The following content is provided under a Creative Commons license. Your support will help MIT OpenCourseWare continue to offer high quality educational resources for free. To make a donation or to view additional materials from hundreds of MIT courses, visit MIT OpenCourseWare at ocw.mit.edu.

CORY: I guess I'll start now since I think this might be the crowd since most people are probably sleeping after the sprint submission deadline. Were most people up at 6:00 AM? Most? Most if not all. Alright, so, my name is Cory. I'm from the winning team last year and what I really want to do today is kind of give a complete information dump. So this lecture will almost seem like a stream of conscious, but these are the things that throughout all of our years at battle code are the things we really wish we'd got in lecture but never actually got.

So my goal is hopefully to give the most relevant-- hopefully like the most applicable lecture to the nitty gritty details of the day to day. Things that you're doing battle code in order to make your bot better and try to improve your performance against the other teams. So whatever competitive edge you can glean from this hopefully propel yourself to the top of the ladder.

So just to give a little bit of background about myself, I've been in three teams since 2010. The first two years where we're kind of meh. But in 2012 we were finally able to win the entire thing with my teammates, Yanping, Haitao and Justin. And in each year was quite different because of the people you work with. So what I want to do is give a lot about our 2012 experiences, and kind of explain the various choices we did.

The tricks and techniques we did in order to give ourselves an edge over the other team. And I also want to give special thanks to these four teams in particular, Gunface, My Archon Died Bellmanfording the Stream, Drunkasaurus and Toothless. Just know that battle code is not done in a vacuum, and you learned from all the other teams. And so these teams were particularly helpful that our team learned from, that our team collaborated with.

That we were able to share code bases and learn from each other, et cetera. So just to preface everything, we're the one's, fun gamers, is the one responsible for the spec this year. So if any of you guys have problems with it, you can't really blame Max or Stephen, you have to blame us. So any cries for like nerfing, OP, whatever, it comes to us. We made a lot of huge changes this year just because we wanted to mix up the game a little bit.

And also we wanted to make it a little bit easier for people to start off code bases and get running players. I don't know how many of you guys played in 2010 or even 2011, but the amount of code required to get even a simple bot up and running was enormous. So in 2010, because you couldn't actually attack a unit until you had equipped a gun on a robot, it took almost one to two days worth of code base before you could even get a robot moving and attacking simultaneously.

So what we did a lot was simplify a huge number of things to make it a lot more higher level strategic control. So no walls, attacking is completely automatic, we removed directionality. I don't know how many of you guys played last year, but this one's pretty big. There's no robot directionality. We added in shared vision, global broadcasting, and a known map. So do most people enjoy these changes?

Or at least people who have played it in previous years? Yeah? We're really excited, at least, about, in particular, having the map completely known and the global broadcasting, because it allows an unprecedented level of robot communication this year. So you can now, instead of having a bunch of almost completely autonomous robots with very limited broadcasting, you can have coordinate attacks.

You know the map. You can design your strategy based on the map. Rather than wandering around for 100 to 1000 rounds, not knowing where to go, getting stuck behind voids, et cetera. The no walls is also very interesting. What we found in previous years is contestants will sometimes, if the math is really stupid or really retarded, what'll end up happening is you just watch for 3,000 rounds both people stuck in their base because of this really interesting concave of walls that prevents the people from getting out and actually engaging each other.

So with no walls you can actually basically control the battlefield. You can defuse mines, you can add your own walls, you can prevent your enemies from attacking. You can create kind of-- you can bend the battlefield to your will, which we thought was really exciting, especially because you're no longer completely screwed over if the map wasn't what you expected.

So to begin, I want to emphasize that battle code is a trade off of time. So this is what battle code ultimately comes down to. You have three to four weeks to basically implement an entire bot. And in these three to four weeks, you have to decide what is most important. If you spend your time on the wrong thing you will not win. You have to spend your time where it matters. And, in particular, I want to say that results to effort vary very highly between what you choose to spend your time on.

So one of the most important things to spend your time on is attack micro. If you can win every engagement you will win the game. It doesn't matter if they're higher level strategy, if you always win the engagement you will beat their army straight up, no matter what. OK? Next is swarm code. If you're swarm moves together you'll usually win the engagement. If you have better swarm code your attack micro can be a little bit off.

But you'll still end up engaging with larger numbers and winning the fight. And winning fights wins games. The navigation code that allows you to actually get to the enemy swarm and defeat them. And then there is some high level strategy that you can't make completely wrong choices. But, once again, if you have good attack micro, you can make questionable high level strategies but still win every engagement.

So the way I'm going to structure this lecture is I'll have various tags. There are certain things that apply to novices. There are certain things that apply to advanced players. And then there are certain things that apply to expert players. This also comes down to a trade off of time. If you have four people, maybe you want to consider doing some of the more advanced tricks and tactics.

But if you only have say, if you're working solo or you only have two people, you want to spend your time where it matters. So don't spend time implementing things that give you very tiny marginal advantages when you could, say, be focusing on your attack micro. And win all the battles rather than saving maybe like one or two bite codes per iteration.

So first off, to do well in battle code you really have to know the spec. And I know we've been really bad at keeping the specs up to date, there've been typos. But hopefully by now everything has mostly been standardized and corrected, and all the typos. And then you guys didn't really like our the diffusion/defusion joke. So we changed the word defusion to diffusion so that you guys didn't think it was a typo and that we were stupid.

