PROFESSOR: Just very quick-- very quick what we're doing today and what we're doing next on Wednesday. Phil's going to start with an estimation lecture. We're going to follow that up with a lecture from me on Sprint Backlogs and Task Lists. And then we're probably going to take a break. You'll come back and you'll have essentially a Sprint planning meeting here in class where you will create a Sprint backlog and a task list, hopefully with estimations down to how many hours you think all those tasks are going to take you.

Then we'll ask you to come up and give a very brief two minute presentation on hey, how did the process work, and what was it like? How did it work for your team? We're not actually interested in hearing about your game or what the top task you have to do is.

We trust that you have a good game in progress and that you've got a reasonable set of tasks. But we actually want to hear about how that meaning worked. So we'll also give you about 10 minutes to have a little sort of post-mort review to talk about the process, after you've made your task lists.

Andrew will then come in. He'll be leading a discussion, sort of a review discussion on good software to see what you've been doing and see if you've remembered anything he said from earlier. And you should have turned in your product backlogs. And you should have code running on your machines. We're not going to walk around and inspect all your laptops to find out if you have running code.

But to be on target, you really should have something running, and if not playable, at least pokable. For Wednesday, we'll be talking about testing. And you'll be running a focus test in house. And Genevieve Conley, yes, from Riot will be also giving a guest lecture after the focus test.

And I believe she'll be here to wander around and help out in the focus test a little bit. So I think that's everything. Now! Those were my two slides. Now you can have the controls.

PROFESSOR: Do try to come in on time for class. If you are unsure about whether you can get your game

set up in time, come in early. And make sure that your game is ready. When we have guests coming to our class, we really will prefer that people don't walk in terribly late. It's just kind of embarrassing.

All right. So over the weekend, I found myself in an unenviable position of having to teach-- prepare for a video game class, and not having a working computer. So I had to prepare for the possibility that I wouldn't have a computer as of today, and redo my entire presentation in analog format. But it turns out that IS&T actually does miracles and got my computer back and working on Monday. If you happen to be working at IS&T, give yourself pat on the back. You are doing God's work.

But I'm going to try to do the analog thing anyway because might is worth a shot. I'm going to do to be very clear, this is the first time that I'm doing this. I'm going to need three volunteers to come down. Come on down. Come on. Come down. All right. OK.

So on Wednesday, we asked you to split things up into small, medium, and large features, right? Small, medium, large. And I'm going to give you an idea of what your product backlog currently looks like. Your product backlog currently looks like a bunch of features. All right.

Now I'm going to assume that all of you have sort of already assigned small, medium, large things to your features. Sort of small, medium, large categories to your features. But I want to try doing it slightly differently. I'm going to give each of you three cards. These are the large currents. Medium.

Can people see this from the back, by the way? The letters on the card? Small.

So this is what's going to happen. I'm going to pull something out from here, and you all are going to all look at it. And you are going to figure out whether that is going to be small, medium, large without actually discussing anything. All right. So you keep the cards hidden, and only you can see it. And then when I ask you to reveal it, show it all at once.

Show me what you've picked. All right. I'm not going to tell you what's small, what's medium, or what's large. I want you to tell me what's small, medium, and large.

All right. So everyone clear on how this works? OK. Don't show it to anyone yet. Make sure everyone's picked. Yes?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Good question. Hang on to that. Is this a small, medium, or large feature with no frame of reference? So everyone's got one? Show it to me.

Medium, medium, medium. All right. I'm going to put this here. Next one. Can everyone see this? All right.

Everyone's picked one? Show it. Medium, small, small. OK. Why is this a medium?

AUDIENCE: Looks similar.

PROFESSOR: It looks similar. All right.

AUDIENCE: [INAUDIBLE].

PROFESSOR: OK. The other one does have a hole in it. If you haven't seen it. OK. I'm going to put this one aside for now. This one. It is red. All right.

Still being picked. Small, small, small. OK. So this is unequivocally small. And this piece. This one also has a color in it.

All right. Show it to me. Large. OK. Going back to this one, what do you think? Everyone show me what you're thinking.

Everyone thinks that now is small. OK. All right. So, don't go back yet. But I want to say thank you very much to all of our volunteers.

Why did I just make you all go through all of that? That seems like a very, very inefficient way of sorting out features. Like if I just gave you this entire box, this is what I want you to do now. Sort this whole box into small, medium, large.

All three of you together. You don't need the cards. Go for it.

AUDIENCE: [INAUDIBLE].

PROFESSOR: OK. Thank you very much. If you can just hang out here, just a little bit longer, that will be great. Now that is probably a fairly good analog of how most of you figured out small, medium, and large on Wednesday. Which was you just compared features next to each other.

Figured out which one's seen on the larger end. Which ones was on the small end. And just

kind of put things in place. But there was some discussion going on. Is this small? Is this large?

It's just like do two of these put together make one medium, for instance? Now think about the first method of first choosing whether this was small, medium, large and revealing it, was that I forced you to actually voice out why you thought that was different. I believe you picked a medium. And the others picked a small. For a lot of these other pieces, you didn't voice anything out.

And one of the reasons why we can do this so quickly, is because human beings are actually pretty good at reasoning about spatial volume. You're pretty good at-- and being at MIT, you also understand geometry, 3D geometry, in a sense pretty well. So you can make these spatial reasonings pretty well. But when it comes to things like time and complexity, human beings tend to have a tougher problem.

So I'm going to say, for instance, I'm going to give you three blocks. This represents for you, seven hours of work. OK? This represents for you, six hours of work. This represents eight hours of work for you. All right?

So you know you've got 16 hours of work to do. Pick out features that you're going to implement in your next print.

AUDIENCE: So this is how much time I have in my life.

PROFESSOR: That is how much time that you spent on the last feature. If all three of you worked on identical features--

AUDIENCE: OK, so this is already done. This is already done.

PROFESSOR: Yeah. This is done. This is a known value. You spent seven hours on that. You have 16 hours.

AUDIENCE: So I spent seven hours on this. And then there are the two small triangles.

AUDIENCE: [INAUDIBLE], right?

PROFESSOR: The first one is just a reference. And the little feature. OK. All right. All right. What did the cube represent to you again? Seven hours of work. And you picked?

And you think that that's a reasonable amount of work to get done in 16 hours? Not including

that one. So this is about 16 hours of-- assuming volume equals time. Sorry, volume equals effort, actually. All right. And over here?

AUDIENCE: How long did it take me to do the other block?

AUDIENCE: Six hours.

AUDIENCE: Oh. I guess I could have done a little bit more. Well, that's fine. I'll just do [INAUDIBLE] like them. And then if I have leftover time, I'll do more.

PROFESSOR: If you went for more, what would you try to pick on?

AUDIENCE: I guess maybe this one. It's a little bit more than perhaps I could do in exactly 16, but it wouldn't hurt to touch it.

PROFESSOR: It's the smallest thing that you can find. OK. All right. And for you?

AUDIENCE: This exactly fills my estimate. But realistically, I'd probably go with this.

PROFESSOR: So you're using kind of programmer patterning. You got a question?

AUDIENCE: Sometimes you do not [INAUDIBLE] the small things, for example, adding balls to your game set is small. But bringing out the framework itself is a large test that must be done before you have the choice of choosing smaller tasks.

PROFESSOR: That's true. That's true. I am not indicating priority in here. I'm assuming that you're picking up high priority tasks. The next thing that needs to be done.

Thank you very much. And you may have a seat. Big hand to our volunteers.

[APPLAUSE]

I'll be going back to this example shortly, as soon as I get my presentation going on. I want you to hang onto these examples because I'm going to come back to them during the talk. On Wednesday-- and what I first asked our volunteers down here to do was to do a small, medium, large, extra large feature sizing, basically. The reason why we use these broad categories is because actually it can be pretty fast. And on Wednesday a lot of you did it pretty quickly.

However, what I noticed on Wednesday was a lot of people weren't actually discussing how

big or how small these features were. Like you all using your own little metrics. So that was a good question. Small, medium, or large to what frame of reference? And everybody had their own frames of reference.

When I forced people to make the assessment and then declare it in front of the entire team, and then discuss if they disagreed with the entire team, then assumptions started coming out. You had to vocalize why you thought a certain feature was a bigger or smaller. You could adjust your assessment after the discussion. But it very, very important to get that discussion out to everybody.

So as long as you're discussing with your entire team, I don't really care what method you use to split things up into small, medium, and large. But you need to be discussing as an entire team. In fact, every single member of the team should be in on the discussion, because somebody in charge of sound design, or somebody in charge of art, or in charge of art of the code architecture would have very different perspective on any given future from other people on the team.

And you want all of those perspectives in the same time. There's oh! Playing a sound effect. That's going to take no time at all, because we already have sound effects in our game, and we know how that code works. That's what a programmer thinks.

The sound designer says wait a minute. I need to create that sound effect. I don't even know what this is going to sound like yet. I have no idea. And that's going to be larger feature from my point of view.

And you want that discussion to come out. So each row should try to lend its perspective. And is very important to listen. The reason why I forced everyone to do this-- the volunteers to do this-- right at the beginning was because it was a very turn-based method of getting that discussion out. I had to force everybody to listen to everybody.

