How Do I Know If My Design Is Correct?

J Paul Gibson, Eric Lallet, Jean-Luc Raffy [1,2,3]

Le département Logiciels-Réseaux (LOR),
Institut National des Télécommunications,
Evry, France

Abstract

Teaching software engineering students about design is very challenging. In general, students will learn about design through a module teaching a graphical modelling language. Our experience shows that this can result in students learning how to represent and comprehend designs but having very little understanding of design as a process. When reviewing design artefacts, students often ask whether the designs are *good*. This leads to the realisation that there is lack of understanding of the fundamental question of whether a design can be said to be *correct*. Of course, the notion of *correctness* will generally be covered by another module, typically called "formal methods". Unfortunately, our experience also shows that formal methods courses can lead to students learning how to build formal models — much like they would build programs — without achieving a good understanding of nondeterminism and abstraction; and without seeing how formal methods can help in the process of design. In this paper, we argue that the teaching of software design needs to be better integrated with the teaching of formal methods. We give some concrete examples of how this can be done.

Key words: Design, Correctness, UML, Formal Methods.

1 Introduction

One of the least well understood aspects of software development is the role of design in bridging the gap between *what* (requirements) and *how* (implementation). Inexperienced software designers fail to treat design as a process, and as a consequence become experts in representing the (static) artefacts using models/languages but fail to master the evolution of design.

[1] Email: paul.gibson@int-evry.fr
[2] Email: eric.lallet@int-evry.fr
[3] Email: jean-luc.raffy@int-edu.eu

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

During the transition from procedural to object-oriented programming languages, there was a realisation that the boundary between design and implementation was becoming even more blurred. In a controversial article in the C++ Journal, Reeves[18], stated: "...about ten years ago I came to the conclusion that, as an industry, we do not understand what a software design really is. I am even more convinced of this today." Reeves goes on to argue that considering the source code as being the design overcomes one of the fundamental issues associated with software design: how can we be sure that it will work correctly?

> "...when real engineers get through with a design, no matter how complex, they are pretty sure it will work. They are also pretty sure it can be built using accepted construction techniques. In order for this to happen, hardware engineers spend a considerable amount of time validating and refining their designs."

These ideas have much more resonance when we consider recent growth in agile development[14].

In fact, the notion that the design is not finished until it has been coded and tested is not, as it would seem at first sight, at odds to a formal approach to software design. In a formal approach, designs are coded (using formal specification languages) and they are tested and refined. Unfortunately, teaching formal methods to software engineers is no guarantee that they will use them during design! Ken Robinson[20] identifies a clear problem with the teaching of formal methods: "It is frequently the case that the other courses make no reference to, or use of, the formal techniques studied in the Formal Methods course."

In this paper we argue that it is the responsibility of the teachers of formal methods to incorporate aspects of *all* other software engineering courses in their teaching (not just design). However, the focus of work in this paper is in the integration of formal methods and design, with specific examples given using UML[7] [4].

2 UML approach: the strengths and weaknesses

The main strength of UML is that it is the standard OO modelling language; with comprehensive tool support and plentiful educational resources. However, it has been openly criticised by a number of high-profile software engineers. For example, we need look no further than Bertrand Meyer for a satirical article that identifies the main weaknesses of UML[16]. In the same spirit one should read the entertaining yet insightful article by Alex Bell[6] which shows the dangers in expecting the adoption of UML to automatically improve your software development process. The UML has been extended with

[4] In our teaching, we use B[1] for formalising aspects of design. In this paper we do not give B models, but make reference to the B-method, refinement and correctness by construction.

a more formal notation in the guise of the OCL. Expressions written in OCL offer a number of benefits: most importantly, they should make the graphical model more precise and more detailed[19]. However, the shortcomings of OCL have been well documented for a number of years[22].

We are not the first to identify the risks of replacing a general software design course with a course on UML. Engels et al. make the point quite simply[11]: "The incorporation of UML in a curriculum can and should not happen by adding a separate UML course. UML is nothing but a means to reach an end (the end in this case being the expression of software models)." We note that, in the above quotation, one can replace all instances of the string "UML" with the string "formal methods" and the resulting sentence is another which we believe to be true!