So quick quiz, what's the base spawn rate? Someone call it out. Ten. Alright, nice. What's the cost for three suppliers? Cumulative? What was that? I think I heard it over there. 60. Right. Because it starts off with 10 for the first supplier, then the cost is the additive. So 20 and then 30 and to 60, 60 the total cost for three suppliers. What is the spawn rate with three suppliers?

It's 8. So total cost of three suppliers is-- the spawn rate of three suppliers is 8 per round because it is kind of a curve. The first supplier will get you down to 9. The second supplier will get you down to eight. And the third supplier is actually the first supplier that doesn't bring you down another point because it's rounded to the nearest integer. Even. So in terms of unit spawn, at out what round does the first supplier break even?

So these are all like critical numbers that if you know you can kind of gain the spec to your advantage. Close. The first supplier breaks even at what round? If I get one supplier, I have one turn spawn reduction, which means I break even, in terms of units, at what round? I sacrifice one unit to get the supplier, so I'll break even when I get back that extra unit, which is at 90 rounds.

So the first supplier breaks even at 90 rounds. So these are key numbers. They maybe tweaked in the future, but if you always have a sense of all these numbers, you can kind of make good high level decisions about what you should be building and what you shouldn't be building at the beginning of the game. So the beginning of the game is very key.

What we did was you have an excess of energy. So you have a choice, like in Max's lecture yesterday, whether to build generators or suppliers. And if you know roughly at what time these things break even, or at what time you gain that the marginal advantage, you can decide when to attack or when not to attack. So know the specs, that's very key.

Second, I want to say testing is probably the most important thing, second from spending your time in the correct place. Because you don't actually know if you're spending your time in the correct place unless you're actually testing the code. So as a high level, you want to write multiple distinct bots. So I know a lot of teams maybe just have a single bot lineage. But our team we wrote like 10 to 12 different bots, each one with different strategies.

And sometimes we wrote bots completely from scratch when we thought we had a really cool strategy. So last year there were flying units. So we thought, well it would be kind of cool maybe if we could have units that only-- or if we had a base that only spawned flying units. And we did a little bit of theory crafting our head and it turned out the DPS was marginally higher.

But the problem is once we tried to implement it, we found out the actual problems with the flyer strategy was that we couldn't sustain the units because of the way that energy transfer was done last year. And we weren't actually able to coordinate the attack well enough to gain that theoretical DPS. So testing is extremely important. Wins, of course, beat theory crafting.

So even knowing the specs and knowing the break off times, you have to be able to implement them in practice, otherwise the theory doesn't pay off. The scrimmage server is your ally in testing. And then you always want to prioritize writing behaviors over spending a lot of time doing framework code. Because the behaviors are what you're actually testing, not the framework.

One thing that novices make the mistake of is they're always afraid of ranked games. So has everyone in here played at least one ranked game? Yes? So what I want to say is it's really a problem more with novices than with the people who have been doing battle code for a long time. And the reason this is, is because sometimes you submit a bot and you're afraid that it's maybe not so good.

Or there's some problems with it. Or that your code is going to do badly and you're going to lose points. But what it gains you is by scrimmaging often you will always know what the top teams are up to. And the thing is, if you do unranked, top teams actually have no incentive to accept your match, and you'll never know what they're actually doing. So the trade off is a top team, if they beat you in a ranked game, they'll actually gain [? ELOW ?] points.

But you actually gain something from that. You gain the understanding of how the top team is micro-ing. The kind of high level strategy the top team is doing. And also any tips or tricks or small things that you can glean from those games to perform better in your next bot iteration. Actually if you send an unranked game to one of the top teams, it's actually a lose-lose situation for them.

Because A they have the probability of losing a lot of points to you. And B they're also giving away their high level strategy. So for the top five teams, at least in years past, if you sent them an unranked match, they'd cancel it and send you back a ranked match. Because there's almost no advantage for them to accept these unranked matches. So don't be afraid of sending ranked games, especially against top teams, because that's where you're going to learn the most about how you can improve your bot.

Another thing that sometimes beginners get wrong is that your code base is actually more than your bot. There is a huge infrastructure that we built, at least around our bot last year, that was not just the simple-- that was not just robot player and everything surrounding robot player. So one of the easiest things you can do is you can make custom maps. Every year when we start battle code, we always make this map on the right, we call it plains, because there's absolutely nothing on it.

The plains for this year, we have a few encampments in the middle just to test like basic encampment capture code. But what plains really allows you to do is it allows you to perfect your attack micro and make it so that you're not worrying about like advance path finding, you're just making sure you can win engagements. Because, as I said, attack micro is so important. Winning engagements wins games.

So spend time, actually, making custom maps. I know we don't really have a map editor out this year. So if you make a map editor, too, make a bunch of custom maps. Test your path finding, test your attack micro. There's so many small things you can do to test your bot very fast. A few small things that you guys may have also not known is with custom maps you can actually declare units to be pre-spawned on the map.

So in the top right you're allowed to declare symbols. You can actually set up unit formations to quickly test different scenarios. And just by hitting play and repeatedly running it, you can tweak whatever heuristics you're running in order to make sure that your code is actually running correctly. And then making custom maps is also extremely good for testing path finding. So you can spawn a map with a unit already in the center, and then just hard code in an endpoint to Nav2. And then see if your navigation is running any better than it was before.