And something that I didn't ask our volunteers to do was to offer alternate solutions. However, they kind of did that automatically by themselves. When I just left them to their own devices, I said, sort these out into features. They started discussing well, this is how I look at a problem, this is how you look at a problem. And they quickly came to some sort of consensus.

Of course, something I mentioned on Wednesday was if you've got a really, really huge feature that's kind of hard to figure out. It doesn't really fit neatly into any of these boxes.

Break it up into smaller features. You want to converge. You want to converge down in to a consensus.

But there are right ways and there are wrong ways to go about that. Every time you have that discussion, you're getting new information into the team about how complex a certain problem is. Maybe sometimes through a discussion, you discover a certain problem is easier-- a certain feature is easier to implement than you had originally expected. And then you redraw a C.

Other people's estimates is actually new information. And that's actually what happened here. There was small, small medium, and the people on the volunteer team figured out how other people were seeing the problem.

Here's something that's very important. Don't negotiate. Don't say I think it's small, you think it's small, most of the team thinks it's small, somebody thinks it's medium. Just change your estimate down to a small. I suppose it's kind of adamant.

No, I think this is a bigger problem than you think it is. Don't try to negotiate with that person because that person has a valid point of view, and this starts to focus a discussion on changing the person's mind, not focusing on the feature. You actually want to figure out how complex a certain feature is. For the same reason, don't take an average.

Someone thinks it's a small feature, someone think that's a large feature. That doesn't make it a medium feature. That means that your team has no idea what this feature is because you aren't in agreement with how complex something is.

So what I did-- does anyone remember what I did when-- let me see. Which feature was the weird one that people were kind of hung up on? I think it was this one. When we got to this one, and there was some disagreement, can anyone remember what I did? Yeah. I just put it aside, right?

I actually gave it away. It is a very, very easy to get into a long drawn out discussion about how large or how small a feature is. It's usually much more clearer if you sort of tackle the easy ones first, and then come back to the ones that you disagreed on. But you remember that you disagreed, and you got new information in the process of talking it out.

So that's a role for the Scrum Master. The Scrum Master says we don't want this feature estimation process to take too long, so we're just going to put that feature aside first, look at

the other features, do the estimation on that, and then review it later. But a Scrum Master does not decide what size that feature is. The whole team has to come to a consensus. You want to converge on a single estimate.

So if you put together your product backlog, and you never really discussed with the rest of your team what those size estimates were, then that's something that you're going to do later today. Before the end of today I want you to have a discussion about whether these things are actually small, medium, or large. And the reason is because estimation-- it is not just a process of just putting down numbers or letters on a spreadsheet. There is a goal for the process of estimation. And it's this.

It's to get a clear enough view of the reality of your project so that you can make informed decisions to control your project in order to hit the targets. Now what's a target? A target is a goal that you wish would happen. It's your fondest desires is to get this project done. Entire product backlog of all features into your game before the deadline.

That's a target, right? Maybe your target's a little bit closer term. What can you get done by this week? What can you get done by this spread? It's still a target, and something that you want to hit.

But you can't base all of your decisions on your fondest desires. You have to base it on what your team's actually capable of doing. An estimation is a process of actually trying to understand what your team is capable of doing. Not whether what you want to happen, but what can happen.

So what are you estimating? This is where you are at right now. You have a product backlog, and you have size. You have feature size that is going to get broken down into a bunch of tasks. Each feature is going to be broken down into individual things that individual people can do.

Maybe a certain feature like jump involves art, involves code, involves sound, involves debugging, involves integration. All of these are individual tasks that might be carried out by different people. So what you're really doing in the second step is to figure out how much time something's going to take.

Remember what Drew said on the very first day of class. It's not so much how hard something is, how hard a feature is, but how long it's going to take. And what you want to do is figure out

how long it's going to take.

Now that's a process of arriving at a number of how many hours this is really going to take, and is it going to fit within the amount of time that we've actually given the team? And you need to translate it first from size into effort, which is like, total amount of effort that's going to be used to tackle the problem. Split it up among the people who are actually going to work on it, and then each person has an individual amount of efficiency.

So for instance, I gave exactly the same cube to all three volunteers. But I told them this meant different things to each one of you. And when they picked off tasks off the product backlog, they picked it based on their own personal metric. That's what you've got to do too. When you decide that you're going to take on a task, you have to assess it based on your own ability to actually execute on a task.

Say it's an AI problem, and you're not very well versed in AI. It's going to take you longer than somebody that has done artificial intelligence before. But that's OK because maybe the person who's doing artificial intelligence is busy trying to solve or harrier problem establishing the base framework of code. And that needs to be done first. And that needs to be done sooner.

So that person is busy. You can start working on certain features, even though you may not be the best person to do it. That's fine as long as I make an honest assessment of how long it's going to take you to do it.

Size estimates are pretty easy. You're just sorting things into buckets. Very large buckets. Time estimates are very much like hitting a moving target. As you start to make estimates, you are going to discover that many of those estimates just flat out wrong.

You're going to say oh, that's going to take me four hours to do. And it takes you four days to do. And you're going to have to communicate that to your team.

Giving an estimate is not the same thing as making a plan. It is part of the same process, but I want to be very clear about what's the difference between estimation and planning. Say somebody on your team asks you for an estimate. [INAUDIBLE] alarm bells going off when he says, how long is this going to take you to do? Everyone starts to get worried because you're not quite sure whether they're actually trying to ask you for how difficult is something, and how much time do you think it's really going to take.

Or are they asking you, can you get it done by Monday? That's kind of like the hidden question

that they're not asking, but they think that they're asking. So you need to be very clear when you respond, whether you're giving an estimate or a plan. You plan to reach your targets. Obviously, none of you-- I hope none of you are planning to not hit your targets.

I plan to get this done one week after the deadline. (WHISPERS) No. But once you've made your plan-- here's our entire product backlog, and we think it looks reasonable, and we think that it can actually be done by the deadline-- you need to start making estimates to see whether that plan is realistic.

You can say, yeah, I think that's a reasonable estimate when you think that assessment is accurate. I've done something similar before, so I figured that this is something that we can get done before the end of the week. You can commit to that estimate.

But you haven't committed to the plan. You haven't said, I'm going to take responsibility for getting it done by the end of the week. That's a very different thing. When you accept the task and say, all right, put my name on that spreadsheet, and I have decided that I think I'm going to be able to get this done in four hours, so I'm now committing to this plan.

Of course then everything goes wrong, and you re-estimate. You're allowed to make re-estimations while you work on tasks. You have to because no plan survives first contact with actual development.

So here's a quick quiz. Here are four statements that you might hear in the middle of a team meeting. OK. I want you to tell me what each one of these things are.

I've already used the words describing each one of these things in my talk so far. It has to be done in two days. What is that? It's a deadline.

I used another word to describe that. It is a deadline. Is it a plan? It's a target. Someone. All right.

So the next one. It'll take about two days to do it. You're talking in a team, you say, this feature looks like it'll take about two days to do it.

That's an estimate. That's right. I'll need two days to do it. That is a plan. That is a plan that you're committing to, more accurately. It's like, I will take two days to do it. You might take three days to do it. You might be able to do it faster than [INAUDIBLE]. But I'll need two days to do it.

And of course the last one is, actual reality. It actually took two days to do it. I mean, in reality it'd be like, it took four days to do it.

So I want you to be very, very clear that when you're communicating with your team, what you are trying to say. Are you giving estimations? Given all the information that you know, this is how tough something is going to take. And this is how long it is going to take to get this feature working. Or are you making a personal commitment, saying this is my responsibility now? I'm going to get it to you by Monday.

Just a quick side note, you've probably seen these two words come up in numerous engineering classes. The difference between accuracy and precision. Which one is more accurate? Raise your hand in the direction that you think is more accurate. OK. That's actually-- OK.

I think that's a majority on this side. Which means that that one is more precise. OK? So that's right. This is more accurate. It's closer to the target that you're trying to hit. Even though it's kind of like broadly spaced out.

None of them actually hit the target, but if you took the average of all, then you get pretty close to the center. That one's a really nice tight grouping. Anyone here do pistol, or archery, or rifle? So we always talk about how tight the grouping is when you're shooting. That's very precise. That can be corrected by say, fixing the sites on a pistol, or on a rifle.

But for time estimates, this is consistent underestimating. Right? It's like, yeah, it's going to take me two days to do it. I'll get it done in two hours. Really, it took four days and four hours. And you're always doing this.

When it comes to estimation, accuracy is more important then precision because you're going to work on multiple tasks. And if your estimates are a little bit off all the time, but sometimes you're under, sometimes you're over, that kind of averages out. Which means that you can still make some effective plans.

But if you are consistently underestimating or consistently overestimating, you need to figure that out quickly. You need to learn that about yourself during this class so that you can start to correct that. And if you can't figure it out, hopefully somebody else on your team can realize that your estimates are always off in a certain direction and then adjust the plan accordingly.

It's also important to know what the uncertainty of your estimates are. It's very valuable to know how imprecise an estimate is. It's like, this could be done in a week if I find someone else who's implemented a feature just like the one that we want. If not, I'm going to have to implement it from scratch, and it's going to take four weeks to implement. I mean, that's a fairly common scenario.

