Goals and Principles for the Redesign of a Programming Course

Erno Vanhala
University of Tampere
erno.vanhala@uta.fi

Jussi Kasurinen
Lappeenranta University of Technology
jussi.kasurinen@lut.fi

ABSTRACT

In this study it is discussed how programming courses have been redesigned. The article is based on three courses lectured during academic years from 2005 to 2016. During these years several modifications and revisions were made, such as moving a course from Python 2 to Python 3, another one from C++ to Java and in one occasions updating the course to include the latest web development concepts and technologies. The first course included video lectures and the latter two were upgraded to utilize flipped classroom teaching method. From these revisions the student feedback was collected and examined to gain understanding on what ideas work and what do not. The overall results could be summarize in four key concepts: 1) provide easy-to-use working environment 2) give students freedom, 3) fortify the frequency of key concepts and 4) separate theory to pre-classroom learning and action to in-classroom learning.

CCS CONCEPTS

K.3.2 [**Computer and Education**]: Computer and Information Science Education – *computer science education*.

KEYWORDS

programming, object-oriented programming, flipped classroom, redesign principles

ACM Reference Format:
Erno Vanhala and Jussi Kasurinen. 2018. Goals and Principles for the Redesign of a Programming Course. In *Proceedings of The 2018 Workshop on PhD Software Engineering Education: Challenges, Trends, and Programs (SWEPHD2018). St. Petersburg, Russia, 6 pages.*

1 INTRODUCTION

Why bother changing a programming course unless you have to? The question is, of course, what is a good enough reason to introduce changes which are likely to cause change resistance, increased effort, and worse short term results [7]?

The 2018 Workshop on PhD Software Engineering Education: Challenges, Trends, and Programs, September 17th, 2018, St. Petersburg, Russia

However, as the passing rates of case courses have improved, a more fundamental question becomes interesting: whether the students actually learn programming skills in the courses. The level of learning has not been studied much in the programming education field but, for example, a Bayesian Knowledge Transfer algorithm has been proposed to fit to estimate the learning of specific programming structures [2,14].

The redesign process started in 2006 with an attempt to remove observed problems in the Fundamentals of programming, CS1 (Case A). Similarly, teaching methods became interesting when it was observed how students appreciated video lectures introduced in Case A. With Case B and C it was developed further with flipped classroom method.

The overall philosophy for revising the case courses has been to make it easy for the students to download the programming environment and start using it. We have a few aims in improving programming courses: 1) reducing dissatisfaction, 2) increasing motivation and 3) estimating learning. In this article it is reported the goals of the course redesigns.

2 RELATED RESEARCH

In this section related research is presented to introduce the topics that are relevant when discussing the reasons and the process of redesigning of a programming course.

2.1 Student motivation

Student motivation has raised considerable interest among the researchers [3,21]. Besides the hygiene factors – elements (e.g. air conditioning or programming manual) that are not the actual key components of the process, but can greatly affect the success of the task (e.g. office work or learning programming) – there are also motivating things on learning, which can greatly affect on the actual course outcome [21].

For example, the assignments and lectures on the course have to be easy enough to be understandable for all, but they also should challenge the most advanced students. Students with less IT skills tend to frustrate near the end of the course when assignments get harder. On the other hand, students with existing programming skills are frustrated since they are not challenged, and are just required to participate. There are, however, ways to detect frustration based on compilation logs and time spent on Virtual Learning Environment (VLE) [22]. When a students get overly frustrated they can easily lose their confidence [10] and will get a grade worse than their actual skill level would indicate.

In principle, the students who have positive impression on a subject tend to be motivated [21]. Motivated students also have a positive perception of the subject and amount of practical work [21]. In this sense, it is obvious that these traits should be supported, by providing programming assignments, which the students find motivational [24].

2.2 Student dissatisfaction

Hygiene factors [11] are not widely studied in the area of introductory programming, but few studies can be found (e.g. [13,15]). Before a student can be motivated, the basic hygiene factors need to be present [11]. In programming courses these Herzberg's hygiene factors include, for example, comfortable learning environment and learning conditions. The first one contains the lecture halls and computer classes, and the latter material supporting the learning and students' time schedules.

When the hygiene needs and requirements are met, the motivational aspects become increasingly important to increase the student satisfaction. Herzberg introduces 7 principles [11] from which three can be easily modified to be used in teaching programming: 1) In the learning environment it is quite easy to give the students access to the scores of one's programming tasks. 2) When giving tasks to the students it is possible to give several tasks covering different degree of difficulty. 3) Access to additional tasks providing deeper understanding to engage advanced students in the course.