This is kind of a little expert thing but we spent a lot last year trying to gain the scrimmage server a little bit. So Justin wrote kind of a scraper that scrapes the rankings every five minutes and determines who's winning to who. If you guys are just starting out don't worry about this too much. But what this allowed us to do, actually, was it allowed us to know which teams were actually rising quickly, which teams had maybe changed their strategy.

And it allowed us to kind of target our ranked matches. Because we would see oh this team won three games in rapid succession, we want to see what they're doing. Or oh this team maybe dropped four games and we probably don't really need to send them a ranked match because they're probably not doing anything different than what they used to be doing.

But, once again, this is a custom tool that was written outside of the framework of just our eclipse project with our battle code bot that allowed us to gain a small competitive edge over everyone else. The current ladder stats ranking system is actually-- I think Justin posted about in the forums, he's actually running it publicly for you guys this year. So if you want to take a peek and just use his code--

Our old one actually recorded [? ELOW ?] gains and losses. So we could see whether a team gained 1 point or whether they gained 10. So if a team gained 30 points in a single match, that's like a warning bell, alright? Because it means they're doing something drastically different and you want to know what they're up to.

This tool was another tool we borrowed from Gunface from 2011. It's basically a mass scrim tester. So the scrimmage server actually is not the only place that you can mass scrim. You can also build your own custom mass scrimming framework, which is what Gunface did in 2011 and what we co-opted from them in 2012. So what this actually allows us to do is it allows us to test in house our own bots against each other.

So you know how I said we wrote like 10 to 12 distinct bots? Well what we would do is we could actually characterize which one was strictly superior to all the other bots by pitting them against each other in thousands of matches. So if you'll see here, what this did was we had a total win counter. So we could submit jobs to our queue server. The queue would run like 50-60 matches on every single known map in the scrimmage ranking.

And then, of course, it takes some time to compute. We actually borrowed a bunch of xVM machines to continuously run this in the background. But you'll see here every single one of our tagged releases, we're actually pitting it against each other. Last year we would consider a bot good if we could beat our older iterations 90% of the time. And until we could get kind of that 90% mark we would consider them about to be equal.

Because some small change, in maybe like the random number generator, whatever, would affect the outcome of the match. But, again, the most drastic improvements-- when you see 31-0, those are from usually from micro improvements where one bot will just-- the two armies will collide and one bot will just obliterate the other one, even with equal numbers.

But these sort of things, running the mass scrim, you can really, really get a sense of. This is an expert level kind of thing. If you don't have-- if you have one person you're not going to blow your time writing a scrimmage based system. But if you have maybe three or four people you could take a day or so to maybe try to repurpose the old 2012 mass scrim tester, which is on GitHub, by the way, to get it working with the 2013 game.

One thing that hasn't really been covered ever, I think, is kind of the byte code system in detail, I think in battle code. And this is a very key part of battle code and yet it's always kind of glossed over. You'll hear the lecturer say you should use less byte codes. Or the less byte codes you use the more units you can have. But what does that actually mean and how does that actually translate into implementation?

So I'll skip ahead and just give a brief overview of the JVM for people who are unfamiliar with it. So the Java Virtual Machine is a stack based system. So what it basically means is if you're familiar with x86 or whatever, there are no registers is just a giant stack. And you can imagine it as values get pushed on the stack and popped off of the stack, and that's how computation is done.

Byte codes are-- when you write your Java it gets compiled into something called byte codes. And byte codes are very similar to assembly, except they work on this stacked based system. And the way this stack is, is you have this operand stack which is your current stack of values. So if you imagine like a prefix operations I can push-- or postfix operations, I can push to values on the stack, I can call add and it would pop off the result, right?

So if you imagine the JVM is kind of working like that, it's very similar. But there's also an array of local variables up top and these are indexed by a number. So, actually, stack operands can operate directly onto the array of variables that are globally accessible. And then there's also a global constants pool that byte codes can operate on.

So this is just an example online. And if you want to read more there's a URL down here. But what happens is you compile your Java, it gets turned into byte codes, and each byte code is actually stepped through by our instrumentation engine. And that's what you're getting counted on. So when someone says you're using 10 byte codes, it actually means that the high level Java call you made is being turned into 10 individual atomic steps on the JVM.

And you're getting penalized for each one of these atomic steps. This leads to a lot of interesting things. So there's a very classic kind of loop structure in battle code. I hear this is so classic, actually, that Dropbox's code base is littered with this structure. So it looks like kind of a normal for loop, but if you look closely, you'll notice that we don't even have a third parameter.

What this says is that for i is equal to the length of the ray, decrement until you're greater than zero. And it's reversed from maybe the way normal computer science would teach it to you. But there's a particular reason we do it this way. And that's because we're actually byte code hacking. So what happens is Java is similar-- kind of like there's legacy in computers testing again zero is actually an operation that comes up very often.

In most branch conditions you're testing against zero. Like Booleans are testing against zero, et cetera. So there's actually a special byte code for testing against zero. And then, also, when you're doing the comparison you don't want to be recomputing, say, msgs.length every single time. So this actually has a net effect of saving byte codes because you can actually write the byte codes out but you can predict what the compiler will spit out.

