Paper ID #37981

Building a model of polymorphism comprehension

Joshua Gross

Joshua Gross is an assistant professor of computer science at CSUMB. He spent nearly a decade as a software engineer, earning an MS in software engineering from the University of St. Thomas in St. Paul, Minnesota. He holds a PhD in information sciences and technology from Penn State, where his research focused on the intersection of artificial intelligence and human-computer interaction. His current research is focused on the psychology of programming, with the goal of better understanding factors that support student success in undergraduate computer science coursework and in software development professions, with the hope of addressing the severe shortfall of qualified developers.

Kevin Coogan

Kevin Coogan is an instructional faculty member at Hampton University in Hampton, VA. He received his Ph.D. from the University of Arizona where his primary research focus was on malware obfuscation techniques. Over the past eleven years he has taught a wide variety of courses including information security, networking, operating systems, data structures, and introductory programming. His current teaching is focused primarily on introductory programming and data structures, and his current research interests center on improving student outcomes in these and related core courses.

Sarah Heckman (Teaching Professor)

Gabriel Silva de Oliveira

Building a model of polymorphism comprehension

Abstract

Mastering subclass polymorphism in object-oriented (OO) programming is critical because polymorphism plays a central role in many commonly used design patterns and in software development generally. However, designing and implementing polymorphic solutions is challenging for novice programmers because polymorphism is an emergent consequence of correctly using multiple OO language features. In order to eventually improve polymorphism instruction, this research focuses on developing a model of polymorphism comprehension, along with a schema for placing students within that model. A case study was conducted with ten students in an OO CS2 course. Participants completed several short assignments, then participated in mock whiteboard interviews. Analyzing these interviews, researchers derived a three-level model of polymorphism comprehension: basic structured software design principles, OO abstraction principles, and OO polymorphism principles. Data show a major gap between OO abstraction and polymorphism, indicating a need to focus on moving from inheritance to substitutability.

Introduction

While basic programming concepts require correctly using a keyword or code structure, polymorphism is emergent. Rosson & Alpert defined subclass polymorphism as a property that ”allows different objects to respond individually to precisely the same message” [1], a definition affirmed by Armstrong’s survey of definitions of object-oriented (OO) concepts [2]. Because implementing a polymorphic solution requires correctly integrating several challenging underlying concepts, polymorphism is among the most complex topics taught in introductory courses.

A very simple example of subclass polymorphism can be demonstrated by calling a method via late binding on each element in an array containing multiple types. In Java, a programmer must (see Listings 1 and 2):

1. Define a superclass with at least one method (Foo, Line 18)
2. Define one or more subclasses (Bar & Grault, Lines 24 & 30) that override the method
3. Declare an array of the superclass type (Line 3)
4. Add instances of the superclass and subclass(es) to the array (Lines 4-8)

5. Iterate over the array, assigning each element in turn to a variable of the superclass type (Line 11)

6. Call the method declared one the superclass-type variable (Line 12)

```java
 1 class Main {
 2   public static void main(String[] args) {
 3     Foo [] f = new Foo[5];
 4     f[0] = new Foo();
 5     f[1] = new Grault();
 6     f[2] = new Bar();
 7     f[3] = new Foo();
 8     f[4] = new Grault();
 9 
10     for(int i = 0; i < 5; i++) {
11       Foo temp = f[i];
12       temp.quux();
13       System.out.println("---");
14     }
15   }
16 }
17 
18 class Foo {
19   public void quux() {
20     System.out.println("Running quux() in Foo");
21   }
22 }
23 
24 class Bar extends Foo {
25   public void quux() {
26     System.out.println("Running quux() in Bar");
27   }
28 }
29 
30 class Grault extends Foo {
31   public void quux() {
32     super.quux();
33     System.out.println("\tRunning quux() in Grault");
34   }
35 }
```

Listing 1: Late Binding Using OO Polymorphism in Java

```
Running quux() in Foo
---
Running quux() in Foo
    Running quux() in Grault
---
Running quux() in Bar
---
Running quux() in Foo
---
Running quux() in Foo
    Running quux() in Grault
---
```

Listing 2: Example Output

Related Work

For the purposes of this paper, we define *polymorphism comprehension* as the ability to select, design, and implement a solution using subclass polymorphism where appropriate. Programmers unable to master these skills may struggle to be hired as software developers. This research was inspired by a discussion with an industry collaborator, who shared that several students failed to complete a whiteboard interview because of this struggle.

Polymorphism has been cited repeatedly as one of the most challenging topics in introductory programming [3, 4]. While Bergin asserted that students can learn polymorphism when beginning to program in an objects-first approach [5, 6], not all programming course sequences use an objects-first curriculum, and despite using objects-first, Ragonis and Ben-Ari did not cover inheritance or polymorphism [7]. Schmolitzky argues for teaching interfaces before inheritance, in part to delay and prepare students for polymorphism [8]; a model of polymorphism comprehension could validate that approach.

