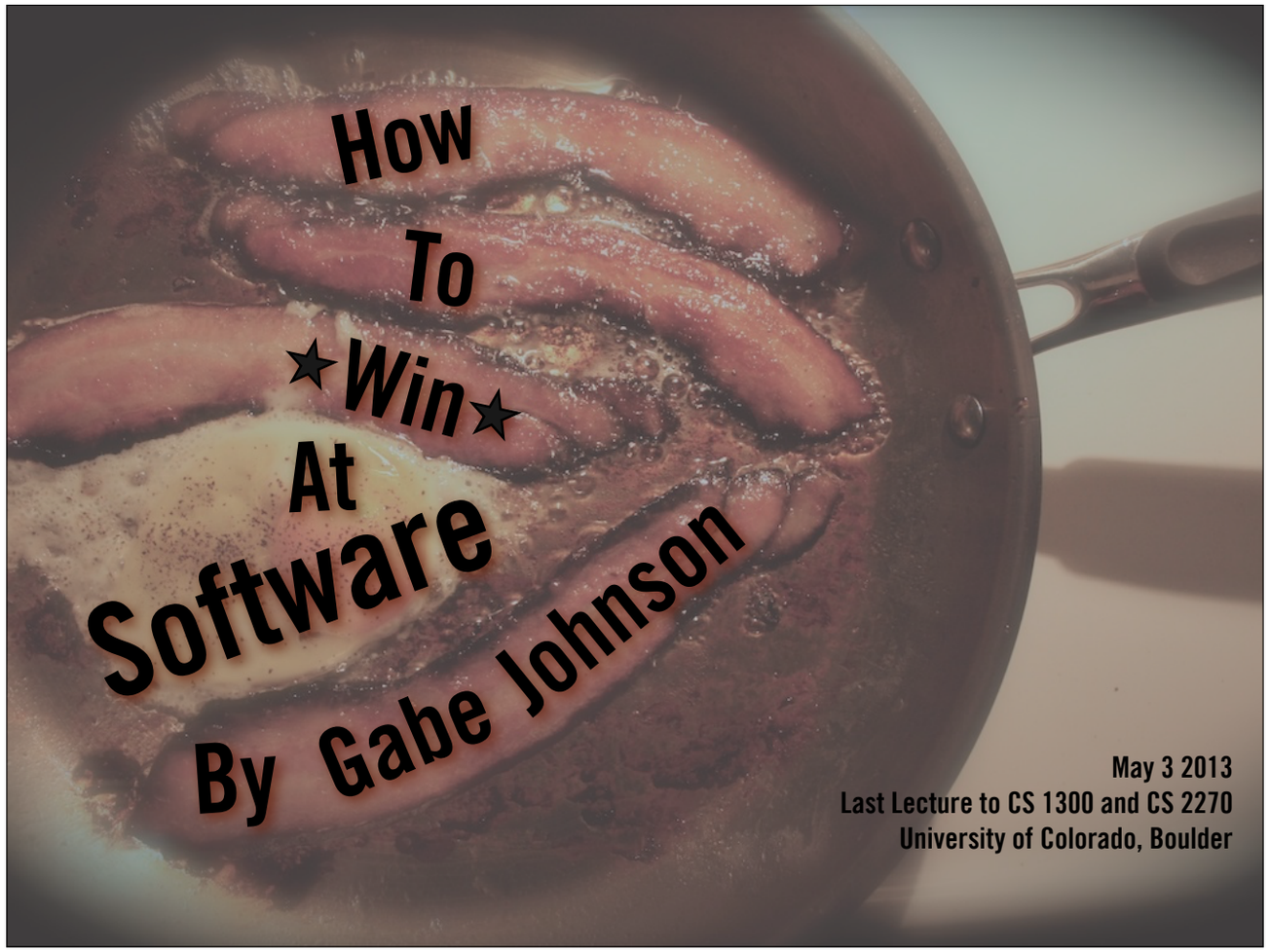

How
To
★Win★
At
Software
By Gabe Johnson
May 3 2013
Last Lecture to CS 1300 and CS 2270
University of Colorado, Boulder

Today:

This is a collection of things I've learned, discovered or was taught over many years. It is not deeply technical but it might help your technical careers.