2.3 Course success measurement

After the course implementation has been carried out, usually the final survey has been given to the students. Based on this feedback it is possible to pinpoint difficult parts of the course topics and unsuccessful ideas or failed implementations.

Yet, a course success measure should reflect the actual objective of course participation, the learning of the course contents. The learning processes have been studied, for example, from the point of view how novice programmers understand programs [23], how they structure their own code [5,20], and how the programming knowledge can be measured – the Bayesian Knowledge Tracing algorithm (BKT) [2,6,8] and Adaptive Control of Thought - Rational (ACT-R) [1]. The ACT-R is focused on long-term learning, skill acquisition, and deterioration while the BKT algorithm has been reported to have a history of success in programming and algebra, and thus the BKT has been reported sufficient for skill mastery estimation [8].

The principles of the BKT algorithm are outlined in Figure 1: the prior knowledge (L_n) is taken into account when the probability of learning the concept in question is calculated (p(T)) to measure the student learning. As student repeats the process, the possibility of guessing the right answer (p(G)) and to err even when the concept is learned (p(S)) are taken into account. It has been estimated that repeating the process for six times predicts the learning of the concepts in question in a reasonable level.

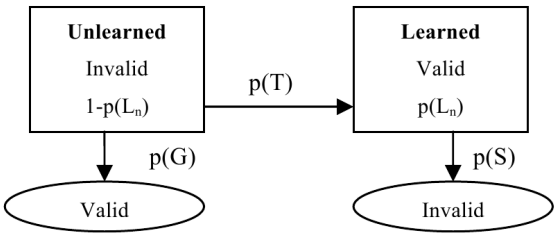

Figure 1. Bayesian Knowledge Tracing algorithm [14].

2.4 Flipped classroom method

As technology develop it also opens new methods to improve teaching methods. The core idea of flipped class room is to let students to study theory on their own outside the classroom and concentrate on actual doing in class with teacher. This method is already used in computer science on various course [12,16]. Flipped classroom can also be used in other areas from primary school to university level education although it originated from economics [4].

3 RESEARCH METHOD

The present study started with two research questions: 1) When should a programming course be revised? and 2) What should be taken into account when redesigning?

In general the study had three objectives: First to demonstrate how the collected data can be used to pinpoint problems and development needs in the course. Second, to report the goals and principles for the course redesign. The third objective was to assess how programming assignments could be developed to take into account the developed principles. In this sense, the study has the characteristics of both natural and design science [17] but it is reported as a case study since the case study methods [25] fit well the study and, in general, the courses referred to in the study represent case studies with literal replication [25].

The data used in the study comes from three courses taught in a Finnish university, between 2005 and 2016. Each course has been concluded with a final survey that has been distributed to all students enrolled in the course. The final surveys have included quantitative questions like the difficulty and usefulness of the different course elements but also open questions to allow the students to express their feelings and concerns about the course.

The analysis has been based on both quantitative and qualitative methods. In particular, the BKT analysis is based on quantitative measurements and statistical analysis while the problem analysis has been based more on the qualitative data acquired from multiple sources. For example, problematic weekly assignments have been identified by analyzing the points each assignment got but deciding about the changes required by each assignments has been estimated case by case by the designer even though the student problem reports and feedback have helped a lot in some cases.

4 CASE STUDIES

The next chapters describe three case courses, their descriptions and upgrades done to them.

4.1 Course descriptions

This study utilizes three programming courses. Table 1 lists key features for the courses. All the courses have positioned in bachelors level, but also not computer science students from masters level have taken part of the courses.

Table 1: Key features for the case courses

Case	Name	Key changes	Data
A	Fundamentals of programming	Introduction of lecture videos, change from Python 2 to Python 3, programming manual	2006 – 2015
B	Object-oriented programming	Change from C++ to Java and change from lecture videos to flipped classroom, programming manual	2010 – 2016
C	Webbed applications	Introduction of flipped classroom and updating all the materials	2015

For Case A, a larger Python programming manual was written to replace a course book, since at the time there was no book available in the local language. When the course was updated to Python 3, the number of weekly programming tasks was increased and the format of the programming project was changed from GUI-based "motivational" project to text- and calculation based real-life problem.

