STATEMENT OF TEACHING PHILOSOPHY

Ahmed Tamrawi

Teaching is a noble profession and a crucial career that holds both challenges and rewards. As educators, we do more than just convey knowledge; we shape the very foundation of our society, building a better world one student at a time. Our role as teachers is crucial as we must *engage*, *contribute*, and *inspire*. By doing so, we are not only passing on knowledge but also cultivating the minds of our students and guiding them toward a brighter future.

As psychologists say, *thinking* is an uncomfortable process; it takes effort. Shaped by my academic and professional journey, I believe that thinking and learning are processes of constructing interconnected chunks of information and embedding them in our long-term memory. Our lives are full of patterns, and our cognition and memory categorize new situations by relating them to previously acquired knowledge.

The memorable lectures and talks are the ones that build on these chunks of information that we previously acquired. These are the lectures that grab our attention through a funny speaker, visually appealing graphics, or self-explanatory stories. The main guiding principle of my teaching philosophy can be summarized as: **make each lecture a memorable, informative, and exploratory time for students**.

All of us watch YouTube videos and often skip the ads after the 5-second grace period. However, there are times when we watch the full ad. This *power of attraction* is what I aim to invest in my teaching to *make students curious about what comes next, to silently trick them to think, and to make their learning experience unforgettable*.

Teaching Experience

I have held several positions as a Teaching Assistant at Yarmouk University and Iowa State University while I was doing my MSc and PhD degrees, and as an Assistant Professor at Yarmouk University and Birzeit University. In these roles, I have taught *Software Construction*, *Numerical Analysis for Engineers*, *Operating Systems Design*, *Software Security*, and *Introduction to Programming* courses.

Software Construction[1] [2] [3]

The *Software Construction* course is a graduate-level class with approximately 20 students. The goal of the course is to equip students with the skills to create high-quality software that is robust, flexible, extensible, scalable, and maintainable. This course emphasizes best practices at each stage of the software development process to ensure the production of clean software.

[1] Software Construction (Fall 2018): https://atamrawi.github.io/teaching/swen6301_fall18
[2] Software Construction (Fall 2019): https://atamrawi.github.io/teaching/swen6301_fall19
[3] Software Construction (Fall 2020): https://atamrawi.github.io/teaching/swen6301_fall20

This course is designed to benefit students both theoretically and technically. The course employs several key strategies:

1. The use of visually appealing lecture slides coupled with in-class activities allows students to experiment with new technologies and tools, and to work with real-world software repositories.
2. A set of open-ended synthesis and conceptual assignment questions encourages thorough research, detailed observations, and practical software implementations.
3. A final programming project emphasizes applying the concepts studied throughout the course to a well-established open-source project with multiple contributors and various open issues. This open-source contribution enables students to interact with actual developers and adhere to project-specific rules and styles.

Through these methods, the course ensures that students gain a comprehensive understanding of software construction, preparing them to create and maintain high-quality software in their professional careers.

Numerical Analysis Methods for Engineers[4]

The *Numerical Analysis Methods for Engineers* course is a 300-level class with approximately 100 students. The biggest challenge in this course was accommodating the diverse backgrounds of the students in a large classroom setting. Together with my colleagues, we worked to evolve the course materials to make them both accessible and engaging for all students. To achieve this, we implemented several key changes:

1. *Real-World Applications:* We added a section to each topic or group of topics to discuss the actual applications of the numerical methods in various fields. This helped students see the relevance and importance of the concepts they were learning. We also introduced a new course outcome emphasizing the applications of numerical analysis techniques in real-world scenarios. We also added exam questions specifically designed to test this outcome, challenging students to apply their knowledge to practical problems spanning different prerequisite courses.
2. *Visually Engaging Presentations:* To capture students' attention and simplify complex mathematical concepts, we incorporated visually appealing graphics into our presentations. We also added proper animations to help students visually comprehend different mathematical algorithms, especially those requiring iterative solutions.

These improvements made the course more interactive and engaging, fostering a deeper understanding of numerical analysis methods and their practical applications. Students found the problems challenging yet interesting, which enhanced their overall learning experience.

[4] Numerical Analysis Methods for Engineers: https://atamrawi.github.io/teaching/cpe310

Operating Systems Design[5]

The *Operating Systems Design* course is a 400-level class with about 50 students. Reflecting on my own experiences as a student, I found the course to be rather dull due to the dense theoretical content. Determined to transform this perception, I decided to revolutionize the course and build it from the ground up, drawing on my five years of research experience with Linux kernel code throughout my PhD degree.

To make the course more engaging, I interwoven theoretical information with practical aspects from the Linux kernel. I created a comprehensive set of slides for each chapter of the selected textbook, ensuring that only 20% of the content was text. The remaining 80% consisted of images, animations, and humorous anecdotes rooted in local culture and applicable metaphors. This approach made complex concepts more relatable and enjoyable for students. For each topic, I related the theoretical material to my industry experience, explaining the algorithmic reasoning behind every concept. Additionally, I connected the studied concepts to actual code from a live Linux kernel repository.