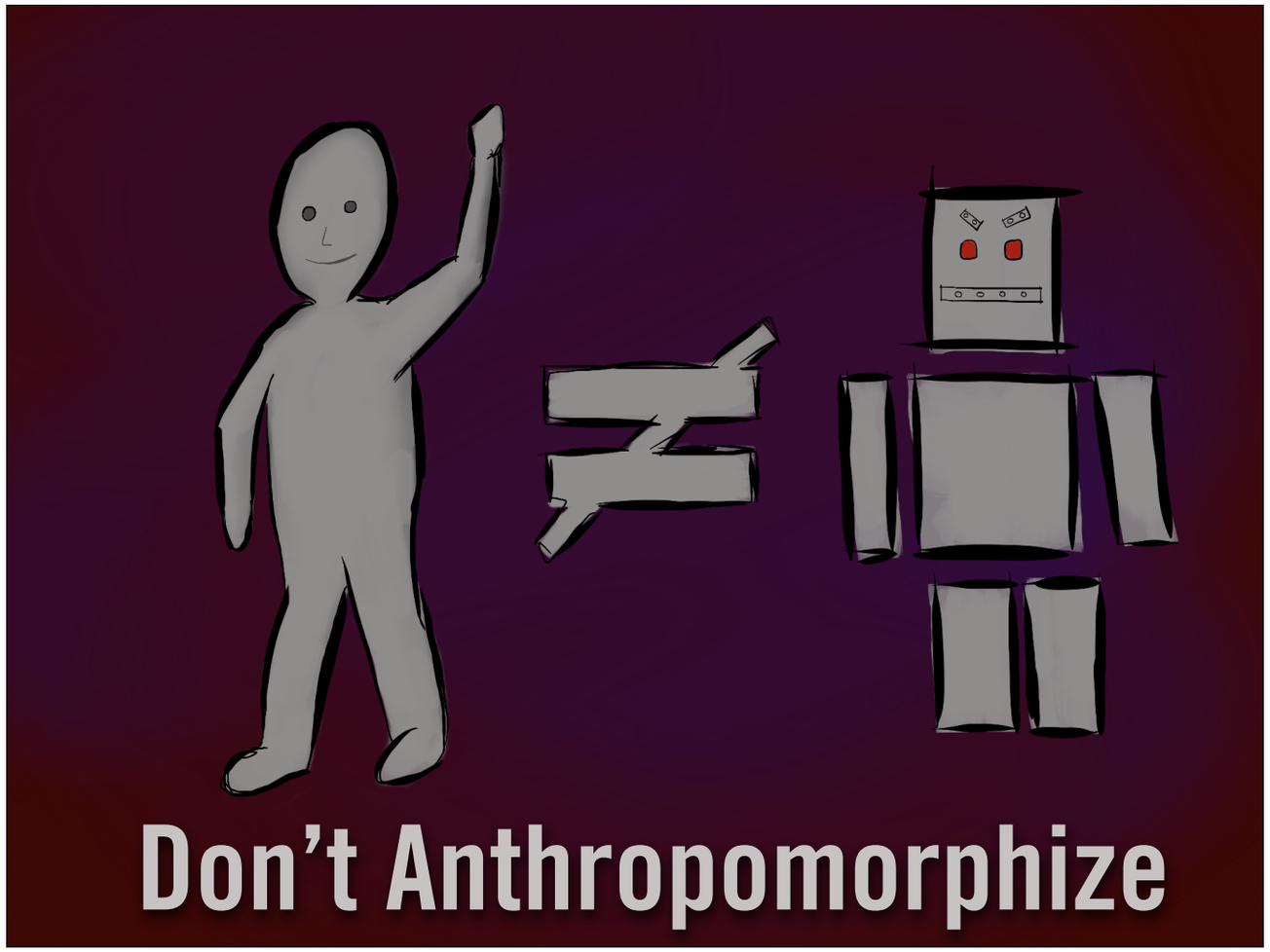
The very first thing: have a good work ethic. Everything depends on this. You are not _purchasing_ an education---you're _earning_ it.

That being said, laziness is a virtue. And by that, I mean you should rely on all the work you've done, all the work your colleagues have done, and all the work done by countless other hackers out there. Like the goblin says, "time is money, friend!"

So if you're "lazy" and use other people's code, that gives you a good starting point for going past them. You have to develop a good work ethic. You will get frustrated. This is natural. You will want to quit and do something easier. Take frequent breaks. Get away from the computer and let your mind wander---you'll be surprised how often it wanders into a solution to whatever vexes you. Develop strategies for learning and overcoming problems. Be deliberately tenacious.

