Acceptance speech by Ms Margaret H. Hamilton

Universitat Politècnica de Catalunya . BarcelonaTech. 18th October 2018

A Preventative Paradigm for Systems and Software

My background in software has been anything but traditional. It began before the discipline of designing and developing software was a field. There were no courses available to understand what it was we were doing or how we should be doing it. Any prior «education», if you will, was a combination of learning from seemingly unrelated life and work experiences before and during college, majoring in math and minoring in philosophy in college, and working in the trenches for years on several and varied software applications that required doing things that had never been done before while at the same time coming up with solutions and associated rules and techniques for doing such things in the future. On hindsight, I think it all began when I was very young. I remember, as a child, the many philosophical «what if?», «why?» and «why not?» talks I had with my father and grandfather. Both taught me to observe and question everything until answers made sense; and I look back at how important these talks were for what was to become important for my work in the future.

I was fortunate to have had unique challenges, responsibilities and experiences in the workplace, even before high school. All because I needed to raise money for college. One experience, in particular, stands out. Before I was 15, I was hired by the owner of an abandoned copper mine, in the Upper Peninsula of Michigan, that had been converted into a storage place for bananas. He wanted to turn the mine into a tourist attraction where people would be taken on tours through the mine and given lectures on how native copper was mined (this was the only place in the world where native copper existed). He wanted me to help jumpstart his new tourist business. He had not gone past fourth grade, and he knew he did not know enough basics (like math) to do everything himself. He asked me to learn as much as I could about copper, become the mine's first tour guide, and hire and train people to work for me as guides. Our tourist business went from giving the tour to a couple of people a day to thousands of people a day. Not only was I in charge of the guides, I was also responsible for almost everything else connected to the mine, including starting and managing a gift shop, being in charge of the financials and running the business at large. During this time I learned quickly that if you made mistakes you did not allow yourself (or others) to ever make them again! After such an experience, when it later became time to become familiar with and responsible for doing other completely new and unfamiliar kinds of things without any prior background or experience, like designing and developing software, it was a non-issue; or so I thought at the time!

I realize also how fortunate I was to have had so many amazing experiences during the earlier days of software, when we were deep in the trenches, to be in a field before it became a field, and to be around those from whom I learned so much. In 1959, Edward N. Lorenz, a professor at MIT, introduced me to compu-

ters; I developed weather prediction applications in hexadecimal and binary on the LGP-30. Known for his ground-breaking work in chaos theory, he was a most humble human being. His love for software experimentation was contagious, and I caught the «bug». Towards this end, understanding the relationships between the hardware and the software was critical, because it was important on this project to develop software that was performance-optimized. We viewed errors, then, simply as a nuisance, especially since debug sessions took forever.

Another early project was SAGE (Semi-Automatic Ground Environment), for which I developed applications in assembly language on the first AN/FSQ-7 (the XD-1), at MIT's Lincoln Labs, to look for enemy airplanes. The machine was huge, the largest computer system ever built. It was especially important not to make an error, because if you did, the computer would tell on you with siren-like and fog horn-like sounds that everyone within a large warehouse-like building could hear, and flashing lights that everyone could see. Operators and programmers would come running to find out whose program had crashed. One time, in the middle of the night, I received a frantic call at home from one of the computer operators to tell me that something terrible was wrong with my program —it no longer sounded like a seashore. It then became clear that we had found a new way to debug, using sound!

Just as it was with the LGP-30, debugging on the AN/FSQ-7 on the SAGE project was time consuming, and tools did not exist for finding an error and that which made it happen. Software errors were unwelcome mostly because they were considered to be a nuisance (or an embarrassment). During this project, as with the LGP-30 project, I became fascinated with errors —looking for ways to understand what made a particular error(s) or class of errors happen and working on ways to prevent it from happening in the future.

