Concurrent Software Engineering and Robotics Education

Jiwon Shin, Andrey Rusakov, and Bertrand Meyer
Chair of Software Engineering
Department of Computer Science
ETH Zurich, Switzerland
Email: {jiwon.shin, andrey.rusakov, bertrand.meyer}@inf.ethz.ch

***Abstract*—This paper presents a new, multidisciplinary robotics programming course, reports initial results, and describes subsequent improvements. With equal emphasis on software engineering and robotics, the course teaches students how software engineering applies to robotics. Students learn independently and interactively and gain hands-on experience by implementing robotics algorithms on a real robot. To understand the effects of the course, we conducted an exit and an 8-month survey and measured software quality of the students' solutions. The analysis shows that the hands-on experience helped everyone learn and retain robotics well, but the students' knowledge gain in software engineering depended on their prior programming knowledge. Based on these findings, we propose improvements to the course. Lastly, we reflect our experience on andragogy, minimalism, and interactive learning.**

I. Introduction

With advancement of technology, the importance of good software engineering has extended far beyond the traditional computing devices and fields; however, most university-level software engineering courses still do not address the need for quality software outside of computer science. Consequently, computer science students rarely gain hands-on experience with a real system, and students of other fields may learn how to *code* but not how to *engineer* software. Even in fields such as robotics, where software engineering is an essential component, current educational system pays little attention to software engineering. Indeed, most university-level robotics courses that teach robotics algorithms cover little to no software engineering topics [1], [2].

Robotics has gained attention as a medium to teach science and engineering concepts [3], [4], and computer science is no exception. Several introductory courses have taught computer science using a robot, initially with a negative result [5] but lately more positive results [6], [7]. This effort has, however, been limited to introductory courses. Although Gustafson proposed the idea of teaching software engineering using robotics over 15 years ago [8], to the authors' knowledge, only one such course exists [9]. This paper presents a new course that tries to address this gap in software engineering and robotics education and evaluates the initial experience.

Robotics programming laboratory is a new, multidisciplinary, elective course for master's-level students at ETH Zurich. Open to students of computer science, electrical engineering, and mechanical engineering, the course teaches software engineering principles, concurrency, and architecture through hands-on learning in robotics context. Independent and interactive learning are at the core of our course. We give each student a personal robot, use online tutorials, require both independent and group work, host in-class demonstrations, and encourage students to communicate via an online forum. To understand the effect of concurrent software engineering and robotics education, we conducted an exit survey and a survey 8 months later. Our analysis of the surveys and measuring software quality of the students' solutions reveal that the course taught robotics well to everyone, but those with little prior programming knowledge did not learn as much software engineering. Based on these findings, the paper proposes an improved course that addresses the issues identified in the pilot course. The paper also reflects the initial experience on andragogy, minimalism, and interactive learning.

This paper is organized as follows. After a brief introduction and an overview of related work in Sec. I, the paper continues with theory in Sec. II. Sec. III presents the pilot course and Sec. IV evaluates how successfully the pilot course addressed its objectives. Sec. V presents an improved version of the course. The paper then discusses the results of the pilot study in Sec. VI and presents threats to validity in Sec. VII. Sec. VIII concludes the paper with final remarks and future work.

II. Theory

The theoretical frameworks for our paper are andragogy, minimalism, and interactive learning. Andragogy and minimalism have been used to explain success in computer science courses that use robots [10]. Interactive learning is hypothesized to be more effective than constructive learning [11]. This section briefly explains these theories.

A. Andragogy

Andragogy [12], [10] aims to explain how adults learn. It theorizes that adults learn differently from children and should be taught accordingly. In particular, it makes the following six assumptions:

1) Adults have self-concept of a learner. Teachers should encourage and nurture adults' need to be self-directing.
2) Adults have prior experience. Teachers should employ experiential techniques to enable adults to utilize their accumulated experience and learn from experience.

3) Adults are ready to learn. Learning programs should be organized around life-application categories and sequenced according to the learners' need to learn.
4) Adults are problem-centered. Learning experiences should be designed around competency-development categories.
5) Adults need to know the importance and the benefits of gained knowledge.
6) Adults are internally motivated, but external motivators can increase the internal motivation.

B. Minimalism

Minimalism [13] assumes that people will reason creatively and improvise when they are engaged in a task. The role of an instructor is to create an environment for creative reasoning and to give appropriate support when an error occurs. Its four principles are:

1) to choose action-oriented approach by introducing the task in the beginning,
2) to anchor tools in the task domain so that the task is easy to understand,
3) to support the learner's error recognition and recovery while keeping the learner active and motivated, and
4) to support reading to do, study, and locate such that every piece of information is self-contained.

C. Interactive learning

Chi [11] classifies different learning activities in terms of observable overt activities and underlying learning processes. The learning activities are passive, active, constructive, and interactive. In passive learning, students enter the learning environment with open mind and accept the information. Active learning requires students to do something physically. Constructive learning involves producing outputs that contain ideas that go beyond the presented information. Interactive learning requires dialoguing substantively on the same topic, and not ignoring a partner's contributions. She hypothesizes that interactive activities are most likely to be better than constructive, active, or passive activities.

III. Pilot Course

Robotics programming is a master's-level, elective, laboratory course that gives 8-credits (240 hours of workload). The course is open to computer science (CS), electrical engineering (EE), and mechanical engineering (ME) students.

A. Course objectives

The main objectives of the course are that students gain

- basic software engineering principles and methods,
- common architectures in robotics,
- coordination and synchronization methods,
- how software engineering applies to robotics, and
- hands-on experience by programming a small robotic system with aspects of sensing, control, and planning

B. Course structure

The course is 14 weeks long and has one lecture (2 hrs) and one exercise session (2 hrs) a week. Each lecture introduces a new software engineering and/or robotics topic, and exercise sessions provide further assistance on the assignments. The course has four graded assignments and no exam.

