SWE 265P
Reverse Engineering and Modeling

Lecture 5

Reality

"My personal strategy is to formulate hypotheses based on heuristics [largely names of things] and then test those hypotheses with print statements." – Crista Lopes [Professor, UC Irvine]

Reality

"So use this documentation as a starting point but definitely you will need to verify it's accuracy." – Alegria Baquero [Senior software engineer, Zocdoc]

Today

- Last week's material
- Key expert practices
- Mental simulation
- In-class practice
- Midterm preview
- Christopher Keller (Consultant Alliance)

Last week's material

- Key expert practices
  - focus on the essence
  - go as deep as needed
  - work with others
- Structural versus behavioral models
  - UML class diagram notation
  - call graphs
  - UML sequence diagrams
- Any questions?

Reminder that reading code is deliberate, and involves being a reflective programmer: on the one hand being engaged with the task, on the other hand observing yourself in how you engage with the task and ensuring that you go about it the right way. Throughout the course, we have been attempting to teach you this. Remember the templates? They are a way of 'slowing you down' and making sure you follow a good process, take good steps, and be very deliberate. In so doing, you learn how to do so eventually without the templates. But you need that template to watch over you. Ditto with last week's 'expert practices', they actually provide you more the reasoning behind having a template, i.e., to train you to become an expert, as well as others ways in which you are to behave when reading go. So let's review the expert practices from last week:

1. Focus on the essence – go for the 'core' of the system, not the cruft around it, to understand what it does and how it achieves what it does [this is what was the point of the homework, where is the essence of your system?] Doing so helps you figure out the big picture, and also is the most direct route to understanding 'how it works'.

2. Go as deep as needed – don't be content with a superficial level of understanding that 'looks right', but convince yourself that you are right [this is what we also asked you to practice, by way of asking you to explain to someone else all that is involved; we will do a lot more of that today]

3. Work with others – don't go it alone. Of course you will at times, but more often

than not, you will want to work with someone – synchronously or asynchronously – to really deepen your understanding of the code. Being forced to externalize your thoughts, hearing other people's perspectives and domain expertise, and the many eyes see many different things benefit are crucial to accelerating your ability to really understand the code.

We then switched back to the topic of externalizing mental models, and particularly reviewed the UML class diagram notation in detail. We made the observation that it is an essential notation that everybody uses, though generally not as advocated (up front design), but much more so as a shared language for documentation of an existing system and how it works or a summary of a design session on the whiteboard, being able to communicate when drawing on a whiteboard, etc.

We also learned that UML class models are just one type of model. UML has many. Beyond UML there are many more. What is important is that these models break apart into two kinds: structural models [documenting the static pieces of a system and how they relate] and behavioral models [documenting how aspects of the system interact]. We showed how different models show different levels of detail, from UML class diagram just being the static structure, to forward/backward chaining with IntelliJ showing individual downstream/upstream relationships, to call graphs showing exactly which methods in which classes call which other methods in which other classes (and do so in full for the program), to UML sequence diagrams which show the sequence of invocations in a very nice visual way, and starting from anywhere in the program. In essence, it shows 'one path' in full, and can only go downstream.

And, of course, many other models exist.

Any questions on your part?

Last week's homework

- Which two features did you decide are essential to your system?
- How difficult was it to locate two essential features?
- How difficult was it to explain the two features to someone else?
- How easy does it think it is for someone else to understand the two features with your packet?
- Any questions?

With respect to the homework that they were given, it is important to spend some time [probably 30-45 minutes or so] reflecting on the exercise thus far; this is perhaps the most important one, also the most nebulous one (what is 'essence'), and the most foundational one. So debriefing is key. All teams are bringing a copy of their document and we should be able to chat with them.

So, then, it is good to have a discussion with the class about this experience. Sample questions are on the slide, but as desired, the discussion can stray from it, of course. There's a purpose to the sample questions (see below). Not every team should have to answer each question, of course, but each team should at least get a turn or two or three.

1. Which features are essential? [Follow-up questions] What makes these two features essential? What other features did you consider? How do you recognize/decide upon among all of these features which ones were 'more essential'? Are these two features different from the two you had last week? Why? Why not? Now that you understand these two features, how do you feel about your understanding of the system versus the understanding you had last week?

