Software Development

Basic Processes

What basics?

- What programs are
- How we design programs
- Types of programming languages

Software

Program or collection of programs
that makes the computer do something.

We're in the business of software development. In this class you'll learn the very basics of how to begin developing quality software. Remember that software is made by people, and we do make mistakes. We want to make as few of those as possible though.

Program

Set of instructions that tell the computer exactly what to do, in which order.

Computer Programs

Must be loaded into physical memory
in order to “run” or “execute.”

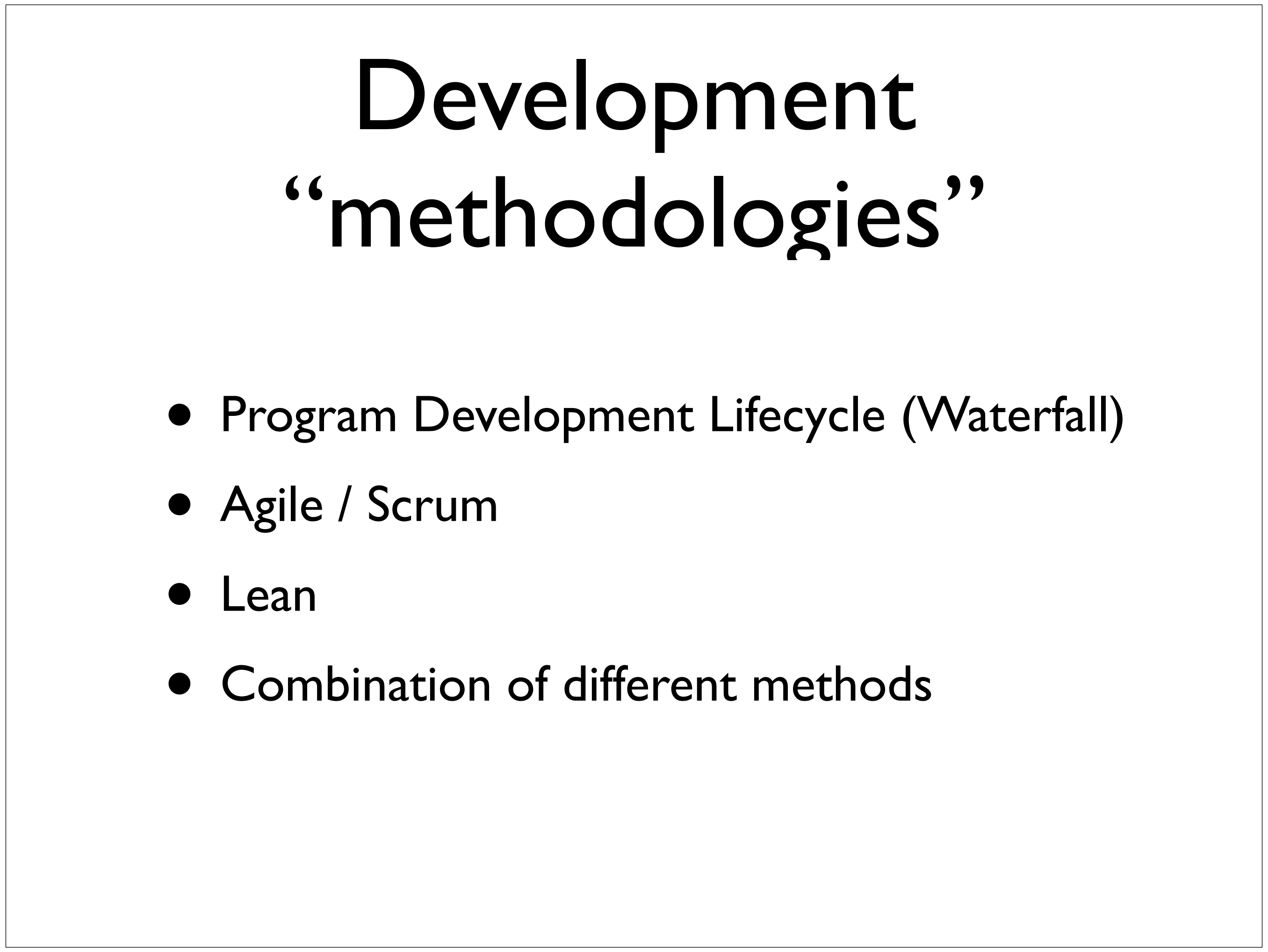
```
document.observe('dom:loaded', function(){

  var name = $('name');
  var newMarkup = new Array;

  name.innerHTML.strip().gsub(' ', '').split('').each(function(c){
    newMarkup.push( ['<div>', c, '</div>'].join('') );
  });

  name.update(
    newMarkup.join('')
  );

  $$('#name div').each(function(div, i){
    div.setStyle({
      fontSize: 600 / i + 'px'
    })
  });

});
```