	Wk	Topic	SE	R
L1	1	Intro to SE and robotics	x	x
L2	2	ROS and Roboscoop	x	x
L3	3	SCOOP	x	
L4	4	Control and obstacle avoidance		x
L5	5	Design patterns	x	
L6	6	Localization		x
L7	7	Mapping		x
L8	8	Modern SE Tools	x	
L9	9	Path planning		x
L10	10	Object recognition		x
L11	11	Software architecture in robotics	x	
		No lecture (weeks 12 – 14)		

Table I. Lecture topics and schedule. Topics cover software engineering (SE) and/or robotics (R).

1) Lectures: The lecture topics are chosen to give students balanced exposure to software engineering and robotics (Tab. I). The software engineering lectures address the first three objectives: software engineering principles and methods (L1, L5, L8), architecture (L2, L11), and concurrency (L2, L3). The robotics lectures introduce algorithms through which students apply their software engineering knowledge and gain hands-on experience.

The software engineering lectures provide an overview of various aspects of software engineering, with more emphasis on topics and tools relevant for the course. The lecture topics are as follows:

- **Introduction to software engineering (L1)**: This lecture introduces software engineering and discusses software product and process. In particular, the lecture delves into software quality factors and different software processes. Mentioned software quality factors include reliability, ease of use and learning, efficiency, extendibility, reusability, and portability. The software processes include CMMI, Agile, Waterfall, Spiral, and Cluster.
- **ROS (L2) and software architecture in robotics (L11)**: These lectures present different robotics middlewares and their architecture. The architecture lecture discusses CARMEN [14] and its model-view-controller architecture, MOOS [15] and its star topology with layered architecture, Microsoft Robotics Studio [16], and ROS [17] and its peer-to-peer architecture. In addition, as the course uses ROS, we dedicate a one-hour lecture to ROS and cover it in detail.
- **Roboscoop (L2) and SCOOP (L3)**: These lectures cover concepts of concurrency and its usage in robotics. Simple Concurrent Object-Oriented Programming (SCOOP) [18], [19] is an object-oriented programming model for concurrency, and Roboscoop [20] is a robotics framework built on top of SCOOP. SCOOP

guarantees the absence of common concurrency errors such as data races, and it has been shown to be easier to learn than Java threads [21]. In the SCOOP lectures, students learn about four risks of concurrency – data race, deadlock, starvation, and priority inversion – and the SCOOP model. The Roboscoop lecture covers how the framework extends SCOOP to bring easy concurrency to robotics.

- **Design patterns (L5)**: This lecture teaches design patterns that are useful for the class, namely, observer, state, strategy, and visitor. The observer pattern is extensively used in ROS for communication. The state pattern is useful for implementing the obstacle avoidance algorithm, which consists of several states. The strategy pattern enables different path planning strategies to be swapped easily. The visitor pattern is useful for implementing different predict and update methods in localization.
- **Modern software engineering tools (L8)**: This lecture covers IDEs, debuggers, refactoring tools, profilers and performance analyzers, automatic testing tools, and configuration management tools. The lecture teaches best practices when working with these tools and gives demonstrations of the tools. The demonstrations help students combine and integrate the tools into their software development process for better time and team management.

The robotics lectures introduce students to essential algorithms in robotics. The lectures cover the following:

- **Control and obstacle avoidance (L4)**: This lecture teaches about differential-drive robot, odometry computation, PID controller, and different bug algorithms for obstacle avoidance.
- **Localization (L6)**: This lecture introduces three different localization algorithms, namely, Markov localization, Kalman filter localization, and particle filter localization.
- **Mapping (L7)**: This lecture covers occupancy grid mapping and Simultaneous Localization and Mapping (SLAM).
- **Path planning (L9)**: This lecture teaches various graph construction methods and path planning algorithms. The graph construction methods include visibility graph, Voronoi diagram, exact cell decomposition, and approximate cell decomposition. The path planning algorithms include deterministic algorithms such as Dijkstra's algorithm and A* search, randomized graph search algorithm, and potential field path planning.
- **Object recognition (L10)**: This lecture provides an overview of object recognition process, from segmentation to feature extraction and classification, both for 2D and 3D data.

The course follows no textbook; instead, we recommend some books as references and utilize online tutorials. Recommended software engineering books are *Object-Oriented Software Construction* [22], *Design Patterns* [23], and *Pattern-Oriented Software Architecture Volume 2* [24]. Recommended robotics books are *Probabilistic Robotics* [25] and *Introduction to Autonomous Mobile Robots* [26]. In addition, we ask students to follow the ROS tutorials [1] and the PCL tutorials [2].

	Week	Topic	I	G
A0	1 – 3	Setup (no grade)	x	
A1	4 – 6	Control and obstacle avoidance	x	
A2	7 – 10	Localization	x	
A3	11 – 12	Path planning		x
A4	13 – 14	Search and rescue		x

Table II. Assignments. Some were completed individually (I) and others in a group (G) of two.

2) Assignments: The course has one ungraded and four graded assignments (Tab. II). The ungraded assignment gives students time to get familiar with the working environment. The four graded assignments require students to apply their software engineering skills to robotics by implementing core robotics algorithms from scratch. The assignments are completed either individually (A1, A2) or in a group of two (A3, A4). Having both individual and group assignments ensures that every student learns the basics and encourages them to share their knowledge. For A3 and A4, we introduce intermediate goals to aid the students' time management.

