Teaching Design-by-Contract for the Modeling and Implementation of Software Systems

Mert Ozkaya
Department of Computer Engineering, Yeditepe University, Istanbul, Turkey

Keywords: Java Modeling Language, Defensive Programming, Design-by-Contract, Education.

Abstract: Defensive programming is considered as a software design approach that promotes the reliable software development via the considerations of different cases for the software modules. Design-by-Contract (DbC) applies defensive programming systematically in terms of contracts that are a pair of pre-conditions on the module input and post-conditions on the module output. In this paper, a DbC-based teaching methodology is proposed, which aims to teach undergraduate students how to use contracts for the modeling and implementation of software systems. The teaching methodology consists of three consecutive steps. Firstly, the students will learn how to model software architectures in terms of components and their communication links. The component behaviours are specified as contracts, which are attached to the messages that the components exchange. In the second step, the students will learn how to implement the contractual software architectures in Java using Java's assertion mechanisms in a way that the contractual design decisions are all preserved in the code. Lastly, the students will learn the Java Modeling Language for combining the contractual modeling and Java implementation in a single model to avoid any inconsistencies between the model and implementation and automatically verify the correctness of the implementation against the modeled behaviours.

1 INTRODUCTION

Defensive programming is a software design technique for ensuring that software systems are developed with the considerations of different cases that the systems can be in. This is also considered as implementing the software modules in a way that the different input cases for a module are always checked before the module operates and the output is generated accordingly after the module call. By doing so, the reliability of software systems can be enhanced as the failures in unexpected cases will be minimised. Meyer proposed the Design-by-Contract (DbC) approach in the early nineties for addressing defensive programming in a systematic way (Meyer, 1992). DbC is based on the notion of contracts, which describes the obligations and benefits for the client and supplier of a module. That is, whenever the client satisfies some obligations on a software module, the supplier of the module guarantees some benefits for the clients. DbC is formally based on the Hoare logic (Hoare, 1969), which defines a contract for a software module in terms of the pre- and post-conditions. The Hoare logic states that whenever the caller (i.e., client) of the software module satisfies the pre-condition, the module computation can be performed and then the post-condition shall be satisfied by the supplier of the software module.

To apply DbC in software development, Meyer proposed Eiffel (Severance, 2012) as a contract-based language for analysing, designing, implementing, and maintaining object-oriented software systems. Following Eiffel, Java Modeling Language (JML) (Leavens et al., 2008; Cok, 2011) has been proposed for combining the contractual specifications with the Java implementation. Spec# (Barnett et al., 2011) applies contracts for the C# programming language. Also, there are some contract-based software architecture description languages, such as XCD (Ozkaya and Kloukinas, 2014), CBabel (Rademaker et al., 2005), and RADL (Reussner et al., 2003).

In this paper, the goal is to teach students how they can use DbC in their software development to enhance the reliability of software systems by means of considering the different cases of the software modules and closing the gap between the software model and implementation. So, a new teaching methodology is aimed to be proposed that prompts the students to model their software architectures contractually and implement their software systems in a way that meets the contractual model and enables the use of any tools

Ozkaya, M.
Teaching Design-by-Contract for the Modeling and Implementation of Software Systems.
DOI: 10.5220/0007950904990507
In *Proceedings of the 14th International Conference on Software Technologies (ICSOFT 2019)*, pages 499-507
ISBN: 978-989-758-379-7

for checking the consistency between model and implementation. The proposed teaching methodology consists of three consecutive stages. First, the students are taught how to specify the contractual software architectures in terms of components that interact with each other by sending/receiving messages whose behaviours are specified as contracts. Second, the students are taught how to implement the contractual software architectures in Java where the message contracts are preserved in code via the Java assertion mechanism. Third, the students are taught to use the JML language for combining the contractual modeling with Java programming so as to minimise any inconsistencies between the model and implementation and thus maximise the reliability of software systems.

In the rest of the paper, the similar works are introduced initially. That is followed by the introduction of the stages that comprise the proposed teaching methodology. Next, the application of the methodology on the gas station case-study is discussed. Lastly, the evaluation of the methodology via an undergraduate course is discussed.

2 RELATED WORK

The literature includes many studies that focus on teaching software engineering or its sub-disciplines (e.g., software modeling and software testing). These studies propose new teaching methodologies, share teaching experiences, or use some well-known techniques for teaching software engineering. However, none of them focus on teaching how to model and implement software systems using the DbC approach.