Drawing on Liberman *et al.* [9] and Chen *et al.*, [10], Mills *et al.* developed a list of four common misconceptions due to ”simplistic alternative programming models” [11], based on the challenges of overriding, dispatch, and inheritance-based type constraints. In particular, Liberman found late binding and dispatch to confuse teachers learning OO; one participant believed that downcasting was necessary to use an overridden method, while another believed that overriding a method in a subclass changes the superclass. The assignment used in this research addresses all four misconceptions, and in the context of design, rather than programming or comprehension, as in Mills.

Research Questions

While the ultimate goal for this research is to improve instruction of polymorphism, it is first necessary to define a model to evaluate outcomes. Assessing polymorphism comprehension in exams requires using low-stakes conceptual questions or simplistic programming problems [12], and completed homework and projects do not demonstrate which aspects of polymorphism a student found challenging to design. The research questions addressed here are:

1. How can polymorphism comprehension be modeled?
2. How can students be assessed within that model?

Methods

Because this research was partly motivated by software engineering hiring practices, whiteboard interviews were adopted as the primary assessment mechanism. In a whiteboard interview, an interviewee is asked to solve a challenging problem by explaining a solution verbally and visually, using a whiteboard to demonstrate design and programming techniques. Despite concerns of bias [13, 14], whiteboard interviews have become a core part of the hiring process for software engineers [15] because they are believed to represent a simplified model of real-world challenges.

Research Protocol

A case study was conducted at a small liberal arts college in the Midwestern United States focused on students from backgrounds historically underserved by higher education, including people of color, people from rural areas, and people from lower socioeconomic strata. Of the 14 students in a CS2 course focused on OO programming in Java, 11 consented to participate, including nine men and two women, three Black students (including one woman), and one student retaking the course. The students were a mix of mathematics and CS majors and CS minors, and in their first through final year. All took CS1 in Java; two had transferred in CS1 credit.

An IRB approval was granted by the host institution. During the consent process, students were informed that the research project would address polymorphism. All students completed all activities as part of the course, regardless of their consent status. Consent documents were kept from the first author (the course instructor) until course grades were submitted. Data collected from non-consenting students was then discarded. (When referring to assignments, this paper will use the term *students*; when referring to data, it will use the term *participants*.) The overall research design included four separate assignments, all involving polymorphism:

1. A one-hour lab completed in programming pairs
2. A one-week homework assignment, completed individually
3. A two-week project developed for this research, completed individually, described below
4. A 10-30 minute whiteboard interview, described below

The project and whiteboard interview both used Corc, a framework for developing interactive, graphical card game assignments in Java. Previous work has explored Corc's effectiveness in motivating students to complete challenging introductory programming assignments redacted.

Corc is a large library that allows educators to build interactive card game assignments, exposing only domain classes to the students. As seen in Figure 1, students are given classes like Card, Face, Suit, and Hand. Per assignment instructions, students use these classes to implement specified parts of a card game, such as an automated player or a hand scoring method. Corc uses library classes, naming conventions, and interfaces to connect to student code and implement graphics and interactivity using JavaFX. Students do not implement any UI code or call methods in the UI code.

Strategy Pattern Project

A new project was developed as part of this study, in which students implemented the *Strategy design pattern* [16]; Bergin suggested its inclusion as an early introduction to polymorphism [5]. Strategy uses a polymorphic callback mechanism, called *inversion of control* [17], typical of many challenging design patterns. It allows a program to dynamically select an approach to solve a problem, often based on system state. For a card game, these classes could be created:

1. A *Player* class that contains the hand of cards and has methods to add and discard cards
2. An abstract *Strategy* superclass, which encapsulates calls to a single method, *playRound*