It is easy to strike up an amazing idea that is way beyond your current capabilities. By all means, try to take these problems on anyway! But realize that you can find yourself stuck if you get too carried away. The worst thing that can happen is that you mistake this experience as evidence that you’re not good at programming.

The converse of avoiding problems that are beyond your capability is working on things that are too easy and don’t challenge you. The worst thing that can happen here is that you wrongly get the sense that you can do anything because all the things you try are easy.



When working on software projects it is easy to lend the machine a soul. People often characterize their computers as being temperamental, angry, confused, or capricious. But computers aren’t people, and even though it might *seem* like they are giving you attitude, they’re just electromechanical systems, and what you think are personality flaws are just bugs (that you might have put there).

The reason this is important to understand is because when you realize that the computer is essentially a very fancy toaster, it no longer makes sense to get emotionally involved with (or perhaps *at*) the computer. Get past the emotional weirdness so you can get to the source of the problem.

The next few slides are general advice about how to tackle big problems. This is in the specific context of software engineering but it really applies to much more, up to and including starting a business.

Failure is everywhere, and how you face and handle failure determines if and how you will succeed.

Fail fast so you can get to the next iteration and hopefully fail a little bit less.

Fail as early as you can so you waste as little time as possible.

Fail as often as you can. If you’re not failing you’re not working on interesting problems.

FAIL

... and
CHARACTERIZE
your mistakes

When you fail, be thoughtful about it. Characterize what went wrong. Was it something with your thinking? Your implementation? Your measurement? The context? Does the failure resemble something else you've seen before?

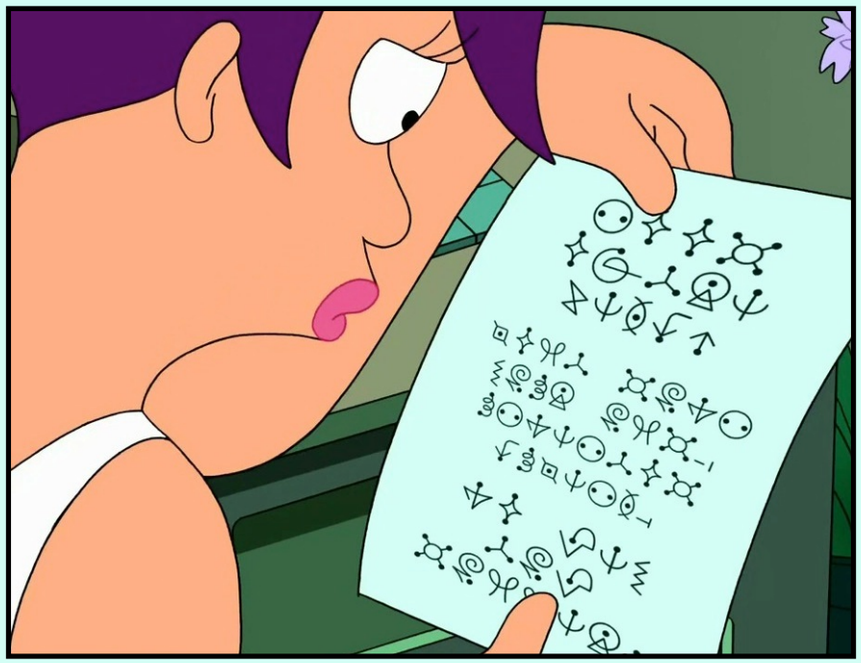
This includes BUGS

And ERROR MESSAGES

We characterize certain software errors in categories. Anybody who has learned C or a similar language has had to figure out why their program suddenly crashes with an error message like “Segmentation Fault.” There’s a few kinds of bugs that can lead to this error, and once you’ve come to understand those bugs it is much easier to find and fix it.

Outside of software, we can characterize other “bugs”. Some device doesn’t work. “Is it plugged in?” is a great question to ask, because “it doesn’t work” can often be tracked down to the thing not having power.

Be Legible



If you want to spot bugs (failures) faster, there's some things to keep in mind as you work.

The first is to be legible. If you're having a hard time being legible, it might indicate that you don't know what you're trying to say to begin with. It's perfectly OK (and necessary) to go through the rough thinking process. But when it is time to externalize your thoughts by speaking or writing them, legibility pays dividends both for you and for your audience.