And because battle code is not run as just in time compilation, actually, the byte code steps through exactly as its outputted. This ends up with a net savings of a few byte codes. So if you'll see down here, this is like the classic way you would write a loop that you might learn in intro to computer science. For i is equal to zero, i is less than your terminating condition, increment i.

What happens here is because you want to compare i every round to msgs.length. Computing msgs.length is actually a separate compare operation with loading the actual object, loading the actual array object. Whereas in our initial version what we've done here is we've set the index variable actually to the final value. And we're comparing against zero, which is a single byte code operation.

So what happens is that this is a net savings of two byte codes. Two byte codes actually might not seem like a lot, but if you're, say, looping through 100 objects that that's 200 byte codes saved in the entire loop. And that's two whole sense functions. So you could send-- that's two whole global sense functions, with the radius and the location. That's actually pretty big.

So saving 200 byte codes can mean that you might have additional data on the battlefield. Or you can loop through one additional object when you're trying to compute a heuristic for map locations or for army positions. And this actually gives you a small marginal competitive edge over the other team. Especially this year when byte codes also translates to energy. So saving byte codes gives you back more energy.

More energy translates to more units. If all your units are saving 500 byte codes per turn that adds up over time, especially with the decay rate. So with byte codes you'll also end up wanting to implement your own data structures. So I don't know if you guys have tried using the native Java data structures like hashset, array q set, et cetera.

But we actually penalize you for the internal calls that Java makes in those data sets. So if Java-- Java likes to be very safe. So a lot of the thread safe code we'll check to make sure another thread is not trying to mutate the data, or that there's on a lock on the data, et cetera. So the thread safe code actually ends up hurting you a lot, because there's a lot of excess byte codes that gets called.

So, for instance, a single hashset.iterator.getnext getting a value from a hashset, worst case is 2k byte codes for a single call, which is absurd. Because you can implement a hashset yourself with an array and maybe like 200-300 byte codes if you custom write it. So what ends up happening is a lot of people will make what's called fast data sets. So we wrote a fast hashset, a fast arraylist.

And these are all custom based on arrays that we could index directly into and pay a very minimal byte code cost in order to get very complex data structures back. Same thing goes for algorithms. You may think that-- a lot of beginners will start off trying to implement something like [? astar ?] or a flood fill algorithm just to do navigation. Yes? Right, this is an example from--

I'll explain what exactly this one is. But just an overview, most n squared algorithms are completely untenable in the byte code system. So if you see an n squared algorithm it's not going to work unless you do something to it. So there are multiple ways. You can distribute it among multiple robots, which is actually very hard. You can break up the computation into multiple steps and run a piece each turn, which is hard depending on the algorithm.

And then you can also just run a completely different algorithm. So the one I actually have up here, this is a cute little example from our old code base. It's not the actual implementation but it's the algorithm itself. This is called a multiply with carry. It's actually a pseudo random number generator. There's a period-- there's like a period of 2 to the 60th. But for all intents and purposes it is a random number generator.

We actually implemented this because we noticed that math.random on seeding pays 144 byte codes. And then each subsequent getnextinteger call is 80. And we actually wrote our own custom random number generator to save 40 byte codes per call to get nextint. And so this is the algorithm actually pulled from Wikipedia, we changed a little bit. But, in essence, it's a bunch of bit operators that gives you a pseudo random number generator to save on byte codes.

And if you have an algorithm that relies on a lot of random numbers the 40 per getnextinteger actually adds up. There's actually a tool, a really nice tool, that we used a lot last year call doctor garbage visualizer. I don't know why it's called doctor garbage, because it's one of the best tools ever. But what it does is it actually decompiles your class into the actual byte codes.

And then it shows the control flow graph on the right. So the control flow graph will actually allow you to reason about where your short circuits are happening, because if you short circuit a code block, you're not actually paying the byte code for what you don't execute, right? So you'll want to terminate loops end-- you'll want to terminate loops early. You'll want to try to avoid code that doesn't have to be computed multiple times in order to reduce your total byte code count.

So by having the control flow graph on the right, it's actually very easy to reason about how to save byte codes, especially when you'll see the control flow graph, plus the instruction, plus kind of a small comment about what the variable is or what the method call is or what the constant is. So that's enough for-- So, ultimately, with byte codes, the last thing I'll say is if you're not sure, test it.

We give you a function called getbytecodesnum and this allows you to see what byte code number you're currently executing. So if I want to profile a particular function call or a particular algorithm, just surround it with two system.out print line getbytecodenum. And this will tell you the total number of byte codes that your algorithm took to execute. And you can use this to kind of profile it down.

You can try different things but ultimately whatever gets that number smallest is what helps you bot when you're optimizing algorithms. We actually, at the bottom, this is a little loop that-- or this is a little function call that I've added at least every year. At the end of every robot's round it actually checks the round number and the current byte code number to see if it went over byte codes.

And it will actually emit a warning to the system console that it went over byte codes. So actually it's really annoying but it's really helpful that if every time a robot goes over byte codes it yells at you. Because then you can play match if you're not actually profiling something-- or if you're not actually explicitly saying that, you can sometimes ignore it because the byte code number is just in the top right of the client and you'll kind of miss it sometimes.