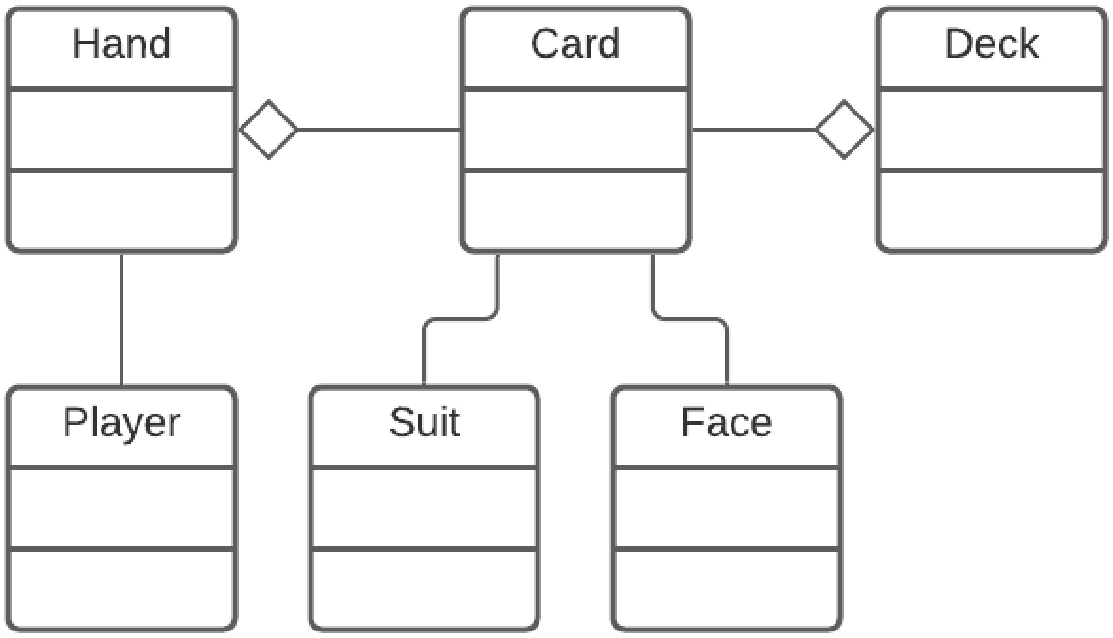

Figure 1: Basic Classes in Corc

3. Multiple subclasses of Strategy that implement playRound using different algorithms

Player contains an attribute of type Strategy; when playing a round, Player passes control to the Strategy object, which calls back to Player to act on decisions. Algorithms are substituted by changing the instance stored in this attribute to a different subclass.

For this project, the card game Gin was chosen. In this version of Gin, each player receives ten cards, which can be formed into groups (called *melds*) of at least three cards in one of two forms: *sets*, the same face (e.g., four kings); or *runs*, sequential in the same suit (e.g., the 4, 5, and 6 of hearts). The first player to get all ten cards into non-overlapping melds wins the game. Each round, a player chooses either a face-up card (discarded by the last player) or a face-down card from the deck, and then discards one card. The only choices made are selecting which card to pick up, and which of the 11 cards to discard.

The student's task was to build a competitive player to play against a human or built-in AI player by implementing both the algorithms and the Strategy pattern itself. The assignment instructions include a class diagram, several sequence diagrams, and high-level descriptions of an algorithm for each Strategy subclass. To familiarize students with the Strategy pattern, they first implemented two simple, naïve algorithms that build only sets and runs, respectively. Students then implemented three final GinStrategy subclasses, each of which chose cards differently based on the game state, and changed to another subclass when necessary. The classes implemented these algorithms (Figure 2):

- At the beginning of the game, *BuildMelds* selects any card that creates a new partial meld or that adds to an existing meld, until each card in the hand is in a meld.
- In the middle of the game, *CompleteMelds* chooses which melds to keep or discard, as new cards become available.
- Toward the end of the game, *WinEndGame* looks for one of a small number of cards that will complete the hand.

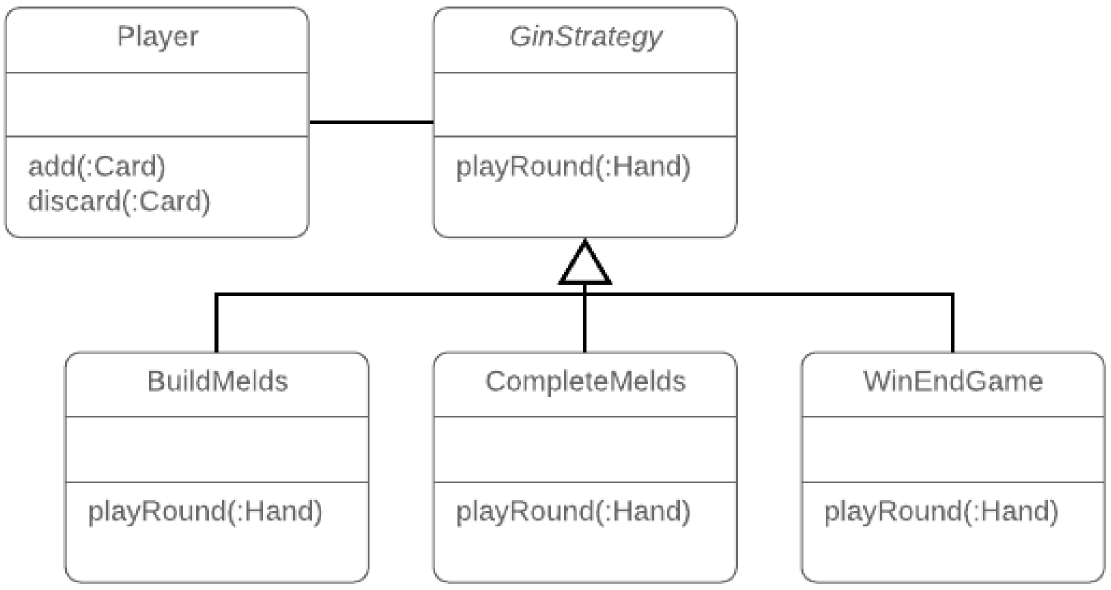

Figure 2: Final Strategy Class Diagram

After completing the project, each student participated in a whiteboard interview conducted by professional software developers. Interviewers had prior experience in conducting whiteboard interviews as part of their company's hiring process, and were trained in research ethics. The researchers wrote the interview protocol with input from the interviewers. The interviews were video recorded for later transcription. Students were able to use verbal and written descriptions, including pseudocode or class diagrams.

The interview protocol consisted of two parts, both of which used Corc. Participants received a simplified class diagram showing Corc's Hand, Card, Face, and Suit classes.