So it's very valuable to know how imprecise an estimate is. But while I welcome that kind of discussion in a team, don't just give a range. Don't just say, well that's going to take anywhere between one week and four weeks. That's too wide a range. You can't establish a plan on that.

How many of you have been in this situation? Two minutes remaining. And it just kind of sits there. And it's really four minutes later, and it's just what the heck's going on?

It would be much better if you had actually, that progress bar. And that progress bar is actually giving you a single point estimate. This progress bar is kind of telling you this job is about 15% complete. And it's telling me there's two minutes remaining. And that bar still has a long way to fill up.

So you realize this estimate may not be terribly accurate, because it not working on a lot of information. It may be a pessimistic estimate. But you still need to commit to a single point estimate, if you want to make a plan around it. Don't just use the average of a range. The difference between a one week estimate and a four week estimate for the same feature, is not somewhere in between.

It's not like it's going to take two and a half weeks. With the one week estimate and a four week estimate, are two very different circumstances. One week, if you found somebody else's code that basically does the same thing. Four weeks if you had to write it from scratch. One week in, you know exactly what situation you're in.

Either the code is done, or the code is not done, and you know it's going to take four weeks. Your estimate is not like, two and a half weeks. You're not halfway in between.

The good news is that the time estimates are entirely internal to your team. No one outside your team needs to know this. I don't need to know this, Sarah doesn't need to know this. None of the graders need to know what your time estimates were. We're not holding you to-- your grade is not dependent on this.

This means that you can revise your estimates over time. And I'm going to encourage you to do that. You are not Neo. You aren't going to be faster, just because you want to be faster. So it's important to actually understand what's your actual pace of work.

Now this is something that I'm going to recommend that everybody do starting from today. And that's to track your own estimates. This is one way that you can do it. It's not the only way, but it's good enough. Where you write down the feature that you're working on.

Say it's the jump feature. And you have the task that you took on. I'm a coder, so I'm going to say I'm going to put some upward velocity when the player hits the space bar. I'm also going to write some code to play a sound effect when that happens. These are two separate tasks.

And with the original estimations, that was going to take eight hours. It's going to take five hours. And then you actually write down how many hours you spent on this task so far.

For the first task, it was done. It took me 10 hours to do. Or longer than I expected. So my eventual estimate was 10 because I actually knew it took ten hours to do. If I ever have to do that same feature again, let's say a feature of the same size, again, a task of the same size. Again to be accurate, then I'm going to estimate, next time if I think it's going to be eight, I'm going to write down 10 instead, because that's what previous history has taught me.

Let's say I'm in the middle of writing my jump sound code. I thought it will take five hours. I've spent two hours on it. And I look at it, and I think actually this is a little easier than I thought it was going to be. So I'm going to say probably only another two more hours of work on this.

And my current estimate is actually being revised downwards. So this is going to get faster. And if I put this in a public location, where my Scrum Master to read this, it can actually see my progress on my work. You might be able to also do this on certain task tracking tools.

Don't estimate in ideal work hours. Don't say, if I had four uninterrupted hours to just work on this game, I'm going to get this jump thing working. How many of you ever have four uninterrupted hours? Really? Is this like between midnight and 4 AM? OK. OK.

Uninterrupted? You're not like, checking Facebook, or having to get a snack in between? No? Really? If there are distractions, if there are team meetings, if there are other problems that [INAUDIBLE] that you expect to have to do, you should count it. You should count it in your task time.

You should be able to account for this. Because say so you've blocked off 12 hours of-- No. Let's say six hours. You are going to spend six hours on this project over the week. And you know you've got that, because you're going to have to spend the rest of your time going to other classes, and doing other work.

And you know that you've got a one hour meeting somewhere in there. Part of your task is actually going to that meeting, and figuring out how a certain feature should be implemented, before you actually start writing the code. Or actually start doing the art for it. You want to factor that into your task estimation. You don't want to say that's a separate thing.

At least if you do count it as a separate thing, make sure that you've actually got that as a separate task with an actual time estimate on it. Count all of your distractions. Do try to schedule and time box all of your meetings, because it makes estimation easier. If you know that a meeting's time boxed to one hour, you can't spend more than one hour in a meeting.

And you can actually account for that in your estimations. Don't just always have ad hoc meetings, for instance, like on Skype. Because then you're just stealing a couple of minutes from everybody's schedule all the time without any one being able to plan for it.

So you've got something like this, where you're tracking all of your progress, then you just add up all of your time estimates when a feature is done. And then you can go back to your original small, medium, and large and say this thing took me seven hours to complete. And if I get another feature that looks like these two features, for instance, and say it's about two half features compared to that seven hours. I said, OK, three and a half, three in a half. And you can use that.

It's not going to be perfect. It's not going to be accurate. But you're basing it on real evidence of how fast you personally work. So all of this information helps you become a better estimator of your own performance. And in the future, you can use those numbers for future sprint planning.

So when you finish one sprint-- say it's project 4, and you've got multiple sprints. You finish one sprint, and you know that this takes you eight hours to complete. So the next time you get this size feature, well it took me eight hours last time. So I'm going to say that that feature looks like a eight hour task.

You should re-estimate every day. So you animate your jump and say that's too big of a

feature, so I'm going to split into two tasks, the animate, the jump take off, and the animate, the jump landing. Since they are two very similar tasks, I'm going to put the same time estimate on both of them. And then I'm going to start working on jump take off since you have to go in the air before you can come down. And I'm going to start working on this.

The task is a little bit harder than I expected, so actually it was a five hour estimate. So I'm going to increase it up to a six hour estimate. And then eventually, I got it done. Down to zero.

I have no more hours on this feature. It works. It's tested. It's integrated. Everything's fine.

Now I have to do the second feature. Animate the jump landing. What should I write down for that? Let's say by the time I actually came to the situation where I wrote down I had six hours remaining, I had already spent eight hours working on the feature. I spent eight hours working on jump take off, and it wasn't done.

Then I wrote down I had six hours left. So what's my new feature estimate? 14. A lot bigger. Because that's real evidence.

It may be that during the process of doing this earlier feature, you've learned certain techniques that's going to speed up your next thing. However, that's wishful thinking. That's hope. That's what you desire will happen in reality.

The actual information that you have is that actually if you add up all the hours I actually spend on this, it took me 14 hours. So I'm going to revise the next feature and let the rest of my team know that I think this next feature is going to 14 hours.

Divide and conquer. If you've got a very big feature and very, very big task, break into little bits, and do your estimations. Don't forget to account for debugging. This is one of the very first actual bugs found by Admiral Grace Hopper, stuck in a vial somewhere in a computer in 1940, that actually caused a program to fail. That's one theory of where the term debugging came from.

Don't forget integration time. Taking your code and your assets and your writing and your dialogue and sound effects, and making them work with other people's code and contributions. Don't forget that some of you are going to get very exhausted and overloaded while you're working on this project. And don't forget to take breaks, and to account for those breaks in your time estimation.

If you know that you don't tend to work well on a four-hour stretch, and you tend to work much better with one hours, with a five minute or 10 minute break, and then another hour, then use that in your time estimates. Try to estimate so that you can work efficiently. Not just how you wish you could be working. Any questions?

I thought that was kind of fire hosey. We'll probably get back to this-- actually we will actually have you practice this later on in class. Five minute break? All right, so right now we'll do a five minute break while we switch our computers around. Be back here by 1:53.

OK. We are back on. So we're back on. Hopefully this will actually not be terribly long, because at least some of the stuff I'm talking about, Philip has already really touched on, and discussed in more detail than I need to hit. Since he just talked about it.

I'm just taking a lot of the terms he has introduced and explained and talked about all the estimations and tasks and so forth, and stuffing them back into our Scrum framework. So you can think about-- given that data, about estimating, what do you do with it? How do you show it? How do you share with it? How do you record it?

But first, let's have a really quick review. Remember I said there would be a quiz later? Here's our quiz. I want to see how many of you actually still remember we talked about on men day. So can I get some volunteers to give you some definitions up here? One per customer.

AUDIENCE: All right. [INAUDIBLE]. In charge of the vision of the products, and also in charge of maintaining the backlogs of the team's [INAUDIBLE].

PROFESSOR: Yep. Specifically prioritizing the backlog, but yeah. OK. Someone else? This is going to be a really long lecture.

AUDIENCE: [INAUDIBLE].

PROFESSOR: Great. Next? Grab them while they're easy.

AUDIENCE: The product backlog is a list of features to be implemented to have the whole product done.

PROFESSOR: OK. And I saw someone.

AUDIENCE: [INAUDIBLE].

PROFESSOR: OK. You're already a repeat customer.

AUDIENCE: The sprint is the period of working on one installation of the product. It should be kept short, so we can [INAUDIBLE]. And each sprint should have [INAUDIBLE] that can be published, or that's running, so that you can only do incremental work.

PROFESSOR: OK. Somebody else. Somebody else. We got two meanings left. Grab the meetings. Team members? Don't make me do the whole lecture over again, because I don't want to give it, and I don't think you want to hear it.

AUDIENCE: The retrospective is the meeting at the end where you review your processes.

AUDIENCE: Team member is really easy.

PROFESSOR: Nobody's grabbed that.

AUDIENCE: Team member is everyone on that's on your team.

PROFESSOR: Well done! Do I have anyone that will take Scrum Master? You've already answered twice. Thank you for noble participation.