3 B approach: the strengths and weaknesses

B is a method [1] for specifying, designing and coding software systems. The concept of refinement [5] is the key notion for developing B models of (software) systems in an incremental way. B models are accompanied by mathematical proofs that justify them. We start from an abstract model and each subsequent model is a refinement of the previous one. Proofs of properties of B models help to convince the user (designer or specifier) that the (software) system is correct, since they demonstrate that the behaviour of the last, and most concrete, system (software) respects the behaviour of the first, most abstract model (which we assume has already been validated).

The main advantage of B is that it focuses attention on the core role of design: moving from abstract to concrete, through a process of refinement that guarantees the correctness of design decisions. The B method has evolved into Event-B, with an associated methodology and industrial strength tools[2], all of which have already been used for teaching formal software engineering[3].

A weakness of teaching B is that students find it difficult to model the system and its components abstractly. The B approach requires students to think in terms of what is required and to start by building abstractions of these high-level requirements. It also needs them to refine their abstract specifications into concrete implementations. Students can manage to do this for simple case studies but fail to appreciate how this can scale up to larger design decisions. They also fail to see how formal proof could be useful to them in their day-to-day design work. There is a mental block between the type of designs they see when using UML and the type of designs they see when working with B. They like the way UML facilitates their visualisation of structural properties. In short, they feel more confortable and confident working with pictures than working with mathematical formalism.

4 Formal Design: integrated teaching

There are many potential benefits to integrating UML with formal methods; the idea is not new[12] but continues to be a challenging topic of research, for example: [8,23,17]. From an educational viewpoint, this research is mature enough to transfer back to our teaching. However, the question of how this integration should be done is one that requires further research. We have experimented with four types of integration:

- *Case-study-driven* - continue to teach UML and formal methods as separate modules but glue them together through common case studies;
- *Formalising UML* - Extend the UML module with material focussing on the OCL and the integration of formal languages;
- *UMLing your formal method* - Extend your formal methods module to show how the models can be specified in an object oriented fashion;
- *Teach Formal (OO) Design* - focus on design as a process and use a range of notations to illustrate design activities.

It is beyond the scope of this paper to analyse these options. However, in the next sections, we give an example of the type of design problem that can be used in any of these teaching approaches.

5 An Educational Example: Queues From Stacks

A typical software engineering problem is to transform a high-level design so that it can be directly implemented on a particular architecture. One aspect of doing this is that one aims to re-use components that already exist in the chosen target architecture.

During teaching of a data structures and algorithms course, students are introduced to the abstract concepts of a queue and a stack. These two examples provide a good opportunity to introduce formal methods. We have used the following problem with 2nd year students (as a Java programming exercise), MSc students (as an OO design exercise), and with fourth-year students (as a formal verification exercise). In this paper, the emphasis is on the design process, whilst the actual modelling languages used by the students (UML, B and Java) illustrate the need to reason about correctness as formally as possible.

5.1 How can we implement a Queue using Stacks?

We specify the requirements as a Queue of integers [5] and state that the students must implement the FIFO behaviour using only two integer Stacks

[5] Note that this problem takes on a different nature if we allow the modelling of parametric classes of behaviour.

(LIFO behaviour) to store the queue contents. As a design exercise, students typically adopt 1 of 2 options:

- Design1: The queue is specified as having two stack components — which we will name as a `pushstack` and a `popstack`. When a push request is made of the queue then this element is pushed directly onto the `pushstack`.When a pop request is made of the queue then move all elements from the `pushstack` on to the `popstack` then pop off the last element of the `popstack` and then move all the elements back on to the `pushstack`.
- Design2: The queue is has two stack components — which we will name as a `mainstack` and a `tempstack` — and a boolean representing whether or not the `mainstack` is `ready to push`. (If it is not `ready to push` then we say that it is `ready to pop`). When `ready to push`[6]: if a push request is made of the queue then this element is pushed directly onto the `mainstack`, if a pop is requested then all the elements are moved from the `mainstack` to the `tempstack`, the `mainstack` and `tempstack` are swapped, the state is changed to `ready to pop` and the element popped off the `mainstack`.

At this stage we ask the students to evaluate the quality of their designs. Most students identify the following inter-related design quality criteria: simplicity, understandability, implementability, extensibility, modularity, maintainability, re-usability, efficiency (time and memory), robustness and reliability. In our experience students will ask about the *correctness* of their design only if they have already studied formal methods. When asked if the design will work, most students reply that they will test their implementation to make sure that it does.