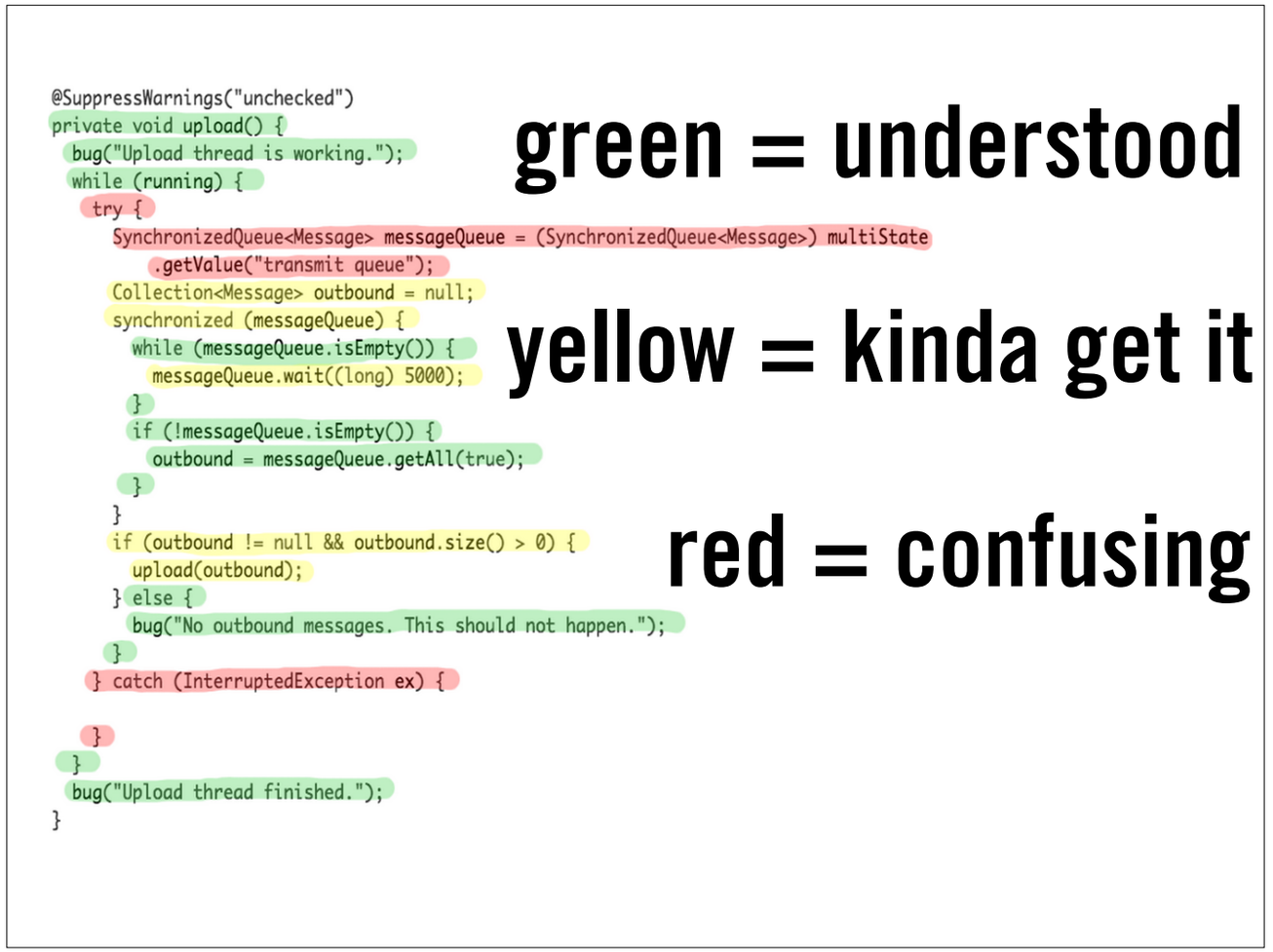
```
int FSM::addTransition(int stateA, int stateB,
int signal, string transLabel) {
int id = -1;
if (stateA >= 0 && stateA < (int) states.size()
&& stateB >= 0 && stateB < (int) states.size()) {
State* st = getState(stateA);
if (st != NULL) {
vector<int>::iterator it = st->trans.begin();
bool found = false;
for (; it != st->trans.end(); it++) {
int transID = *it;
Transition* tr = getTransition(transID);
if (tr != NULL) {
if (tr->signal == signal) {
found = true;
break;}} else { found = true; break; }}
if (!found) {
id = transitions.size();
Transition* tr = new Transition;
tr->label = transLabel;
tr->signal = signal;
tr->next_state = stateB;
transitions.push_back(tr);
if (signal == FAILURE_SIGNAL) {
st->failure_trans = id;} else {
st->trans.push_back(id);}}}}
return id;}

int FSM::countStates() {
  return states.size();
}
```