Fig. 1. Dolls and a rubber duck used for search and rescue.

The four graded assignments are as follows:

- **Control and obstacle avoidance (A1)**: This assignment has two tasks – implementing a PID controller for going to a goal and implementing the TangentBug algorithm [27], an improved Bug obstacle avoidance algorithm for robot with range sensors. For going to a goal, the PID controller simply controls the angle to the goal. For the TangentBug, the PID controller also controls wall following by keeping a constant distance from the closest obstacle.
- **Localization (A2)**: This assignment requires students to implement a particle filter localization algorithm. Given a map and a goal location, students demonstrate how their robot can go from initially unknown location to a predefined location in the map.
- **Path planning (A3)**: This assignment adds A* path planner to their software. Students demonstrate how their robot can go from a starting point to a goal while visiting intermediate stops on the way.

[1] http://wiki.ros.org/ROS/Tutorials
[2] http://pointclouds.org/documentation/tutorials/

- **Search and rescue (A4)**: Last assignment asks students to implement an object recognition algorithm for recognizing dolls and rubber ducks (see Fig. 1) and to search for them in the testing environment. Students take RGBD data as input and use spin image as the feature for object recognition.

3) Grading scheme: The grade depends on in-class demonstration (50%) and software quality (50%). Students submit their software via their SVN repository and demonstrate in class how accurately their implementation works on their robot. For in-class demonstration, exact metric is defined for each assignment. For software quality, percentage breakdown of different components remains the same for all assignments.

In-class demonstration measures the following. In addition, there is a deduction for dumping into obstacles:

- Control and obstacle avoidance (A1)
  - distance to the goal
  - wall following
  - entering and existing obstacle avoidance state
- Localization (A2)
  - distance to the goal
- Path planning (A3)
  - distance to the goal
- Search and rescue (A4)
  - object recognition accuracy (label and location)
  - speed

Software quality [28] (50%) consists of the following:

- choice of abstraction and relations (30%)
  - dependency inversion
  - interface segregation
  - option-operand separation
- correctness of implementation (40%)
- extendibility and reusability (20%)
  - single responsibility
  - open/closed principle
  - hard-coded variable
- comments and documentation (10%)

4) Feedback: Instructive feedback improves students' understanding of learning material [29]. The course therefore provides in-class and individual feedback to students. The feedback sessions, which are held after each assignment, focus on software quality aspect of the assignments, i.e., choice of abstraction and relations, correctness of implementation, extendibility and reusability, and comments and documentation. The in-class sessions focus on mistakes that are frequent and common in the submitted solutions. The individual sessions delve into specific mistakes of each student and provide suggestions for improvement. Each individual session lasts 15 minutes for individual assignments or half an hour for group assignments. Through these immediate and detailed sessions, the course supports the students' error recognition and recovery.

C. Software and Hardware

The course requires a laptop running Ubuntu and a robot with a sensor. Every student receives a complete robotic setup for the semester. They can use either their own laptop or borrow one from the teaching staff.

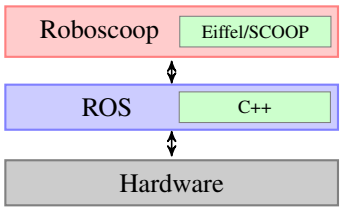


Fig. 2. Software frameworks used in the course.

1) Software: The course uses object-oriented languages, state-of-the-art robotics software, and popular software engineering tools. Programming languages of the course are Eiffel, SCOOP [19] – concurrency language built on Eiffel – and C++. The assignments are either in Eiffel/SCOOP (A1) or both in Eiffel/SCOOP and C++ (A2–A4). The main robotics programming environments are Robot Operating System (ROS) [17] and Roboscoop [20]. ROS is a popular robotics middleware, and Roboscoop is a robotics framework built on SCOOP (Fig. 2). The course also uses Point Cloud Library (PCL) [30] for object recognition. In terms of tools, we use EiffelStudio and Eclipse as IDEs and SVN for the code submission. Working with these software and tools ensures that students gain relevant knowledge and experience, which are readily applicable to robotics and software engineering.

Fig. 3. Hardware setup.

2) Hardware: As with other successful computer science courses with robots [7], our course gives every student a robot and a sensor. Given limited budget and space, giving personal robots would only be possible if the hardware is inexpensive and portable. After considering various robots [3] and sensors [4], we chose Thymio II, a small ($11cm \times 11cm \times 5cm$) differential-drive mobile robot with infrared (IR) sensors, and Carmine 1.09, a small ($18cm \times 2.5cm \times 3.5cm$) RGBD sensor (Fig. III-C2). Both are ROS-compatible and affordable at 99 CHF (about 110 USD) for the robot and 200 USD for the sensor. Since differential-drive and RGBD sensors are popular

[3]Lego Mindstorm, iRobot Create, e-Puck
[4]Microsoft Kinect, web cam, omnidirectional camera

in robotics, using the chosen hardware also gives our students a taste of real, research robotics.

D. Class setting

1) Students: The course enrolment is limited to 16, and the pilot course had 12 students. Of the 12, 11 students – four CS, one EE, and six ME – completed the course successfully. Two were bachelor's students and nine were master's students. Three had completed their bachelor's degree at ETH Zurich. To understand their background, we conducted an in-class, multiple-choice survey in the second week. Ten students filled out the survey; one who did not fill out the survey was a master's student in computer science.

a) Background in computer science: Programming experience of eight students was limited to class assignments or small projects, and two students had no object-oriented programming experience. Our students knew C (six students), C++ (six students), Java (five students), followed by C#, Pascal, Eiffel, Matlab, FPC, PHP, and Python. Concurrency was new to half of the students; the other half had used threading, and some had additionally worked with mutex/semaphore, monitor, or message passing.