Participant were first given Problem 1, which used the game War. Two players each play one card at a time. The higher face wins; matching faces result in a tie. Participants were instructed to solve only the outcome of a single hand. When the participant completed Part 1 or demonstrated that they were unable to do so, they were given an example solution showing a WarComparator class with a compare method that returned an integer. Participants were then given Problem 2, which asked them to solve the same problem for Poker, and were explicitly told to not attempt to write an algorithm for comparing Poker hands, only design the necessary components.

Interviews of consenting participants were transcribed and analyzed. Initial analysis demonstrated that interviewers provided a variety of different kinds of guidance to assist the participants, which led to coding of Interviewer Prompts. The instructions issued as part of the protocol were not counted as prompts. Prompts were assigned a level based on content:

1. Prompting – the interviewer encouraged the participant to be explicit (e.g. ”could you show that in a class diagram?”)
2. Probing – the interviewer addressed a specific problem (e.g., ”And what's the relationship between WarComparator and Comparator [classes]?”)
3. Directing – the interviewer suggested a specific approach (e.g., ”swap between PokerCompare and WarCompare... how would you do that?”)

Results

Analysis of the whiteboard interview data formed the foundation of a three-level Polymorphism Comprehension Model, described in this section. The model emerged from identifying milestones completed by the participants.

Solutions showed a surprising amount of convergence, such that analysis identified a uniform eight separate parts of a complete polymorphic solution, with no valid variations proposed by participants. These were labeled *whiteboard milestones*, although they did not need to be expressed sequentially, although some do have dependencies on other milestones. Participants either successfully completed these milestones or not; no alternative solutions emerged. These milestones were therefore derived deductively from the interviews.

The first three milestones are part of Problem 1, and the last five are part of Problem 2:

1. War Compare Method – a comparison method that takes in two hands and returns the outcome
2. Ternary Return Type – three possible outcomes (Hand 1 wins, Hand 2 wins, tie) require a ternary return type
3. War Comparator Class – a separate comparator class allows for polymorphic substitutability
4. Poker Compare Method – similar to #1 above
5. Identical Signature for Compare Methods – good design practice, and necessary to generalize the solution
6. Poker Comparator Class – similar to #3 above
7. Comparator Superclass – an extension of #3 and #6
8. Polymorphic Selection of Subclass – allows late binding

These data were then analyzed:

- Whether or not the participant completed the milestone
- The number of prompts given for the specific milestone

- The level of each prompt (Prompting, Probing, or Directing)

One participant achieved no milestones and was removed from the analysis. All ten remaining participants achieved three of the milestones (1, 4, and 5), while only two and three participants achieved Milestones 3 and 8, respectively (see Figure 3).

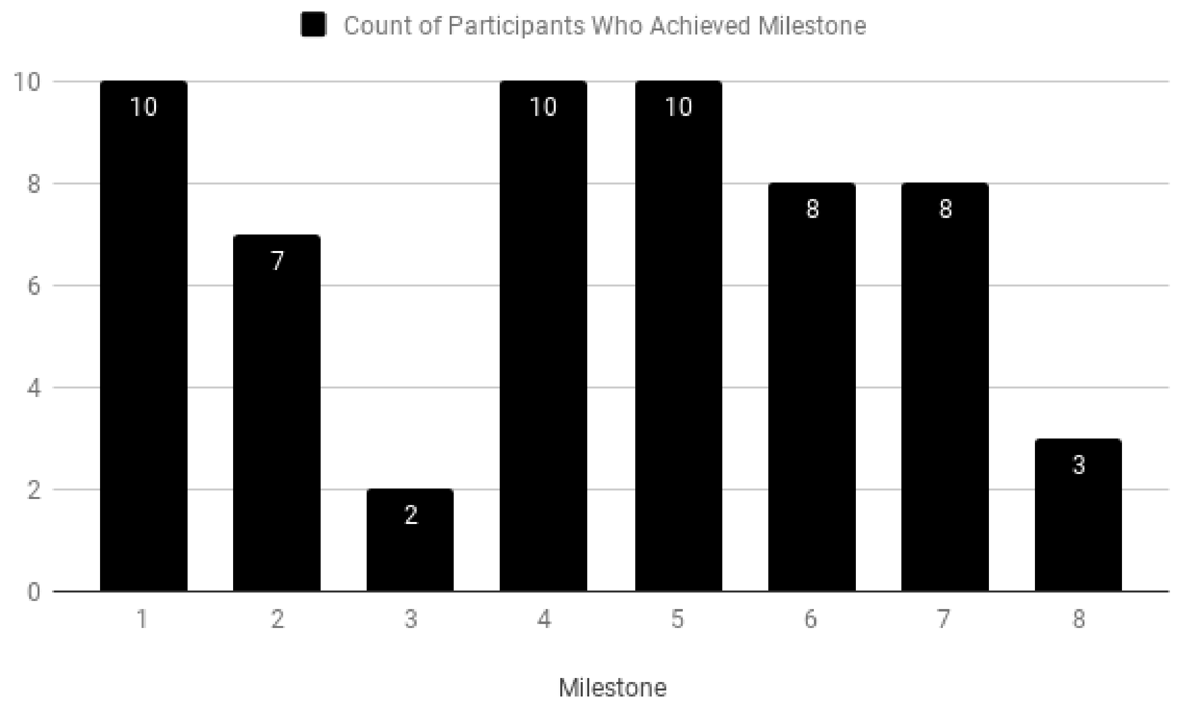

Figure 3: Count of Participants Achieving Each Milestone

Participants were arbitrarily assigned code names from the NATO alphabet (first removing concerning or gender-coded terms). Figure 4 shows the number of milestones each completed. Two participants only completed four milestones, while three completed three milestones. No participant achieved all milestones. Only one common error emerged: four participants made the War Compare Method return a Boolean value, and had to be prompted to deal with ties.