It is really important to challenge them on their decisions for 'essential' features. Part of the discussion has to be to, on the spot, discuss hypothetical other features

that may be more foundational. For instance, some have a database system and may choose 'add table' as essential. We can challenge them and ask about two phased commit. Others might have a game engine as a system and may choose 'build map'. We can challenge them and ask espabout the actual running of the game – how does the game loop operate? Etc.* This part, I feel, the questions should 'stop by' every single team to hear and give feedback.

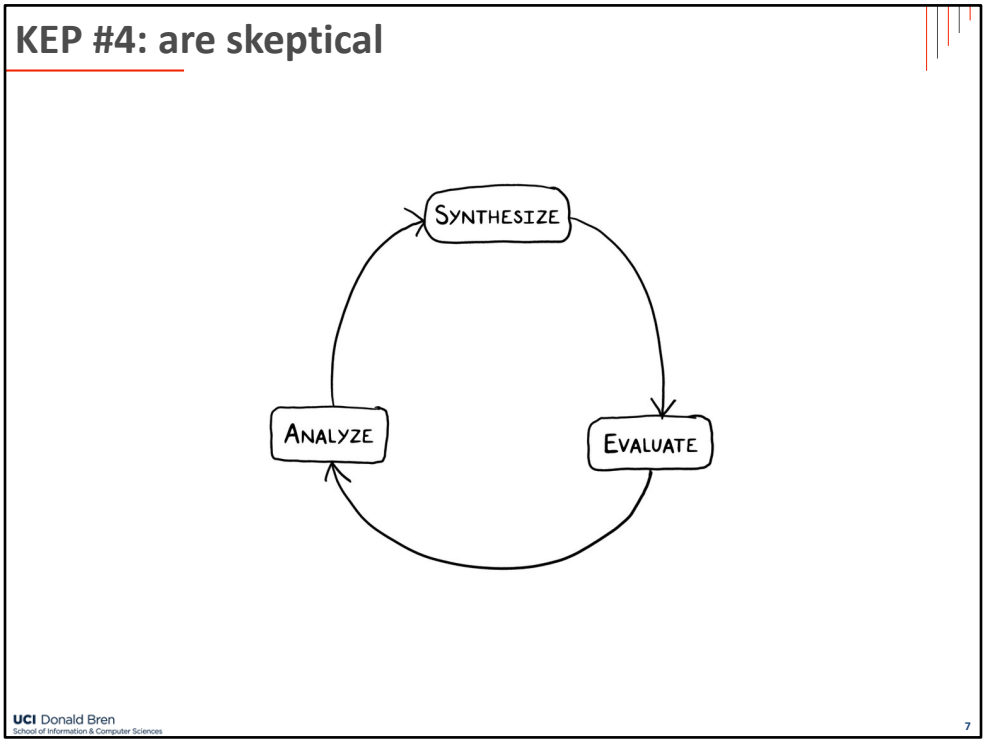
2. How difficult was it to locate the features? [Follow-up questions] Why was it difficult? Was it different trying to find these features compared to the features from last week? How? How spread out were the two features over the code? What tools did you use (templates? UML diagram? Call graphs? Sequence diagram?)? Why? What other 'tactics' did you use to locate the features (did you look on the web site? Read the documentation? Read posts?)? [Essentially: it should not have been an easy exercise, if it was, you are likely wrong; also, the 'tools' you have been provided in the past lecture are not perfect, by any means; also, it is highly unlikely these features were just 'self-contained' in a single spot]

3. How difficult to explain? [Follow-up questions] How difficult was it to put your mental model into something that someone else could understand? Did you use models to try to communicate certain aspects? Which models? Why? How did you feel about having to explain the features to a hypothetical other person? What is your level of confidence that (Kaj) can understand what you are trying to communicate, in full? What are you worries in this regard? How did having to describe change your level of understanding of the features? Did it? Did it not? What exactly did it change? The scope (i.e., where is it implemented); how it actually works; your understanding of the role it plays in the overall system; it was not essential after all and you switched to something else; ...)? [Essentially: this exercise should have increased their understanding, by a lot; if not, they are not taking the whole thing seriously ☺!]

