[talking head] Hi. I'm Eric Hehner. I'm from Toronto, Canada. And I have the pleasure to give this course on formal methods of software engineering. I hope you enjoy it, and I hope it helps you become a better software engineer.

Formal methods are on the theoretical side of software engineering. Every branch of engineering has its theory. Electrical engineering has Maxwell's equations, and Kirchoff's Laws. Civil Engineering has geometry and the theory of material stress. To be an engineer, you have to know the relevant theories, and be able to apply them to practical problems. There was civil engineering long before any theory. People built buildings, but not skyscrapers. Sometimes the buildings collapsed; that was considered normal then. Now there is an advanced theory, and we don't let people build buildings unless they know the theory, and their buildings rarely collapse. Software engineering is newer than the other branches of engineering, and its theory is even newer. Right now, there are a lot of people building software who don't know any theory, and the software often has bugs. That's considered normal. But it's changing. It changed very quickly for the VLSI industry when Intel discovered a bug in their floating-point hardware that cost them many millions of dollars to fix. Now formal methods are used to verify algorithms that will be cast in silicon before they are sold. They are used by software companies that produce safety-critical software, where people's lives are at stake.

[1] All of these applications have made use of formal methods. The banking industry, and creators of text editors and web browsers and mail programs don't yet use formal methods; when that software fails, no-one dies. But Microsoft is now starting to introduce formal methods to some of its programming. The applications where you really have to use formal methods [2] are those where you really must get the software correct. In a few years, I think the use of formal methods will be normal practice. [3] There are two ways to look at programs. One way is that programs are commands to a computer [4] and the other way is that programs are mathematical expressions. We need to think of them as commands to a computer [5] when we want to get the program executed. We need to think of them as mathematical expressions when we're programming, so we can use [6] the theory of programming to help us get the programs right. [7] Why theory? People sometimes use the word "theory" to mean a guess. They might say "here's my theory of what happened". Or maybe they mean an explanation of something. What I mean is a means of proving theorems. It's sometimes called [8] formal theory, because it consists of a [9] formalism, which means the rules for writing formulas, and rules for manipulating the formulas, rules for calculating with them, rules for proving theorems. Whenever I say theory, I mean formal theory, so I won't bother saying [10] formal. We need the theory [11] to prove our programs are correct. We need to program by [12] calculation, just as a theory of motion lets us calculate the trajectory of a planet. We want the same kind of [13] precision.

Most of all, what we can get from a theory is [14] understanding. Just as a theory of motion gives us an understanding of the motion of planets and stars, a theory of programming gives us a better understanding of programs.

[15] When I said formal theory, I did not mean a careful and detailed explanation. Likewise, by informal, I don't mean a sloppy or sketchy explanation. [16] By formal I mean using mathematical formulas, and by informal I mean using natural language explanations. It's quite possible to be informal and careful and detailed. And just as possible to be formal and sloppy and sketchy. There isn't really any choice about whether to be formal or informal. When you start a programming project, [17] you have to start informal, discussing it with other people, finding out what's wanted, maybe drawing some pictures. And [18] you have to end formal, because you end with a program, and that's completely formal. The only question is how to go from an informal start to a formal finish.

When you are finished, [19] you have to test your program to see if it's working. [20] But how do you know if it's working? If it crashes, obviously it isn't working, but if it doesn't crash, how do you know if the answers it gives are correct? [21] And you can test

only some cases, not all of them, because there are too many to test. Maybe even infinitely many. There may be errors that would show up in the cases you didn't test. [22] The only way is mathematical proof. Proof tells whether the program is correct for all inputs, even if there are infinitely many inputs.

[23] Some formal methods tackle the verification problem. Given a specification and a finished program, prove whether the program is consistent with the specification. That's a really hard problem. It's much easier [24] to do the proof as you program, proving each programming step as you make it. That's because the information needed for the proof is exactly the same information needed to make the programming step. What makes verification after development so hard is that it has to reconstruct that information. This course concentrates on program development, with proof at each step, [25] and on program modification, again with proof that the modification is correct.

The first usable theory of programming, [26] in 1969, was Hoare triples, or sometimes it's called Hoare logic. It's still the best known theory, and if you've seen any formal methods before, it was probably that one. Since then, [27] lots of other, better methods have been developed. These have all been used by industry, but only to a very limited extent. The method that has been used most is [28] model checking. Model checking is really [29] exhaustive automated testing. People like it because it's a lot easier than proving. It checks all possible states of all executions against a list of properties that execution should have. So model checking works only on programs that have a finite number of states. [30] But a good model checker can handle an incredible number of states. They say 10 to the 60th power of them. That's something like the number of atoms in our entire galaxy. Amazing. Now [31] 10 to the 60 is approximately equal to 2 to the 200, so that's the state space of [32] 200 bits. And that's about the same as [33] 6 ordinary 32 bit variables. So we're talking about programs with 6 or fewer ordinary variables. That's not very impressive at all. Sometimes there's a way around this limitation. If the program has a state space that's too large for model checking, you might be able to find an [34] abstraction of the program, a simpler program, with a small state space, such that model checking the simpler program still answers your questions about the larger program. Finding a good abstraction is hard, and you have to prove that it doesn't lose the properties of the big program that you want to check. So we're back to proving. There's really no way around it. You're welcome to look up any of these formal methods that interest you, but we won't be using any of them in this course.

[35] The theory we will be using was originally called predicative programming, but now it's called aPToP, which stands for a Practical Theory of Programming. [36] It is simpler than any of the other theories. In those theories, a specification is a pair of predicates, or a function from predicates to predicates, or a temporal logic expression. In this theory, a specification is just a binary expression. That's the kind of expression you write after the word "if" in a program. aPToP is also [37] more general than any of those theories. Some of them are only for terminating computation, and some are only for nonterminating computation. aPToP is for both. Some of those other theories are [38] only for sequential computation, and others only for parallel computation. aPToP is for both. Some of those other theories are [39] only for stand-alone computation with no interaction, and others are only for interactive computation. aPToP is for both. [40] And aPToP also gives time and space bounds, and can be used for real-time programming also. It also works for [41] probabilistic computation that includes random numbers and produces its outputs with some probability distribution.

[42] The prerequisite for this course is that you have some programming experience. I don't care what language you used. Any programming language will do, as long as it has [43] an assignment statement, and an if statement. We'll be using a language that's so small it will take you 30 seconds to learn it.

[44] Here's the textbook for the course. I wrote it, so it fits the course exactly. And the best news of all – you can download it free from my website. We'll start Chapter 1 next lecture.

[talking head] One of the nice things about having the lectures online is that you can watch them on a computer or any portable device anytime and anyplace you want to. You can stop anytime, and resume later. You can replay anything you want to see again. You don't have to play it straight through as though it were a classroom lecture. You can and should pause at any point where you need to think about what was just said before you go on. I hope you enjoy the course, and benefit from it.