Concerning the studies on software engineering, in (Claypool and Claypool, 2005), the authors extended the software engineering course syllabus at the University of Massachusetts, Lowell with game development to maximise the students' understanding on the software engineering concepts, project management facilities, and computer game design. In (Gnatz et al., 2003), the authors proposed the involvement of customers in software projects performed by the students and thus make them better understand the real concerns of software projects, including the communication, technical, management, and documentation issues. In (Alfonso and Botia, 2005), the authors promoted the agile process model for the software engineering course and encourage students to develop software projects iteratively in terms of the inception, elaboration, construction, and transition stages. In (LeJeune, 2006), the author taught a group of students the extreme programming features such as user stories, change management, testing, refactoring, etc.

Concerning the courses on software modeling, in (Tamai, 2005), the author discussed how to teach multiple software modeling techniques (e.g., UML, ER modeling, Petri net, and state chart) that promote the modeling of software systems with components and their relations but differ in the level of abstractions and the models being dynamic or static. In (Starrett, 2007), the author focused on teaching high-school students the essense of modeling and abstraction via UML for improving their problem solving skills. In (Powell, 2001), the author focused on teaching modeling for the management science students to make them capable of describing the solutions for large and complex problems as abstract models and reasoning the models via informal/formal techniques. In (Moisan and Rigault, 2010), the authors shared their experiences on teaching UML to the students and practitioners. Many concerns have been addressed, including the different modes of using UML, UML's acceptance by the students and practitioners, the case-studies to be used, and UML's diagrams types and their usabilities for teaching software modeling. In (Carrington, 1998), the author offered a teaching methodology that focuses on the software design and testing. Carrington promotes the application of the Fusion methodology proposed by Hewlett Packard for teaching software design. In (Börstler et al., 2012), the author focused on determining and analysing the existing problems in teaching software modeling and proposed some improvements on the teaching methodologies for software modeling.

Concerning the courses on software testing, in (Edwards, 2003), the author proposed a web application called Web-CAT, through which students can submit their Java programs and JUnit test-cases for testing their programs against the test-case specifications. The author also surveyed 59 students and observed that Web-CAT helped students enhancing their software testing skills. In (Jones, 2000), the author discussed a methodology for teaching software testing effectively via his SPRAE framework that defines the principles which a student taking a software testing course should learn. In (Mao, 2008), the author proposed a course structure for software testing that consists of four different parts: preparatory testing practice, question-driven classroom testing, experiments in lab, and enhanced training in real context.

3 PROPOSED TEACHING METHODOLOGY

The students are assumed to have taken the software engineering course previously and posses the ade-

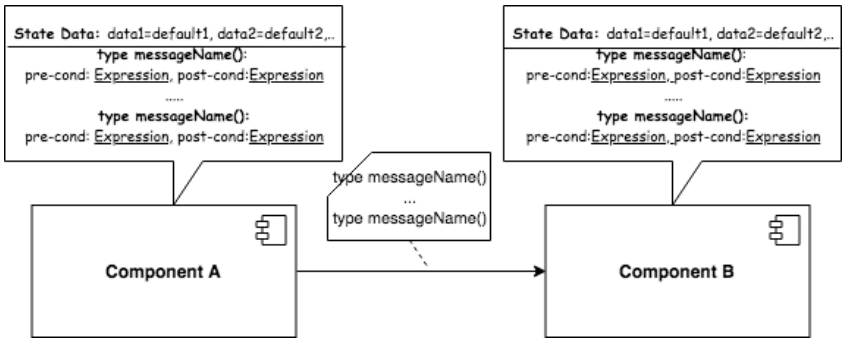

Figure 1: Modeling the contractual software architectures.

quate level of knowledge on the software requirements specification and architecture design in UML.

3.1 Software Architecture Modeling

In this stage, the students are aimed to be taught with how to use contracts for the modeling of software architectures. Given the software requirements descriptions in any natural languages (e.g., English), the students are expected to model the software architectures in terms of the components and their communications first. Here, the students are free to use any visual modeling notations that they are comfortable with (e.g., UML, SysML, or simple boxes and lines drawing). After modeling the components and their communications, the students are expected to determine for each communication link between any two components the list of messages that the two components exchange via the link. As depicted in Figure 1, the message list specification for each communication link needs to be attached to the link to indicate which messages are transferred via that link. Each message can be specified in terms of a return type (if the message includes a response to be sent back), message name, and the list of parameters (if any). The messages in this list are each to be sent (or requested) by one component to the other component that the communication link is directed towards (i.e., component A requests the messages from component B).