(this is not legible)

If you've had to wade through code like this, which is badly formatted, not commented, and using mixed conventions, you understand how annoying it is when the author was not making the effort to be legible. Odds are the author didn't know what they were doing either.

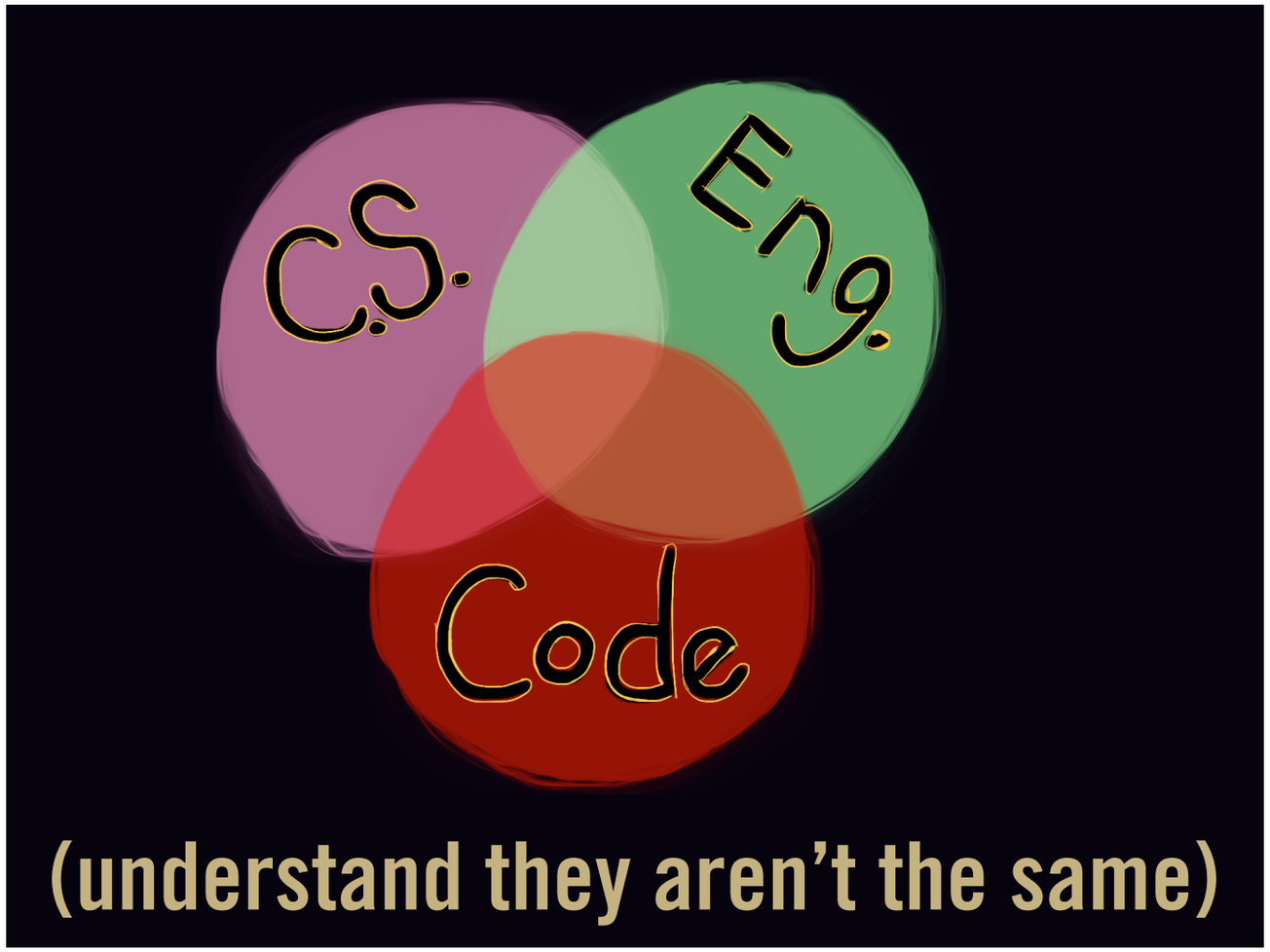
Be Clear

Even legible messages can still be hard to understand. Instead of “int fd01;”, consider “int fixed_disk_01;”