Case B followed examples of Case A as it had video lectures, but in the end it was decided to change the programming language of the course from C++ to Java and in the same time to drop the traditional lectures-exercises-exam paradigm and move to the flipped classroom. In addition, two programming manuals were written; one for Java in general and one for GUI-programming in Java.

Similarly, Case C was transformed to flipped classroom method and all the course materials were updated to reflect the fast development of web programming techniques.

All the case courses consisted of 12–14 weeks depending on the year it was lectured. All the courses followed the format where besides the introductory lecture, each week a new concept was introduced and the previous topics were utilized with the newly learned skills.

4.2 Changes in the course

4.2.1 Hygiene factors

The first major observation was the students were complaining about very basics on case courses, for example "Where do I get the needed software?" or "How do I return my assignments?".

One of the key principles when revising and improving courses was to remove these hygiene factor obstacles. Python was selected as the programming language for Case A as it can be easily installed on Windows, Mac and Linux computers and it already includes IDLE as IDE. IDLE itself is easy-to-use and does not throw everything to students at once, and it has a minimalist and simple look, which replicates between the different platforms, which suits CS1 where the students meet programming code for the very first time.

With Case B it was also thought how to give students the suitable IDE with minimal work to be done. As Case B had also GUI-programming and the JavaFX was chosen as the GUI-toolkit it was decided to use NetBeans as the IDE as both JavaFX and NetBeans were bundled together from the developers of Java.

As web developing can be done with pretty much every tool available, and the course required both backend and frontend development, the Case C was a different scenario. The most pressing issue was the server side system; how this should be addressed in the university infrastructure, since the system had to be usable by the students, while still retaining a certain level of cyber security. In this case, the solution was to use a cloud platform that enabled the option to run the student-generated code without the need of installing web server, database and various libraries. Additionally, some students decided to use their own environments and followed guideline videos on how to install the necessary packages to Linux or Mac.

Case A and half of the Case B utilized VLE for the assignment submission and grading, since the course had thousands of student-submitted works. Since this VLE could not manage graphical user interfaces, the second half of the Case B and Case C applied the traditional teacher grading, but by allowing the students to demonstrate their solutions with quick demonstration at the exercise event, or by sending a video link explaining what they had done. On latter implementations, a peer review approach by other students was also applied.

Teaching materials were provided for every case course. With Case A this meant the programming manual, style guide and Python installation manual with lecture examples and videos. Case B had videos covering all the course topics, two written manuals and code examples available in a git repository. Case C had videos covering all the course topics, short manual presenting the most important techniques and tools used in web programming and code examples in a git repository.

One of the usually overlooked hygiene factors is the exam. When learning to program it is normal to ask students to write a short program in the exam, yet when the exam is pen and paper, it does not reflect to the real world where the students would have compiler messages and manuals with them. To avoid this issue Case B introduced online exam and Case C had no exam at all, but an essay to show how students had learned their new skills. In the end Case A also introduced an online exam for programming as it would decrease the manpower needed to grade all the exams.

4.2.2 Changes done to the weekly programming assignments

With all cases weekly programming assignments were constructed so that they had usually 5 smaller tasks thus students

were able to gain 5 points from one week. With Case A example solutions were provided and they were coded to show how style guide help to build easy-to-read programming code.

With all cases the number of weekly assignments were also increased – though some tasks were divined into two smaller tasks – thus students had to repeat the key structures and concepts at least six times as suggested. Although one has to note that techniques presented in the end of the course could not be covered such many times.

The BKT analysis was done with the program source codes retrieved from the VLE database after the course, and the prior programming knowledge of the students was estimated with an initial survey in the beginning of the course. When the programming assignments were developed in 2006 it was only aimed at reasonable assignments, and only few programming structures met the BKT analysis requirements. The overall outcomes were up 10–20% on all measured categories.

Other positive side of the increased number of assignments was that average students did not need to do all the five tasks per week, but students with advanced skills had more to do. The last task is set to be the hardest one, so the students who were interested in programming could get more challenge when they wanted.

Students focus on tasks that are beneficial to them, that is, tasks giving points to them. All the case courses had previously weekly exercises that were voluntary and there were no points given from a successful solutions. When the case courses were revised all of them shift to a model where students could get points from all the work they do for the course and the final grade was calculated from the points got from weekly assignments, course projects, exam, and other parts of the course.

4.2.3 Course project

All of the cases have larger programming project in the end of the course. The previous version of the Case A was taught with C programming language and its course project was transformed to the first implementation of Python course. The text-based project was not seen up-to-date and it was then changed GUI-based Turtlet programming project. Although it was *supposed* to be motivating, students gave mixed feedback, similarly as observed for example by [18]. Turtlet had limitations on the usefulness and student creativity as well as maintenance problems, so a new project was to developed for the course.