AUDIENCE: A Scrum Master is in charge of just making sure the Scrums go quickly, and everyone knows what they need to do, but he doesn't actually have to do anything. He's not actually in charge of planning anything. He's just in charge of making sure everything goes smoothly during [INAUDIBLE].

PROFESSOR: Yep. Yep. Coach is a great description for that. OK. Now I think we have not hit the sprint review meeting, but other than that, I think were pretty good. So the sprint review meeting is the meeting that you're going to get to do in class today.

No it's not. Sprint planning meeting. Ah! Sprint planning meeting is the one you're going to get to do in class today, where you go ahead, look at your product backlog, decide what you can get done, and make a sprint backlog. And then break that down into sprint task list. And don't think anybody gave sprint review. OK.

And sprint review is just at the end of your sprint, when you were evaluating your project for how close it came to what you had intended to make. And it is a chance to change and update the product backlog by the team, usually led by the product owner, since they're the person with the vision and a lot of strong opinions, to modify any changes to the plan that you guys

have realized now that you see what you've actually made. I'll stop torturing you now.

So backlogs, task lists, and tasks. First step is creating a sprint backlog. It's pretty basic. It's pretty simple.

Usually a product backlog ends up having lots of really big stories. Big fancy features. The things that you want your game to do when everything is in and it's all working at the end. By nature, these are usually big stories.

We want to have physics so that the game feels realistic. I want the player to be able to develop lots of different powers through the game. I want to be a long complicated story. I want there to be lots and lots of characters to interact with. I want a fully immersive world.

Never ever put that in a backlog. But I want a fully immersive world. These are all somewhere passed extra large, often. But in a product backlog, it's OK to go that big.

But when you're actually thinking about what you're going to do this particular sprint, and this particular iteration, you got to get a lot smaller than that. For example, I want a fully immersive world is not getting done in one sprint. Even if it's the only thing you're working on. So clearly you're going to have to break that down into smaller stories. And the place you do that is usually your sprint planning meetings.

So as you break those really big stories into smaller stories, you can leave some of them on the product backlog, because oh my gosh. We're not touching those yet. And then you can decide which of those pieces of the story are complete enough and small enough that you can get them done in the sprint. Be it one week, or two week, or a month. And again, the size of stories you're grabbing is really going to vary, based on how long your sprint is.

By definition, the sprint you guys have coming up is one week. Because your project is due next Monday. So once you've got your stories broken down that you've pulled out, and you've got your sprint backlog, then you can start taking a look at the tasks. Except that's not quite how it works. Because it's kind of easy to grab too many stories onto your sprint backlog at your first pass.

And when you actually are going through and estimating things, and you realize just how much work each of those stories take, you may find you have to put some of those stories back. You may even realize that some of the stories that were at the bottom of your priority list are actually more important than you thought because you've got other stories higher up that

depend on them. So while you're doing your planning, and while you're making your sprint backlog, and making your task list, all a little bit-- you're going to go forward two steps, back a step, forward three steps as you both reorder your backlogs, put stories back, take stories back off as you're trying to figure out how you get all those dependencies and squished into what is actually a really short amount of time.

You guys don't even have what a sprint team thinks of as a full week. You do not have a 40 hour work week ahead of you. You have, per person, probably a 10 to 12 hour work week. Because that's about the amount of time the class expects you to put into it. So small. Small stories. Small tasks.

OK. I just went through all of that, didn't I? How do you break things down? When you're breaking down your stories, think about what's the most important part of what your breaking down into? Because your stories have a because, or so I can, you can start thinking about what is most important part of that story. And which portions of those stories you can just dump.

You don't need a full physics engine to create realistic driving. Because, what the heck does realistic driving mean in this case any way? Does it mean that it accelerates, and it really feels like it's accelerating properly? Doesn't mean that when you handle curves, you can feel it spinning out on you? What does feel mean anyway?

Does it mean that the computer is playing really realistic sound effects for me? That I'm getting really cool sparks off of the wheels. I'm getting this view of the car spinning out. When you understand what you actually want, then you can create smaller stories that let you break it down and sign up just for that. You can also take a look at what the abilities of your team are.

And what you can actually do. If you have somebody who's really get at sound effects, maybe you really want to try to create a realistic aural environment. You're not going to worry so much about the physics. You're not going to worry so much about the sound effects. But boy, is it going to sound sharp. That's OK. That's working within your capabilities and your limitations to do your best to fulfill your overall vision.

Philip really already covered this pretty well. I will zip right through it. Tasks. Think of them in terms of going down to a half hour is probably a waste of your time if you're looking at a whole bunch of tasks that are a half hour long, that's a really small amount of time.

On the other hand, eight hours is probably the biggest you want to look at, given that that's most of a week. Ideally, you don't have one task that takes up your entire week's worth of work. If so, you're rolling some really big dice there. Because you're assuming that your estimate is pretty good. And especially the first time you start estimating, your estimate probably isn't that good.

It is more common for people to overestimate by up to twice as much. So, I can get this task done in eight hours often means I can get this task done in 16 hours. Think about that as you are tasking things out, and as you are creating your sprint task lists.

So I think I've mostly covered that. Finally, once you have this big pile of tasks on an Excel spreadsheet that no one actually ever wants to look at again, because they're kind of hard to pick out there, what do you do with them? I mentioned Scrum boards. I mentioned them being preferably there a physical Scrum board. That's really not realistic for us.

So you're going to want-- and I really strongly do recommend that you use-- a visual Scrum board solution. Two that I have used on projects that work well for different situations are Trello and Scrummy.com. Don't feel like you have to use either of these. If you find something better, use it and tell me about it. Because I want to know. Especially if it's free.

So let me show you very quickly-- let's see if I can pull them up. So here it is-- is that showing? Here's scrummy.com which I actually-- I think it's kind of the easiest and quickest one to use. This is always going to be more work for your Scrum Master.

So when you're thinking about tasks, set some time aside for your Scrum Master, who's probably going to create this board. It's going to be your responsibility to maintain it, and move things over, but somebody's going to have to type tasks into it. And it would work best if everyone could be relied upon to type in their tasks, but it rarely works that way, especially since you often have a whole bundle of tasks that haven't been assigned, at least at the beginning of the sprint, because people start picking them up as they discover that they've gotten stuff done.

So pretty much it works much like a Scrum board ought to. So you can create all your tasks. It divides things up naturally by stories. Although if you'd like to put it by user, you can.

They're two standard ways to organize a Scrum board. Either organizing all your tasks by the story it belongs to, or by the user who's working on. Feel free to do it how ever works well for

you. Scrummy, because it's free, and you don't want to pay for the upgraded version, won't let you change the names of your columns. But it's pretty hard from the end of the world.

So if my stories demonstrate so many of the [INAUDIBLE], I have recorded the URL, so I can get back here. That's done. I've made a new board. That's all done. And I confirmed the board was there.

I've assigned a task to Philip, so that there can be a task there. And it's a really pretty quick and easy-- ah! No, don't delete it. It's really pretty quick and easy to go ahead and edit things.

It changes color, so you can have different colors. Easy to tell apart. And it's really just sort of a click and grab. Click and drag, except that I don't have a mouse.

Sorry. I don't have a track counter, and so it takes these fancy two-fingered maneuver to actually move it. So I'm going to fail to move it over, but they're pretty easy just click and drag to the various things. To the various sides. So that's Scrummy, which has less functionality than Trello, but it's actually, in exchange for that, it's quick and easy. You type in a task, you type in a name, you're done.

Trello-- and I apologize about the resolution here. Let's see if I can close this. Trello let's you go ahead and create as many columns as you'd like. You can name them whatever you want. You can put in all the information you want.

If you want to track your estimations here, if you want to have a checklist of subtasks that are related to it, if you want to put on pretty little labels so that every user has a different color, or every story has a different color, as long as you only have five stories-- but on the other hand, you can also waste a lot of time organizing your project on Trello. And it will look really nice, and it will have all the data there, and you will have sucked all those hours that you could have been coding, or drawing, or actually making a game. So Trello can be really nice to use, but don't let it steal your project from you, if you choose to use it. I was told that there's a cool Scrum-free download extension. Runs in the Chrome browser that makes it work better for Scrum because it adjusts lists, and it does some previous set up for you.

I haven't tested that, so I don't know how it works. If you'd like to try it out, do. But don't do it this project, because you don't have time to waste mixing it around and fixing that. Do that on outside time, if you really want to is my advice to you. I don't want you to spend a whole lot of time getting the perfect system together. I want you to get a system that works well enough

and use it.

Yay. Back to the slideshow. So that is Scrum boards. Ah. Don't use PowerPoint either. Apparently it has to be difficult.

So the classic line up on a Scrum board is you've got all the tasks in your to-do pile. You've got the in-progress. And anything in-progress should be assigned to someone, should have a name attached to it. It can't be in-progress if someone isn't working on it.

And it should have a time estimate attached to it. Ideally, the time remaining. Although, it's good to keep track of how much time has actually gone by on it. The task cards are really good places to keep track of your estimates as they change. Then it's all in together in one place.

Once you get finished, it should not go over to done. It's going to testing. Someone who is not the person who did it should test it, and then move it over to done. That way you've got a second set of eyes on every piece of code and every feature, just confirming it's working.