Back then, you were on your own, and knowledge (or lack thereof) was passed down from person to person. A manager hired you if you «knew» the commands in his computer's native language. Like «knowing» a set of English words would mean you could write a novel. I was mystified by this attitude. If a system crashed, software was always the one to blame. Terms were undefined, leading to errors, misunderstandings and drama. Terms like «software», «design», «error», and «computer systems» meant different things to different people. In fact, one of the managers I worked for at first thought the term «software» meant «soft clothing». Also, at the time, the world of software was tribal. Software «types» often could mix things up if and when other software areas were unfamiliar to them (e.g., mixing up the operating system functionality with the target system functionality). Although some things were quite different then, the life cycle process itself by default was not unlike today's traditional life cycle (e.g., the waterfall, spiral, and agile models), going from requirements to coding to endless testing and maintenance.

Apollo On-Board Flight Software

Sage definitely came with drama, especially having to do with errors; but this was only the beginning of what would come next: the Apollo on-board flight software project at MIT, under contract to NASA. The challenge was that the software was man-rated, meaning astronauts' lives were at stake. It had to WORK —the first time. Not only did the software itself have to be ultra-reliable, but it would need to be able to detect an error and recover from it in real time. Learning was by «doing» and «being». Hardware engineers came with formal rules for designing and building hardware; for us, the «software engineers», this was not the case. Problems had to be solved that had never been solved before. At times, we made it up. Most developers were fearless and young, yet dedication and commitment were a given. Managers (mostly from hardware backgrounds) for whom software was a mystery gave us total freedom and trust. There was no time to be a beginner. I began by building software for the

unmanned missions, concentrating on the areas of the systems software, areas used by and impacting all of the flight software. The systems software included the software for error detection and recovery. The software system for the unmanned missions was synchronous.

Manned missions were next. I was now in charge of the team that developed the on-board flight software for the manned missions and the on-board flight software itself; but I made sure to keep my technical hand more often than not in the systems software area. The software was more complex for the manned missions. Our environment was asynchronous (a multi-programming environment), where higher priority jobs interrupted lower priority jobs based on every job's priority relative to every other job's priority. It was up to us, the developers, to manually assign a unique priority to every process in the flight software to ensure that all events would take place in the correct order and at the right time. I was always searching for new ways to prepare for and recover from the unexpected: going from program-specific to system-wide protection. I began to think about all of the possible «what-ifs» and worked on ways to address them. For example, I asked, what if there was an emergency during flight and we wanted to warn the astronauts? I thought there had to be a way to solve this. There was!

Each mission was exciting, but Apollo 11 was special. We had never landed on the moon before. Everything was going perfectly until something totally unexpected happened. Just as the astronauts were about to land on the moon, the flight computer became overtaxed! The software's Priority Displays (AKA Display Interface Routines) of 1201 and 1202 alarms interrupted the astronauts' normal mission displays to warn them there was an emergency, allowing NASA's Mission Control to understand what was happening and alerting the astronauts to place the rendezvous radar switch back in the right position. It quickly became clear that the software was not only informing everyone there was a hardware-related problem, but the software was also compensating for it. The Priority Displays gave the astronauts a go/no-go decision (to land or not to land). With only minutes to spare, the decision was made to go for the landing. The rest is history. The Apollo 11's crew became the first humans to walk on the moon, and our software became the first software to run on the moon.**[1, 2, 3]**

The software's systems-software error detection and recovery mechanisms had taken control. System-wide «kill and recompute» from a «safe place» snapshot-rollback restarts were triggered by the overloading, keeping only the highest priority jobs. During this moment, I remembered the most exciting and memorable discovery related to it. That was when the realization came to me that the steps taken earlier within the multi-programming environment could become the basis for solutions within a multi-processing environment. That is, even though there was only one process executing at one time within a multi-programming environment, other processes were waiting in parallel to that process. With this as a backdrop, the Priority Displays were able to be created, changing the interface between the flight software and the astronauts from synchronous to asynchronous (the software and astronauts becoming parallel processes within a system of systems). This would not have been possible without an integrated system of systems (and teams) approach and contributions made by other groups to support this becoming a reality. The hardware team at MIT changed their hardware and the mission planning team in Houston changed their astronaut procedures, both working closely with us to accommodate the Priority Displays for both the Command Module (CM) and the Lunar Module (LM), for any kind of emergency and throughout any mission. Mission Control was well prepared to know what to do if the Priority Displays interrupted the astronauts.