In terms of software engineering concepts [5] and tools, design patterns was a familiar concept to four students, algorithms and data structure also to four, followed by programming paradigms and verification. Five students had never used any configuration management tools. Three students had used SVN, one of which had additionally used CVS. Two had used Git. Five students had used a debugger, one a profiler, and no one for automatic testing tools.

b) Background in robotics, control, and vision: Many students had learned control theory (eight students) and kinematics (six students) or perception (three students). Only two CS students had no prior knowledge of robotics. Five students had programmed control algorithm for a real or a simulated robot; the rest had no hands-on experience with any robotic system. No one had used a robotics middleware.

Seven students knew signal processing, and some additionally knew detection/recognition and classification. Three students had no knowledge of computer vision. Only four students had some vision programming experience, three in object detection/recognition and feature extraction or classification and one in segmentation.

2) Teaching staff: Multidisciplinary cooperation can address the challenges of teaching a multidisciplinary course [4]. The course is thus jointly taught by a professor in software engineering and a lecturer in robotics. Two graduate students, one in software engineering and the other in mechanical engineering, assist the lecturers. Diversity in expertise enables the teaching staff to better understand the students' needs.

3) Interaction: The course promotes interaction among the students via online forum, in-class demonstration, and group work. We encourage students to post their questions in the forum and answer them when they can. In the pilot class, students posted 104 questions, 29 of which were answered by the students. Of the 11 students, nine asked at least one question and eight answered at least one question. On average, our students spent 42.5 days online, viewed 103.6 posts (questions and notes), and made 19.5 contributions (posts, responses, edits, follow-ups, and comments to follow-ups). Everyone made at least one contribution and used the forum throughout the course (Fig. 4). We noticed no correlation between grade and forum usage.

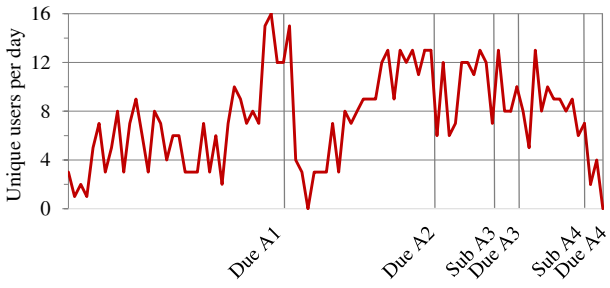


Fig. 4. Forum activity.

In-class demonstration that is held at the end of each assignment is another medium for interaction among the students. Moreover, two of the four assignments are completed in a group of two, and this furthers the interaction. As predicted [29], seeing everyone's work increased communication among the students and helped students learn from one another. In terms of group work, three students found group work easier than individual work while four pointed out both advantages and drawbacks. Despite the division of labor, most students felt that they understood their partner's work because of joint debugging and testing sessions. They shared their knowledge and skills, and two found this experience of group work relevant and realistic. Only two students were unhappy with the experience, and they gave their weak partner as the reason for dissatisfaction.

IV. Results

This section evaluates how well the students of the pilot course learned the five course objectives. To this end, we conducted an exit and an 8-month survey, consisting of long-answer questions, and analyzed the submitted software using software quality metrics.

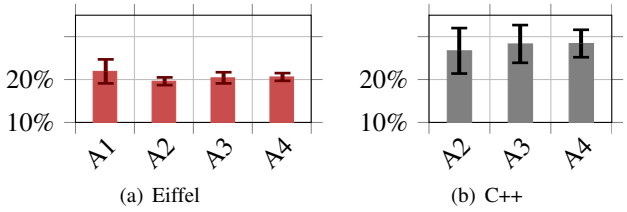
A. End of the semester

The exit-survey was conducted electronically at the end of the semester, and all 11 students returned it in one week.

1) Software engineering principles and methods: The course taught the students various software engineering tools – IDE, debugger, profiler, refactoring features, and configuration management system – and encouraged them to use these tools to write correct, modular and well-documented software. To analyze their progress, we measured software quality of the students' work. As metrics, we used those that are directly related to modularity and documentation and also others that capture common mistakes. The final set of metrics are

[5]One CS and one ME student skipped this particular question.

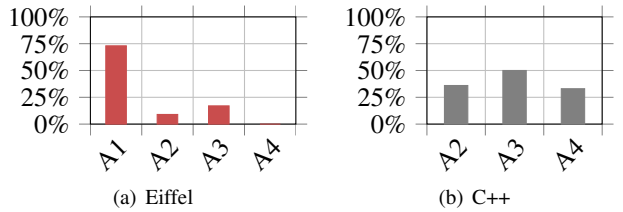
percentage of comments, percentage of routines with hard-coded values, lack of parametrization, and number of SVN commits.



(a) Eiffel (b) C++

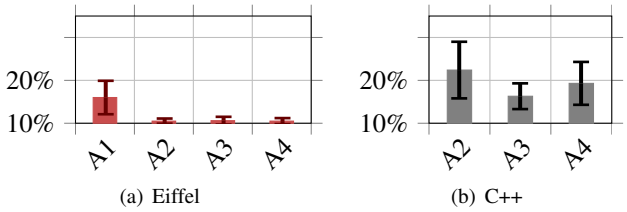
Fig. 5. % of comments.