For each message that the two linked components exchange, the expected behaviours of the components need to be specified in a precise and understandable way. So, the DbC approach is followed for that purpose and each component notation is expected to be attached with a specification as depicted in Figure 1, which describes the component state data and the message contracts. The component state data are represented with a list of variables and their default (i.e., initial) values. The message contracts for a component include a single contract specification for each message that the component exchange via its communication link(s), and that contract is specified with pre- and a post-conditions. Whenever the pre-condition is satisfied for the message, the component can send/receive the message, and then, the post-condition is expected to hold. Note that pre-condition is defined over the component (pre-)state and message parameters, while post-condition is over the component (post-)state and the message result (if any).

3.2 Model Transformation

3.2.1 Model to Java Transformation

After modeling the contractual software architectures, the goal is to implement the architecture models in a way that preserves all the design decisions specified in the architecture model. The Java programming language is considered here, which is one of the top popular programming languages used in industries, and most universities offer mandatory courses on Java.

```
1  public class component.Name {
2    FORALL data ∈ component.DataList
3      data.Type data.Name = data.InitialValue;
4
5    FORALL link ∈ component.CommunicationLinkList
6      IF link IS_REQUEST
7        link_INTERFACE link.Name;
8      FORALL message ∈ link.MessageList
9        public message.Return message.Name(message.ParameterList){
10         //pre-condition
11         assert(message.pre-condition);
12         //computation ..........
13         IF link IS_REQUEST
14           link.Name.{message.Name(message.ParameterList)};
15         //post-condition
16         assert(message.post-condition);
17       }
18 }
```

Listing 1: Translating system components in Java.

Implementing the contractual software architecture models in Java is expected to be performed via the algorithms given in Listings 1 and 2, which are considered for the components and communication links respectively. As shown in Listing 1, each component is implemented as a Java class, which consists of an instance variable for each class data (lines 2-3) and a class method for each message that the component receives/sends (lines 9-17). The component

class methods consist structurally of three parts: the pre-condition, computation, and post-condition. The pre- and post-conditions herein are translated from the contract specifications of the component messages. Whenever the method is called, the pre-condition needs to be satisfied first. If so, the computation can be performed, and the post-condition needs to be satisfied at the end. If either the pre- or post-condition is violated, the method-call execution fails. The pre- and post-conditions are implemented using Java's assertion mechanism[1], which can be used in Java programs for checking any condition and throwing an error if the condition is not met. Note that if a method is translated from a request message, its computation part in the method body delegates the call to the receiving component's class instance via the link instance variable (line 14). The link instance variable used herein is to be instantiated with the class instance of the link that connects the receiving and requesting components (line 7). Thanks to the link instance variable, the classes of the components that are linked are essentially decoupled from each other, and their instances may communicate only via the class instance of the communication link, which acts as the adapter between the linked components' class instances.

```
1  FORALL link ∈ component.CommunicationLinkList
2   public interface link.Name_INTERFACE {
3    FORALL message ∈ link.MessageList
4      public message.Return message.Name(message.ParameterList);
5   }
6
7   public class link.Name_CONNECTOR implements link.Name_INTERFACE {
8    link.ReceiverComponent receiverComponent;
9
10   public link.Name_CONNECTOR(link.ReceiverComponent receiver ){
11         receiverComponent = receiver;
12    }
13    FORALL message ∈ link.MessageList
14      public message.Return message.Name(message.ParameterList){
15         receiverComponent.{message.Name}();
16      }
17  }
```

Listing 2: Translating communication links in Java.

The communication links are implemented following the adapter design pattern. As shown in Listing 2, each communication link is implemented as an adapter class (lines 7-17). So, the link class implements an interface that comprises the definitions of the methods which correspond to the message list for that communication link (lines 2-5). The adaptor link class implements the interface methods (lines 14-16) in a way that the methods direct the calls to the receiver component class instance (i.e., adaptee) which is received via the class constructor (lines 10-11).

[1]Java assertion statement: https://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html

```
1  public class component.Name {
2    FORALL data ∈ component.DataList
3      data.Type data.Name = data.InitialValue;
4
5    FORALL link ∈ component.CommunicationLinkList
6      IF link IS_REQUEST
7        link_INTERFACE link.Name;
8      FORALL message ∈ link.MessageList
9        /*@public normal_behavior
10           @requires: message.pre-condition
11           @ensures: message.post-condition
12       @*/
13       public message.Return message.Name(message.ParameterList){
14         IF link IS_REQUEST
15           link.Name.{message.Name(message.ParameterList)};
16       }
17  }
```

Listing 3: Translating system components in JML.