Next, to measure the importance of interviewer feedback for each successfully completed milestone, a composite level of prompting was developed to incorporate both the number of prompts and the amount of guidance. Recall that interviewer guidance was classified as Level 1 (Prompting), Level 2 (Probing), or Level 3 (Directing). By far the most common prompt level was Level 1 (61 of 112 prompts); participants had to be encouraged to demonstrate (verbally or in writing) the ideas that they hinted at.

Each relevant prompt preceding a participant's completing a milestone was counted, along with its level. Each prompt was multiplied by its level, and these products were summed in a Prompt Score for each achieved milestone. For example, if a participant received one Level 1 prompt, two Level 2 prompts, and one Level 3 prompt, they received a Prompt Score of 1*1 + 2*2 + 3*1 = 8. Prompt Scores ranged from 0 (no prompts) to 9.

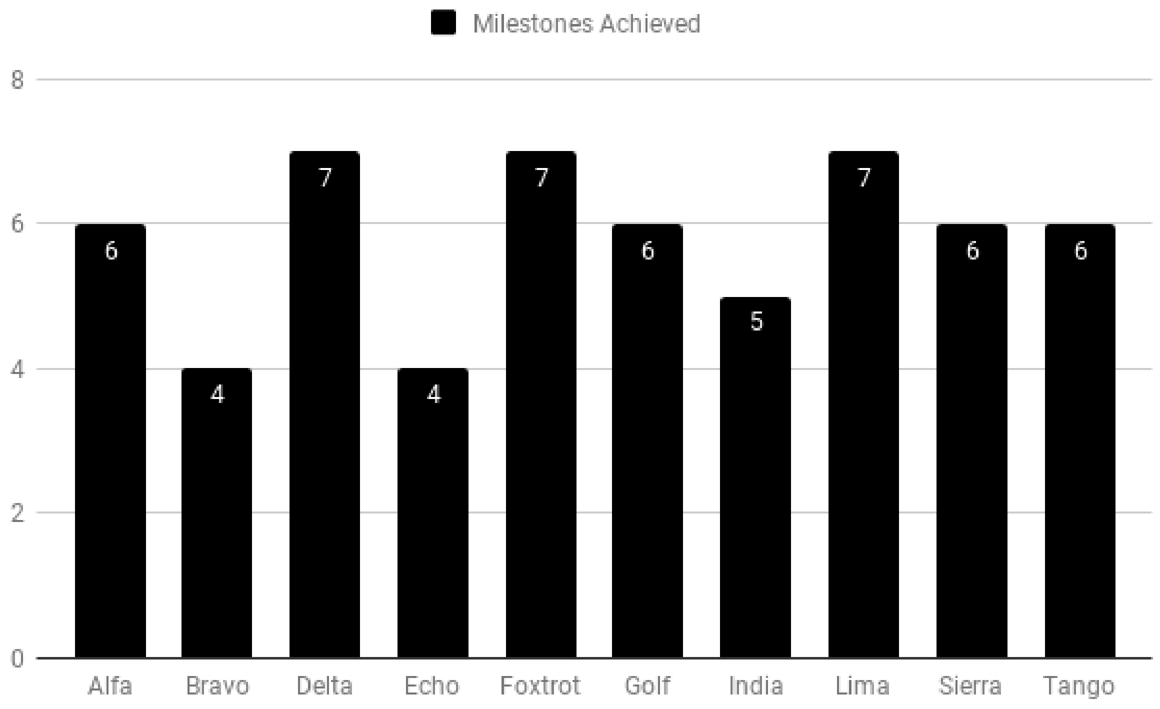

Figure 4: Milestones Completed by Each Participant

Prompt Scores were averaged per milestone (Figure 5), indicating that milestones represented rather different levels of challenge. This can be seen most dramatically for Milestones 3 and 8, where high levels of prompting were necessary to produce the few answers that satisfied those milestones. Milestones fell into three frequency groups:

1. Three milestones were achieved by all ten participants (Universal Group): #1 (War Compare Method), #4 (Poker Compare Method), and #5 (Identical Signature)
2. Three milestones were achieved by 7-8 participants (Common Group): #2 (Ternary Return Type), #6 (Poker Comparator Class), and #7 (Comparator Superclass)
3. Two milestones were achieved by few participants (Rare Group): two achieved #3 (War Comparator Class), and three achieved #8 (Polymorphic Selection of Correct Subclass)

A Three-Level Model

Further analysis derived three separate levels of novice ability to design a polymorphic solution, based on participants producing clusters of related design features. These levels represent Structured Software Design Principles, OO Abstraction Principles, and OO Polymorphism Principles. Each of the three levels is associated with a distinct set of milestones from the analysis above.