Many Scrum boards also have another column called Blocked or On Fire. Which is to say this is a task that someone can't get done until someone else gets something out of my way. It's nice to have. It's not required.

And if you're working on a board that you're having a hard time fitting everything into one screen, Trello has always a problem for that with me, it's OK to leave it off. But it can be really helpful to have. That sort of is a Scrum board. And I do really recommend them. They are useful.

I know it feels like a lot of extra time often, when you're setting them up. But if your team is actually looking in there and checking it every day and updating, it's a really good way for the team to know where it's at without having to spend a lot of time talking about it. Because all the really basic stuff is there to be seen, which means you could freeze you up to talk about any real problems that are coming up.

OK. So that said, it's your turn. You've got a week left on your project, so let's try creating a sprint backlog and a task list from that backlog for what you're going to get done for Monday when you have your turn in. I'm going to give you-- like all meetings, they should be time boxed. So I was going to give you half an hour to do it. I will check in at half an hour, and if people are screaming oh no, we're nowhere near done, I will give you another 10 minutes or

15 minutes. Not actually a whole lot more time, but I'll give you a little more time.

This is probably doable in half an hour if you guys have been working on your project, and you've got your product backlogs in good shape. But it's the first time you're doing it, so it's always-- it's hard to know what the right amount of time is. After that, I will give you five or 10 minutes-- probably call it five minutes to talk about the process, and have someone come up and give a very short presentation of what the process was like.

What worked, what didn't, whether you liked it or not. It's OK to come up here and say bleh. I don't feel like this helped us at all. Just come up and tell us what happened, and how it works for your team.

[CHATTER]

PROFESSOR: OK. So [INAUDIBLE] meaning we used Scrummy to try to organize our product backlog into our sprint task list. We found Scrummy to be a little annoying, because it doesn't automatically update instantly, you have to refresh the page. And whatever happens last, we find, is what it shows as what's edited. So it shouldn't break, but it's not the most useful thing.

I think more useful would be to Google use the Google Doc of the product backlog. Section out the stuff that you do in that sprint, and then just update it there. We had it all color coded, and so it looks really nice. And that's much easier to look at and do than importing that into some other program, and then doing everything there and then going back. It just seems better to use Google.

PROFESSOR: OK. Thank you.

[APPLAUSE]

[INAUDIBLE].

AUDIENCE: OK, so we used Trello for our Scrum planning. And we liked Trello because you can just create cards that have lists of tasks and easily move them around between different columns. So when someone takes on a task they can move it to in-progress and assign their name to it. And it' really easy to look at and see what's going on at a glance. Trello does have a lot of features like checklists that you might not want to use a lot of, especially if you have really complex tasks.

.

You don't want to put all of those inside the checklist because that gets confusing. The thing that we had the most problems with was doing time estimates for the tasks, because no one in our group has used Phaser before. And so we have a very kind of shaky grasp on how long things are going to take, so our guesses were mostly random at this point in time. But we will update that as we get more experienced.

One thing that we did find was really helpful was when we were drawing up goals, we broke down all the features that the game needed into as fine detail as we could because we know what results we need to have to get the game to work. But we don't know Phaser yet, so we don't know exactly what the processes are going entail yet. But having like the list of things that need to go into the game gives us some guidance on what we should be developing towards.

PROFESSOR: Thank you.

[APPLAUSE]

Sparkling Redemption.

AUDIENCE: Sparkling Redemption. We basically just opened up Google Docs and made a whole bunch of Excel spreadsheets. I think we're kind of trying to use Scrum Do and see how that works, but we figured it'd be best to just kind of finish the thing first, and then take a look at that. So no thoughts there yet. I guess we basically just took a look at what we wanted to get done by Wednesday, and what we wanted to get done by the end of the week, and then broke that down into the tasks, and then worked with that.

I guess estimation was-- it wasn't too bad. Because some of us have worked with Phaser before, but there was also a lot of things where we realized oh, well, assuming x is finished first, then y would be really straightforward. But do we count x's time inside of y? Well maybe, but maybe not really. So that was as complicated as it got, I think.

PROFESSOR: Thank you.

[APPLAUSE]

Blundermans.

AUDIENCE: So for us, we started out with a fairly detailed plan. And we used Asana, which is multi-player to-do list. So, similar to Scrum.

But ultimately, our entire team more or less got sick over the weekend, and we didn't actually get much work done. So we can't speak to exactly how good are estimates would have been, or how useful our plans would be. So all the planning in the world doesn't help if your entire team is in bed.

PROFESSOR: Yep. Thank you.

[APPLAUSE]

Blind Aliens.

AUDIENCE: So our team used primarily Google Docs. What we did was we had our product backlog. And we were a little confused about the product backlog. So we made it way too detailed. But that really helped with this.

So we ended up copying and pasting our product backlog into our sprint task list, deleting everything we'd already finished. And then adding based on what we had done. And we found that because we were further along in our project, it was really easy to tell what needed more work, and to get the tasks more detailed and better estimates, just because we had done a lot of already.

[APPLAUSE]

AUDIENCE: So we started out planning on using Trello, but did the product backlog in Google Docs. And since it's sort of this information repeated twice, we just stopped using Trello, and ended up using Google Docs anyway. And that sort of helped us out now, because all we did was sort the things in Google Docs by priority.

So we looked at the things that were highest priority so we're doing this week, and then we just sort of thought about each one. How can we break this down? What are the specific tasks? You started working on this. What's left to do here? And then we put that into the sprint task list.

PROFESSOR: Thank you.

[APPLAUSE]

AUDIENCE: So we've been using Google Docs from the beginning. We decided to not use any other site, because we had already started working on Google Docs. And it's worked out pretty well for us. We were able to take our product backlog and take the highest priority that hadn't been completed yet, and just split them up into a finer detail task that we can use for the sprint task list.

And I think that that worked fine for us. It helped us really scope out how much time we had-- helped us reevaluate how much time we needed to complete the task that we had to complete. That is going to help us for planning for the rest of the week.

PROFESSOR: Thank you.

[APPLAUSE]

All right, so I have in fact, as we say, iterate on everything. I've iterated on this presentation. And been changing the order of it. So it may be I get confused a couple of times. I hope not. We'll see.

We're going to talk a little bit about high level and low level here. But as we said, in this class we want to iterate on as to the every process you might use. This is how you control your team. This is how your game works. Everything.

And this is actually not a bad philosophy to do everything in your life, that you might want to consider trying stuff, fixing what doesn't work, embracing what does work, and moving on. But this also applies to how you write your code. And I can talk about-- on a high level and a low level. The where you put your parentheses, and exactly what kinds of variables you use is a process. And you make decisions on that basis every moment that you're coding.

And it's a really good idea to think about what you did and how well it worked for you, or someone else. And this is the kind of thing that maybe in your early programming career you don't want to think about too much, because just getting the stupid thing to work in the first place is kind of hard. But I'm going to try to convince you that it's worth your time to get these habits going, so it's easier for you in the future to debug your code, or someone else's code.

So I want to talk a little bit, very briefly, about something I mentioned in the review on the

lecture on game engines, which is that all software sucks. And it sucks in different ways and all that, right? Well, the problem with this is is that yours does too. Your software is going to have bugs in it. And your job is to try to make that happen as little as possible.

You want your software to suck as little as possible. Now we also talked about one reason we use a game engine is just so that you write less code overall. You spend less time doing that. That's because everything that you do when you're running code is slow. And in particular, debugging is the slow part.

It's not too hard to write new code, but it's actually pretty hard to go into old code and fix what happens. At it varies, depending on the complexity of your task, but generally speaking, when you've written a bug, later, you have to spend the time to figure out that there was a bug, figure out where it was, and then you actually have to fix the stupid thing. And that is actually where most of your time is going to go, when you're trying to develop code.

So again, figure out the problem, and then change the code to fix your bugs. But I acclaim that most of the time, figuring out the problem is more of your time. Now this isn't true for the small bugs, when you're typing something at your computer, and it doesn't work when you compile it or whatever, and it runs. You go back and fix it right away.

Yeah, that part of debugging is not so much figuring out the problem. Because you just wrote it. It's fresh in your mind. You know what's going on there. But if it's been a couple of days or a couple of weeks, then actually the hard part and the slow part, is figuring out what went wrong.

And so one of the things that you need to think about when you're writing your code, is you want to make it as simple as possible so that you're less likely to write a bug in the first place. Or once you've written a bug, it sticks out. It's easier to see. And the important thing to remember here is that all this code you have written is not as easy to read as you think it is.

And the right way to do this is to make it as easy as possible. And the way I like to think about it, at least today, is to make it require as little knowledge as possible to figure out what's going on in your code. If you use some really fancy language features, that's really cool, and it feel like you're really smart when you're doing it. But a week later you back to look at that code to try to figure out, I don't even know what I did there. And heaven help you if someone else tries to read your code.

So ideally, you don't want to go too obscure into the computer language fancy features. You

don't want to go too much into your subject matter. But there are exceptions to this. If for example, you work on a very specialized piece of software where everyone has to have an arrow engineering astro degree to actually work on the project, you can assume some level of competency there.