The more memory your computer has, the more programs can be loaded at the same time. The more CPU power you have, the faster you can process program instructions.

Basic instructions

- Input
- Processing
  - Math
  - Logic
- Output
  - Storage

Developing Software



Development “methodologies”

- Program Development Lifecycle (Waterfall)
- Agile / Scrum
- Lean
- Combination of different methods

There are many ways we can design and develop software. Here are just a few of the more popular ones.

Program Development Life Cycle

- Traditional and widespread approach to software development
  - It might take a team to get the project done
  - Programs can get very complex very fast
  - Programs, files, databases, etc must all work together.

The PDLC is a traditional way of developing software that comes from traditional manufacturing processes. Programs get complex fast and we have to design them, test them, and make them work together. There are five steps and we're going to go over them and then we're going to look at how using this method can get us in real trouble if we aren't careful.

1. Define Program Specifications

- Obtain a complete description of the problem to solve
  - Define the input
  - Define the processes
  - Define the output
- Understand user requirements

First, we want to design the program's specifications. We have to figure out what the program is going to do and what the users want it to do. That means defining the inputs, processes and outputs.

2. Program Design

- Define input and ouput (top-down design)
  - Identify any data the program needs to obtain from a source outside the program
  - Identify anything the program should produce; reports, screen of information, invoices, data file, etc...

2. Program Design

- Determine the major tasks needed to solve the entire problem
- Plan the logic and steps needed to solve the problem and major tasks
- Define the sequence of operations to be performed to solve each major task and then the whole problem

Design Methodologies

- Procedural
- Event-driven
- Data-driven
- Object-Oriented

Different types of problems require different types of programming approaches

Procedural design

- Focus is on the process
- Every time the program runs, the process follows a sequence of steps from start to finish.

Procedural (top down)

- Start with big picture
- Break down into smaller steps

The most common way that beginners approach programming is the top-down procedural model. They look at the big picture and then try to break it down into steps. That’s where we’ll spend a lot of our time in this class. We keep breaking things down until we have all the pieces.

Procedural (Modular)

- Group tasks into modules
- Design each module separately
- Connect with top-down design

Another way to do this is to divide up the tasks we have to accomplish into modules. We look at the major things the program does and design them as modules, then connect them together. So instead of breaking the big thing down further and further, we just divide things up. This works for larger programs that have many many steps, where we may even need to break them into smaller programs.

Event-driven

- Focus on the event
- Programs respond to events
  - Code is attached to these events
- Clicking a button runs code.

Event-driven design works for situations where we want a program to respond to an event. The focus is on an event or something that happens. Think about a Windows app. When you click a button, something happens. We tie code to those events. Instead of the process just happening when we run a program, the process happens in response to an event.

Data-driven design

- Focus on the Data
- Analysis of data and relationships between data
- Look at outputs
- Figure out what processes convert input data to output

Data-driven design is common in situations where there's lots of data processing going on. Database administrators love this approach. They look at the data and its relationships, then look at the outputs, and then figure out how to make that happen. This is common in querying, reporting, etc.

Object-oriented design

- Focus on the Objects (things) in our system
  - Customer
    - has properties (things it is)
    - has things it can do
  - Objects talk to each other by passing messages.

Focus on the real thing we're building. Like a customer. We think about all the things a customer can do and things it is, and all the code that goes with the customer is bundled up. Object oriented program is something we'll talk briefly about at the end of the course, but it's something you'll do in the rest of your courses because it's very popular, very powerful, and is perfect for larger, complex systems that need to be maintained and expanded.