The course included a weekly three-hour lab, for which I developed new materials focused on learning operating system concepts as well as software design, development, and testing. I taught students how to use Linux-based operating systems, create build files for projects, use software repositories (e.g., GitHub), and write proper test cases for projects. We also discussed various algorithm design strategies to choose the most efficient solution for addressing assigned requirements.

By integrating practical experience with theoretical learning and using engaging teaching methods, I aimed to make the Operating Systems Design course an informative and enjoyable experience for all students.

Introduction to Programming[6]

The *Introduction to Programming* course is a 100-level class with approximately 120 students. This course was one of the most challenging to teach, not due to its difficulty, but because of the students' diverse backgrounds and lack of coding skills. To address this, I encouraged students to think creatively about how to teach a computer to perform tasks.

I began by comparing a computer to a simple calculator, capable of basic operations. I then posed questions on how to use these operations to perform tasks such as averaging 50 grades, performing multiplication through repeated addition, and executing division through subtraction. This approach helped students understand fundamental concepts before moving on to C++ syntax, the programming language used in the course.

In class, I always prepared a couple of programs to illustrate multiple concepts. Using the preferred IDE, I engaged in live coding sessions. We start the lecture by discussing a problem and exploring potential solutions, which highlighted the necessity of certain language constructs.

[5] Operating Systems Design: https://atamrawi.github.io/teaching/cpe460
[6] Introduction to Programming: https://atamrawi.github.io/teaching/cpe150

Then, we wrote the program interactively, with me posing questions and deliberately making logical mistakes to capture students' attention. I frequently paused to ask whether to use a less than or greater than symbol or if a double array should have seven elements or fewer. After completing the first draft, we explored different approaches to solve the same problem. If we encountered erroneous output, we used the debugging feature, which was especially useful in teaching arrays, pointers, and function calls.

The course included a weekly three-hour lab where I developed tasks centered around toy-sized applications mimicking real-world scenarios such as coffee shops, salary computation, and wind forecasting models. The goal was to emphasize software design, development, and testing.

Theoretical and practical exams aimed to assess two outcomes: program comprehension and debugging, as well as programming skills. The first outcome was evaluated through output-related problems, logical error fixing, and describing mathematical expressions between variables induced by the program. The second outcome involved writing complete programs that were slightly different from those taught in lectures or lab tasks.

By employing interactive teaching methods and relatable examples, I aimed to make the Introduction to Programming course both engaging and educational, equipping students with the foundational skills needed for further studies in computer science.

Software Security[7]

The *Software Security* course is a comprehensive 400-level class designed for about 50 students. I have introduced this course as an elective course option. The course addresses the critical need for secure software in our increasingly software-driven world. Drawing from my extensive experience, research and my work on two high-profile DARPA projects (APAC and STAC), I developed this course to provide students with both theoretical knowledge and practical skills in software security.

The course covers a wide range of topics, including operating systems and applications security, web security, secure design and development, malware, algorithmic complexity and side-channel attacks, as well as an introduction to software analysis and penetration testing. These topics are essential for developing software that remains functional and secure even under malicious attacks. Key elements of the course include:

1. *Engaging Lectures:* Each lecture is designed to be interactive and visually appealing, incorporating animations and real-world examples to explain complex security concepts. This approach helps students better understand and retain the material.
2. *Practical Assignments:* The course includes 5-6 assignments that consist of synthesis, conceptual and practical questions. These assignments require students to analyze programs with respect to the security concepts studied in class and implement working vulnerabilities on virtualized machines to test their knowledge. This hands-on approach ensures that students can apply theoretical knowledge to real-world scenarios.

[7] Software Security: https://atamrawi.github.io/teaching/comp4384_fall20

3. *Comprehensive Exams:* The final exam includes questions from the course notes and additional synthesis questions to test students' understanding of the material and their ability to apply it.
4. *Real-World Applications:* Throughout the course, we discuss the practical applications of software security techniques in different fields. This includes examining actual vulnerabilities and attacks, and understanding how to defend against them.
5. *Ethical Hacking:* The course encourages a collaborative learning environment while maintaining high standards of ethical hacking in mind.

By combining theoretical foundations with practical applications and emphasizing real-world relevance, the Software Security course aims to equip students with the skills and knowledge necessary to develop secure software. This approach not only prepares students for careers in software development and security but also instills a deeper understanding of the importance of security in the digital age.

Teaching Methodology

As an educator, I believe that the central goal of teaching is to *nurture the ability of students to inquire and learn by themselves*. This is particularly important in the fields of computer science and engineering, which are rapidly evolving. Many programming languages and software techniques in use today did not exist twenty years ago and may become obsolete or be replaced twenty years from now. However, the ability to learn remains valuable throughout one's lifetime. From my personal experience, I taught myself Java, C#, C/C++ programming skills, Android development, web frameworks and full-stack web development through online tutorials and empirical projects. The success of a career in computer science and engineering largely depends on one's ability to keep pace with the field's daily innovations.