We assume that comments improve understandability of code and thus increase reusability. Therefore, we measured the percentage of comments, i.e., the number of commented lines divided by the number of lines of code, as a metric for reusability. As the metric assumes that the code is clean, we ignored the commented out code blocks and only counted real comment lines. On average, there was a comment once every six to ten lines of code (Fig. 5). The percentage of comments was higher for the first assignment than the other assignments in Eiffel. This may be because for the first assignment, many students reused and modified the example code we provided at the beginning of the semester. From A2 to A4, the percentage of comments increased 1% in Eiffel and 1.7% in C++.



(a) Eiffel (b) C++

Fig. 6. Lack of parametrization. N=11 for assignments 1 and 2. N=6 for assignments 3 and 4.



(a) Eiffel (b) C++

Fig. 7. % of routines with hard-coded values.

Lack of parametrization and occurrence of hard-coded values dropped significantly after the first feedback sessions. From A1 to A2, the number of submitted solutions that lacked parametrization dropped in Eiffel (Fig. 6). Hard-coded values and "magic numbers" were also prominent in A1, but after our recommendation of using variables and language support for *constants*, there was a drop in occurrence from A1 to A2 in Eiffel (Fig. 7).

In terms of SVN, the students increased their activities right before the deadline for A1 and A2 but used it more

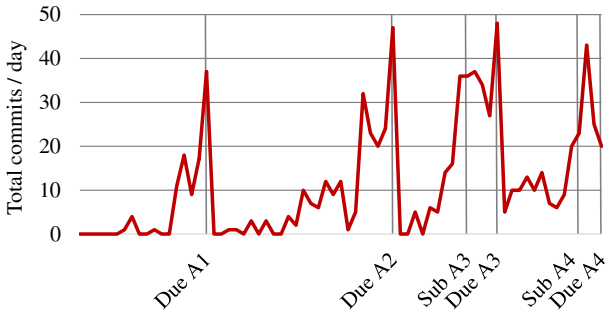


Fig. 8. SVN repository usage.

consistently for A3 and A4 (Fig. 8). This is partially due to the introduction of intermediate goals and group work in A3 and A4, and partially because they improved their time-management skills as some noted in the exit survey. By A4, our students worked more consistently, or at least, made more regular commits, during the assignment.

According to the exit survey, our students used various software engineering concepts and tools in the class. Mentioned concepts include reusability (five students), object-oriented programming (four students), documentation (three students), abstraction (three students), concurrency (two students), and genericity (one student). Tools included IDE (five students) and debugger (three students).

2) Most common architectures in robotics: The course exposed the students to different software architectures and extensively used event-driven programming and publish-subscribe pattern in ROS. In the exit survey, five students correctly explained the difference between ROS publish/subscribe and ROS service communication models and four attempted an answer that was either incomplete or not fully correct. One person gave a completely wrong answer, and another one skipped the question entirely.

3) Coordination and synchronization methods: The concurrency model the course uses is SCOOP. SCOOP has two main synchronization mechanisms – *wait conditions* and *wait by necessity*. When asked of these mechanisms in the exit survey, only five students were able to answer the question. All five students only mentioned *wait conditions*, and none mentioned *wait by necessity*. The submitted code revealed that while everyone used *wait conditions*, only one student also used *wait by necessity* mechanism. The student became familiar with *wait by necessity* only after receiving help from the teaching staff, and thus may not have understood the mechanism fully.

4) How software engineering applies to robotics: Our students improved their software engineer skills by completing the four assignments. In the exit survey, seven students stated that they changed their approach during the semester. They now think and plan the architecture beforehand. One student drew "a UML diagram before starting to write the code". Another found it "easier to write nice code" for the last assignment "because code architecture was decided at the beginning". Four students made no change either because they have always planned the interface first (three CS students) or

"due to being late with everything" (one student).

Four students expressed their appreciation for learning software engineering for robotics. Seven students have increased awareness of uncertainty in real systems, a key challenge in robotics programming. Three pointed out debugging and testing to be important.

5) Hands-on experience with a real robot: Most students noticed additional challenges that come with a real system. By the end of the semester, five students felt comfortable or more comfortable working with a real system than before; working with a real system "was one of the major reasons to visit this course" for one student. Four students defined their experience with a real robot as hard, complex and troublesome. One of them, who has experience in game programming, thought robotics would be similar, but "uncertainty in the real world (made) things more complex". Another said working with a real system was difficult but more satisfying. Two students made no remark.

B. Eight months later

To understand a longer-term effect of the course, we conducted a short survey eight months later. Five (1 CS, 1 EE, 3 ME) of the 11 students returned the survey. They were two top, one middle, and two below-average students.

1) Retained knowledge: Everyone stated that they remember the robotics algorithms they implemented, but few stated that they also remember software engineering components. Only one (EE student) said he remembers software engineering aspects, namely, concurrency and design patterns. The CS student remembers software engineering as well, but this is due to his prior knowledge.

2) Software engineering principles and tools: Three (all ME students) have used or plan on using object-oriented programming in their study or internship. The other two have used design patterns, IDEs, configuration management tools, and a debugger or a testing tool.

3) Benefits: Everyone stated that they appreciate gaining practical, hands-on skills from the class. Most also appreciated having better programming or software engineering skills. Two mentioned that their knowledge in robotics has been helpful. One student found the course especially helpful in securing an internship. Another indicated that he is now "able to find solution(s) online to some extent by (him)self".

4) Complaints: The students complained about various aspects of the course. Biggest complaints were the workload, not having enough time to learn the basics, and using Eiffel (2 students each). Other complaints were A0 being too long, not implementing more algorithms, and using Roboscoop.

V. Current Course

Based on the lessons we learned from the pilot course, we introduced several changes to the course. Main changes include shifting more lectures to the beginning of the semester, introducing an intermediate goal to every assignment, having both individual and group components in every assignment, improving the integration of software engineering topics into the course, and removing the ungraded assignment.