Analysis of Design1 and Design2 usually leads to students identifying that Design1 is easier to understand and implement, but that Design2 may be more efficient. Representing the two designs in UML often leads to the students realising that the two designs appear to be structurally the same, but quite different in terms of their dynamic behaviour. The class diagram, in figure 1, illustrates that using UML leads to further investigation of design alternatives that are not so obvious from working only with a formal modelling language.

Most students choose to model the assocation between the Queue and its class components as composition. The remaining students usually model this using aggregation. We ask the question as to why a hybrid model (using both composition and aggregation) is, in general, never considered.

5.2 Rigorous Analysis

We ask the students to argue (demonstrate) whether the designs are correct before they implement them. Typically, they are not able to convince themselves that the designs are correct but they do identify unsafe states of the designs that should not arise. We then show them how these can be modelled

[6] The `ready to pop` case can be treated similarly.

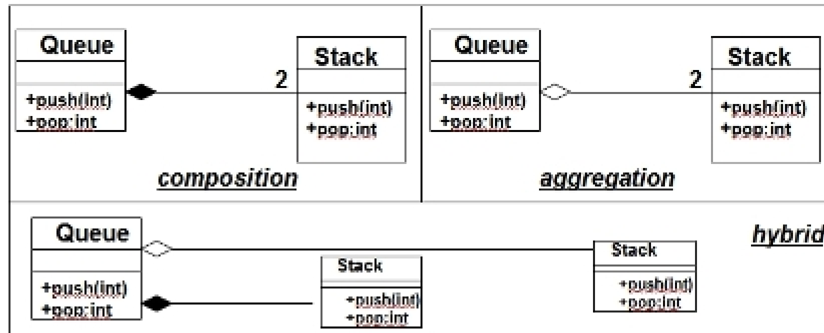

Fig. 1. Aggregation or Composition?

using invariants. For example, in Design1 the Queue system is unsafe if the `popstack` is not empty when an element is pushed on to the `pushstack`.

We have followed two different routes from this point. Firstly, students can implement their designs (typically in Java). Secondly, students formalise their designs in B and attempt to prove that the designs are correct. In the first instance, student implementations often do not meet the queue requirements (which can be found through testing): students then need to discuss whether this signifies that their designs are incorrect. In the second instance, students usually manage to model the abstract queue requirements, but fail to see how they can refine their queue into two communicating stacks. The best students manage to model the designs in B but fail to prove the refinement relation (and hence the correctness). However, when asked to implement their B designs (again, in Java) they usually do not make the same programming errors.

5.3 How can we implement a queue of integer pairs from stacks of integers?

This extension to the problem is stated as: we wish to store co-ordinates in a queue with standard FIFO behaviour, and co-ordinates are specified as pairs of integers `(x,y)` where the class methods allow reading and writing of `x` and `y`. Our underlying implementation architecture allows us to store integers on Stacks. Most students quickly realize that they can re-use their designs to the queue of integers problem. Then, they typically propose 1 of these 3 designs:

- DesignA: propose some isomorphic function between integers and co-ordinates. To push a co-ordinate onto a queue we need only transform it into an integer (using our function); and then push this integer onto an integer stack. To pop off a co-ordinate, we just pop off an integer and transform it into a co-ordinate (using the inverse of the transform function)
- DesignB: To push on a co-ordinate `(x,y)` just push `x` onto the integer queue and then push `y` onto the integer queue. To pop off a co-ordinate then pop off an element `a`, pop off an element `b` and return the value `(a,b)`.
- DesignC: Use 2 integer queues - one for the x co-ordinate, the second for the y co-ordinate. To push on a co-ordinate `(x,y)` just push `x` onto the xqueue

and then push y onto the yqueue; and similarly for popping.

Following their experience from the first simpler design exercise, the students immediately identify unsafe aspects in each of the designs. DesignA will certainly cause problems if we cannot prove the transform function to be isomorphic. DesignB could give rise to unsafe system states if the number of elements on the queue is odd. DesignC could give rise to unsafe system states if the number of elements on each of the queues is not the same. They are then asked to use B to model the designs and the appropriate invariant properties.

5.4 Interesting Design Aspects from the UML

In figure 2, we see an interesting design question: is it possible to combine Design2 and DesignC in order to provide a single temporary stack (shared by both Queue components) that is used for reversing the elements when moving elements from one stack to another inside the queue components?