Here's a good strategy to know if you (or somebody else) is being legible and clear. Pretend you have three highlighters: green, yellow, and red. Go through your code (or term paper, or whatever) and color every line. Green is for code that makes perfect sense. Yellow is for code that you kind of understand if you think about it for a few seconds. Red is everything else: even if you *can* understand it, it takes too long. Bugs love red code. Your colleagues will love your green code.

You should strive to eliminate all red code. You do not get points for cleverly writing code that nobody can understand. (You might get yourself fired or refactored out of the position.)



Computer science is a theoretical thing. It's really a renegade branch of mathematics.

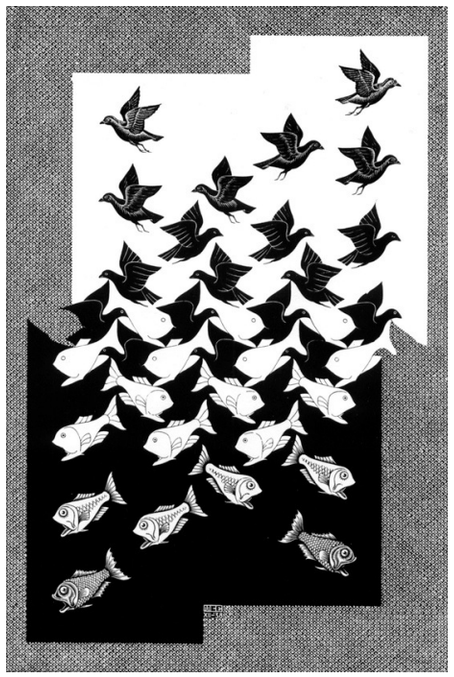
Engineering is a practical thing. Teams and whole companies base their livelihoods on this.

Coding is also practical, but it is personal. It's the language individual people use to train computers to do tricks.

There's a lot of overlap between these things, but they aren't the same. A fantastic software engineer might be able to create beautiful architectures, but not be very good at coding. And vice versa. Theoretical computer scientists often do their best work with pencil and paper.

My grandfather had a war story. He was a lead engineer at the Rock Island Arsenal during World War Two. In the fall of 1944, he was called on by the DoD to redesign a piece of artillery so it could withstand being dropped from a very high altitude (so planes could avoid the German AA). Several teams of engineers had tried to solve this problem before, and they had focused on the gun itself, just as the DoD asked my grampa. But instead of focusing on the stated requirements, he looked at the surrounding context and problem. The gun was fine. So he spent a couple days redesigning the box and throwing it out of airplanes. He solved the problem by reframing the _problem_. This is a play every good designer should know. Design is just as much about problem _finding_ as it is about problem _solving_.