3.2.2 Model to JML Transformation

Maintaining consistencies between the contractual specification and the Java implementation is also an issue and needs to be addressed for reliable software systems. Note that any changes made on the contractual design may be neglected for the Java implementation; and this will make the design and implementation inconsistent. To minimise the gap between model and implementation, the students are encouraged to learn and use JML, which is one of the top-used software modeling languages in industry (Ozkaya, 2018) and allows for combining the contractual software model with the Java implementation. In JML, the Java class methods are annotated with the contract specifications that are attached at the top of the method declarations. Thanks to the verifier tools that support JML (e.g., KeY (Ahrendt et al., 2016) and OpenJML (Cok, 2011)), the Java programs can be formally verified for the contractual JML specifications.

Since JML is essentially the extension of the Java language, the same rules discussed in Section 3.2.1 can be used by students to model their software systems in JML. That is, a separate class is generated for each component and connector link and each class has the same structure as given in the Java transformation algorithms in Listings 1 and2. However, while the Java translations use Java's assert statements for checking contracts, the JML translation includes the contract annotations attached to the class methods. Listing 3 gives the modified algorithm for transforming the component specifications in JML. JML contracts are specified as comments (/*@ ... @*/) that are attached to the head of the class methods. JML essentially offers an expressive notation set for modeling contracts[2], but for simplicity, only the basic JML constructs are considered. The JML constructs considered herein are the *normal_behaviour* construct for describing the normal case for a method behaviour that is supposed to end with a successful

[2]JML reference manual: http://www.eecs.ucf.edu/ leavens/JML/jmlrefman/jmlrefman_toc.html

termination, the *requires* pre-condition, *ensures* post-condition, and *result* construct for representing the result to be returned after the method-call.

4 A CASE STUDY - GAS STATION SYSTEM

The author considers the gas station system (Naumovich et al., 1997) as the case-study for demonstrating the use of the teaching methodology introduced in Section 3. Given its appropriateness for the DbC-based software modeling and simplicity, the gas station system may further be used in the classrooms as the medium for making the students understand the teaching methodology. Indeed, the students can initially be provided with the informal requirements of the gas station system (in textual, plain English format) and expected to use DbC for modeling and implementing the gas station architecture in the way introduced in the rest of this section that consider the translation algorithms discussed in Section 3.

4.1 Software Architecture Modeling

Given the textual description of the gas station system, the students may specify the the system components and their communication links as the author did in Figure 2. So, the gas station system comprises the customer, cashier, and pump components. The customer interacts with the cashier to make the gas payment. Upon receiving the payments, the cashier asks the pump to release gas for the customers. Then, the customer may receive the gas paid from the pump.

The next step is to extract the behaviour descriptions and document them contractually as given in Figure 2. So, the customer's pay message contract states that the pay message may only be sent if the *requestMade* data of the customer component is *false* and the *amount* parameter is in equal to *1*. After the customer requests the *pay* message, the component state must be updated so that the *requestMade* data is set to *true* and the *chosenAmount* data is set to the *amount* parameter. Customer's *pump* request is performed if the *requestMade* data is *true* (i.e., the customer already made the pay request). Then, after the pump message is requested, if the result to be returned is equal to the *chosenAmount* data, the *requestMade* data can be set back to *false* again so as to perform new payment requests. Note that the pump message is specified with a return type (i.e., *int*) in Figure 2.

The cashier component receives and accepts the *pay* message only if the component's *paymentAmount* data is *0* (i.e., the pre-condition). After the *pay* message is accepted and the necessary computations are performed, the *paymentAmount* data must be set to the amount parameter of the message (i.e., post-condition). The cashier component may request the *releasePump* message if the *paymentAmount* data is not equal to *0* and the amount parameter is equal to the *paymentAmount* data (i.e., the pre-condition). That is, before the releasePump message is requested, the pay message request needs to be received first and the amount of the gas to be released must be equal to the amount paid for. After the *releasePump* message is requested, the *paymentAmount* data must be set to *0* (i.e., post-condition). Note that this allows for receiving the *pay* message again.

Lastly, the pump component receives and accepts the *releasePump* message if the component's *pumpReleased* data is *false* (i.e., pre-condition). Then, after the message is received, the *pumpReleased* data must be set to *true* and the *paymentAmount* data set to the *amount* parameter. The pump component receives and accepts the *pump* message if the *pumpReleased* data is *true* and the *paymentAmount* data is not *0* (i.e., the releasePump message received first). Then, after the *pump* message is received, the *pumpReleased* data must be set to *false* and the result to be returned back to the customer must be set to *paymentAmount* data.