4. How easy? [Follow-up questions] Throughout, did you put yourself in the shoes of the other person (Kaj)? How did you do that? What did you imagine? What did you assume? How did that impact your packet? [Essentially: remember, the other person knows nothing, so how much context are you giving them… Also, by placing yourself in the shoes of the other person, I bet your packet changed. In what ways?]

Overall: what have you learned from this exercise? What is the one key lesson for you? What would you do different next time around?

Any questions on your part?



A reminder, once more: When working with code, reading it, changing it, designing it, etc., there are a number of things that the true experts engage in that distinguish their behavior – and thus their results – from others. In this course, I will call them expert practices and I will introduce them as we go along. Last week we did three. This week, another three.

The key once more is not just knowing them, but practicing them. It's a huge difference. Can you, in the moment of deeply being engaged in work, be present twice: once doing the work, once reflecting on your engagement with the work and steering yourself in proper ways? Think about your homework – how much did you really stick to what you have been taught, versus how much did you divert and 'forget'? It's alright if you digressed, it just says you have to be extra careful – and that is where reflection comes in. Can you think about your work while you work? Reflective programmer.

Are skeptical: they kind of don't believe what they believe until they see appropriate proof. So when they think a line of code does 'X', they find ways of figuring that out (print statement, debugger). When they think it works under normal circumstances, they are skeptical that it does under non-normal – and will try to prove to themselves

that it nonetheless works. Etc. How do they do this, they analyze the situation to form a goal ("gotta understand <this particular> bit of code", then make progress (<let's read it>) and then evaluate (adding print statements, asking around, debugger, etc.). And they do this over and over again. This, btw, is our normal thinking cycle.

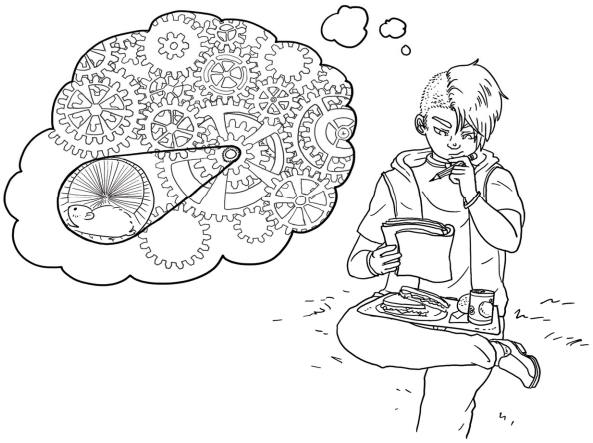
There's other ways in which they might do this too.

- to convince themselves that something works as it looks like it does (very often, it does not) [Crista: hypotheses!]
- to study for potential side effects they may not be aware of
- to not gloss over detail that may matter
- to externalize thought, but in a very different way than just documenting

Overall: because their experience has been that things are never as they seem, so they are skeptical when things look too easy, too clear, too straightforward, … A significant chunk is too that they don't trust themselves and the conclusions they draw. They know they have blind spots. Etc.

Useful audience engagement: did you have any mea culpa moments in your homework, where all of a sudden you realized it wasn't the way you thought it was?

KEP #5: simulate continually



One way in which they address being skeptical is by simulating continually.

Simulate continually: when engaging with a code base, and as they build their understanding, experts continually 'mentally run' the parts they are trying to understand. They go through the code step by step, or method by method, or line by line, or invocation by invocation, essentially checking/verifying/debugging/adding to their understanding of what the code does. And they don't just do it once, they do it continuously. With every new bit of understanding they attach it to their existing mental model, perhaps updating aspects of it because they gained some depth, etc., and then 'running the code again', and so on and so on.

