat has been proposed for specifying software design models using Gamma et al.'s six well-known design patterns, which are composite, singleton, visitor, observer, abstract factory, and facade. *DesPat* offers a graphical notation set inspired from UML's class diagram, which is believed to be familiar to many practitioners in industry. So, given any software system under design, users may use the *DesPat* modeling editor to draw class diagrams for different patterns supported, combine the pattern models by re-using the same components (i.e., classes) in different pattern models, and analyse the pattern models according to the pattern rules at modeling time. *DesPat*'s modeling toolset also transforms the pattern models of any software system into Java skeleton code.

In this paper, we introduced an extension to *DesPat* for the capability of specifying software design models according to the user-defined patterns. We developed a toolset that enables for defining software design patterns in terms of the component and interface types, and the relationships between them. So, any software system can be designed by using as many instances of the component and interface types as needed in a way that satisfies the relationship rules. We also developed a converter that takes any pattern definition and produces the meta-model file in XML which can be imported in the Metaedit+ meta-

modeling environment. Metaedit+ then provides a modeling editor for specifying pattern-centric models, analysing models, and generating Java skeleton code. We illustrated our extension toolset with the well-known gas station system architecture.

In the future, we will improve our pattern definition notation to support different aspects of software design such as the behaviour and interaction. We will also extend *DesPat* so as to enable the isolated components that are not involved in any pattern to be specified as part of the system design and interact with the components of the pattern-centric models and other isolated components.

REFERENCES

Alexander, C. (1979). *The Timeless Way of Building*. Oxford University Press.

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., and Angel, S. (1977). *A Pattern Language: Towns, Buildings, Construction*. Center for Environmental Structure (Book 2). Oxford University Press.

Alexander, C., Silverstein, M., Angel, S., Ishikawa, S., and Abrams, D. (1975). *The Oregon Experiment*. Oxford University Press, New York.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software*. Addison Wesley. ISBN-13: 978-0201633610.

Hedin, G. (1997). Language support for design patterns using attribute extension. In Bosch, J. and Mitchell, S., editors, *Object-Oriented Technology, ECOOP'97 Workshop Reader, ECOOP'97 Workshops, Jyväskylä, Finland, June 9-13, 1997*, volume 1357 of *Lecture Notes in Computer Science*, pages 137–140. Springer.

Kelly, S., Lyytinen, K., and Rossi, M. (2013). Metaedit+ A fully configurable multi-user and multi-tool CASE and CAME environment. In Jr., J. A. B., Krogstie, J., Pastor, O., Pernici, B., Rolland, C., and Sølvberg, A., editors, *Seminal Contributions to Information Systems Engineering, 25 Years of CAiSE*, pages 109–129. Springer.

Kim, D. (2015). Design pattern based model transformation with tool support. *Softw. Pract. Exp.*, 45(4):473–499.

Mak, J. K. H., Choy, C. S. T., and Lun, D. P. K. (2004). Precise modeling of design patterns in uml. In *Proceedings. 26th International Conference on Software Engineering*, pages 252–261.

Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., and Tang, A. (2012). What industry needs from architectural languages: A survey. *IEEE Transactions on Software Engineering*, 99.

Mapelsden, D., Hosking, J., and Grundy, J. (2002). Design pattern modelling and instantiation using dpml. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications*, CRPIT '02, page 3–11, AUS. Australian Computer Society, Inc.

Mikkonen, T. (1998). Formalizing design patterns. In Torii, K., Futatsugi, K., and Kemmerer, R. A., editors, *Forging New Links, Proceedings of the 1998 International Conference on Software Engineering, ICSE 98, Kyoto, Japan, April 19-25, 1998*, pages 115–124. IEEE Computer Society.

Naumovich, G., Avrunin, G. S., Clarke, L. A., and Osterweil, L. J. (1997). Applying static analysis to software architectures. In Jazayeri, M. and Schauer, H., editors, *ESEC / SIGSOFT FSE*, volume 1301 of *Lecture Notes in Computer Science*, pages 77–93. Springer.

Nicholson, J., Gasparis, E., Eden, A. H., and Kazman, R. (2009). Verification of design patterns with lepus3. In Denney, E., Giannakopoulou, D., and Pasareanu, C. S., editors, *First NASA Formal Methods Symposium - NFM 2009, Moffett Field, California, USA, April 6-8, 2009*, volume NASA/CP-2009-215407 of *NASA Conference Proceedings*, pages 76–85.

Ozkaya, M. (2018a). Analysing uml-based software modelling languages. *Journal of Aeronautics and Space Technologies*, 11(2):119–134.

Ozkaya, M. (2018b). Do the informal & formal software modeling notations satisfy practitioners for software architecture modeling? *Information & Software Technology*, 95:15–33.

Ozkaya, M. (2019). Are the UML modelling tools powerful enough for practitioners? A literature review. *IET Softw.*, 13(5):338–354.

Ozkaya., M. and Kose., M. (2021). Despat: A modeling toolset for designing and implementing software systems using design patterns. In *Proceedings of the 16th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE,*, pages 251–260. INSTICC, SciTePress.

Rumbaugh, J., Jacobson, I., and Booch, G. (2004). *Unified Modeling Language Reference Manual, The (2Nd Edition)*. Pearson Higher Education.

Saeki, M. (2000). Behavioral specification of GOF design patterns with LOTOS. In *7th Asia-Pacific Software Engineering Conference (APSEC 2000), 5-8 December 2000, Singapore*, pages 408–415. IEEE Computer Society.

Taibi, T. and Ling, D. N. C. (2003). Formal specification of design pattern combination using BPSL. *Inf. Softw. Technol.*, 45(3):157–170.

Zhang, C. and Budgen, D. (2013). A survey of experienced user perceptions about software design patterns. *Inf. Softw. Technol.*, 55(5):822–835.