4.2 Model Transformation

After documenting the contractual gas station architecture as given in Figure 2, the students may use the algorithm rules discussed in Section 3.2.1 to implement their architecture specification in Java in a way that the behavioural design decisions for the components are all preserved. The students may further modify the Java implementation in accordance with the rules discussed in Section 3.2.2 so as to obtain the JML model for the gas station. Figure 3 depicts how the author implemented the customer component specification in Java and transform the Java code in JML. Also, Figure 4 shows how the author implemented the communication links specified for the customer and cashier components in Java, which essentially act as the adapter between the components. Indeed, in Figure 3, the class instance for customer uses the link instance for making a method-call to the class instance for cashier. The author's full Java code and JML model for the entire gas station architecture can be found in the web-site of the software testing course[3], which has been used for evaluating the teaching methodology as discussed in the next section.

[3] Software testing course web-site: https://sites.google.com/site/drmertozkaya/softwaretest

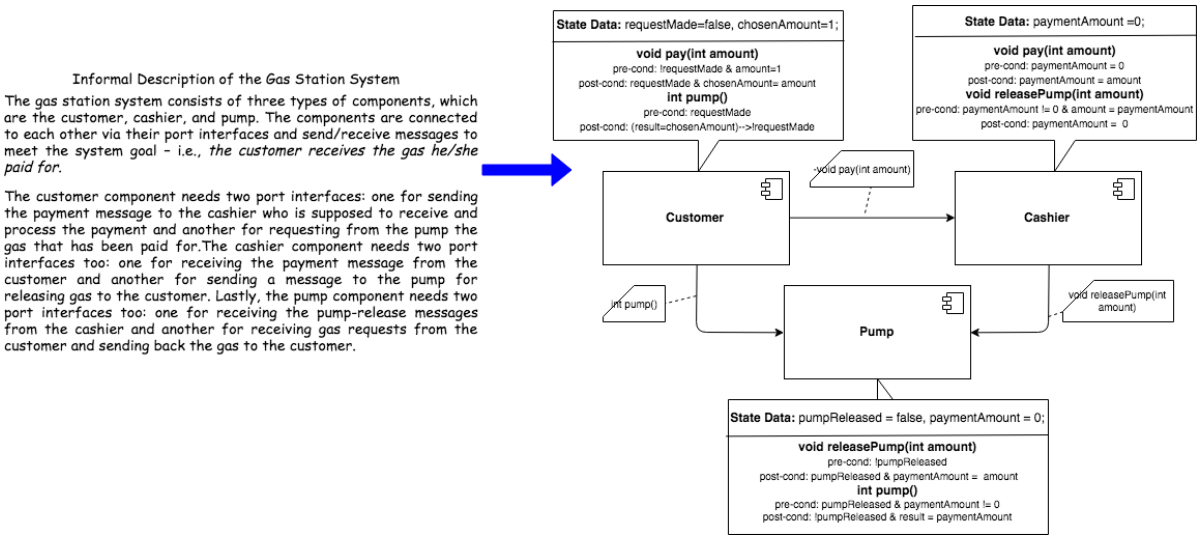

Figure 2: The contractual specification of the gas station system.

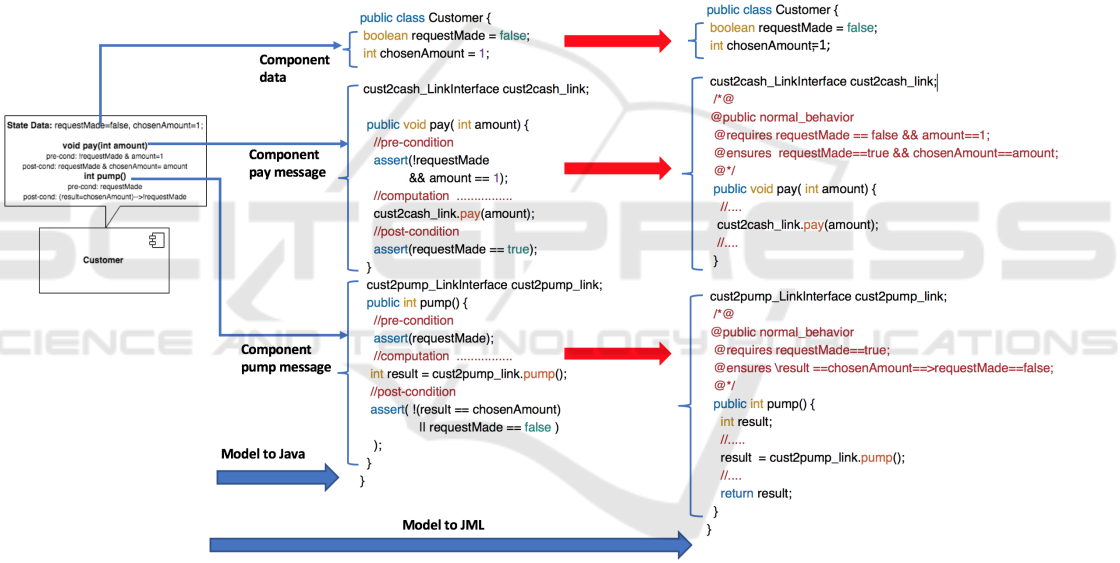

Figure 3: Transforming the customer component specification in Java and JML.