The students in this course initially learned Java as a procedural language, only using static

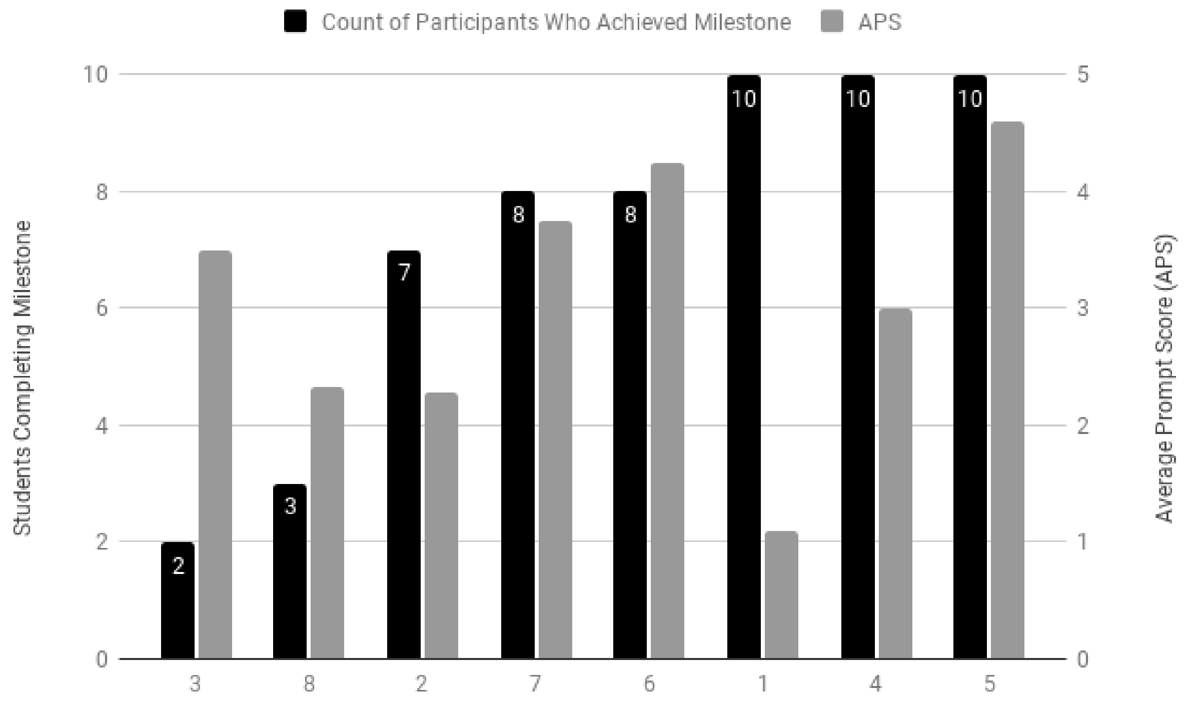

Figure 5: Milestones vs. Average Prompt Score Per Milestone, Ordered by Count of Participants Completing Milestone

methods. Subclass polymorphism in Java rests on a foundation of good *structure software design principles* that do not imply any aspect of object-oriented design:

- Milestone 1 – War Compare Method
- Milestone 2 – Ternary Return Type
- Milestone 4 – Poker Compare Method
- Milestone 5 – Identical Method Signature

This is the Universal Group, with Milestone 2 added, as it is conceptually related to the others and not specific to OO. All ten participants achieved Milestones 1, 4, and 5. The three participants who did not include a ternary return type simply never addressed the return type for the method. Of the seven who did, four started with a Boolean return value, were prompted to address ties, and resolved the issue by returning a string or integer.

Lima achieved Milestone 2 without prompting:

> *[Interviewer] [Reads Problem 1]*
> [Lima] ...I guess the first would be check to see if... they are [the same], then that would return a tie. Else if hand 1 is less than hand 2, some sort of method declares a winner.

Foxtrot needed two prompts:

> [Foxtrot] What would it output? I guess it could be a Boolean that outputs true if the first one beats the second one and false if the second one beats the first one.
> *[Interviewer] What if there is a tie?*
> [Foxtrot] Null. Don't know if you can do that actually.
> *[Interviewer] So maybe there is a better type for this?*
> [Foxtrot] You can just do an int and 1,2,3.

Milestones 1 and 4 are nearly identical (War and Poker Compare Methods), but Milestone 4 had a much higher APS. This seemed to be caused by two issues. First, some participants had to be reminded to not try to write an algorithm for scoring poker hands, and two needed multiple reminders. Second, some participants had to be prompted to make their solution explicit. Here is the exchange after Delta drew a Poker Comparator Class:

> *[Interviewer] Well, there's a line [reads problem aloud]. How would this Poker class implement CardGame [the superclass]? So what methods would you have in the Poker class?*
> [Delta] So per class we'd still have the Compare [method] and probably, possibly its core method to run each hand.

The next level of comprehension demonstrated participant mastery of *OO abstraction principles*, adding the use of classes and inheritance to abstract from the problem domain to an effective generalized OO solution; these are the remaining elements of the Common Group. Java abstraction uses classes to organize and isolate functionality. Once given the model solution for War, which includes the WarComparator class with a compare method, it is reasonable to expect participants to see the value of both a Poker Comparator class and a Comparator Superclass. Eight participants achieved both milestones.