But generally speaking, in game development, and most application development, you want to assume was little knowledge as possible, so that when you want to debug your code, you can look at the little part of the code you're at right now, and see if you can fit what's going on there. If possible, you want to assume no knowledge on the part of who was reading your code. And I'm going to talk a little bit about why that is.

This is a useful thing to think about for user interfaces or really any problem solving. Humans can't hold that much in their brains at once. And if you're trying to write code that requires you to think about 10 different things at once, you're in trouble, because humans really don't do that.

Now generally we get around this problem with what's called chunking. So when you started with math, you thought about the number five wasn't even a numeral to you. It was five objects. But you couldn't really wrap your brain around 12, because you can't look at 12 objects and think oh, I understand what that means.

But as soon as you practice with your numbers more, you begin to think of number 12 as a unit. It's not 12 things, it's one thing. It's the concept of 12, and that's really useful to you. And then you started working through algebra and you got variables. And now you're using calculus, and all the other cool things.

You talk about vectors in math. A vector represents three numbers, or a direction or position in space, but you don't think about all three numbers. And you certainly don't think that 35 is one of those numbers, and that means that there are 35 things. Your brain isn't wasting time on that. Instead your brain is thinking, it's a vector, it's a point space. That's all I need to know.

So your brain is ignoring a lot of information by clumping a bunch of information into one spot. I often use phone numbers as an example for this. And it's an increasingly poor thing to use, because I bet most of you don't memorize phone numbers, because why would you?

But when they started [INAUDIBLE] phone numbers, there are three parts to a phone number. First part is an area code, the second part is a sub area code. The third part is the four digits

that are more unique, right. It's really hard to remember those 10 numbers.

But actually, once you learn the area code, the area code is three numbers. You think of it as one number. Boston is mostly 617. The second set of numbers is also three numbers. You clump that as well.

And the last four numbers, yeah, you have to memorize those. But basically instead of remembering 10 things, you have to remember six things. The four numbers and then the two prefixes.

And another problem with coding is that as you keep doing it, you're making decisions constantly in your coding. What's the right way to do this? What's the right variable you're going to use? Et cetera, et cetera. And as you do this, you get tired. Humans only have so much decision making in them before they need a break.

So a couple of things we can do is we want to reduce the amount of things you have to remember while you're looking at your code, and we want to reduce the number of decisions you're going to have to make. And to some extent, although I'm not really clear on a psychological basis for this, I'm going to make a crazy claim that figuring something out and deciding what this thing must mean is a decision.

So even if you're just trying to figure out something, that also is a burden on your brain. And what you want to do is keep your brain alert and hopping as much as you can, for as long as possible. And to do that again, we want to simplify your code if you can.

So to go back here, simplicity-- when I say simplicity, I mean you want fewer things to think about. Should be easy to read the code, so you don't have to memorize what the code is doing. You can just quickly glance and get a reminder of what it's doing. You have fewer bugs because there's just fewer moving parts. Fewer things to go wrong.

I'm going to say this hopefully a couple of times, but a short variable name is not simpler, it's just shorter. If you start thinking about what you spend time doing when you're writing code, and trying to get something done, what takes the most time? I'm willing to bet it's probably going to be debugging part, but maybe you think it's going to be getting the stupid thing to compile in the first place or whatever, but I'm willing to bet almost none of you think that your typing speed is the thing slowing down your coding. If only I could type 80 characters a minute, I could double my coding speed.

It doesn't work that way. You do a lot of thinking, and just a little bit of typing. So if your variable name is 12 characters long instead of four characters long, that is not cost you time. It just costing you eight keystrokes.

And in a modern development environment, you've got auto complete. You type the first four characters, you hit control space, and you get all 12, hey. You can't even claim that you've saved eight characters, you've only saved six.

So anyway, I'm going to talk about that a lot, because I believe that the longer variable names are really useful cues to remind you what you're doing. And one of my favorite things to talk about is write once, read never code. Who here has used Perl? Anyone?

Have you tried to read other people's Perl code? Yeah. Why would you try? Have you tried reading your own Perl code? He goes, no. That's right. You just write it again. Absolutely.

So, Perl if you're not familiar with it is this cool scripting language that's not used as much now, but it was once the open source darling. PHIP is kind of modern equivalent of, my god, don't try to read that code. Perl had the nice feature where pretty much every symbol on your keyboard has a meaning. And if you want to do something, there's probably three different ways to do it, with three different obscure characters on your keyboard. Worse, each of those characters has a different meaning, depending on what you're using it on.

But this is an example of APL code, which a stands for A Programming Language developed in the late '60s, early '70s. And it's really painful because in APL not only does every character on the keyboard have meaning, they've minted special keyboards with an extra three characters per key, so that you could have more characters. And this is awesome if you think of writing code as if piece of paper with your pencil and you're doing math, and you're writing fancy integral symbols, and this and that, sure.

It's handy in math to chunk those concepts this way. But for coding, generally speaking, we're stuck on our alphanumeric keyboard with a couple of symbols. And I advise you to use as few of those symbols as possible. Try to stick with your language if you can. All right.

Who here thinks that you write enough comments in your code? Wow! Some of you do. But almost none of you-- maybe a quarter of you are like, yeah, I'm proud of my comments. It's cool to be proud of your comments if you're writing them. And if you're not writing them, don't be so proud of them.

But sometimes I see comments that are just duplicating what the code is saying. If your if statement says, if x is less than 5-- don't put a comment up there that says if x is less than 5 do something.

That's a waste of time. I can see that in your code. But I want to know why you care if x is less than 5. That's the kind of comment I would like to see in your code.

And we'll get to this a little bit later, but I also talk about longer variable names, longer function names. Those are useful ways to dodge writing comments if they are correct. But really quite frankly, if you're trying to figure out gosh, I don't know if I should comment this or not, write one. It's not going to hurt you.

So here's an example. These comments don't mimic the code, but what they do do is they tell the story of what the code is actually doing. Random aside, font is actually very important with your editor. Make sure you use a fixed width font because pretty much all programmers do. And sometimes you use spacing to have everything line up nicely, and you want to have that be crystal clear to anyone else using your code.

But there's another way to this. So this is code. You can figure it out. Go ahead and figure it out. There's no rocket science going on here.

We're using the Pythagorean Theorem to calculate a distance, and make sure that if it's small, we return 50. There's nothing hard here. A programmer can figure this out, but the programmer shouldn't have to figure this out. And that's my argument.

If you takes you-- I don't know, maybe you're great and it takes you two seconds to figure that out. Maybe it takes you 30 seconds to figure that out. It doesn't really matter. It should take you less than that if the code has been written correctly.

So for example, I'm just looking at the 900 to 30 times 30. Because why? The compiler doesn't care. It's a little bit easier to read. You can say if the distance is less than 30, as opposed to, if the distance squared is less than 900.

AUDIENCE: And it's also the bug. This also has a bug.

PROFESSOR: This does have a bug. Good catch. There's a bug here. This code doesn't work. Because we're saying y0 minus 1 instead of y0 minus y1.

So there's a better way of doing that. This is even easier to read. We are using a vector instead. The vector class has some internal knowledge of what it needs to attract vectors, and what the length of a vector is. Not only is this less bug prone, but it's just easier to read. And that bug that you caught there can't happen. Now this code is a little bit cheating, because I used v for the name.

So you can guess that they're vectors, but that's fine. Here's comments. Basically the same code. I'm saying, hey, here's some context.

This is a lot more code. In some ways it's harder read. In some ways it's easier to read. That's one of the things that you're going to have to figure out for yourselves. What coding styles make sense for you?

Obviously, simpler is a subjective question. But I kind of like the style of code because what it does is it means that even if you don't comment your code, your code still is commented. Because you can read this and know what's going on. More importantly-- and this is the one I've change since the last year I gave this presentation. I thought that the comments saying why, and then the code that did it was probably the better way to go. But a friend of mine pretty quickly convinced me that actually as codes get maintained for a year or so, you might change the code and forget to change the comment.

And this actually happens a lot in real software projects. With the kind of commenting or the kind of self documenting code that I'm using here, that can't happen. If you change the code, you change the comment. It's innate. They're intrinsincally linked because they are the same thing. If you can do this, it actually helps a lot.

So I'm going to go through a variety of tips and tricks and things that might make things a little bit easier to read. But before I do that I want to talk to you a little bit about to why and who it is you're helping. So a while back, I was complaining to my grandmother that wow, you know, last year I made a bunch of stupid decisions. I didn't do this right, I didn't do that right. I'm so much smarter now this year than I was last year. It's amazing.

My grandma said, yeah, I feel that way too. All the time. She was 84. And I thought oh no, when I'm 83 I'm still going to be a moron.

So one of the people that you're writing code for-- one of the people who will be reading your code is not other people on your team, although they're very important, but it's you. You will

read your code later. Have any of you looked back at code you wrote six months ago yet? You're early in your career, so not many of you have.

But I'm willing to bet when you look back at the code you wrote six months ago you think yourself a couple things. One, wow, I was an idiot. And two, you're going to think to yourself, I could write this better now. And three, you're probably also thinking even if the code works, the algorithm is solid, this is decent, it could be more readable.

Maybe you're not thinking that yet. I want you to start thinking that because you're going to forget the code that you write. And again, when you're just starting out, that seems impossible. That you can't forget code that you just wrote. But I guarantee you after you've been writing code for five years or more, a lot, you're going to discover that you don't remember what you wrote yesterday, or last hour, as well as now when you're just learning and getting deep into it, you remember a lot more than you're going to.