The advantages and disadvantages of such a design should lead the students to identify that this may give rise to performance and synchronisation issues. However, they still need to ask if such a design is correct. In fact, this question is more subtle than it first seems as the question is really whether the high level structure can provide a framework for the desired behaviour.

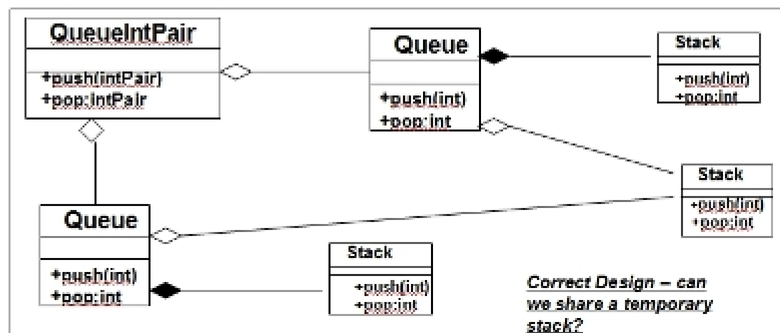

Fig. 2. Can We Build A Correct Design From This High Level Structure?

5.5 Formal Methods and High-level structure

In figure 2, we have very clear high-level structure; but no clear underlying model of how the components will co-ordinate in order to provide correct Queue (of IntPair) behaviour. In the UML we could model this using sequence diagrams or collaboration diagrams. However, without providing these diagrams with formal semantics, it is not possible to establish the correctness of the design to a high degree of confidence. Note that we do not reject the use of UML: the high level structural properties are much easier to represent (and validate) using UML than with a more formal notation.

5.6 Formal Methods Observation: Class invariants are fundamental

This use of invariants is key to integrating the formal with the informal. Every class in the system needs to have an associated invariant which glues together the class associations/components/attributes. For example, in our design in figure 2, we need to specify that the number of elements in each of the component queues must be the same. This is part of the invariant of the system that specifies that if this is not true then the system is in an unsafe state. A formal notation (like B) can then be used to verify that every event that changes the system state (corresponding to a class method) respects the invariant. If this proof cannot be established then the designers must make special note of the fact that this property needs to be tested in the final implementation.

The best lesson that students can take from these types of design exercises is that abstract models must specify (sub)system invariants. Concrete models must guarantee that these invariants are respected. We have evidence that students have (partially) learnt the lesson: in their subsequent software development projects, we have seen invariant checking methods in their implementation models (code) and a structured approach to testing system component invariants at runtime. However, we have yet to see any students formally state and verify their invariants using B.

5.7 Extending and refining the example

The design problem in this paper has been re-used in a number of other different modules —

- Fault tolerance, robustness and reliability: if we know that the fundamental components can fail (following different failure patterns) then can the designs be analyzed in order to reason about the reliability of the system?
- Performance: if we have very strong requirements concerning the speed at which the system must operate then can we analyse the design options in order to reason about system-wide performance in a compositional manner?
- Maintainability and extensibility: if we extend our co-ordinate system so that we have more than 2 dimensions then do we have to change the design?
- Automated software engineering: how likely is it that such a design could be automatically compiled into code?

We do not claim that this design example is without fault. For example, it is slightly contrived and rather simplistic. However, we have found it to be a good case study for teaching formal design concepts, and for showing that the best approach is to try and integrate different modelling languages.

6 Related Work and Conclusions

The (pedagogic) integration of design with testing is discussed in [10] but does not address the role of formal methods. Formal methods and requirements modelling is treated in [13], where the step from requirements to design is identified as being difficult to teach. In [4] there is a discussion on teaching design by contract using the OCL of UML. Recent work on teaching design by contract[15] advocates a mixed semantic approach to teaching formal design. The EU FrameWork6 project RODIN [3] acknowledges the need for research and development into the teaching of formal design, and the Eclipse-based platform provides specific plug-ins for integrating B and UML[21].

This paper reports on our view on the teaching of formal object oriented design. In a new MSc programme (starting in the next academic year) for *software engineering (of smart devices)* we have attempted to distribute formal methods throughout the programme modules and not to fall into the trap of teaching a stand-alone formal methods module that students fail to relate to all other software engineering material. The future of software design is for students to realize that formal methods are just another tool in their toolbox. This message needs to be transmitted from lecturers to students and from students to industry.

Formal methods lecturers should not just encourage their colleagues to talk about formal methods; they should also make more of an effort to incorporate other aspects of software engineering in their own formal methods modules.