A. Lectures

	Wk	Topic	SE	R
L1	1	Intro to SE and robotics	x	x
L2	2	ROS and Roboscoop	x	x
L3	-	Control		x
-	-	Modern SE Tools 1	x	
L4	3	SCOOP	x	
L5	-	Obstacle avoidance		x
-	-	Modern SE Tools 2		x
L6	4	Design patterns	x	
	5	No lecture		
L7	6	Path planning		x
	7	No lecture		
L8	8	Object recognition		x
L9	9	Software architecture in robotics	x	
	10	No lecture		
L10	11	Localization and mapping		x
	12 – 14	No lecture		

Table III. Lecture topics and schedule. Topics cover software engineering (SE) and/or robotics (R).

Current course covers the same content as the pilot course (see Sec. III-B1), but the lecture schedule has changed to provide more information in the first third of the semester (Tab. III). Although the course still runs with two 2-hour sessions a week over 14 weeks, it no longer follows "one lecture and one exercise session a week" format. Instead, more lectures are held in the beginning of the semester and the beginning of each assignment phase, and more exercise sessions are held towards the end of each assignment phase. This change ensures that students get essential knowledge as soon as possible and enables them to get started on the assignments early.

Main changes in the schedule are the order of tools and localization/mapping lecture. The new schedule has the modern software engineering tools lecture as two one-hour lectures in the beginning of the semester and the localization and mapping lecture at the end of the semester. Current course gives the software engineering tools lecture early because the lecture teaches tools that are useful throughout the course but are new to students. Having the lecture early ensures that students get a proper introduction to these tools before using them. The lecture is divided into two so that students have the time to try the tools before delving into them deeper. For the localization and mapping lecture, the decision to move them to the end came because the students in our pilot course found it to be the hardest to implement. Having to implement the algorithm earlier in the semester meant that they had to improve their software engineering skills while tackling this challenging algorithm. Moving the topic to the end would give students enough time to acquire software engineering knowledge and experience that are necessary to handle the task more smoothly.

B. Assignments

Assignments have the same content as those in the pilot course (see Sec. III-B2) but different schedule and individual/group work allocation. Main reasons for the changes are

	Week	Topic	I	G
A1	1 – 5	Setup and control	x	
		Obstacle avoidance		x
A2	6 – 8	Path planning in simulation	x	
		Path planning with robot		x
A3	8 – 10	Object recognition	x	
		Object recognition with robot		x
A4	11 – 14	Localization in simulation	x	
		Search and rescue		x

Table IV. Assignments. Every assignment has individual component (I) and group component (G).

to distribute the workload more evenly throughout the semester and to minimize the gap between stronger and weaker students in group work. For the former goal, the ungraded assignment is now part of the first assignment and localization is moved to the last assignment. Through these changes, we hope to motivate students to get familiar with the working environment earlier and to prepare them well for the last, most challenging assignment. For the latter goal of preparing students evenly, every assignment now has both individual component and group component. Individual component requires students to implement core algorithms while group component, completed in a group of two or three, asks them to extend the algorithms to handle more complex scenarios. As everyone must complete the individual component and demonstrate it in class before they start working on the group component, fewer groups would face problems that stem from a weaker member.

C. Grading scheme

Having equal emphasis on in-class demonstration (50%) and software quality (50%) remains the same. The only change brought to the grading scheme is the metric for the in-class demonstration portion. In the pilot course, all assignments except for the last one gave a demonstration grade based on an absolute scale. This proved to be problematic as some students found full-credit performance unattainable and subsequently gave up on doing their best. In the current course, the performance is measured relative to the best performance, i.e., whomever performs best gets full credit. This would bring out students' competitiveness and result in overall performance improvement.

In-class demonstration measures the following. As in the pilot course, bumping results in a deduction:

- Control and obstacle avoidance (A1)
  - Individual: go to goal
    - distance to the goal
  - Group: obstacle avoidance
    - distance to the goal
    - wall following
    - transition in and out of wall following mode
- Path planning (A2)
  - Individual: path planning in simulation
    - different robot sizes
    - 4-connected and 8-connected map
  - Group: path planning with a real set-up
    - distance to the goal
    - time to completion
- Object recognition (A3)
  - Individual: object recognition
    - correct labeling and bounding box
  - Group: object recognition with a moving robot
    - object recognition (label and location)
    - time to completion
- Search and rescue (A4)
  - Individual: localization with recorded data
    - predict, update, and resample steps
    - stability of localization
  - Group: search and rescue
    - object recognition (label and location)
    - distance to the final goal
    - time to completion

D. Improved integration of software engineering and robotics

One change we specifically paid attention to is the integration of software engineering and robotics. To this end, we introduce more live demonstrations and code snippets to the lectures so that we can better teach students how an algorithm can be implemented in software. In addition, we offer more practical help for the assignments during the exercise sessions in both robotics and software engineering. For instance, we make it clear that having a separate PID controller class is more flexible and reusable than having it integrated into go to goal controller. Although this may seem unnecessary extra work at first, as students implement obstacle avoidance, usefulness of the class becomes clear. We also increase the frequency of feedback sessions to twice per assignment; students now get feedback after each individual assignment and group assignment. Frequent interactive communication allows students to learn software engineering from their own experience and resolves questions earlier on.

VI. Discussion

This section discusses the results and reflects the experience on andragogy, minimalism, and interactive learning.

A. Achieving the course objectives

Background survey revealed that our CS students had limited knowledge of robotics and our non-CS students had limited knowledge of software engineering. In fact, most non-CS students found object-oriented programming difficult. Teaching such a bimodal course proved to be challenging as every content had to be easy enough for beginners and interesting enough for advanced students.