You'll see like, oh, maybe it's like 10k or whatever. But if your robot is yelling at you every time it goes over 10k and your entire army just screens in the middle of battle, there's something that has to be changed in order to make it so that you're not missing an attack-- or you're not missing a movement every turn that you could be moving. So this year, actually, it's even more critical because there are no cool downs.

Every robot, every turn, can move. So if you go over byte codes once, that's a lost movement, which could translate to a lost attack. So it's very critical in battle situations that you do not go over byte code so you can get your movement. And I'll show you why in a bit especially in micro situations. So micro is really important, attack micro, I've been emphasizing this over and over.

You should know the execution order of the robots. But you should know what happens when. So when do attacks happen. So when do attacks happen? At the end of the round, right. You should understand the discrete nature of the map and you should know that micro's better than everything else. So I don't know if Max has covered this, but in two robots which robot actually wins the one v one engagement?

It is-- yes, it's the one who closes the gap first, is the one who wins the engagement. So if I have two robots, two squares apart, right now it's a stalemate. But if red makes the mistake to move in, what happens is blue closes the distance and gets the attack off first. And, all things being equal, red will attack second. But blue has already done the initial damage, which means on blue's turn where he kills red, red will not deal back the corresponding damage to kill blue.

Blue will win. So blue will have one more robot in the next upcoming engagement and therefore higher DPS and will win the next battle. This situation is also particularly bad because red can't actually retreat. If red moves back, blue can still close the gap next turn. So understanding how the micro works in this game is actually very key to winning engagements.

So in a one v one situation the gap closer always has the advantage. Just a straight up. So you will want to write code that takes this into account, so that you can actually win engagements. In a two v two situation, it's a little bit more complex. So what happens in two v two? So red can move forward. Blue can close the gap, but red can close the gap with the second unit.

And now he's dealing twice DPS that blue has and will eventually win. Blue can actually-- if blue is clever, blue can actually micro to focus fire on a single unit, because damage is split evenly across all enemy robots. So blue can actually try to micro little bit to kill the bottom red unit, if red makes a mistake. But red will actually-- can move down and continuously attack blue until the blue robot is dead.

And the two v one will win. And if red micros correctly he won't lose a single unit. So in a large flight-- in larger fights, the larger army has the advantage given correct positioning. Because you could assume in this sort of scenario blue can actually reduce this to a one v one flight if red does not make the correct choices in the subsequent five or six moves.

So blue ball will move up and attack red and reduce it to a one v one scenario unless the red robot can retreat backwards while still attacking blue towards the bottom red unit and continue fighting so that they can turn the one v one into a two v one engagement. But these small, discrete steps are what you should actually be thinking about when you're writing microcode.

It's very critical that you don't get indefinitely kited. It's very critical you do not write code that ends up in this scenario where you're attacking and you try to retreat, but you can't actually retreat in this game, because the opponent will close the distance. So there's only two scenarios in which you can retreat. You can retreat if there's an army behind you willing to back you up.

Or you can treat if you're retreating over a mine. That gives you more DPS. So there's some team dynamics things that I just want to cover. This is mostly geared towards the novices in the room. But make sure you guys are using source control management. I think this goes without saying, every MIT class now emphasizes the importance of source control management.

And I'll even say you should just use hosted source control management. So if you're using Git, if you're using Mercurial, you should just throw it up on one of these free hosted SCMs just because you don't want to deal with the server going down. So my first year we made the terrible mistake of hosting our own SVN server on an xVM machine from [INAUDIBLE].

And then randomly there was some down time that was uncontrolled and we didn't have access to our code base for like two days. So it was terrible. So just-- GitHub, these guys are paid to keep your source up, OK? So [INAUDIBLE] is not paid to maintain that your battle code source does not go down. So trust these large source control guys. And especially trust distributed source control, if you guys are familiar.

I think Max gave a lecture on Git, so use it effectively. And also figure out your SCM-- your source control strategy. So this is actually a big one. Every team does they're kind of release cycle or, I would say, release cycle but you're like releasing bots, a little bit differently. So I'll explain the way we did it. We had a single package called Ducks, just because we really like ducks. Don't ask.

We forked off a bot every time we release it. So in our mainline Ducks package, when we get to a state, we're saying this bot's reasonable. We want to submit it to the server. We'll actually copy the Ducks package, rename it to our kind of bot naming scheme, which is based on SC2AI levels, if you guys play Starcraft. And so we started off with a medium player. So we'll copy the Duck's package into the medium player package.

And what this allows you to do is always have a copy of the medium player in your repository. So you can always pit your mainline bot against any of the other heads you've cut off from the repository. So some teams like to do it differently. I know Gunface in 2011 liked to use source control tagging. So what this allowed them to do was they could update to any version in the source control.

And they could recompile it. But the issue with that is if you update you have to update your entire mainline package backwards. You have to recompile it, and you have to keep the class files around so that you can pit them against your current bot once you re-update to the current head. This gets a little bit annoying. And for us it was just simpler to persist like 30-40 copies of the same file just in different packages.

Because it saves on time. It's not elegant, but elegance doesn't really matter that much when you're trying to finish a bot in the short amount of time that you have. We actually went one step further. This is technically part of the custom tools, but we actually had One touch deploy system. So we had a single script that was responsible for building the release, tagging the release, testing the release on our automated mass testing system.