References

[1] Abrial, J.-R., “The B Book - Assigning Programs to Meanings,” Cambridge University Press, 1996, ISBN 0-521-49619-5.

[2] Abrial, J.-R., *A system development process with Event-B and the Rodin platform*, in: M. Butler, M. G. Hinchey and M. M. Larrondo-Petrie, editors, *ICFEM*, Lecture Notes in Computer Science **4789** (2007), pp. 1–3.

[3] Abrial, J.-R., M. Butler, S. Hallerstede and L. Voisin, *An open extensible tool environment for Event-B*, in: Z. Liu and J. He, editors, *ICFEM*, Lecture Notes in Computer Science **4260** (2006), pp. 588–605.

[4] Baar, T., D. Chiorean, A. L. Correa, M. Gogolla, H. Hußmann, O. Patrascoiu, P. H. Schmitt and J. Warmer, *Tool support for OCL and related formalisms - needs and trends*, in: Bruel [9], pp. 1–9.

[5] Back, R. J. R., *On correct refinement of programs*, Journal of Computer and Systems Sciences **23** (1981), pp. 49–68.

[6] Bell, A. E., *Death by UML fever*, Queue **2** (2004), pp. 72–80.

[7] Booch, G., “The UML User Guide,” Addison Wesley, 1999, ISBN 0201571684.

[8] Borges, R. M. and A. C. Mota, *Integrating UML and formal methods*, Electronic Notes in Theoretical Computer Science **184** (2007), pp. 97–112.

[9] Bruel, J.-M., editor, "Satellite Events at the MoDELS 2005 Conference, Doctoral-Educators Symposium, Jamaica, October 2-7, 2005, Revised Selected Papers," Lecture Notes in Computer Science **3844**, Springer, 2006.

[10] Carrington, D., *Teaching software design and testing*, in: *Frontiers in Computer Science Education*, 1998, pp. 547–550.

[11] Engels, G., J. H. Hausmann, M. Lohmann and S. Sauer, *Teaching UML is teaching software engineering is teaching abstraction*, in: Bruel [9], pp. 306–319.

[12] Evans, A., R. B. France, K. Lano and B. Rumpe, *The UML as a formal modeling notation*, in: J. Bézivin and P.-A. Muller, editors, *UML*, Lecture Notes in Computer Science **1618** (1998), pp. 336–348.

[13] Gibson, J. P., *Formal requirements engineering: Learning from the students*, in: *Australian Software Engineering Conference* (2000), pp. 171–180.

[14] Martin, R. C., "Agile Software Development, Principles, Patterns, and Practices," Prentice Hall, 2002, ISBN 0135974445.

[15] McKim, J. C. and H. J. C. Ellis, *Course module: Design by contract*, in: *CSEET '05: Proceedings of the 18th Conference on Software Engineering Education & Training* (2005), pp. 239–241.

[16] Meyer, B., *UML the positive spin*, American Programmer **10** (1997).

[17] Oliver, I., *Experiences in using B and UML in industrial development*, in: J. Julliand and O. Kouchnarenko, editors, *B*, Lecture Notes in Computer Science **4355** (2007), pp. 248–251.

[18] Reeves, J. W., *What is software design*, C++ Journal **2** (1992).

[19] Richters, M. and M. Gogolla, *Validating UML models and OCL constraints*, in: A. Evans, S. Kent and B. Selic, editors, *UML*, Lecture Notes in Computer Science **1939** (2000), pp. 265–277.

[20] Robinson, K., *Embedding formal development in software engineering*, in: C. N. Dean and R. T. Boute, editors, *Teaching Formal Methods*, Lecture Notes in Computer Science **3294** (2004), pp. 203–213.

[21] Snook, C. and M. Butler, *UML-B: Formal modelling and design aided by UML*, ACM Trans. Software Engineering Methodology **15** (2006), pp. 92–122.

[22] Vaziri, M. and D. Jackson, *Some Shortcomings of OCL, the Object Constraint Language of UML*, in: Q. Li, D. Firesmith, R. Riehle and B. Meyer, editors, *TOOLS (34)* (2000), pp. 555–562.

[23] Younes, A. B. and L. J. B. Ayed, *Using UML activity diagrams and Event-B for distributed and parallel applications*, in: *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference - Vol. 1- (COMPSAC 2007)* (2007), pp. 163–170.