Why do they do this? Because…:

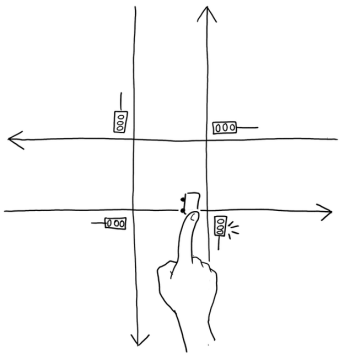
1. To close gaps in their understanding – they may know what it does, but not exactly how; or they may want to know what other utility routines are being leveraged; or…
2. To begin to understand where in the code they may need to make changes to create a difference in behavior. Such an understanding has to be at individual lines of code, but also
figure out 'where to intercept' the current programmatic behavior and alter it into the newly desired programmatic behavior. For one to create a new behavior,

knowing the
current behavior in detail is critical.
2. The devil is in the details, and small mis-understandings overall add up. So, they go as deep as needed, and mentally run the code.
3. It allows them to test whether the diagrams they are working with [if they have them] are correct, and overlay their understanding, thus creating a mapping from diagram to code. These mappings
will serve as future indices. Often, the model is marked up with where future changes may need to occur.
4. It allows them to verify whether the documentation [if they have any] is correct, and overlay their understanding, thus creating a mapping from documents to code. Often, the documentation is
marked up with where important detail and context is, and where it may need to be updated. [Alegria's comment]

Useful audience engagement: to what degree did you find yourself doing this (informally) as part of the homework?

KEP #6: draw examples alongside their diagrams

```
IF (CAR APPROACHES INTERSECTION)
  IF (INTERSECTION.LIGHT IS RED)
        CAR.SPEED = 0
  ELSE (INTERSECTION.LIGHT IS YELLOW)
        SPEED += 10
```



Draw examples: so when doing mental simulation, it is important to externalize our thoughts. We have seen ways in which we can build models that help us with that, particularly call graphs and sequence diagrams. Experts, though, go further: they actually will put real values next to those diagrams (or, next to the code, or real values next to a representation that is more in the problem domain). Why?

1. Mental simulation is running the code, and running the code always is done with concrete examples. So the values represent those examples, and the effects the code has on the values.

2. Nothing beats the ability to link 'visual' behavior to lines of code; after all, the code represents a use case (or many), and that use case has to be completed successfully.

3. Externalizing the examples often helps in articulating what the code should be doing, and identifying potential problems with/limitations of the existing code.

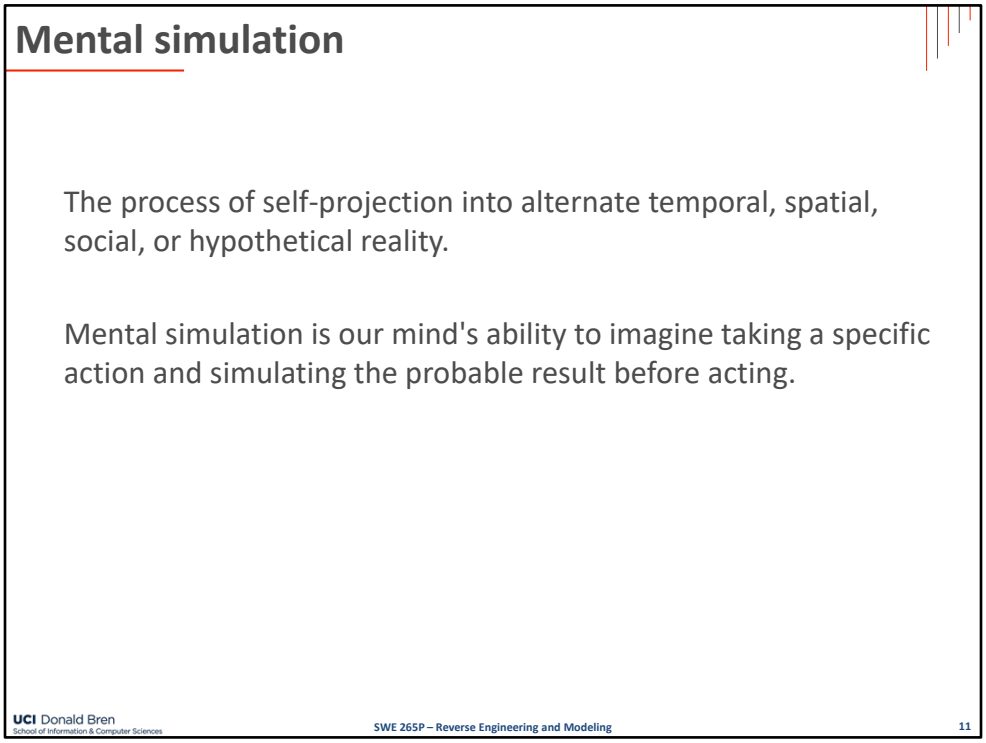
4. Externalizing the examples helps one understand with what other values the code may need to be simulated.