With Python 3 the programming project used more engineering approach. First theme on the project was to calculate district heating systems. The project was approximately double the size of the old Turtlet version, yet it did not yield complaints more than the Turtlet. Nor was it completely success. The project required a lot of precise mathematical calculation that the students found hard to get right. The project was later moved to work with temperatures, data files and generating svg-graphics.

Case B had had various humorous – game-like – projects when it was implemented with C++. The projects required to understand the concepts of C++ but they did not reflect to real problems or were not in an area, which would have been useful experience for the future. When Case B was transformed to Java the course project was also redesigned.

The new project included parsing publicly available open data sources and building a graphical user interface to visualize the data on a map overlay. This gave students a feel of working with the real-world problems, tools and data.

With Case C the course project was not focusing on any specific area. As the Case C is more advanced course than A or B, it was decided to give students more creativity and responsibility with the course project. There were elements that gave certain amount of points and combining several elements students could get the number of points they wanted to get certain grade. For example creating responsive design gave 5, utilization of cache gave 3 and front-controller design pattern gave 3 points.

4.2.4 Lectures vs. flipped classroom

The Case A provided video lectures from the beginning of Python era. Students found the videos very useful. Also with Case B video lectures were provided from the beginning of data collection period. When Case B was revised it started to use flipped classroom method. Similarly Case C was revised to flipped classroom when data collection started with new administrating lecturer.

Both video lectures and flipped classroom seem to be suitable when teaching programming. Students have given feedback on how they value the opportunity to watch theory when they have time and repeat it as many times they need.

With flipped classroom, although the initial cost of creating new course format is high it later gives more man power to be used in classroom teaching then benefiting the students.

5 DISCUSSION

In the beginning two research questions where set: 1) when should a programming course be revised? 2) what should be taken into account when redesigning? Here the questions are discussed based on the experience gained from three cases described in the previous chapter.

5.1 When should a programming course be revised?

The present study explored the reasons for initiating the programming assignment revision and found four key reasons for the revision:

Problems with existing assignments. The students reported different kind of defects in the assignments over a number of years which indicated that there were problems with the assignments. Collected data suggests that problems in the programming assignments reduced the student motivation to complete them, and thus such problems became the hygiene problems in these courses. Thus, to avoid dissatisfaction – hygiene problems – among the students, the programming assignments should not pose undue problems for the students.

Mismatch between the students and the assignments. Even though the assignments would not have any problems, the

students may not be motivated by them. Collected data suggests that the students were concerned about the usefulness of the assignments they had to complete and especially the game-like project with graphical user interface (Case A) had been criticized every year it was used. Thus, to keep the programming assignments motivating for the students they should be aligned with the student experiences and expectations as the student body evolves. This observation is in line with the earlier work from for example [3].

Technological development. Programming languages and tools develop quickly. With Case A the change from Python 2 to 3 required to check all the material and assignments in the course. In Case B, where the programming language changed, all the assignments needed to be redone. And web programming (Case C) changes all the time so the revision of everything was a mandatory task to be carried out at least once in five years.

Pedagogical development. The teaching methods and tools are improved all the time. Where PowerPoint slides came in the 90s, this millennium has given video lectures and flipped classroom. Teachers need to match the new ways of studying and thus courses need to be revised also from pedagogical point of view. The traditional lecture-exercise-exam model can be replaced with modern ways where students get more individual time from teachers. With Case A video lectures where provided all the time with Python course and with Case B and C the transition to flipped classroom was carried out when courses where revised. Both the video lectures and flipped classroom method generated praises from students.

5.2 What should be taken into account when redesigning?

The response to the second research question of what should be taken into account in the redesign is twofold. The problems leading to the revision should, of course, be fixed but a revision provides also an opportunity to improve the course. In the present study the revisions made it possible to assure that the assignments were fit estimating the learning outcomes of the course with the BKT algorithm. Overall the following design principles for programming assignment revision can be pointed:

- Motivating assignments reflecting the real world problems, useful for the studies as well as the future careers and assignments as engineers
- Give an option to do more assignments when the topic requires them
- Repeat all the main programming structures at least six times in the weekly assignments to allow accurate mapping of the learned topics with BKT or ACT-R algorithm
- Follow both the technological development and pedagogical development to be able to utilize up-to-date tools and methods

5.3 Retrospective

The reported modifications to the programming courses have been made during the years 2005 – 2016. After few years of cool down it can now be discussed what parts of improvements were success and what parts still require work.