And then cutting the branch from mainline and pushing it back to the repository. This is very key, actually, if you're doing what some of you guys may have been doing last night in trying to submit like 20 seconds before the deadline. Because if something goes wrong you're completely screwed. So if you have a tool that you know works and you know roughly the amount of time it takes for it to work, it's standardized and you're less prone to human error.

So we ran the script. Everything's done. We have a jar. We upload the jar. We're done. And this is reliable that we're not manually trying to create the zip file. I know some people have been having issues with bots randomly exploding. Or sometimes a submission works or sometimes a submission doesn't work. If the whole thing is automated it's going to work barring some catastrophic failure.

This is also a random cute little tip. Sorry for this stream of conscious, but I'm just trying to mind up as much useful information on to you. This one I'm actually really surprised, I couldn't figure this out for like three years. How to deal with shared debugging. So for those of you guys working in teams, you'll have print lines, you'll have indicator strings that one day you'll be testing something and you'll merge in someone else's pull.

And then you'll be like who overwrote my indicator strings? Because now I don't know what I'm debugging. Have any of you guys had this issue? So you'll have-- you're debugging nav and you're printing out the current Nav state-- your current state machine's navigation position, and all the sudden the guy working on attack micro has overwrote your indicator strings with current number of robots attacking you, or whatever.

And this gets really frustrating really fast. Because then you end up with merge conflicts, when people try change each others indicator strings. The guy working on nav will con out the attack guy's indicator strings. And then you'll end up with a merge conflict and then you guys will screw up the merge conflict. And stuff will just go to hell.

So what we did was the bots can actually read from the Java bc.config. And actually what we built was this custom system, this custom debug system, that allowed us to specify who was running the bot based on the current eclipse launch profile, and only print out indicator strings from the person watching it. So you'll see on the left we made four custom launch profiles.

One for me, one for Justin, one for Haitao, one for YP. And when you ran your custom launch profile, it would only display debug information from your bot. So you'd only see your own print lines. You'd only see your own indicator strings. And then we never had a merge conflict. Well, we did have merge conflicts but not related to indicator string setting. So this is just a small little trick that just saves a lot of time.

Because stuff will add up across your total bot's development time. You don't want to spend time arguing about people-- I know some teams will say, OK, you can only use indicator string one. I'll only use indicator string two. But someone will need two lines of information. Then you guys will fight over it and it's really stupid.

Other general, useful advice. All the past teams have actually put a lot of information on. So there's a lot of stuff that I don't need to reiterate. So how many of you guys actually read the blog post that I put on hacker news? Yeah, nice. From 2012 to 2009, actually, all the old strategy reports, all the winning teams, are actually online.

So I have the bug post from our team, but there's also a strategy report that corresponds to that, to our bot, that's in our repository. The 2011 post-mortem is actually an excellent write up by Steve Arcangeli, who wrote the massive scrim tester about various other tricks you can do to save byte codes. 2010 is by Spencer Skates on finding the shortest path to victory. It's kind of their whole like Bellman Ford pun team thing that they had going on.

They detail a lot of high level mechanics that help you when you're in combat. They also-- Spencer Skates and Steve Bartell also won in '09 and they were called G2G Ice Skating Lessons, which was-- it was like an inside joke that none of us got. But they wrote about their 2009 experiences as well. They actually set the trend of the winning team usually ending up writing the best strategy reports, which is a trend you guys should continue.

Some other random tips. Messaging. I'm really excited about the new changes this year, because messaging is global. So what this allows you to do is you no longer have to have robots really close together. So if you guys are familiar with the 2011 game, what you would have to do is, because you could only broadcast eight squares away, you had to set up a relay network of robots that would propagate information outwards. And it was a whole amount of overhead just associated with writing the relay network that could get a message from robot a to your entire army. That was at least a day's worth of work.

Setting up the re-broadcasting. Setting up the hashing. Setting up all that code. This year, because everything is a global message board, it makes it easier to broadcast very important information very fast to all your robots. But what this also does is battle code has a huge history of message based attacks. So it also makes the message based attacks more interesting because now the enemy very easily can see everything you're writing to the board.

So some of the more famous message attacks have been one year-- you guys are familiar that the robot VM limit is eight megs, right? So if you exceed eight megs, your robot actually automatically uploads. So one year, a team went around just broadcasting eight megs arrays. So the enemy team would either get it, try to process, and completely freeze. They'd get it, try to process it, and explode immediately, because they loaded into memory or something stupid like that.

Or they would just stop moving because they constantly exceed their byte code limit. Then, I think in like '07, teams were trying to exploit the fixed cost of array.hashcode. So array.hashcode used to be a single byte code. We used to-- there is a package in your distributable battle code installer called method costs, and that explains the cost of every single method that we don't explicitly list a cost for--

Or if a method is not listed in there it's free unless it's in one of the restricted packages. But what happened was array.hashcodes was free. So teams thought they were being really clever by using the o of one hashcode cost to basically do an o of n algorithm to get entire hashcode and secure their messages that way. But then one team just took the fact that every team was using array.hashcode and he could mutate the array without changing the value of the final hash.