- Milestone 6 – Poker Comparator Class
- Milestone 7 – Comparator Superclass

As seen in Figure 5, these milestones required high levels of prompting. Typically, the participant had to be encouraged to isolate the functionality of the Poker Compare Method after completing Milestone 5. Several participants immediately jumped from seeing the value of isolating the method to seeing the need for both a Poker Comparator Class and a Comparator Superclass. As noted above, when participants achieved both milestones together, the prompts counted for both.

For example, after Alfa created a Poker Compare Method, the interviewer prompted as below. Immediately thereafter, Alfa added a Poker Comparator Class and a Comparator Superclass.

> *[Interviewer] If you had to design a system that contained both of these comparator methods... is there a way? Cause these two aren't related ... you just have a duplicate method definition.*

[Alfa] ...So, the question is if is it the same comparator? So this would be PokerComparator [writes the name of the class].

OO subclass polymorphism is often dependent on late binding, allowing a specific implementation of a method to be invoked. This requires mastering both static and dynamic aspects of polymorphism, selecting the appropriate subclass at runtime.

- Milestone 3 – War Comparator Class
- Milestone 8 – Polymorphic Selection

This is the Rare Group. Only three participants achieved polymorphic switching, out of eight who had the underlying components (superclass, subclasses, and appropriate methods). All three needed significant prompting to articulate it. As an example, after Sierra added a Poker Comparator class (Milestone 6), the interviewer eventually explained the goal (swapping which method was called) in order to clarify, at which point Sierra immediately described a full solution. The success of this prompt exposes a potential lever for future educational interventions: focusing additional attention on the mechanism and value of switching.

[Interviewer] If you... wanted to swap between PokerCompare and WarCompare, how could you go about accomplishing that?
[Sierra] So, like, changing games with a single change of code?
[Interviewer] Yes.
[Sierra] I could do a superclass and just call it Comparator, and then [make] PokerCompare and WarCompare a subclass and basically I would want to, so, let's say I guess, `Compare c1 = new` and I could basically put the PokerCompare or WarCompare to swap between games.

Creating a WarComparator class in Problem 1 (Milestone 3) can be seen as *anticipatory design*, a controversial practice in software engineering [18]. No overlap existed between the two participants who achieved Milestone 3 and those who achieved Milestone 8. This interaction between the interviewer and Foxtrot is typical; they do not see any inherent value in isolating the compare method:

[Foxtrot] You'd have like a compare method, so...
[Interviewer] Can you put that in a class diagram?
[Foxtrot] If it's a method, not a class itself [confirming with interviewer], I need to put it into a class, so...
[Interviewer] Would this be a new class or would you use one of those other classes?
[Foxtrot] I mean, couldn't you put it in Card?

Table 1 shows participants, the count of milestones each completed, and their APS. Bravo and Echo did not achieve either milestone for OO abstraction, which emphasizes classes and inter-class relationships, but the remaining eight participants achieved both. Five of those eight achieved one of two milestones associated with OO polymorphism, but none completed both, and all needed high levels of prompting. This supports the idea that the gap between OO abstraction

Table 1: Participant Placement by Level

Level	Participant	Milestones	APS
Level 1	Bravo	4	2.00
(Structured Software Design)	Echo	4	2.00
Level 2	Golf	6	1.67
(OO Abstraction)	India	5	3.60
	Tango	6	1.33
Level 3	Alfa	7	1.67
(OO Polymorphism)	Delta	7	1.43
	Foxtrot	6	2.71
	Lima	7	1.43
	Sierra	7	2.67

to OO polymorphism is large. Further refinement of the model will hopefully confirm the model and create additional delineation and clarity on how to assess students within it.

Discussion

The research questions for this project were:

1. How can polymorphism comprehension be modeled?
2. How can students be assessed within that model?

The case study analysis supports a Polymorphism Comprehension Model with three distinct levels (Structured Software Design, OO Abstraction, and OO Polymorphism Principles, respectively); participants were placed into that model based on design milestones completed. Ten participants achieved Level 1 (completing at least three of four milestones), eight achieved Level 2 (completing both milestones), and five achieved one milestone within Level 3. This confirmed study assumptions that polymorphism would represent a significant gulf in student comprehension.

While qualitative case studies are useful to *build* models, less subjective validation is necessary to *confirm* the model. Future work will need to validate the model to address methodological limitations and investigate what the model implies for improving polymorphism comprehension.

Limitations include:

- The need for different prompting meant that interviews proceeded differently
- The interviewers decided when and how to prompt, adding subjectivity and possibly implicit bias
- The size, origin, and objects-later education of the participant group may have influenced findings
- The specific nature of the assigned problems may have influenced findings

- Recording interviews may have influenced participant reaction; in one study on whiteboard interviews, no women successfully completed a task in a public interview, but all succeeded in a private interview [14]

Improving Polymorphism Comprehension

While substitutability is not the only benefit to subclass polymorphism, it offers a clear justification for the complexity introduced by inheritance. Driven by participant interactions and milestone achievement, further emphasis on substitutability may improve student comprehension. With that in mind, a simplified strategy project that focuses more on substitution than algorithmic problem-solving may better address the gap between student abilities to comprehend OO abstraction and OO polymorphism.