But keep in mind that experience. And you go back to your code and you don't remember what you wrote, that's the experience your teammates have every time they look at your code. They didn't write it. They cannot remember it. Your code is a communication opportunity to talk your team. To let them know what it is that your code does.

So one way to make code more readable, shorter functions. You hear this sometimes, all of us are eventually write that 500-line procedure that does this, then that, then the next thing to the next thing. We all do it, because it's the easy way to do it. You think, I'm going to do this thing, so I do step 1, step 2, and you march on through. It's reasonable. It's a way to get things done, especially when your work is in progress.

But if you ever want to go back and fix that code, and now you have to figure out what pretty much all 500 lines are doing. And that's kind of dangerous. The easiest thing I can recommend there is really if it is a step 1, step 2, step 3 kind of thing, why not divide steps and sub functions? The subfunction may only get called once, but that's OK. The reason this helps you is twofold.

One, it gives you a nice name. For this section of the code is where I am going to tell all the planets to do build spaceships this turn. I make that a function call. I don't have to worry about what's inside that function if I'm trying to debug a war mechanic. Because probably, the part where all planets are building spaceships isn't relevant to the gigantic space battle somewhere else.

But 2, when you subfunction this, inside that subfunction you are guaranteed that the local variables there are unimportant for everything in the big function. The scope is such that they can't affect the big functions. They are local variables. You don't to worry about that. So this is a tiny couple little class of bug that you can kind of dodge by something as simple as subdividing a function.

Two-- and this is the other important thing-- you've got this function, you strip it down so hopefully you can see it on a page in your editor, and you can read in English kind of, what it's doing. You see here the five steps. There they are. It's a little bit easier. And that's kind of a simple, straightforward example, but it's that way of thinking again, that I want you to think about.

If you ask five programmers, what's the right thing to do? They're all going to give different answers. And that's fine. But the thing I want you start doing is thinking about what is it that you think is the right thing to do? Why would you consider doing it this way versus that way? And if you just write the code and don't think about it, you're not going to get better as quickly as you will if you think about why you wrote the code that way, and how it's working for you.

Here's another fine example. Making wrong code look wrong. This is perfectly valid code unless I make my variable names a little more descriptive. And then you discover that that's wrong code. That code is terrible.

Degrees times radians is a classic mistake. Centimeters and inches has destroyed Mars rovers. Milliseconds, kilometers, and miles per hour. That's probably not going to work so well. And these are actually-- you say, I never do that.

But again, I kid you not, we have lost billions of dollars to unit conversions because people didn't do this. It's going to be annoying to you, when you're typing it in, perhaps. But I guarantee you, you'll be safer if you do it.

Again, random stuff. You've probably heard all this. And we you're doing games, you get a chance to really experience it firsthand. Try to avoid sticking numbers deep in your code. No magic numbers. You want to stick it in a variable somewhere. This has a couple good features.

One of the biggest is, you've got someone else on your team can easily look up that number

and change it. They don't have to find it in your code. They can just look at your list of variables somewhere, and figure it out. Immunity, it's even more awesome, because you can just go to the editor and change it around.

Also, you can change it globally everywhere. If the number 50 is magical, great. But call it something else so that you can change it to 51 when you're tweaking your game plan.

I've talked with this before, and I will talk about it again if I get half the chance. You want longer variable names that say exactly what they are. You don't want to have to guess. They should be pronouncable. And this isn't true for everyone, but most people, will tend to in their mind, to say a variable name out loud. And so if it's a bunch of random consonants that don't go anywhere, it's a little bit harder to read.

If they look similar, you don't want to have a typo kill your project. Spell them correctly. And this is actually really important, especially in interpreted language that just doesn't do anything except running at run time. In Python, for example, I believe, if you declare a variable that's never been seen before, it probably has a valid value.

No. OK, good. I'm glad that's true. JavaScript. That's what I'm thinking. But JavaScript-- be very careful, especially if you're using Phaser JavaScript.

If you mistype the name, it's 0. That's no good. More and more modern languages are rejecting that. But the other thing is just it's easier for your teammates to know how to spell it. If you all agree on a spelling.

And if you're all going to agree on a spelling, you might as well agree on the correct one. And then reusing a variable name is also a little bit dangerous sometimes. If you're exiting a loop, and you've maybe failed to initialize the thing properly. You reuse a variable name elsewhere. You got this weird value coming in, and that's a bug waiting to happen. And it'll be kind of hard to find.

Function Names. All the same rules basically apply. You want to have it be distinctive, spelled correctly, you want them to say what they do. It's tempting sometimes, to make your function name really, really short. Again, that doesn't speed you up.

If a function is longer, you get a chance to read what it actually does. But the other thing about function names-- and this is even more true than it is for other parts of coding-- is that your function name tells your other programmers what the heck you just did. I wrote this function.

Well, what does it do? Well, read the function name.

It's right there. That's ideal. You can't always pull that off. And when your function does something more complicated than you can describe in four, maybe five words, then you need a lot of comments to describe what that function is doing. Ideally, your function should be simpler than that, if you can manage it.

By the way, I took to mention variable names and links, and a lot of you are probably thinking right now, yeah, a longer variable name sounds like a good idea in principle, but I don't want to type that. It would slow me down to read it. There's actually been research done on this topic. 8 to 20 letters on average, is often a marker for good code that works with fewer bugs.

So I'm not talking a little bit longer. If you can push it, push it harder to be as long as you can stand. And then you'll be a little bit happier with that.

If you are coding in VI or Punch Card still, you could complain about a 20-letter variable name, but you aren't. You're using a modern development environment, with auto complete, I hope. Variable scope and names. Ideally, and this is a thing we strive to do, and don't always succeed at, you should name your variables so that you can tell by looking at them where they come from.

By which I mean is it a global variable? Is it a local variable? Is it an argument to the function? That kind of thing. You can sometimes do this by prefixing, or post fixing, or camel casing, or not camel casing. That kind of thing.

It doesn't matter which system you use, as long as you're mostly consistent. Trying to be consistent among different programmers is often difficult, because everyone will horribly disagree as to what the correct answer is. But if you can try to sort of remove that from your brains and say, actually it doesn't matter what it is this time, I just need to know what you're doing. That's the important bit.

Parallel rays. These are free arrays. And number 3 in each array is related. This is something that you're going to find yourself very tempted to do a lot, especially in a game. And I recommend against it.

Even if you only have two parallel arrays, I recommend making a container classic that contains both pieces of data, and have them stick in one array. Just because you're going to

get bit sometime by having these arrays get out of synch. You just want to avoid that if you possibly can.

Order of Operations. There's an expression up there, which is probably legal code in most languages. And we are taught in algebra that we should memorize the order of operation so we know what gets multiplied when, and all this. But in computer language we've got a lot more, we've got incrementation, we've got raise to the power of, we've got modulo, we've got all this crazy stuff. And my argument is don't.

Why waste brain space on memorizing the order of operations for the eight different kinds operations you have, when you can just use parentheses and remove all doubt. Plus, you could also even divide that-- I'd divide that up into a bunch of statements, actually. There's no reason to keep it like that. There's no way that someone looks at that expression and thinks, this is the most intuitive way to understand this.

If you're writing on paper in math class, you might find that because you can use two dimensions, you actually can make your equation more intuitive. But in code, you mostly only have the one dimension. And so it's kind of hard to really make your math intuitive. So I would put extra effort into just split it up.

I also mentioned this last time, I'm going to mention it again. It's very important. Warnings should always be treated as errors. I can could repeat it again, but basically every new warning is a hint something might be wrong. And if you let your build be cluttered warnings, you will not notice when the important warning pops up. And you can lose hours to that, just because the build wasn't at zero warnings.

Backwards Conditionals. Which of these looks wrong? The second one looks wrong. I agree.

The second one totally looks wrong, and yet it's useful. Because what happens if you mistype the equal to equal sign? If you mistype the equal to equal sign, that second one doesn't compile. Depends on your language of course. But generally speaking, it's a handy little trick that I find useful, and everyone else I know thinks is terrible. So don't necessarily use it, but try to find tricks like this that cause to be the case, that when you mistyped your code, you get a compile error instead of a bug in your code.

AUDIENCE: Do you think what the signal equals might be a client warning?

PROFESSOR: It will be a warning, absolutely. Which is an even better reason to not to go down to zero

warnings, right? Doesn't help you if it's a warning that you just let go by. And it should be a warning. I'm glad they added that. When I start programing, it wasn't a warning. Wasn't that wonderful?

I mentioned splitting up the math. I think that that first thing is, you can read it. You can figure it out. I think that the second thing might be a little bit more comprehensible. Again, it depends on who you are.

This is one that I'm emphatic about. I mentioned order of operations before. Bar minus minus. That's fancy. In some languages, and I assume most of the modern ones these days, bar minus minus means do the math, and then decrement bar by 1. If I'd said minus minus bar, I'd have meant decrement bar by 1, then do the math.

And this is pretty cool. It's this nice little language trick. But if you don't happen to know what that means, you just confuse your fellow programmer with your language trick. That second thing is perfectly clear. You know exactly what happens. Do

The math, then decrement the variable. And if you really want to get pedantic, I guess you could say bar equals bar minus 1. But you have to draw the line somewhere.