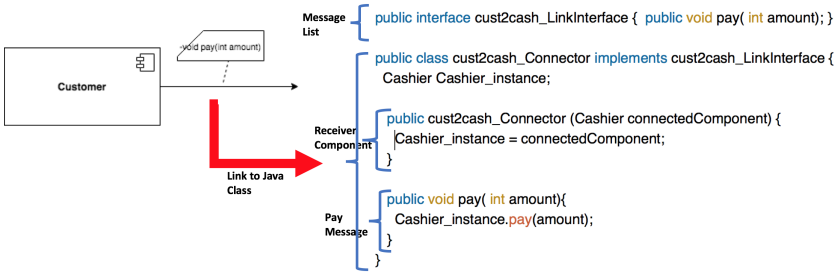

Figure 4: Transforming the link specification between customer and cashier components in Java and JML.

5 PRELIMINARY EVALUATION

The DbC-based teaching methodology discussed in this paper has been used in the software testing course that are offered by the author as a technical elective course at Yeditepe University, Istanbul for the 3rd and 4th (final) year computer science (CS) un-

Table 1: The course syllabus for teaching how to design and implement reliable software systems.

Week	Activity	Total Hours	Teaching Method
1	Introduction to Defensive Programming and DbC	1	Lecture Slides
1	Jazequel and Meyer's article entitled "Design by Contract: The Lessons of Ariane" (Jézéquel and Meyer, 1997)	1	Reading and interactive discussion
2	Introduction to Modeling with DbC	1	Lecture Slides
2	Modeling with DbC via the Gas Station System	2	Guided Class Practice
1	Introduction to Java's Assertion Mechanism	1	Lecture Slides
3	Introduction to Implementing DbC-based Models in Java with Assertions	2	Lecture Slides
3, 4	Implementing the Gas Station System in Java	2	Guided Class Practice
4, 5	Introduction to JML	3	Lecture Slides
5	Modeling the Gas Station System in JML	2	Guided Class Practice

dergraduate students. Note that the software engineering course is a pre-requisite for the software testing course. The software testing course has been enrolled by 19 Yeditepe CS students in Autumn 2018. The course syllabus has been enriched with the 5-weeks long materials shown in Table 1, which has been taught in 15 hours in total (i.e., 3 hours per week)[3]. So, the goal here is to introduce students the important concepts of model-driven software development (e.g., modeling software architectures, DbC, and model transformation) and make them gain some practical experience on using DbC for the modeling and implementing software systems.

As indicated in Section 3, the defensive programming and DbC topics have been introduced initially. To attract the students' interest further, Jazequel et al.'s article (Jézéquel and Meyer, 1997) on the DbC approach has been studied and discussed in the classroom in an interactive way, which aid in illustrating the importance of DbC via the Ariane 5 space shuttle explosion. Following that, a 1-hour introductory session has been conducted on the DbC-based architecture modeling with the goal of enhancing the students' capability of using contracts for the specifications of architectural components as discussed in Section 4.1. Then, the students have been involved in a guided class exercise for the contractual modeling of the gas station system architecture. Note here that the students have been initially given the informal, half-page description of the gas station system requirements in English and free to use any modeling tools (or draw on a paper). In the next step, another 1-hour introductory session has been conducted on the Java assertion mechanism so as to teach the use of assertions in Java. This has been followed with a 2-hours session for teaching how to implement the contractual software models in Java using Java's assertion mechanism in accordance with the algorithm rules discussed in Section 3.2.1. Then, the students have been expected to implement their gas station models in Java with its assertion mechanism. Lastly, JML's notation set has been introduced to the students, and the students have been taught with how to transform any Java program with assertions into a JML model with contract annotations that needs to satisfy the translation rules discussed in Section 3.2.2. Then, in another 2-hours practice session, the students have been asked to combine their gas station architecture model and its Java implementation into a JML model this time. After modeling with JML, the students used the OpenJML tool (Cok, 2011)[4] to verify their JML models for the gas station example and check exhaustively whether their Java program satisfies their contractual specifications. Note that OpenJML also provides a free online verification tool that saves the students from having to download and install any applications.