Choosing a methodology

- The problem you're trying to solve
- The technology you want to use
- Your past experience

We'll use Procedural methodologies in this class.

And some Event-driven.

Procedural Example

I have to take attendance

I open the class roster web page that has the students names on it. I want to know who is here and who is missing. So I read each person's name off the list and they tell me if they are here. If they respond, I mark them present, and if they don't respond, I mark them absent. When all the students are done I submit the roster page, closing the roster.

In this class we'll do most of our development using the procedural approach. Our programs will be small, and we want to focus on a few important concepts. And once you have those concepts down, you'll move on to object-oriented design.

Structure Theorem

- Sequencing (putting things in order)
- Selection (Making decisions)
- Repetition (Doing things over and over)

Computer programs do at least one of these three things.

Planning the Logic

- Develop the logic to get from the input to the output
  - What are the steps in the process?
  - Do I need to make any decisions to solve this task?
  - Does the task (or any part of the task) need to be repeated?

So we apply the structure theorem. What are the steps? What decisions need to be made?

Inputs, processes, outputs

- What are the inputs?
- What are the processes?
- What are the outputs?

Start by figuring out what goes in, what needs to be done, and what comes out of the program.

Do I need to make any decisions to solve this task?

Do I need to make any decisions to solve this task?

Yes! If they say they're here then I mark them as attended.
If they don't answer I mark them absent

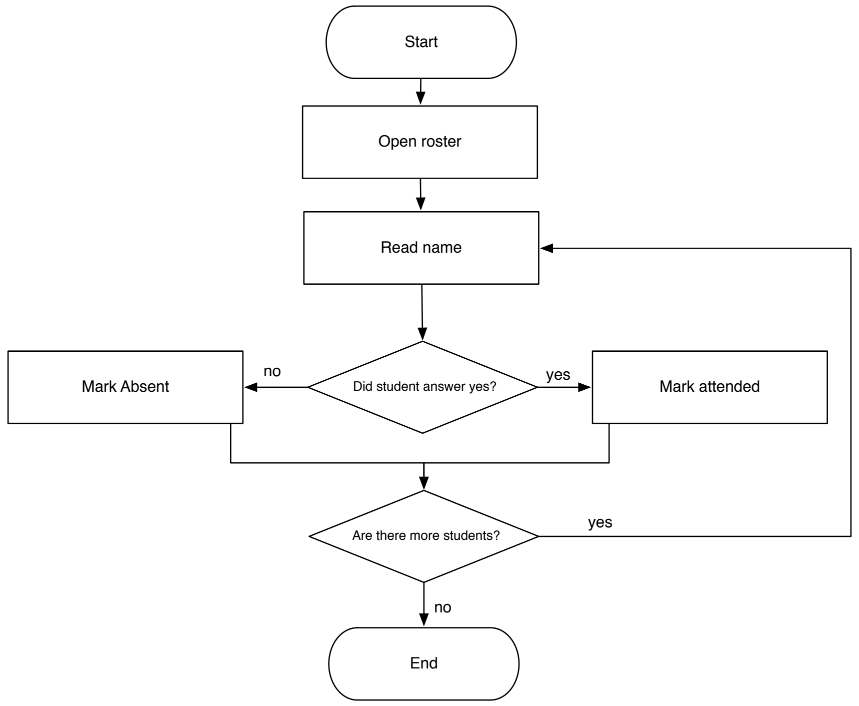
Do we need to do something repeatedly?

Do we need to do something repeatedly?

Do I have to repeat the process for more than one student?

