Using LEGO to Teach Software Interfaces and Integration

Stan Kurkovsky
Central Connecticut State University
kurkovsky@ccsu.edu

ABSTRACT

Software design is a complex endeavor because it requires mastery of engineering practices, insight into the domain knowledge, exploring of alternative ideas, and, most importantly, plenty of practice. Principles of good software design should be introduced early in the curriculum and practiced whenever possible. This work describes a LEGO-based activity for multiple teams to practice collaborative design, parallel development, and component integration to illustrate the advantages of well-designed component interfaces.

CCS CONCEPTS

• **Social and professional topics** → **Software engineering education**; • **Software and its engineering** → *Agile software development*; *Object oriented development*;

KEYWORDS

Software interfaces, software design, integration, active learning

ACM Reference Format:
Stan Kurkovsky. 2018. Using LEGO to Teach Software Interfaces and Integration. In *Proceedings of 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE'18).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3197091.3205831

1 INTRODUCTION

As Hu noted [5], it is important to start teaching software design skills early in the curriculum. However, given the relative simplicity of problems that can be given to students with limited skills and experience, it is difficult to achieve significant results aside from teaching basic good design practices and offering ample opportunities to apply them [1]. Despite collaboration and teamwork being integral principles of modern software engineering, students in introductory programming courses rarely participate in any collaborative design activities because the main focus of these courses is often on learning how to program. Furthermore, instructors may be reluctant to assign any team-based design work in introductory courses because of the general concern about each student's accountability [9].

Class, component, or service interface is a fundamental principle in software reuse. The notion of "programming to an interface, not an implementation" is a key principle in creating well-designed software systems [4]. When designing a software component or service, it is necessary to define its interface in a way that makes it easier for other components and services to integrate with it. These design decisions must take into account the tradeoff between exposing too little functionality and making the service inflexible or difficult to use, and exposing too much functionality, which may make it difficult to limit the use cases of the component only to those that are legitimate. Such design activities are rarely practiced extensively in typical university-level courses.

A solid grasp of of the concept of software interface enables students to become professionals who are comfortable with many modern practices, such as continuous integration, as well as software frameworks, including many cloud-based platforms built on the application/platform/infrastructure "as a service" model. However, a number of reports indicate that many students experience serious difficulties with the mastery of this concept [5, 7]. Santos [8] suggests that one of the difficulties of teaching effective object interface design is a common lack of collaborative learning experiences, in which student teams would develop interoperable components for a common system. Such learning experiences can help students focus on interoperability of the solutions they design. Additionally, these experiences can help students better understand the consequences of poor interface design choices that they are likely to face when integrating their components with those designed by other teams.

Clear lines of inter-team communication are also extremely important in modern software projects developed by multiple teams. Many agile projects developed by distributed teams are completed under pressure resulting from the tension between the project agility and the need for effective communication, which is often compromised by differences in cultural backgrounds, time zones, and geographical locations. Well-defined software component interfaces play a very important role in establishing and supporting these clear communication channels [3]. However, replicating this experience in course projects is often challenging. For example, Billingsley and Steel [2] report that student teams started to actively communicate with each other only in the final week of a semester-long course-project. Furthermore, the teams barely cooperated with each other earlier in the project when designing component interfaces and their functionality.

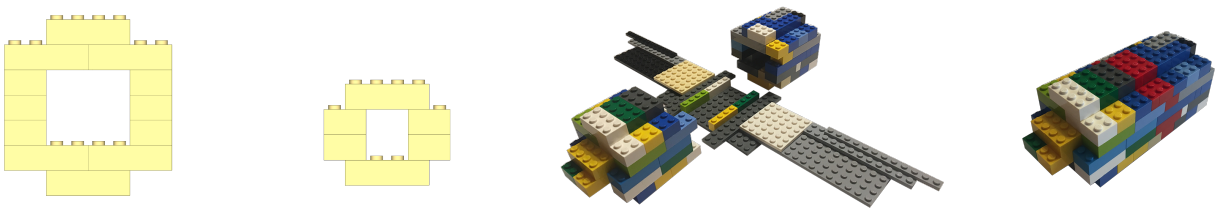
2 BUILDING INTERFACES WITH LEGO

The advantages of using LEGO to teach software engineering have already been established [6]. Here, we are relying on the analogy between building software and constructing a LEGO model: creating software out of multiple lines of code and using software interfaces is not unlike locking LEGO bricks together while following certain rules about how you can and cannot interconnect the bricks.