I thought of the years of preparing for this day and how fortunate I was to work with and share this experience with the many talented and dedicated people who made this possible. Coming up with discoveries, new ideas and solutions was an adventure.

From my own perspective, the software experience itself (designing it, developing it, evolving it, watching it perform and learning from it for future systems) was at least as exciting as the events surrounding the mission.

As developers, we had been given the opportunity of a lifetime: to make every kind of error humanly possible, each holding answers to questions we had not thought of asking. On hindsight, a blessing in disguise. The task at hand was to develop the CM and the LM software. This included the systems-software that resided within both the CM and the LM and was shared between them, and the structure of the software («glue») that defined the relationships between, among and within mission phases. Updates were continuously submitted from hundreds of people (including «guests») over time and the many releases for each and every mission (where the software for one mission was being worked on concurrently with the software for other missions). We had to make sure that everything would play together; that the software parts would successfully interface to and work together with each other as well as with other systems (including the hardware, peopleware and missionware).

We were handicapped by the hardware's time and space constraints, giving software «experts» the license to be creative. «Creative» code (i.e., tricky programming) was admired more than the number of lines of code a person wrote. Requirements were «thrown over the wall» by «non-software experts» who assumed that all the software programs would somehow «magically» interface together. Fortunately, this was not the case. For, if it had been, we would never have learned what was to come next. Further, because of the computer's constraints, data storage locations were shared between, among and within mission phases; and because of the multi programming environment, responsibilities and data storage locations were also shared among many programs continuously interrupting lower priority programs based on time and priority during each and every mission phase. This included both synchronous and asynchronous man-in-the-loop multi-processing within a system of systems. Although there were more than enough opportunities to make errors, there were now the opportunities to come up with solutions to prevent them. «Software engineering» rules evolved with each relevant discovery, while top management rules from NASA went from complete freedom to overkill. *Even though there was no lack of opportunity to make an error and just about any kind of error possible, no on-board flight software errors were ever known to occur during flight.*

Having been through these experiences, one could not help but do something about learning from them. We asked, «What can we do better for future systems? What should we keep doing because we are doing it right?» With initial funding from NASA and the DoD, we performed an empirical study of the Apollo effort. Our analysis took on multiple dimensions, not just for space missions but for systems in general. Lessons learned from this effort (and their impact) continue today: always ask "what if?" and always expect the unexpected. We learned that systems are asynchronous, distributed and event-driven in nature, and that this should be reflected in the language to define them and the tools to build them —characterizing their natural behavior in terms of real-time execution semantics. Having done so, there is no longer a need to explicitly define schedules of when events occur. By describing interactions between objects, the schedule of events is inherently defined. We also learned that the life cycle of a target system is a system with its own life cycle and that every system is inherently a system of systems.

Most interesting of all were the lessons we learned from the errors found during pre-flight testing. They were full of surprises. They told us what to do and where to go. After categorizing the errors, we concentrated our efforts on analyzing three categories of errors: interface errors (data, timing and priority conflicts), which were 75% of all the errors found, errors found by manual means with the Augekugel («eyeballing») or «scanning-listings» methods, and errors previously existing on earlier missions, which were usually the most subtle and hardest errors to

find.[4] Our analysis resulted in a theory based on lessons learned from Apollo and later projects. From its axioms, we derived a set of allowable patterns for defining a system. This led to a universal systems language (USL) together with its automation and preventative development paradigm, development before the fact (DBTF).[2,4,5,6,7,8,9]

It became clear one day that systems defined with the USL behaved differently than those defined with traditional languages. Of greatest interest was the realization that the root problem with traditional systems engineering and software development languages and their environments is that they support users in «fixing wrong things up» («after the fact») rather than in doing things in the right way in the first place («before the fact»). In contrast, with a preventative paradigm, instead of looking for more ways to test for errors, and continuing to test for errors late into the life cycle, the majority of errors, including all interface errors, are not allowed into a system in the first place, just by the way it is defined. Testing for non-existent errors becomes an obsolete endeavor.

