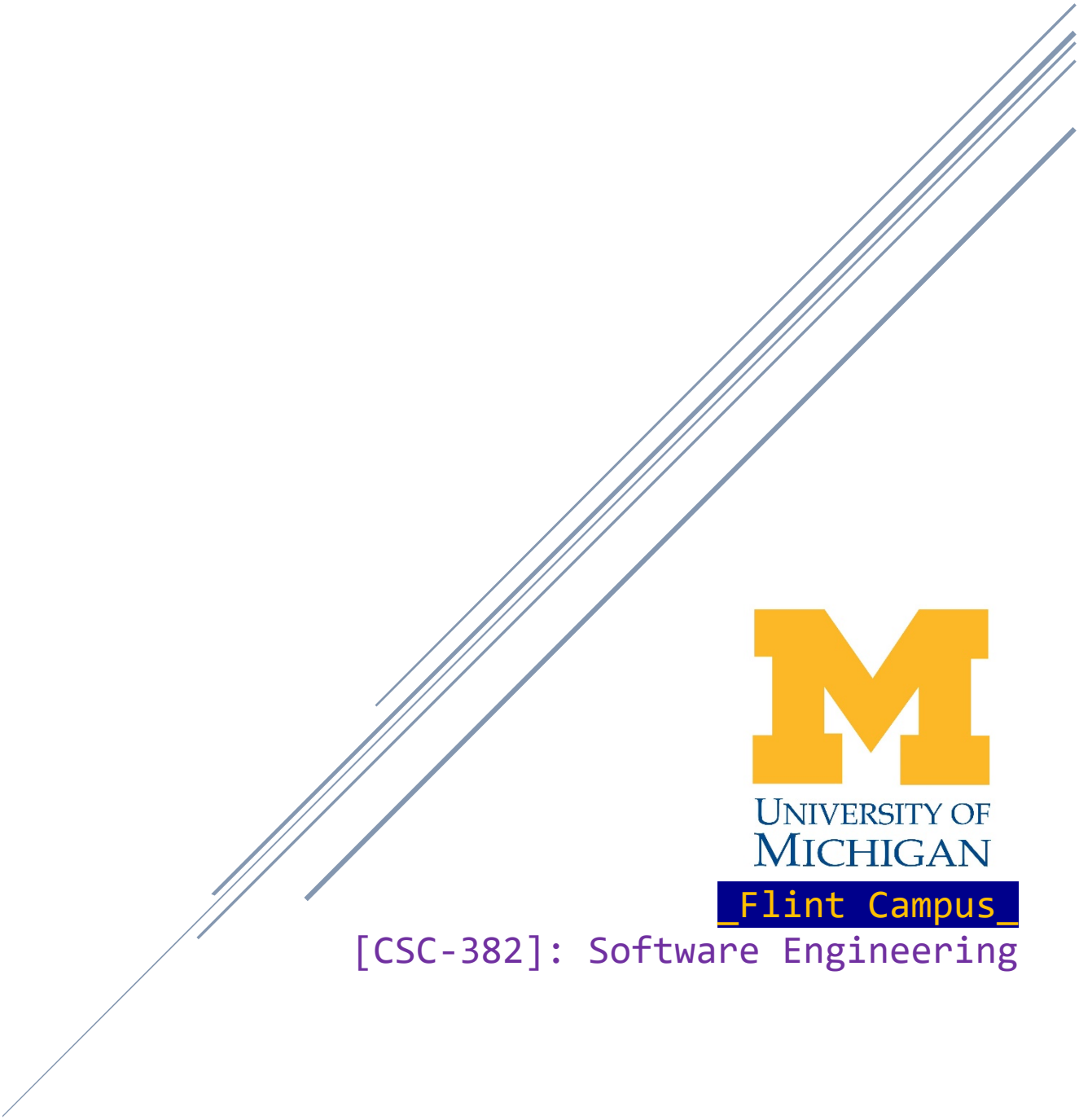
ASSIGNMENT #1

Cason Konzer 7/8-11/21



UNIVERSITY OF MICHIGAN
Flint Campus

[CSC-382]: Software Engineering

Contents

[Note]:

For all written homework in this class, a page is defined as single space, 12-point font, with one inch or less side margins, and 1.5 inch or less top and bottom margins. Write in complete sentences and provide references with citations as needed.