We observed that most non-CS students acquired only basic software engineering skills during the semester and failed to utilize it since the semester ended (Sec. IV-B). Over the course of the semester, the students' software improved in quality, namely, modularity and documentation. The students also learned basic software engineering principles and utilized

software engineering tools (Sec. IV-A1). On the other hand, many students did not learn much of architecture (Sec. IV-A2) nor concurrency (Sec. IV-A3) despite using them throughout the semester. This may partially be due to the fact that the SCOOP lecture ended up focusing more on the basics of Eiffel and the ROS and the architecture lectures were too advanced for beginners. Consequently, we predict that our students managed to program without understanding the concepts behind the mechanisms they used.

In robotics, most students learned how to apply software engineering in robotics and some understood the importance of testing (Sec. IV-A4). Many also learned about uncertainty in real systems and improved their hands-on skills (Sec. IV-A5). Eight months after the course ended (Sec. IV-B), our students still remembered what they implemented and appreciated the knowledge and experience gained in the course. Our students desired to spend both more and less time with the basics and implement more robotic algorithms with reduced workload. Meeting these conflicting demands remains a challenge.

B. Andragogy

As in other robotics courses [10], our course had elements of andragogy. First, we promoted the students having ownership of work by giving every student a robot and encouraging them to learn from online tutorials and actively participate in the online forum. Our students gradually became independent learners, actively seeking and giving help online and offline (Sec. III-D3). One student remarked that his research skill has improved in our class.

Second, prior experience played an important role. Although our students were initially shocked that the others did not know what they found so basic, they gained an appreciation for each other's expertise and exchanged their knowledge more actively. In general, our ME students struggled with programming but did better with calibration while our CS students found programming easy but struggled with the uncertainty in real systems. Our one EE student had the easiest time as he had more programming experience than ME students and more hardware experience than CS students.

Third, using well-established software and hardware made our course relevant. In particular, using ROS was exciting to many as they discovered how big it is in robotics. Everyone found the course useful, and most enjoyed working with a real system and learning ROS.

Fourth, in emphasizing hands-on learning and requiring in-class demonstrations, the course was performance-centered. With individualized feedback and observation of peer's work, the students improved their skills over time. Many appreciated the hands-on experience and retained their knowledge.

Fifth, our students were excited to work with a real robot, and as they came to understand the benefits of tools we used, they also came to appreciate our decision. SVN in particular was not well-received in the beginning, but with group work, the students came to see its usefulness.

Lastly, as an elective course, only intrinsically motivated students took the course. Hands-on learning with a robot was definitely attractive to many, and multidisciplinary aspect made the course approachable to various students.

C. Minimalism

Our course also had elements of minimalism. First, the course took action-based approach. Every assignment, which is given at the beginning of each phase, required an implementation of one specific algorithm. Lecture topics for each phase were chosen accordingly so that students could complete the assigned task. Second, the course anchored the tools in robotics programming by using popular robotics hardware and software and utilizing online tutorials. Third, we encouraged our students to be independent learners and supported them when needed via the exercise sessions, online forum, and individual feedback sessions. Lastly, although we did not follow a textbook, we pointed out specific chapters of the recommended books (Sec. III-B1), gave papers to read, and wrote a short tutorial as needed.

D. Interactive learning

Our course promoted interaction among the students via the online forum, in-class demonstration, and group work. The students became increasingly active in sharing their knowledge with others via online forum, and more students communicated with one another during the demonstration. Group work had a mixed review depending on the difference in knowledge between the group members. Although we did not evaluate the effect of the interactive environment, we hope that it was positive as hypothesized [11]; one student remarked that he "enjoyed the open, positive atmosphere among the students".

VII. Threats to Validity

There are several limitations and threats to validity of this study. An obvious limitation of the study is that data are drawn from a single course offering at one university. While the pilot study provides some understanding of the effect of teaching software engineering in a robotics programming course, a longer and broader study is necessary to generalize the claim. In addition, the pilot study's small data size limits generalization of the results. More fundamentally, as it is uncommon for computer science students to take robotics courses and mechanical engineering students to take computer science courses, effectiveness of our course compared to two separate courses – a robotic and a software engineering course – is unclear.

The study contains several potential sources of bias. First, as an elective course, no student was required to take the course. Our students may have been more motivated to learn about software engineering in robotics setting than regular students, and this may bias our results towards better learning outcome. Second, the authors designed, implemented, and executed the course described in the paper. Even though we tried to be neutral in analyzing data, the results may nonetheless be biased towards success. Third, as the students got to know the authors very well by the end of the pilot course, their exit and 8-month survey responses may have been influenced by their

feeling towards the authors. In fact, for the exit survey, top third of the students gave longer and more-detailed responses than most other students. Consequently, their opinions are better represented in the qualitative analysis. This may also bias towards better results. For the 8-month survey, however, the five responses were evenly distributed across the grades.

In our analysis, we assumed that the submitted solutions and the survey responses capture the students' understanding of software engineering. In reality, correlation between the submitted code and the students' knowledge is not clear. Many students found the course challenging and time-consuming and tried to optimize the way they spend their time. Consequently, the software they submitted is not necessarily representative of what they actually learned in class. Due to time limitation, the students may not have applied everything they have learned but instead only those that result in a higher grade for the effort. Also, the last two assignments were completed in a team of two students, and it is not clear how the knowledge of two students maps to a single piece of software.