We continue to discover new properties in systems defined with the USL. Each property «*comes along for the ride*» throughout a system's own development; the derivatives of the system's definition (its software being one of them) inherit the properties of the definition from which they are derived; integration from systems to software is seamless; reuse is inherent or derivable; a system defined with this language has properties in its definition that inherently support its own development «before the fact»; and[2] each of its systems has «built-in» reliability and «built-in» productivity throughout its life cycle. In contrast to the traditional paradigm with its «test to death» philosophy, with the preventative paradigm the more reliable the system, the higher the productivity in building it, minimizing the need for most testing. Much of the design and all of the code is automatically generated by the USL's automation, inheriting all of the properties of the definition from which it came. The developer doesn't ever need to manually code or manually change the code; only the changed part of the system is regenerated and integrated with the rest of the application, automatically.

We learned that, because the USL is syntax-, implementation-, and architecture-independent, its systems can be developed for diverse architectures, and that the syntax of other languages can be mapped to the underlying semantics of the USL.[7] Unlike a formal language that is mathematically based but limited in scope from a practical standpoint (e.g., kind or size of system), the USL extends traditional mathematics with a unique concept of control, enabling it to support the definition of any kind or size of system. With this paradigm, the language also provides input to and serves as the basis for its automation to inherit and pass on the definition's properties of control for the system's own development process. The USL's automation, a large system (millions of lines of code) in its own right, is completely defined with and generates itself.

It is not magic. These things are possible because of the USL's mathematical foundation. Not only does it take its roots from Apollo and later systems, it also takes roots from other formal methods, formal linguistics, and object technologies. Evolved over several decades, it has always stood its own in several different kinds of environments including academic[6,10,11], government agencies[12], commercial[8] and other environments.[13] Used in research and «trail blazer» organizations, it has been positioned for more widespread use. A radical departure from the traditional, it redefines what is possible. New to the world at large, it would be natural to make assumptions about what is possible and impossible based on its superficial resemblance to other languages —like traditional object-oriented languages. We have learned that it helps to suspend any and all preconceived notions when one is first introduced to this language, because it is a world unto itself —a different way to think about systems.

When today's developers of middle to large systems are asked to list their most pressing issues, this is what they say: integration

is too late, if at all; traceability, flexibility and evolvability are lacking; reuse is ad hoc and error-prone; software is unreliable even with extensive testing; software costs too much and it takes too long to make. Clearly, the last issues on the list, which have to do with reliability and productivity, become solved if most of, if not all of, the other issues on the list are solved. Most people say it is impossible to do much about addressing these issues —at least in the foreseeable future— and so software by its very nature is destined to have these kinds of problems. I ask, «Why is it that today's developers list the same issues as they listed 50 years ago when the field was brand new, and why is it that they give the same reason they gave 50 years ago?»

It is true that many of the pressing software issues that existed in the earlier days still exist today. What we have learned, however, from our own work, is that the reason they gave both then and now is not the one I would give. That is, I believe that the reason for the issues related to developing software is not the nature of software, per se, but it is instead largely because of the traditional paradigm used to build it —a paradigm that has been around since the beginning and continues in force to this day. Many of the well-known problems that exist with the traditional paradigm need no longer exist with a preventative paradigm. Moreover, what works best for developing ultra-reliable systems just happens to work best for systems in general, no matter the application. It has been shown that many aspects of the pressing issues (both 50 years ago and today) can be addressed; if not, ultimately eliminated altogether, with the use of the preventative paradigm.**[2,4,7,8,9,10,11,12,13]** The preventative paradigm with its language and its automation did not disappoint when put to test. In fact, the larger and the more complex the system, the better the results. What is able to be done with what has been learned, however, depends on how ready and how open developers are to a change in how we build software. Given the kinds of systems we need to build for today and tomorrow, I believe that change has to, and will begin to, happen, sooner rather than later.