Important frequent question #4

How does this code work?

To guide us in the remainder of this lecture, and particular 'mental simulation', we introduce IFQ #4: how does this code work? This is probably the one question that gets asked the most, and that is of course not a surprise.

It's as simple as deceiving a question. It might need to be asked more like 'How does this code really work?' or 'truly work' or 'work in ways that I did not think it could' or… Basically, the need to know how some (typically smaller part of the system) works in detail. This clearly relates to the three expert practices we just talked about, and especially the reasons why developers engage in them. Long list of why and when we want to know how a piece of code really works. Most often – because we need to change it.



Attribution
first definition: https://www.ncbi.nlm.nih.gov/pubmed/25603379
second definition: https://personalmba.com/mental-simulation/

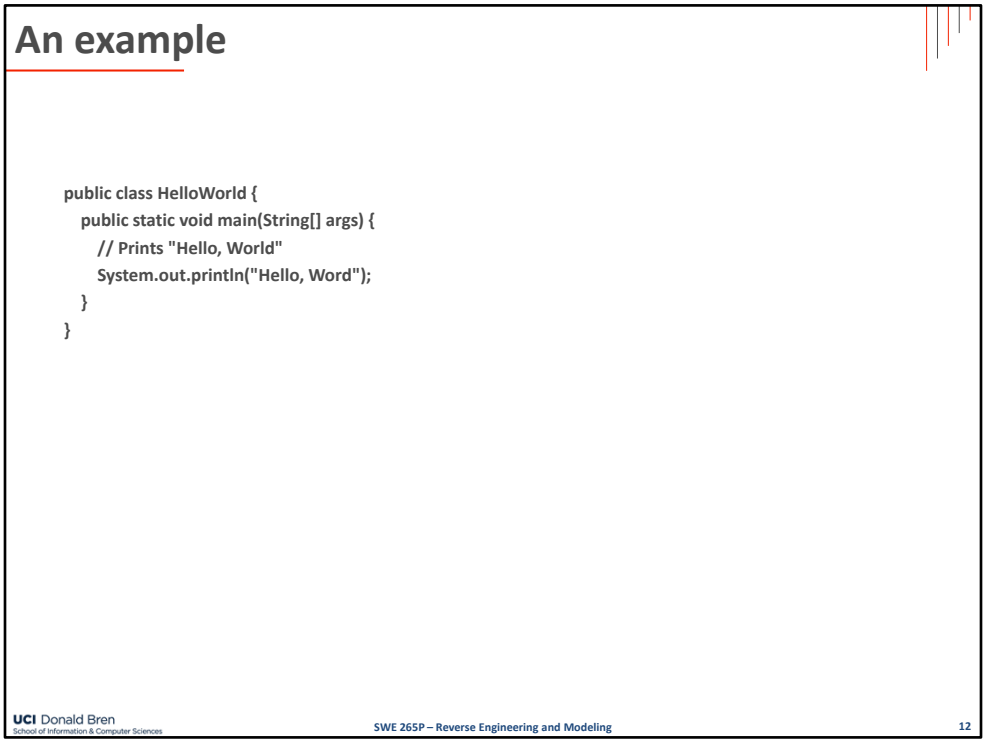
At the basis of this lies, of course, reading code. But when we do so, we do something more. As we have already seen, we form mental models. As we have also already seen, many models exist, across structural and behavioral. The mental models that we tend to create on one hand exhibit structure (which classes relate to which) and on the other hand behavior (which one calls which one when). But often we go even further. We mentally, in our minds, build operational models that move from the abstract to the concrete: we 'run' the program mentally using specific examples. This is what we call mental simulation.