VIII. Conclusions

This paper reported a new, multidisciplinary robotics programming course and evaluated it with respect to its objectives using surveys and software quality metrics. It then presented an improved course that addresses the challenges identified in the pilot course. Although our students found concurrent software engineering and robotics education beneficial, the pilot course was only successful in teaching robotics and had limited success in teaching software engineering. Current course tries to minimize the learning curve by providing relevant information earlier and giving feedback more frequently.

Next step of this research is more thorough evaluation of the course. Due to limited budget and resources, we can only offer the course to 16 students at a time. Subsequently, data collected from the pilot study are too small to draw any concrete conclusion. We continue to offer the course and collect more data to better understand the effect of this course on students' learning of software engineering and robotics.

Acknowledgment

This work was partially supported by the European Research Council under the European Unions Seventh Framework Programme (ERC Grant agreement no. 291389), the Hasler Foundation under the SmartWorld programme, and ETH under the Innovedum fund.

References

[1] J.-F. Lalonde, C. Hartley, and I. Nourbakhsh, "Mobile robot programming in education," in *ICRA*, 2006.

[2] N. Correll, R. Wing, and D. Coleman, "A one-year introductory robotics curriculum for computer science upperclassmen," *Trans. on Edu.*, vol. 56, no. 1, pp. 54–60, 2013.

[3] R. D. Beer, H. J. Chiel, and R. F. Drushel, "Using autonomous robotics to teach science and engineering," *Commun. ACM*, vol. 42, no. 6, pp. 85–92, Jun. 1999.

[4] J. B. Weinberg, G. L. Engel, K. Gu, and C. S. Karacal, "A multidisciplinary model for using robotics in engineering education," in *ASEE Annual Conference and Exposition*, 2001.

[5] B. Fagin and L. Merkle, "Measuring the effectiveness of robots in teaching computer science," *SIGCSE Bull.*, vol. 35, no. 1, pp. 307–311, Jan. 2003. [Online]. Available: http://doi.acm.org/10.1145/792548.611994

[6] J. Summet, D. Kumar, K. O'Hara, D. Walker, L. Ni, D. Blank, and T. Balch, "Personalizing cs1 with robots," *SIGCSE Bull.*, vol. 41, no. 1, pp. 433–437, Mar. 2009.

[7] M. M. McGill, "Learning to program with personal robots: Influences on student motivation," *Trans. Comput. Educ.*, vol. 12, no. 1, pp. 4:1–4:32, Mar. 2012. [Online]. Available: http://doi.acm.org/10.1145/2133797.2133801

[8] D. Gustafson, "Using robotics to teach software engineering," in *FIE*, vol. 2, 1998, pp. 551–553.

[9] "Software engineering in robotics," 2014, http://www.cc.gatech.edu/~hic/8803-SER-10 [Online].

[10] M.-I. Koski, J. Kurhila, and T. A. Pasanen, "Why using robots to teach computer science can be successful theoretical reflection to andragogy and minimalism," in *Proceedings of the 8th International Conference on Computing Education Research*, ser. Koli '08. New York, NY, USA: ACM, 2008, pp. 32–40.

[11] M. T. Chi, "Active-constructive-interactive: A conceptual framework for differentiating learning activities," *Topics in Cog. Sci.*, vol. 1, no. 1, pp. 73–105, 2009.

[12] M. Knowles, *The Modern Practice of Adult Education: from Pedagogy to Andragogy*. Prentice Hall, 1980, ch. What is Andragogy?, pp. 40–59.

[13] J. M. Carroll, Ed., *Minimalism beyond the Nurnberg funnel*. MIT Press, 1998, ch. Reconstructing Minimalism.

[14] M. Montemerlo, N. Roy, and S. Thrun, "Perspectives on standardization in mobile robot programming: The carnegie mellon navigation (CARMEN) toolkit," in *Proceedings of the Conference on Intelligent Robots and Systems (IROS)*, 2003.

[15] P. Newman, "Moos - mission orientated operating suite," Department of Ocean Engineering, MIT, Tech. Rep., 2006.

[16] J. Jackson, "Microsoft robotics studio: A technical introduction," *IEEE Robotics Automation Magazine*, vol. 14, no. 4, pp. 82–87, Dec 2007.

[17] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "ROS: an open-source robot operating system," in *IROS*, 2009.

[18] P. Nienaltowski, "Practical framework for contract-based concurrent object-oriented programming," Ph.D. dissertation, ETH Zurich, 2007.

[19] B. Morandi, "Prototyping a concurrency model," Ph.D. dissertation, ETH Zurich, 2014.

[20] A. Rusakov, J. Shin, and B. Meyer, "Simple concurrency for robotics with the Roboscoop framework," in *2014 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, 2014.

[21] S. Nanz, F. Torshizi, M. Pedroni, and B. Meyer, "Design of an empirical study for comparing the usability of concurrent programming languages," in *Proceedings of ESEM'11*. IEEE Computer Society, 2011, pp. 325–334.

[22] B. Meyer, *Object-Oriented Software Construction*, 2nd ed. Prentice-Hall, 1997.

[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[24] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, 2000.

[25] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. Cambridge, MA, USA: MIT Press, 2005.

[26] R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza, *Introduction to Autonomous Mobile Robots*, 2nd ed. Cambridge, MA, USA: MIT Press, 2011.

[27] I. Kamon, E. Rimon, and E. Rivlin, "Tangentbug: A range-sensor-based navigation algorithm," *The International Journal of Robotics Research*, vol. 17, no. 9, pp. 934–953, 1998.

[28] B. Meyer, *Object-oriented software construction*. Prentice hall New York, 1988, vol. 2.

[29] J. Ormrod, *Human Learning*, 5th ed. Pearson/Merrill Prentice Hall, 2008.

[30] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011.