In a *lecture setting*, I like to interleave theoretical material with live tool demonstrations, slides and animations, and hands-on exercises that students and I collectively solve in class. I have used this approach in my guest lectures and tutorials at Iowa State University, Yarmouk University, and during a tutorial on program verification and analysis tools at Amazon's campus in Seattle while I was an intern. This method creates a feedback loop between theory and practice, keeping the audience motivated and engaged. I often reference my work at EnSoft and Amazon, sharing stories that spark students' curiosity about how theoretical concepts and learned principles are applied in real-world situations.

One principle I emphasize to my students is that there is no such thing as a stupid question. *Curious questions can lead to groundbreaking discoveries or inventions*. This approach has encouraged students to ask more questions, often requiring me to research answers. I am delighted to draw out these questions from students who are still mastering the topic.

From my past experience, interactive lecture components such as open-ended problem sets and large projects are more effective than traditional lectures and quizzes in nurturing students' ability to inquire and learn independently. I strongly advocate using large open-ended team projects to teach computer science and engineering subjects. There are at least three significant benefits: First, working on a large project promotes active learning. Second, it bridges the gap between theory and practice more effectively than any other method. Lastly, projects are a more engaging way for students to learn, providing them with the satisfaction of inventing and creating something independently. To ensure the success of a large project, it should be broken into step-by-step incremental milestones to maintain a steady learning curve, and course staff should monitor each team's progress. Additionally, assistance and guidance from the course staff should be readily available to prevent students from straying too far during the project's early stages.

In a *practical lab setting*, I usually attend the entire three-hour lab, engaging students in discussions and assisting the teaching assistants. When students are shy or looking for ideas, I simply grab a chair and help them, which is rewarding and relaxing for both me and the students. I challenge top-notch students with bonus questions that require additional research and effort. In the lab, I have an open-resources policy, allowing students to collaborate to a certain extent and use available online resources to solve the given problems.

In an *exam setting*, I have observed that visually appealing exams contribute to student satisfaction and better scores. Additionally, including humorous questions after challenging ones helps students take a mental break from intense technical focus and think creatively.

After grading exams and analyzing ABET results, I engage with students in *post-exam sessions*. During grading, I note where each student encountered difficulties or should have performed better based on their past performance. Comparing current results with previous semesters helps me gauge my teaching effectiveness and identify gaps in each class. Once the exam is graded, I meet with each student individually to review their performance, discussing why they answered as they did and what led them to change their minds from the correct answers. This process has revealed instances of ambiguous question statements and misinterpretations of concepts, making it an exciting and valuable learning experience for both the students and me.

Teaching Interests

I am enthusiastic about teaching both undergraduate and graduate courses in programming languages, compilers, software security and analysis, and software engineering. Additionally, I am keen to contribute to introductory programming or algorithms and data structures courses, integrating elements of program analysis and security into these foundational subjects. I am also open to teaching undergraduate classes outside my immediate areas of interest.

I would particularly enjoy teaching an undergraduate course on "*The Mechanics of Programming.*" This course would introduce students to the intricacies of program structure, execution mechanics, and translation processes, including essential programming languages and compilers. It would also cover supportive operating system features, such as key operating system design and concepts. Furthermore, the course would highlight critical security and performance issues in program design, focusing on software analysis and security.

At the graduate level, I am interested in teaching courses on software analysis and security, as well as software testing and maintenance. My extensive research experience in various areas of program analysis and security positions me well to design a comprehensive course covering a wide range of topics. Additionally, I aspire to offer a vital course to both graduate and undergraduate students titled "Bridging the Academic-Industry Gap." This course would focus on assembling the pieces of the puzzle, from building a resume and preparing for technical interviews to recapping data structure concepts, software design, software testing, and software engineering. This comprehensive approach, especially in the final year, would ensure that graduating students are well-prepared to achieve their career aspirations.

Advising Approach

During my time as an assistant professor and as a PhD student, I had the privilege of advising numerous students at both the graduate and undergraduate levels, each with their unique needs. I have always enjoyed working with students in an advisory capacity, sharing my knowledge and providing them with the most efficient pathways to success.

I am an engaging person, and I take a keen interest in discussing students' future careers, guiding them with the support of other professors. As a graduate student, I supervised two undergraduate research projects and two master’s theses. This experience underscored the importance of striking a balance between allowing students to explore their ideas and providing the guidance they need to succeed. This guidance often took the form of weekly discussions, intermediate deadlines, and reading lists.

Additionally, I realized that each student is different, and a single style of guidance does not fit all. Tailoring my approach to meet the individual needs of each student has been a key aspect of my advising philosophy, ensuring that every student receives the support and encouragement they need to thrive.