It is useful to actually define mental simulation. Here, we present two.

The first is from the world of psychology, and basically points to the human ability to 'anticipate', 'dream big', 'fantasize', 'forecast what may happen', etc. This, of course, is what we do on a daily basis. Anticipating what may happen in traffic and how we might respond. Or thinking about an upcoming meeting and what someone might say. Or looking at a particularly gnarly puzzle or knot, and mentally solving or untying it before really starting that process, because wrong moves can have

significant consequences. In all of these cases, we think before we act. We anticipate rather than only react. [Of course not always, there still is plenty in the moment, but it - mental simulation - is a basic human capacity that we have and use all the time.]

The second definition is not that different, but takes a somewhat broader perspective – it is less about ourself (i.e., 'self-projection' from definition #1) but more so about the consequences of actions we might take. It is a slight reframing, but it is useful for our purposes. When it comes to code, indeed, what this definition allows us to say is "we read the code, and anticipate what it does, without actually taking the action of running it". So that is, we are anticipating what the code does, imagining what it does, based on the code only, not based on compiling and actually running it. Developers do this *all the time*.



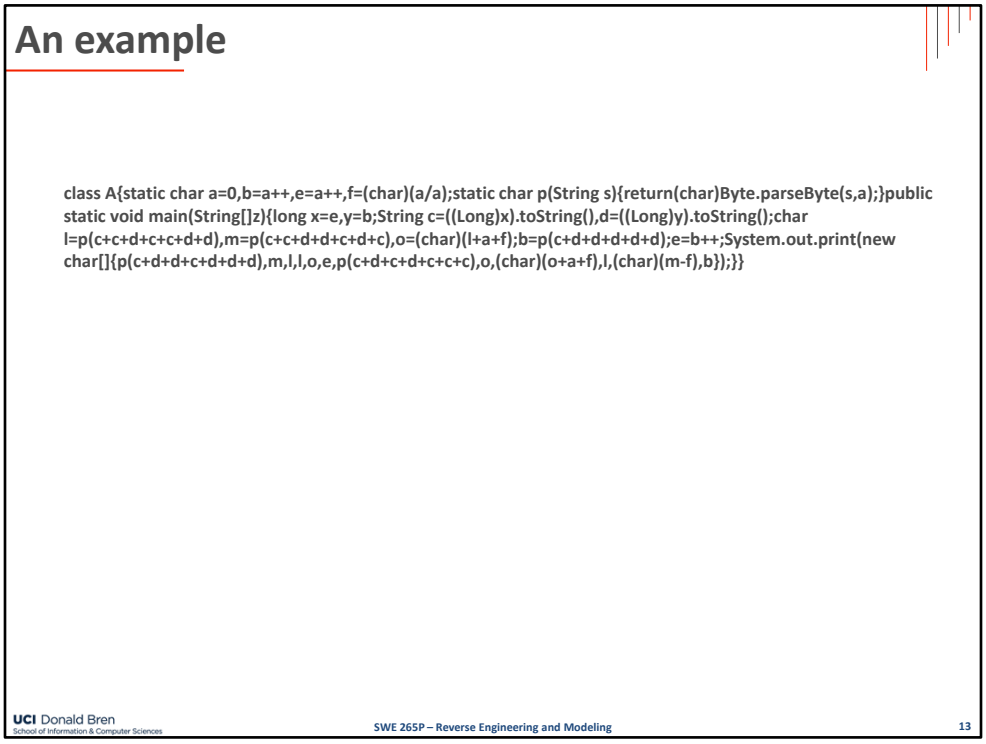
Credit:
https://introcs.cs.princeton.edu/java/11hello/HelloWorld.java.html
Sedgwick & Wayne.

So let's look at an example. What does this bit of code do? [It may be interesting to do this as a 'who can tell me the fastest']

Some folks may be tricked into thinking it prints "Hello, World", though in reality, the print statement actually says "Word". If we are lucky, everyone figured out it does not quite do what is documented.

The point here is that we need to actually go slow, simulate, and read the code in detail. The point, too, is to never believe the documentation. Be skeptical.