Future Use of Software Engineering

One definition of software engineering is the systematic approach to software development and its process throughout. As a result, software engineering can be broken down into task groups. Common framework consists of communication, planning, modeling, construction, deployment, and maintenance; most often the process follows these tasks in the given order. This framework can be set up as a one-time pass, an iterative loop, or a dynamic process with flexibility and simultaneity between the groups. Software engineers can be deployed to individual groups or work dynamically throughout them. The framework presented by software engineers is applicable to most any engineering field. Working now in automotive engineering, I can see a direct application of the framework within my daily job. Working additionally in undergraduate research I can see a similar structure provided with more flexibility and less time constraints. Within my current employment, the software engineering grants a new level of abstraction to view operations management. When working it is easy to become nearsighted with tasks at hand and miss out on the bigger picture. Through software engineering methodologies I can view the production process from a higher viewpoint, with a larger focus on end goals and systems thinking.

As this class progresses, I hope to greater hone my analytical thinking and gain a deeper understanding of the engineering process. Within my research, I plan to utilize these skills to my advantage. Currently my research tasks are that of software development and directly relate to software engineering. Often, I find myself doing too little planning and modeling before diving into construction. The culmination of coding and testing is most realized and building a complex system from scratch requires frequent proof of concept. Before implementing any new functions or modules I need to first unit test their outputs. In practice implementation on trimmed datasets are necessary before moving to the whole. What I do find useful is exceptional prior planning; without a vision of what software should look like at the deployment phase it is easy to spend time wasted. Within software development optimization is king; two programs with equal functionality are thus judged on their resource consumption. To implement programs with low cpu, gpu, and memory requirements is crucial. Through proper modeling, forethought allows for optimization. Combining modeling and construction, prototypes serve as a proof of concept before deployment. Knowing in advance the methods of

implementation planned for future functions and modules allows for consistency throughout software. This idea extends to optimization through the utilization of consistent datatypes. Similarly simple functions can be built out withing one module, allowing for reuse in new development projects. As I continue to expand my knowledge on the topic of software engineering, I hope to further expand my capabilities of implementing the KISS ideology (Keep It Simple Stupid). In the present, I plan to implement the development stages framework to mitigate time lost and improve efficiency.

For my full-time job, I am a node of communication. Technically I am an engineering project manager as a co-op/intern. In application I communicate with application engineers (software & systems implementation), engineering technicians (prototype builders), management, and our testing team. I joined my project on the testing launch date, amid the construction phase. For testing we deploy a 'working' prototype of the system in place to collect data and diagnose software issues (false positives). From the software development framework my team is best described as agile. There is currently production versions of the same software deployed and active on road; we implement a continuous integration strategy to push out new versions each year. As I learned the typical development stages I can see many faults that are hindering the performance of my team. Communication is key and necessary to be implemented throughout. Without clear knowledge of what issue a tester is experiencing it is nearly impossible to diagnose root cause and implement a solution. To implement the system extensive modeling is required, and a large sum of domain expertise as they become technical. We set our goals based on requirements and respective estimations for data collection. A schedule is then made, and progress is tracked throughout the project. Before can deploy the software there are many phases of testing. There is first systems testing by our engineering team, followed by additional systems testing done by an agency, and last software testing based on data collection. As I finish a full development cycle, I see the framework making itself more present.

Engineering is akin to problem solving in many ways. Being able to document issues and learn from them is a great way of preventing future problems. What I have learned so far in this class have granted me a new viewpoint of the work I do, and the evolutionary process of a product. I plan to utilize this viewpoint and new lessons learned in my future endeavors with team projects and independent development.

This page is intentionally left Blank.

Future Use of Software Engineering

Loka in “Software development: What is the problem?” and Denning and Riehle in “Is software engineering engineering?” take the opportunity to compare and contrast software engineering with traditional engineering. The strong and weak suits of software engineers are pointed out while constructive criticism is given to the field for future improvements. Both articles provide a unique view on software engineering and agree the field has growth and maturity to come.