And then he would send that back to the robot and all their robots would just completely mess up. Because they thought the hash was correct, but the data was completely garbled, which which was a really interesting messaging attack. So you need to make sure when your messaging to check the integrity of the data and make sure that the data-- check the validity of the data and also make sure that the data actually gets to where it's supposed to go.

So, this year the most obvious attack, of course, is to wipe the entire board. And you can do this every round if you have some obscene number of generators, like 30 or so. But the thing is, you can pool energy this year. So actually a team that, say, just wants to wipe the message board during critical situations, like during an engagement, can actually pool enough energy so that they can wipe the entire message board in one go.

So you can't actually assume that what you wrote into the message board one round will actually be their the next round, or even between rounds. Because a robot-- The enemy robot who executed between you could have wiped it. So make sure that whatever messaging scheme you use, there's a way to check the validity either by hashing and storing maybe the top four bytes as your hash.

Or by duplicating the information in multiple places and checking that all the information matches with each other. But as long as you can reliably get important information over. Like if you had a retreat signal, or an all out attack signal, just make sure it's secured and it's safely propagated to all your robots. Here's a random piece of advice. Keep your frameworks lean, especially with the byte code limit.

You don't want to create this beautiful hierarchy of abstract interfaces that just has a ton of intermediate layers before you actually get to the nitty gritty code. Because for every additional layer of abstraction, you're increasing the byte code cost. So actually one of the cool things that the byte code limitation actually enforces is it enforces really lean code, which is not something you typically see in Java.

So if you'll actually look at our class structure, this is from our 2012 bot, we have a very, very flat class hierarchy. One, because we just didn't want to deal with extra subpackages. And two, because there's really not too much to be gained from really elegant architecture other than just wasting time. Because the behavior code is ultimately what matters. So we wrote extremely lean frameworks.

So this is actually our main run loop. You'll see here all we do is we reset our internal timers. We update a few key round variables, like the current time, and my current energy, and the current position of all the bases. And then we run our message system. We actually run the main run call, which would basically be a giant state machine, probably like 500, 600 to 1,000 lines long of just completely messy code that determines what the robot wants to do and does it.

And then we actually abstracted movement out. This was probably the biggest piece of abstraction that we did, was we moved a movement out to a separate position where we knew that each robot was reliably moving every single round, because it's very key that you always move because that is an impact on your total damage per second. And then we did some generic things.

And we also have this cute little thing down here at the bottom which is if we had some spare byte codes we would use the extra byte codes for computation. So navigation last year was-- because there were walls, the algorithms you had to use last year were actually quite different. Most teams would write something called bugnav, which-- you guys from the navigation lecture, I suppose you discussed a little bit, where you see an obstacle, you hit the obstacle, you try to trace around the obstacle, and then once you're free of the obstacle you keep moving.

But the more advanced teams would actually write a more complex version of bugnav which is called tangentbug, which is-- you basically can project a virtual bug and if you see you've rounded a corner, instead of going forward, hitting the wall, and going around it, you can just cut the corner. And we actually used our extra byte codes to pre-compute steps of tangentbug in order to make our navigation maybe five or six squares better.

But that better means you get to the enemy two rounds faster because you cut a corner. And then two rounds faster means you have extra DPS on the enemy and so you win that initial engagement. This year because there are no permanent walls, bugnav and tangentbug may not actually be some of the best algorithms. Haitao and I have kind of a little bet running on what the best nav will be.

We're thinking it's probably going to be some sort of very discretized floodfill algorithm or Astar algorithm that doesn't try to compute every square but actually breaks everything up into discrete segments. So you want to be able to rush to the enemy base if they're just a straight up nuke bot. We guarantee you a round-- we guarantee you a minimum of 200 rounds if you follow the optimal path.

But you have to know what that optimal path is or you won't get there in time.

Maximum of 200 rounds. So our bets are actually on the most advanced teams writing kind of an Astar algorithm or a floodfill algorithm that takes the map, divides it into discrete chunks so it doesn't consider every individual square, and then it kind of picks sort of roughly the best path to take.

So you can avoid areas with more or less mines than the other areas. And then this will still get you to the nuke bot in enough time without having to blow the entire cost of the complete Astar algorithm. So Astar, in maybe like an eight by eight or ten by ten is tenable, but it's not tenable in twenty by twenty or a hundred by a hundred situations.

This is kind of a big one. Don't be afraid to write messy code. I think it speaks for itself, but in case you guys are like spending a lot of time trying to make beautiful code, the thing is what you ship is ultimately-- what you ship that changes your bot's performance is ultimately what matters. And a lot of beginners have this issue where they're afraid to just hack something together, or hack a one off bot to test.

But it's so key that you kind of overcome this fear to write messy code, write stuff that works, just get something hacked out that's working. And then use that to gain yourself an advantage in what matters, which is the actual scrimmage, the tournament matches, et cetera. So we actually have code that's terrible. Our navigation code actually is completely-- is not understandable by the person who wrote it anymore because it's so messy.

Like you can imagine, bugnav, for anyone who's tried to implement bugnav it sounds really simple, but there are actually like 100s or so of edge cases that you have to account for. So if you end up bugging around another robot or you end up in some sort of infinite loop bug, there's a lot of extra edge conditions that you have to account for that none of the previous lecturers have actually talked about from '09 or 2010 or 2011.