Do mental simulation! What could you have drawn next to the code to help you?



Credit:
https://codegolf.stackexchange.com/questions/22533/weirdest-obfuscated-hello-world/22628#22628

Sooo… what does this program do? Anyone? Without running it? Why not? If anyone guesses 'Hello World', let's ask them what they base their guess on. Is it Hello World? Or Hello Word? How do you know?

[Meta comment: Poking a little fun at them here. The formatting is really crazy and is not helping us making any sense.]

Key observation: there's little we can do but run. Trying to dissect this is tricky. Though, we can ask how to do that. What are their thoughts about making this more accessible? Someone might say 'let's use pretty printer'. Let's do so.

An example

```
class A
{

  //initializing some constants needed
  static char a = 0, b = a++, e = a++, f = (char) (a / a);

  //shorthand for parseByte() (codegolfing handyness)
  static char p(String s)
  {
    return (char) Byte.parseByte(s, a);
  }

  public static void main(String[] z)
  {
    long x = e, y = b; //needed for some weird reason to save bytes
    String c = ((Long) x).toString(), d = ((Long) y).toString(); //creating a 1 and a 0 a string
    char l = p(c + c + d + c + c + d + d),  //binary digit voodoo
    m = p(c + c + d + d + c + d + c),  //more commonly used letters prepared
    o = (char) (l + a + f);
    b = p(c + d + d + d + d + d);
    e = b++;
    System.out.print(new char[]  //assembling the string with arithmetic and more binary stuff
    {
      p(c + d + d + c + d + d + d)
      ,
      m, l, l, o, e,
      p(c + d + c + d + c + c + c),
      o, (char) (o + a + f),
      l,
      (char) (m - f), b
    });
  }
}
```

https://github.com/SWE-265P/obfuscated

Credit
https://codegolf.stackexchange.com/questions/22533/weirdest-obfuscated-hello-world/22628#22628

Here it is. Pretty printed (and magically with a little commentary now too). Are we in a better state now? Actually, let's play a little bit with this program. I will give it away: it prints 'Hello World'. That's fantastic!

But… you have been tasked to change it to print 'Hello Word'. How do you want to go about it? Running the code helps little. You could try to use the debugger, but save that for a later lecture.

Let's mentally simulate, using drawing examples next to the code. So, can you get together in small groups of 3, grab a copy of the printed code [Andre printed 35], and go from there? Step through the code, and annotate it.

Let's do this for five maybe ten minutes.

Are you progressing? What were you drawing alongside? Are you running into trouble anywhere? Are you desperately wanting to run individual lines of code?

My hypothesis is that you are [because you really want to know the very detail of Byte.parseByte() or what b=a++ does. You are wanting to inspect /verify individual values – to check your assumptions and be skeptical that it works. Because the code still is not so clear.

So let's go ahead, and actually have you add some print statements to the code. Download the example from the class GitHub, and talk among your group about which print statements you want to add. Go ahead and add those, run the code, and see what you are starting to learn. How does this code really do its job? Where is the 'essence'? Where is the magic? Can we answer? [Let's spend another 10 minutes on this.]

Moral of the story – mental simulation needs to be externalized [by drawing examples] but also has its limitations when the code is too unreadable, complex, unfamiliar, strange, … In such cases, mental simulation needs to become a mix of mental simulation and actually running the code – with the help of print statements (or as we shall see in a future lecture, the debugger). That is how programmers most often go about understanding code. Why are they still using print statements? Because it forces them to think about what they want to print and why. That is different from the debugger that gives you everything. By being forced to think, you are forced to create hypotheses (see Crista quote!) and verify them by choosing what to print.

Template

Line number	Values of key variables	What changed?	Why?	What am I unsure of?	Why?	Notes

Here's part a potentially useful template: as we progress line by line, we should keep track of the values of key variables. But we do so by noting when they change, why, and what we still do not really know.

As with previous templates in class, the why is really important – it is the question that forces us to reflect. The unsure part is also important – it allows us to express 'alternate realities' that may happen if our assumption does not hold, and the value changed in a way we did not anticipate.