With Case A the most difficult part has been developing a programming project that would be easy enough for beginners but would also let students to show their skills if one wants to. This is an issue that has no real solutions, but it can be iterated towards project that would have all the necessary parts required and still be easy and useful.

Case B failed in grading for the first implementation after revision. Points were provided too easily and students got high grades – although they did good work. Some example solutions were also not perfect and issues arose when JavaFX required special version of Java, which was not installed by default. These hygiene factors were then hot-fixed, but would still require more work to be done.

The biggest issue with Case C is the number of tools and techniques web development has. In the course dozen of different techniques are introduced to students and then they do not have enough time to repeat newly learned skills. This problem would be solved by extending the course somehow or by splitting it into two separated courses.

With all the cases teachers are required to do a lot of grading. With Case B it was changed so that students peer reviewed programming project so that teachers could focus more on teaching. This similar method could be considered also for Case A and C.

5.4 Limitations and validity of the study

As this is a partly qualitative study, the observations presented here are not strong, confirmatory results, but guidelines and sophisticated suggestions on things that have been discussed in this article [9]. Although we have triangulated the data against quantitative data sources, this study is not free from possible bias towards any direction that has been missed. Bias is addressed in the following ways. We have followed the three principles of data collection [25]: we have used multiple sources of evidence, created a multicase study [19], and maintained a chain of evidence. We have also triangulated our data from multiple sources (e.g., student surveys and feedback, VLE program database, and grading data), both authors have participated in the analysis, we have used different theories, and we have used both qualitative and quantitative methods to analyze the data.

6 CONCLUSION

This article presented three university level case courses, which had all been revised and improved. When revising a course one should note, for example, to follow the latest technical and pedagogical tools and methods, make sure students repeat learned tasks at least six times and motivate students with real-life assignments.

This is a latest wrap up of the on-going improvement of these three courses. The shift from lecture room lectures to video lectures and flipped classroom method has already widely begun, but there is still much to do. For example, only Case A was a mass course and only Cases B and C where flipped. In the future

it is required to study whether same methods can be used when flipping a mass course.

7 REFERENCES

[1] J. R. Anderson and C.D. Schunn. 2000. Implications of the ACT-R Learning Theory: No Magic Bullets. *Advances in instructional psychology 5*.

[2] Ryan Baker, Albert T. Corbett, and Vincent Aleven. 2008. More Accurate Student Modeling through Contextual Estimation of Slip and Guess Probabilities in Bayesian Knowledge Tracing. 406–415.

[3] Tiffany Barnes, Eve Powell, Amanda Chaffin, and Heather Lipford. 2008. Game2Learn: improving the motivation of CS1 students. In *Proceedings of the 3rd international conference on Game development in computer science education*, 1–5.

[4] Jacob Lowell Bishop and Matthew A. Verleger. 2013. The flipped classroom: A survey of the research. In *ASEE National Conference Proceedings, Atlanta, GA*.

[5] Janet Carter, Kirsti Ala-Mutka, Ursula Fuller, Martin Dick, John English, William Fone, and Judy Sheard. 2003. How shall we assess this? In *Working group reports from ITiCSE on Innovation and technology in computer science education*, 107–123.

[6] Cristina Conati, Abigail Gertner, and Kurt VanLehn. 2002. Using Bayesian Networks to Manage Uncertainty in Student Modeling. *User Modeling and User-Adapted Interaction* 12, 4 (November 2002), 371–417. DOI:https://doi.org/10.1023/A:1021258506583

[7] P.D.I. Elrod and D.D. Tippett. 2002. The "death valley" of change. *Journal of Organizational Change Management* 15, 3 (2002), 273–291.

[8] Kevin A. Gluck. 2004. Knowledge Tracing for Complex Training Applications: Beyond Bayesian Mastery Estimates. 383–384.

[9] Nahid Golafshani. 2003. Understanding Reliability and Validity in Qualitative Research. *The Qualitative Report* 8, 4 (2003), 597–606.

[10] Stuart Hansen and Erica Eddy. 2007. Engagement and frustration in programming projects. In *Proceedings of the 38th SIGCSE technical symposium on Computer science education*, 271–275.

[11] Frederick Herzberg. 1968. One more time: How do you motivate employees? *Harvard Business Review* 46, 1 (1968), 53–62.