They'll just say oh yeah, it's bugnav, just go around the thing. But what happens if you have two units bugging around the same wall and then they hit each other, then one unit has to go around. But if one unit goes around and the unit on top moves, all of the sudden you have a unit who's no longer next to a wall but though he was bugging around a wall. But he doesn't see any walls next to him anymore.

And then you're states get messed up. And everything just goes to hell. So don't be afraid to just write edge case code that deals with that, gets your bot up and running so you can test the frameworks, and the strategies, and everything that matters. There's also-- last minute hacks actually win games. So I don't know if you guys are familiar with our 2011-- or 2012 championship.

But what we did was on the very last day we were scrimmaging on the server and we found out that the second place scrim team, team 16, had modified his attack code, that it was actually slightly better than ours. And because having better attack code means you win the engagement, it means he beat our bot actually three times out of four.

So we got, actually, really scared about this. And don't forget we scrimmed this the day of the final submissions. So an hour before the deadline, me, Haitao, and YP are just thinking about well what should we actually do? So our ultimate hack was we actually figure out what the team is by the characteristic of their message broadcasting. So you're not actually given the team number.

But if you look at what kind of messages they're sending, you can kind of guess, right? And throughout the whole battle code 2012 we had been kind of on the side just curious, looking at the structure of other teams messages. And so an hour before the deadline submission we wrote something that determined whether the team we were playing against was team 16.

And we would change our strategy. And this was an hour before. And it was completely untested code. But we had been thinking about it, kind of in the back of our mind, and we just threw this together. We uploaded it. And if you were at the finals it was a pretty tense match where we actually had a bug in that code that triggered once and would not trigger a second time.

But the one time it did trigger was the very last match, and so we ultimately won the final tournament. But this is like a piece of code, incredibly critical. It's like three or four lines, but it was the difference between first or second place. So once again, spending time where it matters is really key. You can probably do these sort of hacks this year. Determining what the team is based on the message structure is a little bit harder.

Because last year people would send discrete message objects. And so you could count the total number of integers, the total number of map locations. Whether there's a secret key and then you could correlate the secret keys. This year you'll have to see how they channel hop or what they store on the various channels, or the channel distribution of the different messages. But it's still possible.

So last minute hacks do win games. This is also another small one. You guys should all-- if you're not already using IRC, you should use IRC, because all the devs are on IRC, especially us who are remote. I was the only person working on engine who was actually willing to fly out six hours to give this talk. So everyone else is on IRC. If you have any questions you should go on IRC and you should actually ask your question.

So, for those of you who don't know, there's a web client. Just use the freenode web client, connect to channel battle code, ask whatever questions you have. We'll answer them pretty fast. If you guys are feeling a little more advanced, you should use IRC better. If you actually parsesys in the channel there's a lot of banter that goes on regarding game balance. Or teams will discuss strategies or whatever.

So if you actually get a good IRC client and stay in the channel, you'll actually learn a lot. Because sometimes some of the old battle code champions will just come in, just talk a little bit about strategy. Or people will just discuss what they've been doing on the bots, et cetera. And so just use a good IRC chat client. If you were to use pigeon, pigeon has a built-in IRC client. There's no reason to not connect to it.

Talk to the devs, ask questions. There's just been a lot of very simple questions, but we're more than happy to answer. So questions about round number, or if you have an issue with your bot, it's much faster than the forums, because we'll answer the

IRC probably in minutes rather than hours on the forum. And, most of all, you guys should just have fun. Battle code, at least for us, was a very fun experience.

We worked hard on it. But we also-- we goofed off a lot. We probably spent more than 40 hours playing Civ 5. And before the sprint tournament, or before the Qualls tournament, this is usually a bad idea. But it's about-- this is IP, it's about having fun. So work hard, but also have fun. And also I have a few requests. So there may potentially be the 2013 winner in this group.

So, if it is one of you guys, just a few requests. You guys should write a good strategy report, because battle code is not done in a vacuum. We learned from the teams before us. The teams before us learned from the teams before them. This information really needs to be more out and open, which is why I'm giving you kind of a brain dump of a lot of the very small tricks and tips that we did in order to gain a competitive edge.

So write a good strategy report, help your fellow competitors. This is a friendly competition, it's not cutthroat. You guys can make fun of all the other lesser competitions, but within battle code keep it a good family. And then join the devs if you guys win, because we need good desks. So thanks.

[APPLAUSE]

CORY: So any general questions or whatever? Once again, yeah, it's our fault if you guys really hate nuke you should say it now. Yes?

[INAUDIBLE]

AUDIENCE: It changes from year to year. So how did you choose your team?

CORY: The 2012 team was all Star League people. So there's also another Star League team. I have some high hopes for the Star League guys. First year I did it with people from my high school. Second year I did it, I did it with Star League people. The third year was also all Star League people as well. You guys should go to Star League if you like Starcraft. That's a pitch. Every Friday at 7:00 o'clock for 253. Go play Starcraft

It'll make your meta game analysis better. Any other questions? Balance? General comments? So I'll be here afterwards if you guys want to ask any questions. If you want to look through our old bot. At some point I will write out a code walk through of our 2012 bot. But until then just ask, or we'll be in the IRC channel or whatever. So hopefully this was useful to you guys. So thanks.

[APPLAUSE]