During the course, the students' feedback have been received regularly via the Q&A sessions that have taken place in the last 15 minutes of each session. Also, at the end of the course, a quick survey has been conducted among the students so as to understand their thoughts on the taught topics. So apparently, none of the students have ever heard about the important concepts taught in this course, including defensive programming, DbC, and model transformation. Also, the students stated that they have not considered so far any means of enhancing the reliability of the software systems. The students' understanding on software reliability is basically restricted with testing software implementation for a set of expected inputs, and other relevant topics such as model checking, verifying software implementation for the model have not been studied before. While the students are all familiar with Java, none of them ever used Java's assertion mechanisms before. The students stated that they viewed a software module as a unit of computation only without considering to check for any pre- or post-conditions. JML is also completely new to the students. As the students completed each lecture session given in Table 1, they have been observed to show a great interest to the contract-based modeling and the ability of JML in combining the contractual modeling with implementation for promoting the reliable software systems that are guaranteed to satisfy the design decisions. The students got highly impressed with how easy it is to model software behaviours in terms of contracts that simply require specifying conditional statements without the

[4]OpenJML's web-site: https://www.openjml.org/

need to learn and use any complex techniques. JML's tool support for the exhaustive checking of the Java code for the contract specifications via formal methods has also been so interesting to the students. Indeed, the capability of checking all possible execution paths is a great facility to the students. However, some students are quite concerned that JML's notation set is more than just simple pre- and post-conditions and requires some learning curve. The students were keen to visually specify the JML models and verify, and implement the models in Java automatically, rather than writing textual specifications as is the case with JML.

Concerning the students' experiences observed, most of the students suffered from obtaining a contractual software architecture model from the requirements statement of the gas station system. While the students were all able to specify the structural aspect of gas station, they found it quite challenging to figure out the component behaviours. Even if the students managed to determine the messages that the components exchange, they hardly determined how the messages impact on the component state. So, the author provided further clarifications on the requirements and helped the students grasp some implicit details. Moreover, most of the students were not so good at translating the software architectures into software implementation manually in accordance with the rules given. This was essentially due to the lack of knowledge on the adapter design pattern that the translation rules are based on.

6 CONCLUSION

In this paper, a novel teaching methodology has been proposed for the undergraduate students to learn how to use DbC for the modeling and implementation of reliable software systems. The teaching methodology comprises three steps to be performed. The students are initially taught how to specify the contractual behaviours of the components that compose the software architectures. That is, each component notation is enriched with the contractual behaviours of the messages that the component exchanges. In the second step, the students are taught how to implement their contractual software architectures in Java in accordance with a simple algorithm to be followed that uses the Java assertions for checking the message contracts. To avoid any potential consistency issues between the software model and implementation, the methodology further proposes in its last step the algorithm for combining the contractual modeling of a software with its Java implementation via the use of the JML language, which is also supported with many verifier tools that can automatically prove the correctness of the Java programs for the contractual models.

To evaluate the DbC-based teaching methodology, the author extended the syllabus of his undergraduate software testing course at Yeditepe University and the students have been taught with the necessary materials along with a guided class exercise on the gas station example in autumn 2018. According to the students' feedback received, the topics such as defensive programming, DbC, and JML seem to be completely new to them and the students got really surprised on how easy it is to model the behaviours of software modules contractually. The students also showed a great level of interest towards JML and its capabilities for verifying the Java implementation against the contractual models. However, the students have been observed to face some difficulties in extracting the behavioural design decisions from the requirements statement and also expressed their desire on using JML via some visual notation set.

While the teaching methodology introduced in the paper got so many positive feedback, the lack of language and tool support highly limit its applicability. So, we have currently been developing a software modeling language and its toolset for teaching students the important concepts of model-driven software development such as the software architecture modeling, DbC, multiple-viewpoints modeling, model analysis, and code generation. The new language will allow the students to use a simple boxes-and-lines based visual notation set for modeling software architectures from multiple viewpoints, which are the use-case, logical, behaviour, and physical viewpoints. Note that the behaviour viewpoint is inspired from the contractual behaviour modeling discussed in this paper. The students will also be able to use the toolset for performing many operations on their architecture models. These include checking the consistency between different viewpoint models, checking the model completeness, transforming the models into Java and JML in accordance with the algorithms discussed in the paper, and transforming the models into any formalisms for formal verification. The new toolset is expected to be evaluated via the software engineering, software architecture, and software testing undergraduate courses offered at Yeditepe University.

REFERENCES

Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P. H., and Ulbrich, M., editors (2016). *Deductive Software Verification - The KeY Book - From Theory to*

Practice, volume 10001 of *Lecture Notes in Computer Science*. Springer.