[12] Diane Horton and Michelle Craig. 2015. Drop, Fail, Pass, Continue: Persistence in CS1 and Beyond in Traditional and Inverted Delivery. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (SIGCSE '15), 235–240. DOI:https://doi.org/10.1145/2676723.2677273

[13] Jussi Kasurinen and Uolevi Nikula. 2007. Revising the First Programming Course - The Second Round. 92–101.

[14] Jussi Kasurinen and Uolevi Nikula. 2009. Estimating programming knowledge with Bayesian knowledge tracing. In *Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education*, 313–317.

[15] Jussi Kasurinen, Mika Purmonen, and Uolevi Nikula. 2008. A Study of Visualization in Introductory Programming. 181–194.

[16] Mary Lou Maher, Celine Latulipe, Heather Lipford, and Audrey Rorrer. 2015. Flipped Classroom Strategies for CS Education. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (SIGCSE '15), 218–223. DOI:https://doi.org/10.1145/2676723.2677252

[17] S.T. March and G.F. Smith. 1995. Design and Natural Science Research on Information Technology. *Decision Support Systems* 15, 4 (1995), 251–266.

[18] William Isaac McWhorter and Brian C. O'Connor. 2009. Do LEGO® Mindstorms® motivate students in CS1? *SIGCSE Bull.* 41, 1 (2009), 438–442.

[19] Christine B. Meyer. 2001. A Case in Case Study Methodology. *Field Methods* 13, 4 (November 2001), 329–352. DOI:https://doi.org/10.1177/1525822X0101300402

[20] Keir Mierle, Kevin Laven, Sam Roweis, and Greg Wilson. 2005. Mining student CVS repositories for performance indicators. In *Proceedings of the 2005 international workshop on Mining software repositories*, 1–5.

[21] Matthew Mitchell, Judy Sheard, and Selby Markham. 2000. Student motivation and positive impressions of computing subjects. In *Proceedings of the Australasian conference on Computing education*, 189–194.

[22] Ma. Mercedes T. Rodrigo and Ryan S.J.d. Baker. 2009. Coarse-grained detection of student frustration in an introductory programming course. In *Proceedings of the fifth international workshop on Computing education research workshop*, 75–80.

[23] B Simon, R Lister, and S Fincher. 2006. Multi-Institutional Computer Science Education Research: A Review of Recent Studies of Novice Understanding. 12–17.

[24] Alan L. Tharp. 1981. Getting more oomph from programming exercises. In *Proceedings of the twelfth SIGCSE technical symposium on Computer science education*, 91–95.

[25] R.K. Yin. 2002. *Case Study Research: Design and Methods* (3rd edition ed.). SAGE Publications., Thousand Oaks, CA.

programming, object-oriented programming, flipped classroom, redesign principles. ACM Reference Format: Erno Vanhala and Jussi Kasurinen. 2018. Goals and Principles for the Redesign of a Programming Course. In Proceedings of The 2018 Workshop on PhD Software Engineering Education: Challenges, Trends, and Programs (SWEPHD2018). St. Petersburg, Russia, 6 pages. 1 INTRODUCTION. Why bother changing a programming course unless you have to? The question is, of course, what is a good enough reason to introduce changes which are likely to cause change resistance, increased effort, and worse short term The Object-Oriented Design Principles are the core of OOP programming, but I have seen most of the Java programmers chasing design patterns like Singleton pattern, Decorator pattern, or Observer pattern, and not putting enough attention on learning Object-oriented analysis and design. It's important to learn the basics of Object-oriented programming like Abstraction, Encapsulation, Polymorphism, and Inheritance.Â Composition allows changing the behavior of a class at run-time by setting property during run-time and by using Interfaces to compose a class we use polymorphism which provides flexibility to replace with better implementation any time. Even Effective Java advise favoring composition over inheritance. The Object-Oriented Design Principles are the core of OOP programming, but I have seen most of the Java programmers chasing design patterns like Singleton pattern, Decorator pattern, or Observer pattern, and not putting enough attention on learning Object-oriented analysis and design. Itâ€™s important to learn the basics of Object-oriented programming like Abstraction, Encapsulation, Polymorphism, and Inheritance.Â Another thing you can do is to join a comprehensive object-oriented design course like SOLID Principles of Object-Oriented Design by Steve Smith on Pluralsight.Â A programmer should always program for the interface and not for implementation this will lead to flexible code which can work with any new implementation of the interface.