Student ability to design solutions with subclass polymorphism has not been well studied. Having the three-level model provides guidance for future work, and maps well to a scaffolded approach of mastery of advanced language features.

The ultimate goal of this research is to improve polymorphism instruction. While most participants understood the underlying language features, polymorphism remained confusing, demonstrated by the fact that only three participants implemented polymorphic selection, and needed high levels of prompting. For that milestone above, Sierra struggled to identify the relationship between poker and war comparator classes, but the mention of swapping between the two immediately led them to both the comparator superclass and polymorphic selection milestones. This seems to encourage the importance of emphasizing substitutability [19] as the primary benefit of inheritance, rather than reuse of code.

This work demonstrates both the viability and limitations of using whiteboard interviews as a method to evaluate student comprehension with respect to design. Despite challenges, this approach offers a novel means of evaluation, creates an opportunity for industry partnership, and can aid students in preparing for real interviews.

The most pressing concern is to validate the three-level model. Original plans for a larger-scale replication study have been hampered by the pandemic. Plans now call for two separate approaches: semi-structured interviews with industry professionals, and the development and validation of an assessment focused on separating the students at Level 2 (OO Abstraction) from those at Level 3 (OO Polymorphism). While designing the assessment, researchers will focus on the critical prompts leading to participants achieving milestones.

Acknowledgements

This material is based upon work supported by the NSF under Grant Nos. 1525028, 1525173, 1525373.

References

[1] M. B. Rosson and S. R. Alpert, “The cognitive consequences of object-oriented design,” *Human-Computer Interaction*, vol. 5, no. 4, pp. 345–379, Dec. 1990.

[2] D. J. Armstrong, “The quarks of object-oriented development,” *Communications of the ACM*, vol. 49, no. 2, pp. 123–128, Feb. 2006.

[3] C. Schulte and J. Bennedsen, “What do teachers teach in introductory programming?” in *ICER ‘06*. ACM, Sep. 2006, pp. 17–28.

[4] I. Milne and G. Rowe, “Difficulties in learning and teaching Programming—Views of students and tutors,” *Education and Information Technologies*, vol. 7, no. 1, pp. 55–66, Mar. 2002.

[5] J. Bergin, “Teaching polymorphism with elementary design patterns,” in *OOPSLA ’03*. ACM, Oct. 2003, pp. 167–169.

[6] J. Bergin, E. Wallingford, M. Caspersen, M. Goldweber, and M. Kolling, “Teaching polymorphism early,” *SIGCSE Bulletin*, vol. 37, no. 3, pp. 342–343, Jun. 2005.

[7] N. Ragonis and M. Ben-Ari, “A long-term investigation of the comprehension of OOP concepts by novices,” *Computer Science Education*, vol. 15, no. 3, pp. 203–221, Sep. 2005.

[8] A. Schmolitzky, ““objects first, interfaces next” or interfaces before inheritance,” in *OOPSLA ’04*. ACM, 2004.

[9] N. Liberman, C. Beeri, and Y. Ben-David Kolikant, “Difficulties in learning inheritance and polymorphism,” *ACM Transactions on Computing Education*, vol. 11, no. 1, pp. 4:1–4:23, Feb. 2011.

[10] C.-L. Chen, S.-Y. Cheng, and J. M.-C. Lin, “A study of misconceptions and missing conceptions of novice java programmers,” in *Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS)*, 2012, p. 1.

[11] N. Mills, A. Wang, and N. Giacaman, “Visual analogy for understanding polymorphism types,” in *Australasian Computing Education Conference*. New York, NY, USA: ACM, Feb. 2021, pp. 48–57.

[12] B. Y. Alkazemi and G. M. Grami, “Utilizing BlueJ to teach polymorphism in an advanced object-oriented programming course,” *Journal of Information Technology Education*, 2012, accessed: 2021-8-5.

[13] M. Behroozi, A. Lui, I. Moore, D. Ford, and C. Parnin, “Dazed: Measuring the cognitive load of solving technical interview problems at the whiteboard,” in *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*. ieeexplore.ieee.org, May 2018, pp. 93–96.

[14] M. Behroozi, S. Shirolkar, T. Barik, and C. Parnin, “Does stress impact technical interview performance?” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Nov. 2020, pp. 481–492.

[15] D. Ford, T. Barik, L. Rand-Pickett, and C. Parnin, “The Tech-Talk balance: What technical interviewers expect from technical candidates,” in *2017 IEEE/ACM 10th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. IEEE, May 2017, pp. 43–48.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1994.

[17] R. E. Johnson and B. Foote, “Designing reusable classes,” *Journal of object-oriented programming*, vol. 1, no. 2, pp. 22–35, 1988.

[18] A. Hunt and D. Thomas, “The trip-packing dilemma [agile software development],” *IEEE Software*, vol. 20, no. 3, pp. 106–107, May 2003.

[19] B. H. Liskov and J. M. Wing, “A behavioral notion of subtyping,” *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 6, pp. 1811–1841, Nov. 1994.