I'm a fan of Booleans being asking a question. Like the Boolean purple probably means is purple. The Boolean hungry probably would mean are you hungry. But the Boolean desk might not mean anything.

But in any event, try to make your Boolean into a question that has an unambiguously yes, no answer. Which way is true? Which way is false? Status is a particularly bad one, because I don't know what that means. Is status true good? Is status true mean that something bad happened? I don't know without some help.

You've probably heard this before, mysterious constant, don't do it. Ideally, if you can define it in your language, you get an error. When you mistype it, and that's much better. Mysterious string constants is particularly the worst. And you're going to be very tempted to do it in JavaScript. But you should probably have a defined of some kind at the top, so you don't get that problem.

This is a language construct which is very handy, but I recommend using it sparingly. If you are asking a question-- if you want to set a variable to a value, it's often very convenient to do

this syntax. But really, you don't want to ever nest these. If you nest them it becomes unreadable gobbledygook. And if your expressions become complicated, it also becomes unreadable gobbledygook.

Again, I'm not saying you can't figure it out, because I'm confident that you can. But why do you want your future you to have to spend 30 seconds to a minute, figuring out what the heck you were saying instead of two seconds figuring out what you were saying?

Goto. We are told that goto is evil all the time, always. I claim it's not always evil, but you should be careful when you use it. It's particularly useful when you're exiting a horrible set of nested if statements. But the main reason goto is evil, is because of a thing called proximity.

Which is that you should use things near each other in the code. If I declare a variable, and it's two screen pages up from when I use it, that's kind of a difficult thing to figure out. That's one reason to keep your functions short.

But instead, if I can clump it together, I'm less likely to make simple little errors. Right here, I'm doing the same thing to x and y, and I'm interleaving it. Because I'm thinking of it as, I want to prepare them, I want to calculate it, I want to print it. But it's a little bit easier, I would say, to debug and think about it, if you treat each one separately.

And if you get a lot of these, write a loop or something. There's no reason to actually type it all out. All I'm saying is keep it simple. As simple as you possibly can, so that later, when you go back to it, you can figure it out quickly.

You're going to very tempted to make complicated systems in your prototype games, even though you're game spec is changing, probably daily. So you might want to be a little bit cautious about that. Rather than making a complicated system, solve the problem you have today, and then maybe two days from now, when you know for sure you're actually doing that, you might want to put more effort into it.

This is advice that's not good for all projects, but it is good for very short projects. It's tempting to create a complicated system that will do everything you need. And it's tempting to create one solution that will solve all of your woes, but you might very well be borrowing trouble and solving a problem that you're never going to have.

So given that you're probably talking about fast iteration. Your game is changing all the time. Don't-- and this is counter to my architectural design thoughts-- but don't design too much,

because you're going to change it all anyway.

Other things that cause bugs are recursion. Recursion is awesome. Recursion is fun. Recursion is really hard to debug.

Try debugging someone else's recursion, and you'll be very sad. To try debugging two things that call each, other sadder. And it looks really cool, but I recommend don't do it.

Similarly, optimizing. Wait. You think you know what slow and what's fast. Chances are, you don't. If your development system has a nice profile, it'll tell you exactly where the bottlenecks are, exactly why your code is slow.

Wait for that, and fix the actual problems. Don't waste time chasing ghosts. Particularly this happens a lot when you're trying to do some code, and you're thinking oh, this is really important. It ends up being called once per second.

Well, once per second in computer time is never. So don't bother optimizing that. But the nice thing here is that a profile can tell you what you need to actually fix.

I've mentioned fixed treat warnings as errors before. I'll mention it again, probably. But similarly, you want to fix a bug as soon as you find it, if it's in your head.

Don't think oh, that bug is easy to fix, I'll get to before the end of the sprint. Because you might not remember it four days from now as clearly as you do right now. Also, if you've just written the code, it's much, much easier for you to actually fix it, because it's all much more fresh in your mind.

Strongly consider not fixing bugs that don't matter. And this is a tough one because I just contradicted myself by saying fix them right away. And I'm telling you sometimes don't fix them. But basically, if fixing the bug is going to cost a lot time, and it's not going to improve your game enough, then consider leaving it. That's going to be a thing you'll have to talk about with other people your team. But hey.

So for example, if you've got every j in your text renders green for some reason. And to fix that, you need to go into the font library and do some crazy code. Well don't. Claim it's a feature. J's are green.

Obviously, if you're doing Microsoft Office, you can't do that. But we aren't. We're doing

games. So we can cheat like that. Use a debugger.

If you do not yet know how to use a debugger, take advantage of this class to learn how to use a debugger. I can't really say more about that. Do it. When you're also try to figure out a bug, talk to a teammate. Even if your teammate has no clue about your part of the code, the act of putting into thoughts and words, this is what my system is doing, will, a good third of the time, cause you to realize where the bug is.

It's kind of silly, but it totally works. So if someone comes to you on your team and says hey, I need help. You're like cool. I'll listen to your problem. And they talk to for 15 minutes, and then say, I got it, and walk away, don't feel useless.

You just saved them an hour. And it cost you 15 minutes. And that's a bargain. And don't feel embarrassed about it, if you're the person talking about your bug. It's great.

They may ask you questions that are irrelevant, and you're like, no, no. That's OK too. You have to explain it. That orders it in your head, and then you can solve the problem.

And then of course, take a walk. I have solved more bugs going to get my lunch, or in the shower in the morning, or whatever, than I have at my desk, I would say. At least more tough ones. Binary search is a thing that's kind of hard to describe.

But rather than necessarily saying, this could be the problem, I'll test that. This could be the problem, I'll test it. If you can do a test that says, well, roughly half of the potential problems that could cause this bug are over here, and half of them are over here, and I can do one test that won't tell me where the bug is, but will eliminate this entire tree, that's a good test to do. You can quickly narrow in on where your bug actually is.

And this one happens a lot too, with beginner programmers. If your bug goes away, it probably hasn't gone away. It's just hiding now. So be sure you know why the bug went away.

If it's the green j's, you might not need to worry about it. But even then, you might want to think about it. Because you thought you knew what the green j's were, and they went away, and you didn't fix it, but who knows what's going on here. It's worth putting a little bit of time in to make sure that it isn't some serious bug hiding under the covers.

I'm not going into source control again. We've already done a little bit of that. You need to do source control. If it all possible, everyone on the team should be able to just get the code and

do a build. That should be as hard is it is.

And on that realm, daily builds are awesome. If you can afford to, have some on your team whose job it is, once a day, midnight, pick a time, do a get, a fresh get, do a build, make sure everything works. This sort of forces all the programmers on the team to sort of be more careful about making sure their code works. And also if something gets checked and it breaks everything horribly, you know about it right away, not three days later.

There are automated tools that will do this for you. At least they'll check the compilations step. That might not check that makes sure everything is still working, but, you know. It gets there a lot.

Coding Standards. Why we use them. We like tidy code. And obviously, the braces go in one place or another.

Those are terrible reasons to have coding standards. Instead, these are the reasons that I like a little bit more. Which is my code-- don't touch my code. Well, you're on the same team.

I'm going to touch your code if I need to, and it's midnight, and you're asleep and I need to fix the big. That's the way it is. You shouldn't be mad about that. We're trying to fix the game. The fact that that you wrote the first pass in the code doesn't really matter.

And then, of course decision fatigue is a thing I mentioned earlier. We talk about what variable names to use, and what algorithm to use, or this or that. We got plenty of decisions to make. Where you put your curly braces should not be something you waste time stressing about. If you have a coding standard, use that.

If you don't have a coding standard, be consistent. If you're modifying someone else's code, stick with their coding standard. Because if nothing else, when you do the check in, it will look like the entire file has changed just because you changed the indentation strategy. And that's not very useful if I want to know what changes you made.

I'd much rather have a minimal set of changes that I can look through and see what the actual meaningful changes you made were. And of course, easy to read. Easy to debug.

A lot of things I've been talking about are things you can read a lot more about. There are people that have written serious books on the topic of making your code as easy to read as possible. Steve McConnell in particular, is at this point-- his books are older, but they're still

perfectly valid. Because the science of writing code has not really changed much.

Joel Spolsky also has a much easier to read set of blog posts that talk about this kind of thing. It's much more accessible and it's free. And he's the person who-- his company is behind Trello. Which kind of indicates that his method of thinking kind of matches with our method of thinking. There's a lot more things to think about in these realms, in terms of ways to make your code easier.

But again, I want to go back to the core concept we talked about before. You're going to have a couple projects during the semester where you're going to see a bunch of different coding styles. Yours and other people's. Yours might even change over the course of the semester, and that's good.

But you should include, perhaps as part of your post mortem process, what kinds of things did you like in the code you saw. What can you pull into your coding style that is going to make your code easier to read, or easier to debug? And maybe you can try to influence other people to write prettier code.

You're not always going to agree what pretty code is. Try to let that go when it happens. The important thing is you should think about what you're seeing, so that after your project is over, you are better programmer for it.

And I think at this point, unless you have any questions, we'll go on to have this project time with the remaining 15 minutes.

[CHATTER]