For whatever success I may have experienced, much of it was because I was in the right place at the right time, with the right opportunities and the right people. In some ways, I had the benefit of beginning with no preconceived notions, since it was necessary to go into areas that had never been gone into before. Many of the things we were doing had not been done before and for that I feel very lucky. Much of the credit goes not only to those I have learned so much from, but also to the errors I have had the opportunity of having had some responsibility in their making, without which we would not have been able to learn the things we did. Some with great drama and fanfare, and often with a large enough audience to not want such a thing to ever happen again!

The errors showed us how to exist without them. They led to a language with a preventative paradigm where a system's definition *inherently replaces* much of what used to be aspects of the system's own life cycle, and which now becomes no longer needed; and it *serves as the input* for the language's automation of what used to be manual processes in the system's own life cycle, and therefore results in many parts of the system's own life cycle becoming no longer needed. In essence, trail blazing and taking risks in unknown territory led us among other things to the errors; the errors led us to a paradigm that leads «before the fact» to the future. Educating people how to think, do and be in

terms of the paradigm becomes the next real challenge.

References

[1] Snyder, L and Henry, RL. «Fluency7 with Information Technology». Pearson, New York, NY; 2018, p. 173-176.

[2] Hamilton, MH. «The Language as a Software Engineer». 2018 International Conference on Software Engineering; 27 May - 3 June, 2018; Gothenburg, Sweden. Celebrating its 40th anniversary, and 50 years of Software engineering.

[3] Hamilton MH. «Computer Got Loaded». Letter to Datamation. Cahners Publishing Company; 1 March 1971.

[4] Hamilton, MH and Hackler, WR. «Universal Systems Language: Lessons Learned from Apollo». IEEE Computer. doi:10.1109/MC.2008.541. December 2008.

[5] Hamilton Technologies, Inc. 001 Tool Suite (1986-2018). Example demo: «System: do_all_taxes». Cambridge, MA; 25 Aug. 2018.

[6] Hamilton, MH and Hackler, WR. «Universal Systems Language for Preventative Systems Engineering». Proc. 5th Ann. Conf. Systems Eng. Res. (CSER), paper #36; Stevens Institute of Technology; Mar. 2007.

[7] Hamilton, MH and Hackler, WR. «A Formal Universal Systems Semantics for SysML2». 17th Annual International Symposium, INCOSE 2007, paper #8.3.2; San Diego, CA; Jun. 2007.

[8] Hamilton, MH. «Universal Systems Language (USL) and its Automation, the 001 Tool Suite, for Designing and Building Systems and Software». IEEE Computer Society/Lockheed Martin Webinar Series, slides 36-41; 27 Sept. 2012.

[9] Hamilton, MH. «What the Errors Tell Us». IEEE Software-Special issue, «50 years of Software Engineering». September/October 2018; 35(5).

[10] Ouyang, M and Golay, MW. «An Integrated Formal Approach for Developing High Quality Software of Safety-Critical Systems». Report No. MIT-ANP-TR-035; Massachusetts Institute of Technology, Cambridge, MA; 1995.

[11] Krut Jr., B. «Integrating 001 Tool Support in the Feature-Oriented Domain Analysis Methodology». CMU/SEI-93-TR-11, ESC-TR-93-188. SEI; Carnegie Mellon University, Pittsburgh, PA; 1993.

[12] Department of Defense. «National Test Bed Software Engineering Tools Experiment'final report», vol. 1, Experiment Summary, Table 1, p. 9. DoD Strategic Defense Initiative Organization; Washington, DC; Oct. 1992.

[13] Schindler, MJ. «Computer-Aided Software Design: Building Quality Software with Case». John Wiley & Sons; New York, NY; 1990.