In Loka’s synopsis, software development is classified as an art or craft, following the ideas of Donald E. Knuth (Loka, 1). A key issue that Loka points out is the methodology encompassed by the consortium of software engineers today; rather than solving problems at their root cause, teams often ‘fix’ issues by pushing them to a different aspect of the system. As a result, software engineers today need to adapt a more systematic approach and understand the problem fully; doing so is a viable way to cut down on maintenance work (Loka, 2). In general, Loka suggests a mediocrity approach present in the industry that stems these problems. As a result of low tier software; there is an expectation imposed by Loka that the customer also holds responsibility in their buying activity. This approach is counter to common human computer interaction theories as it shifts blame on the user rather than the developer. As a side note, Loka comments on software project management being very little of an engineering discipline. Like any management, those who are overseeing a project should have expertise in the industry, their job is more business and operations focused than anything. In traditional engineering; numerical metrics drive product quality and sales. One cannot deny that precision, accuracy, and recall are well know metrics that give insight into quality of a product. Loka does just this, stating “developers place an undue emphasis on numbers. Yet numbers can only be used for extrapolation. Nor are numbers themselves very dependable” (2). While when used improperly, numbers can give a false sense of security; If a product is used for many years and completes the tasks required with no faults, there numbers clearly convey a positive message. What seems to be the crux of Loka’s issue with numbers is their implementation within the software engineering field. Just as in typical engineering most products require minimum benchmarks to be met before production can happen. Unlike traditional engineering, software engineering allows for higher manipulation of numbers to meet such

benchmarks. A renowned example of this is the Volkswagen emission scandal within recent years. Rather than complying with legal regulations; software developers added in a use case for emission testing such that it would dial back emissions in order to pass required benchmarks. Loka suggests than the solution to these issues is a culmination of competency within developers and management. Software developers must gain expertise and management must encourage knowledge sharing within their teams (Loka, 3).

Denning and Riehle classify software engineering as a culmination of mathematics, science, and engineering (1). Software engineering is classified as an engineering process, such that requirements, specifications, prototypes, and testing define. With this said there is still a prominent view that software engineering has much to learn from traditional engineering (Denning & Riehle, 1). It is noted that there is a high tendency to propose hypothesis without testing and evaluation of them within the software field. Riehle quotes 6 concepts software engineers are not particularly good at: Predictable outcomes (principal of least surprise), Design metrics, including design tolerances, Failure tolerance, Separation of design from implementation, Reconciliation of conflicting forces and constraints, and Adapting to changing environments (2, 3). To address these problems, 2 solutions are proposed. First, the software engineering practitioners must take a systems approach. In doing this software should be isolated into components which can all be unit tested, and those working on the software should know how each component interacts with others, as well as its function within the system. Second, software development teams need members with defined roles. There should be, software architects, software engineers, programmers, and project managers (Denning & Riehle, 3). In addition to these four crucial roles, implementing a systems engineer role will help to adapt the systems viewpoint relevant in traditional engineering. In conclusion, the authors suggest that implementing these methods of systems thinking and team management from the more traditional engineering disciplines will be crucial for getting software engineering widely accepting as a true engineering field.

Within both the papers a systems approach is suggested for the current software dilemma. If the teams can adapt a bigger picture view, they will be able to minimize wasted time and introduce more dynamic systems. Loka views software engineering as an art or crafts and does not suggest a move towards traditional engineering. Denning

and Riehle view software engineering as a true form of engineering despite its lack of traditional engineering discipline. Additionally, both papers address the need for interactive project managers to keep the development teams on track. Automated management will not replace a person as face-to-face interaction is always the best mode of communication. In either case, the field of software engineering is still in need of development before it is considered routinely optimized.

In my opinion software engineering is a culmination of both artistic and engineering approaches. I believe that the best results will be achieved with high levels of forethought before construction. With this said, software development is full of nuances such that not everything can be predicted beforehand. From an engineering perspective software development should utilize reusable code throughout their projects. Additionally, working code should be written in a few common languages, and optimized for each. As a result, development saves wasted time by utilizing past works. Additionally, with a reuse principal in mind, it pushes companies to add an extra optimization or algorithms engineer on projects that will encourage teaching and learning within teams. When the development process takes place, the software engineers will still have their artistic form in choosing their language and methods for each project. In the end, we should never take an artistic approach at the sacrifice of performance (exceptions may take place in fields such as web or application design such that UI/UX is heavily valued to the software buyer). Many applications of software are best ran in the background and require no user interaction. It is these implementations that most resemble traditional engineering: kernel design, embedded systems, logic, and assembly language.

Works Cited

Denning, P., and R. Riehle, "Is software engineering engineering?", *Comm. of the ACM*, vol. 52, no. 3, pp. 24-26, 2009.

Loka, R.R., "Software development: What is the problem?", *IEEE Computer*, pp. 110-112, Feb 2007.

Pressman, R., and B. Maxim, *Software Engineering: A Practitioner's Approach, Eighth Edition*, McGraw-Hill, 2015.