The **learning objective** of this exercise is to illustrate that designing and specifying component interfaces will make it easy to integrate them into a whole system. This exercise simulates a distributed software project, in which a number of components are

ITiCSE'18, July 2–4, 2018, Larnaca, Cyprus

ACM ISBN 978-1-4503-5707-4/18/07…$15.00
https://doi.org/10.1145/3197091.3205831



(a) Large airplane profile (b) Small airplane profile (c) Airplane components (d) Component adapter

Figure 1: LEGO-based activity to practice component integration and software interface design

built in parallel by different teams. The specific objective of this exercise is to build an airplane consisting of four components: cockpit, fuselage, tail end, wing assembly. There is no landing gear or engines. The wing assembly is made with thin plates and must be attached to ether top or bottom of the fuselage. A cross-section profile of the airplane is provided as shown in Figure 1a. Students are instructed not to use red-colored bricks, which are reserved for integration as discussed below.

This activity typically is run with **four teams** (ideally consisting of up to five people), each building one of the components. With a smaller number of people or with a limited number of bricks, it is possible to reduce the number of components to three: instead of the cockpit, fuselage, tail end use the front half of the fuselage plus the back half with the tail end. Required **materials** include an ample amount of standard LEGO bricks: 2x4 are ideal, but any bricks ranging from 2x2 to 2x6 or longer are suitable. LEGO Classic sets such as 10698 are suitable for three teams working with the smaller profile (Figure 1b). The activity consists of two parts, each completed within about 10 minutes.

In **part 1**, each team is given five minutes to build their components without negotiating or designing the interfaces. Typical resulting LEGO models of the airplane components (without the tail end) are shown in Figure 1c. Then all teams need to integrate their components together without changing them. The objective here is to show that integration will be very difficult and time consuming. When the teams see that it's impossible to integrate their components without any modifications, they are allowed to use red bricks to connect the components (Figure 1d). This is an analogy of applying the adapter design pattern. Any bricks that have to be removed and placed back in the process will also need to be replaced with red bricks. It is important to measure how much time is needed for integration.

In **part 2**, the teams are rotated so that each team would build a different component. Each team delegates one member to negotiate the component interfaces. These must be documented/sketched and reviewed before each team representative takes them back to their team. This should be done in under one minute. Then all teams have five minutes to build their components using the previously designed interfaces. A result, integration should be easy and straightforward. If necessary, red bricks can be used for the same purpose as in part 1. As before, it is important to measure the amount of time the teams need to integrate their components.

As with many active learning exercises, the key of this activity is in its **debrief** session. There are a number of open-ended questions that should be discussed. *Why did we have so few problems with the wings?* Wings are always easy to attach because the interface is already explicitly provided by the cross-section diagram. They simply need to fit the top or bottom of the fuselage, whose shape is known in advance. *How much time did we save by providing explicit interfaces in part 2 and why? Did everything work flawlessly in part 2? What were the issues and how they could be avoided? Is it ever possible to build software components that would integrate perfectly?*

This exercise has been offered to students in both upper-level software engineering and lower-level object-oriented programming courses over the last two years. Current results suggest a positive effect on student understanding of the need for software design and improved mastery of designing and using software interfaces.

ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation award DUE-1611905.

REFERENCES

[1] A. Baker, A. van der Hoek, H. Ossher, and M. Petre. 2012. Studying Professional Software Design. *IEEE Software* 29, 1 (Jan 2012), 28–33. https://doi.org/10.1109/MS.2011.155

[2] William Billingsley and Jim Steel. 2013. A Comparison of Two Iterations of a Software Studio Course Based on Continuous Integration. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '13)*. ACM, New York, NY, USA, 213–218. https://doi.org/10.1145/2462476.2465592

[3] Jan Bosch and Petra Bosch-Sijtsema. 2010. From Integration to Composition: On the Impact of Software Product Lines, Global Development and Ecosystems. *J. Syst. Softw.* 83, 1 (Jan. 2010), 67–76. https://doi.org/10.1016/j.jss.2009.06.051

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[5] Chenglie Hu. 2013. The Nature of Software Design and Its Teaching: An Exposition. *ACM Inroads* 4, 2 (June 2013), 62–72. https://doi.org/10.1145/2465085.2465103

[6] Stan Kurkovsky. 2015. Teaching Software Engineering with LEGO Serious Play. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '15)*. ACM, New York, NY, USA, 213–218. https://doi.org/10.1145/2729094.2742604

[7] Stacey Omeleze, Vreda Pieterse, and Fritz Solms. 2015. Teaching modular software development and integration. In *6th Annual International Conference on Computer Science Education: Innovation and Technology (CSEIT 2015)*. 178–187. https://doi.org/10.5176/2251-2195_CSEIT15.25

[8] André L. Santos. 2015. Collaborative Course Project for Practicing Component-based Software Engineering. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research (Koli Calling '15)*. ACM, New York, NY, USA, 142–146. https://doi.org/10.1145/2828959.2828972

[9] S. Sheth, J. Bell, and G. Kaiser. 2013. A competitive-collaborative approach for introducing software engineering in a CS2 class. In *2013 26th International Conference on Software Engineering Education and Training (CSEET'13)*. 41–50. https://doi.org/10.1109/CSEET.2013.6595235