PATTERNS



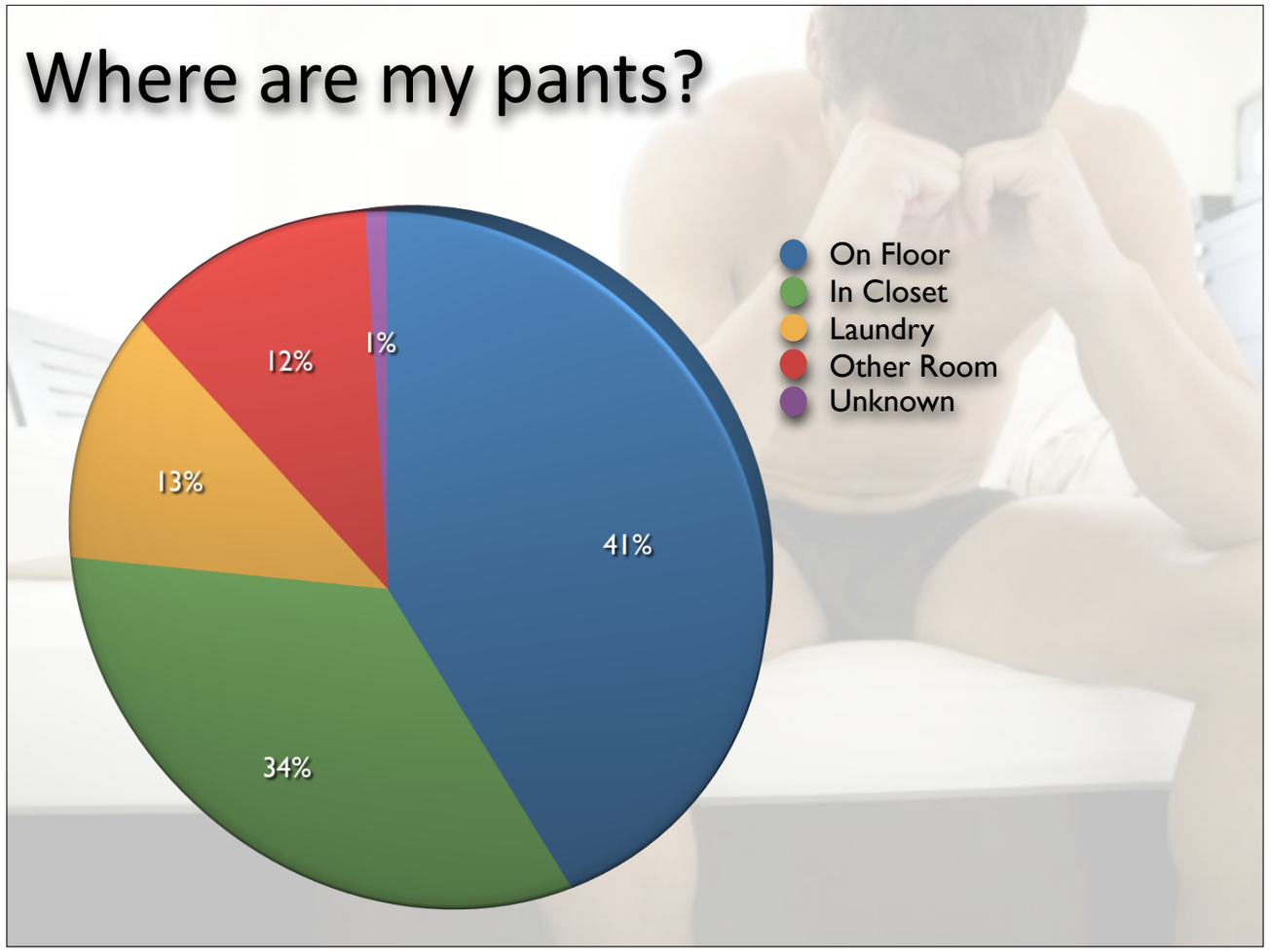
Learn,
Recognize,
Use

they help us manage complex things, make predictions and answer common questions.

Once you've encountered something—either a tricky problem, a bug, or usage pattern—try to see if you can spot a pattern. Next time you encounter something like it, you are more likely to recognize the pattern and apply an appropriate solution. They help us manage complex things, make predictions about what's going on, and answer common questions. Questions such as…

Where are my pants?

They could really be anywhere. Up a tree! In Denver! Nailed to a door! Maybe Vladimir Putin annexed them. Or launched into space?



In reality, there are likely spots where you left them. For example, most of the time they're either crumpled up on the floor, or hanging on the closet. Maybe they're in the laundry, or the other room, depending on what you were doing last night. It is pretty rare that they aren't in one of these spots. So you have a pattern and a history that you leverage. You will find there are patterns in software that will help you as well.

We've got patterns!

Factory - Adapter - Memento

Thread Pool - Visitor - Flyweight

Blackboard - Monitor - Encapsulation

Iterator - Mutator - Template

Observer - Decorator - MVC

Loose Coupling - Prototype

Singleton - Tail Recursion

In fact software developers have formalized a lot of these patterns and given them names. All of these things on the slide are shorthand for a fairly specific technique that addresses one or more problems. Some of them introduce problems, but hopefully fewer than they solve. Learn to identify, use, and talk about patterns in software and you'll be much more effective on your own and in groups.