The last field is for us to make any notes that we see fit "boy this was hard to digest" or "there's more to it than I can get right now", "need to return here after I look elsewhere", "definitely here", etc.

Clearly, meant to be updated as you go, and to revisit old parts as you go, so best on a piece of paper or whiteboard.

This, btw, can work really well with print statements – they inform what you need to write down here. Similarly with the debugger. Writing down key bits what you are finding with the debugger on a template like this can serve as your external memory for the complex dynamics that might be going on.

JPacMan4

- By what rules does the ghost Clyde move?

https://github.com/SWE-265P/jpacman4

[FOR THIS TO WORK, THEY SHOULD UPDATE THEIR CODE B/C JPACMAN3 GOT AN UPDATE FROM THE TA, TO REMOVE THE COMMENTS from INKY, so they all need to sync]

Scenario: not uncommon – you want to understand how one of the ghosts (Blinky) moves, perhaps to verify it does it according to spec, or perhaps just out of pure interest, or perhaps to augment the behavior. Fortunately, you have the comments and the code!

So let's have them do this for a little while, with the template. The basic question for the students: does this code do what it says it should? No running the code

LESSON EMBEDDED IN THIS EXERCISE:
- walking through the code meticulously
- actually it does work as advertised (hooray)
- but you really should update the comment on line 87 and 89 to reflect the variable SHYNESS (!) Leave the code a better place than when you got there…

Questions:
how did you feel about this exercise?

what was surprising and/difficult?
how confident do you feel that what you figured is actually what it does?

JPacMan4

- By what rules does the ghost Blinky move?

Scenario: not uncommon – you want to understand how one of the ghosts (Clyde) moves, perhaps to verify it does it according to spec, or perhaps just out of pure interest, or perhaps to augment the behavior. Fortunately, you have the comments and the code!

The basic question for the students: does this code do what it says it should? No running the code, just reading and mentally simulating.

LESSON EMBEDDED IN THIS EXERCISE:
- walking through the code meticulously
- but wait, this is the exact same code as Clyde
- and note the comment: @TODO – welcome to a mini reality in programming. Stuff to do. But oh so nice they flagged it!!!

[Take at most 2 or 3 minutes with this slide…]

JPacMan4

- By what rules does the ghost Inky move?

Same approach, though they should feel free to actually run the code. Can you tell how it works?

LESSON EMBEDDED IN THIS EXERCISE:
- there is no commentary, so now you are on your own
- the template does not help enough – unless you use a template per 'scenario' – still need to abstract up from the findings of the templates
- choosing multiple examples to work through
- did anyone draw a map of PacMan to support their mental simulation?

- how hard was it to create the abstract, human understandable rules from the code?

Midterm

- Next week
- About 1 hour and 30 minutes to 2 hours (still to be decided)
  - brief lecture afterwards
- Covers all material to date
- Theoretical and practical
- Closed book

Sample questions (theoretical)

- What is a mental model?
- List three important frequent questions.
- What is a call graph [draw a small example]?
- What is the difference, in UML class diagrams, between association, aggregation, and composition?
- Discuss two reasons why experts work with others?

Sample questions (practical)

- Draw a UML diagram accurately representing [some classes].
- For [some piece of code], explain how you would apply opportunistic comprehension, and illustrate it by annotating the code.
- For [some piece of code], what would be a good beacon? Explain why.
- For [some piece of code], explain the approach you would take to go 'as deep as needed'.
- Explain what [some piece of code] does.

Midterm additional notes

- These are sample questions only
- Bring your laptop and make sure IntelliJ and the plug-ins we have practiced work properly

Homework (individual)

- How to quickly and effectively read other people's code
  - https://selftaughtcoders.com/how-to-quickly-and-effectively-read-other-peoples-code/
- How does one begin to tackle understanding a large open source code base?
  - https://www.quora.com/How-does-one-begin-to-tackle-understanding-a-large-open-source-code-base
- Learn to read the source, Luke
  - https://blog.codinghorror.com/learn-to-read-the-source-luke/

Homework (individual)

- Make sure to regularly update your personal diary, including an entry for today's lecture

Break

And now…

- …welcome Christopher!