Alfonso, M. I. and Botia, A. (2005). An iterative and agile process model for teaching software engineering. In *18th Conference on Software Engineering Education Training (CSEET'05)*, pages 9–16.

Barnett, M., Fähndrich, M., Leino, K. R. M., Müller, P., Schulte, W., and Venter, H. (2011). Specification and verification: The spec# experience. *Commun. ACM*, 54(6):81–91.

Börstler, J., Kuzniarz, L., Alphonce, C., Sanders, W. B., and Smialek, M. (2012). Teaching software modeling in computing curricula. In *Proceedings of the Final Reports on Innovation and Technology in Computer Science Education 2012 Working Groups*, ITiCSE-WGR '12, pages 39–50, New York, NY, USA. ACM.

Carrington, D. (1998). Teaching software design and testing. In *FIE '98. 28th Annual Frontiers in Education Conference. Moving from 'Teacher-Centered' to 'Learner-Centered' Education. Conference Proceedings (Cat. No.98CH36214)*, volume 2, pages 547–550 vol.2.

Claypool, K. and Claypool, M. (2005). Teaching software engineering through game design. *SIGCSE Bull.*, 37(3):123–127.

Cok, D. R. (2011). Openjml: Jml for java 7 by extending openjdk. In Bobaru, M., Havelund, K., Holzmann, G. J., and Joshi, R., editors, *NASA Formal Methods*, pages 472–479, Berlin, Heidelberg. Springer Berlin Heidelberg.

Edwards, S. H. (2003). Teaching software testing: Automatic grading meets test-first coding. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, pages 318–319, New York, NY, USA. ACM.

Gnatz, M., Kof, L., Prilmeier, F., and Seifert, T. (2003). A practical approach of teaching software engineering. In *Proceedings of the 16th Conference on Software Engineering Education and Training*, CSEET '03, pages 120–, Washington, DC, USA. IEEE Computer Society.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.

Jézéquel, J.-M. and Meyer, B. (1997). Design by contract: The lessons of ariane. *Computer*, 30(1):129–130.

Jones, E. L. (2000). Software testing in the computer science curriculum – a holistic approach. In *Proceedings of the Australasian Conference on Computing Education*, ACSE '00, pages 153–157, New York, NY, USA. ACM.

Leavens, G. T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Mller, P., Kiniry, J., and Chalin, P. (2008). *JML Reference Manual*. Iowa State University, 1.220 edition.

LeJeune, N. F. (2006). Teaching software engineering practices with extreme programming. *J. Comput. Sci. Coll.*, 21(3):107–117.

Mao, C. (2008). Towards a question-driven teaching method for software testing course. In *2008 International Conference on Computer Science and Software Engineering*, volume 5, pages 645–648.

Meyer, B. (1992). Applying "Design by Contract". *IEEE Computer*, 25(10):40–51.

Moisan, S. and Rigault, J.-P. (2010). Teaching object-oriented modeling and uml to various audiences. In Ghosh, S., editor, *Models in Software Engineering*, pages 40–54, Berlin, Heidelberg. Springer Berlin Heidelberg.

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.

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

Ozkaya, M. and Kloukinas, C. (2014). Design-by-contract for reusable components and realizable architectures. In Seinturier, L., de Almeida, E. S., and Carlson, J., editors, *CBSE'14, Proceedings of the 17th International ACM SIGSOFT Symposium on Component-Based Software Engineering (part of CompArch 2014), Marcq-en-Baroeul, Lille, France, June 30 - July 4, 2014*, pages 129–138. ACM.

Powell, S. G. (2001). Teaching modeling in management science. *INFORMS Transactions on Education*, 1(2):62–67.

Rademaker, A., de O. Braga, C., and Sztajnberg, A. (2005). A rewriting semantics for a software architecture description language. *Electr. Notes Theor. Comput. Sci.*, 130:345–377.

Reussner, R., Poernomo, I., and Schmidt, H. (2003). Reasoning about Software Architectures with Contractually Specified Components. In Cechich, A., Piattini, M., and Vallecillo, A., editors, *Component-Based Software Quality*, volume 2693 of *Lecture Notes in Computer Science*, page 287?325. Springer Berlin Heidelberg.

Severance, C. (2012). Bertrand meyer: Software engineering and the eiffel programming language. *Computer*, 45(9):6–8.

Starrett, C. (2007). Teaching uml modeling before programming at the high school level. In *Seventh IEEE International Conference on Advanced Learning Technologies (ICALT 2007)*, pages 713–714.

Tamai, T. (2005). How to teach software modeling. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 609–610.