Program Defensively

When you write code you should assume that you or your best friend will have to maintain it: finding bugs, fixing them, looking for opportunities to improve, and so on. You can't do this if you do it quickly and barge ahead without regard for tomorrow. Program defensively! Just like defensive driving is the best way to stay alive on the road, defensive programming is the best way to write fantastic code.

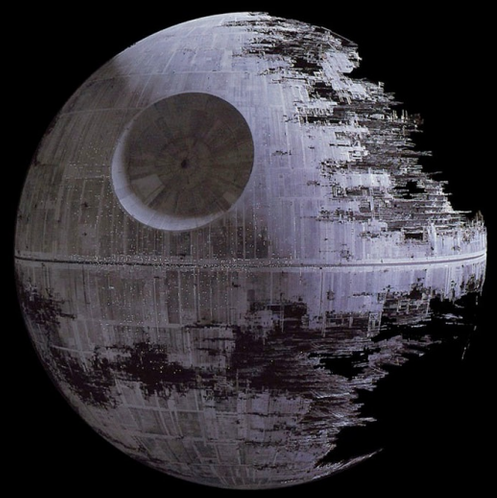
Eisenhower once recalled an old Army saying: “Plans are worthless, but planning is everything.” This means that you can get a lot of mileage by systematically imagining what would happen if you did this, and then that happened, and so on. In software, this means that you write code on purpose. And I’ve seen a lot of code that looks like it was written by accident. Plan ahead. Be purposeful.

FUNCTION

Good code →has← purpose.

Another problem with code is when you write that has no functional purpose. Good code can be artful, sure, but it should have a purpose. Otherwise it is just taking up memory, processing time, and (probably most importantly) the programmer's time and attention. And I'm not talking about computer art projects (those are great!) — I'm talking about 'dead code', and code that is way more complicated than it needs to be to do its job.

EFFICIENCY



“There is nothing so useless as doing efficiently that which should not be done at all.”

— Peter Drucker

Another way of putting it is this one from Peter Drucker, who was a management professor. There’s nothing so useless as doing efficiently that which should not be done at all. Programmers LOVE to write efficient code. Sometimes they write magnificently efficient code that _does nothing useful_.

Write Code Incrementally

When you're writing code, do it incrementally. Write a little bit, then test it. Some people use engineering techniques like unit testing to do this. That's good, it has its place. Other times, it is usually good enough to just have it print something out, so you can sanity check if it is working as you thought. The important part here is that you write little bits at a time. That way when there's a problem you have a good shot of identifying where it comes from since you likely just wrote it.

A line of code should do → one thing ←

Instead of:

```
position += heading++;
```

Consider:

```
position = position + heading;
heading = heading + 1;
```

On the topic of writing incrementally: I've found the best way to screw something up badly is to have a line of code that does a dozen things. Don't do that! You do not win points from the elite hacker society by making your code hard to read. Make it as easy as possible to understand what your code is doing by making each line do a single thing. (For those of you who understand C, convince yourself that these two things are the same. Or not.)

The best technology in the world was flawed, sometimes deeply, when it started getting used. And even deeply flawed things can be useful for creating other things. Besides, “good enough” is good enough. It depends on what you’re doing. Medical devices or missile defense code probably has a higher standard than the next social app.

It is good to have lots of tools. When you’re off on a project, especially if it is a super secret corporate thing, or if you have a startup, you might be the only one in a position to address a problem. Or build a new thing.

This means:

→ Know many languages
→ And paradigms
→ And patterns
→ And debugging strats
→ And libraries ...and much more...

Learn lots of tools: lots of languages, frameworks, editors. Work at high levels (web apps?) and at low levels (arduino or embedded systems). You'll find that they all have their place and can teach you a thing or two about things you thought you had mastered.

Last Thing.

PEOPLE.

One more thing. People!

Who is this?

You probably recognize the guy in the middle. That’s Jimi Hendrix. But he wasn’t a solo act. He had a band, those two guys supported him. Hendrix would likely have been a phenomenon without these two guys specifically, but he still needed a band.

We tend to idolize individuals while ignoring all those who contributed to their success. Great things happen when teams work together. This is a skill that must be learned.

Self-reliance: awesome!

Being incapable of collaboration: terrible!

So while self reliance is pretty awesome, it should not come at the price of being unable to collaborate.

So Long!
It was real!

I hope you win!

That’s all I know.