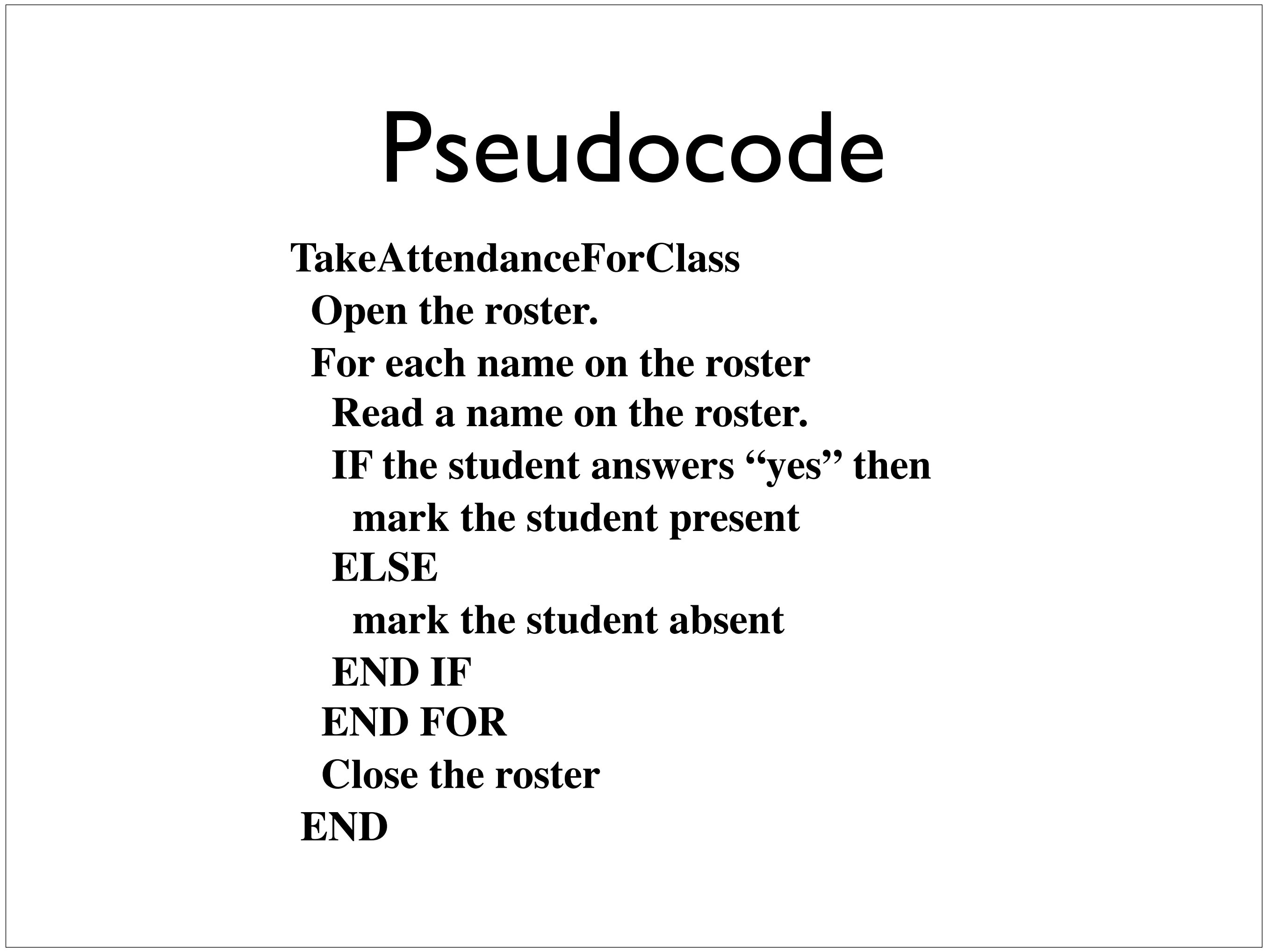
Oh yes...I have to do it for EVERONE in the class!

Flowcharts



A diagram that uses symbols to show how input is
processed to produce output

It shows each specific step and sequence needed to solve a problem.



Pseudocode

```
TakeAttendanceForClass
  Open the roster.
  For each name on the roster
    Read a name on the roster.
    IF the student answers “yes” then
      mark the student present
    ELSE
      mark the student absent
    END IF
   END FOR
   Close the roster
 END
```

Is an English-like representation of the design steps
that represent the solution to a problem

It also shows the steps and sequence to solve the problem.

3. Program coding

- writing instructions in a programming language to implement the logic determined from the Program Design step

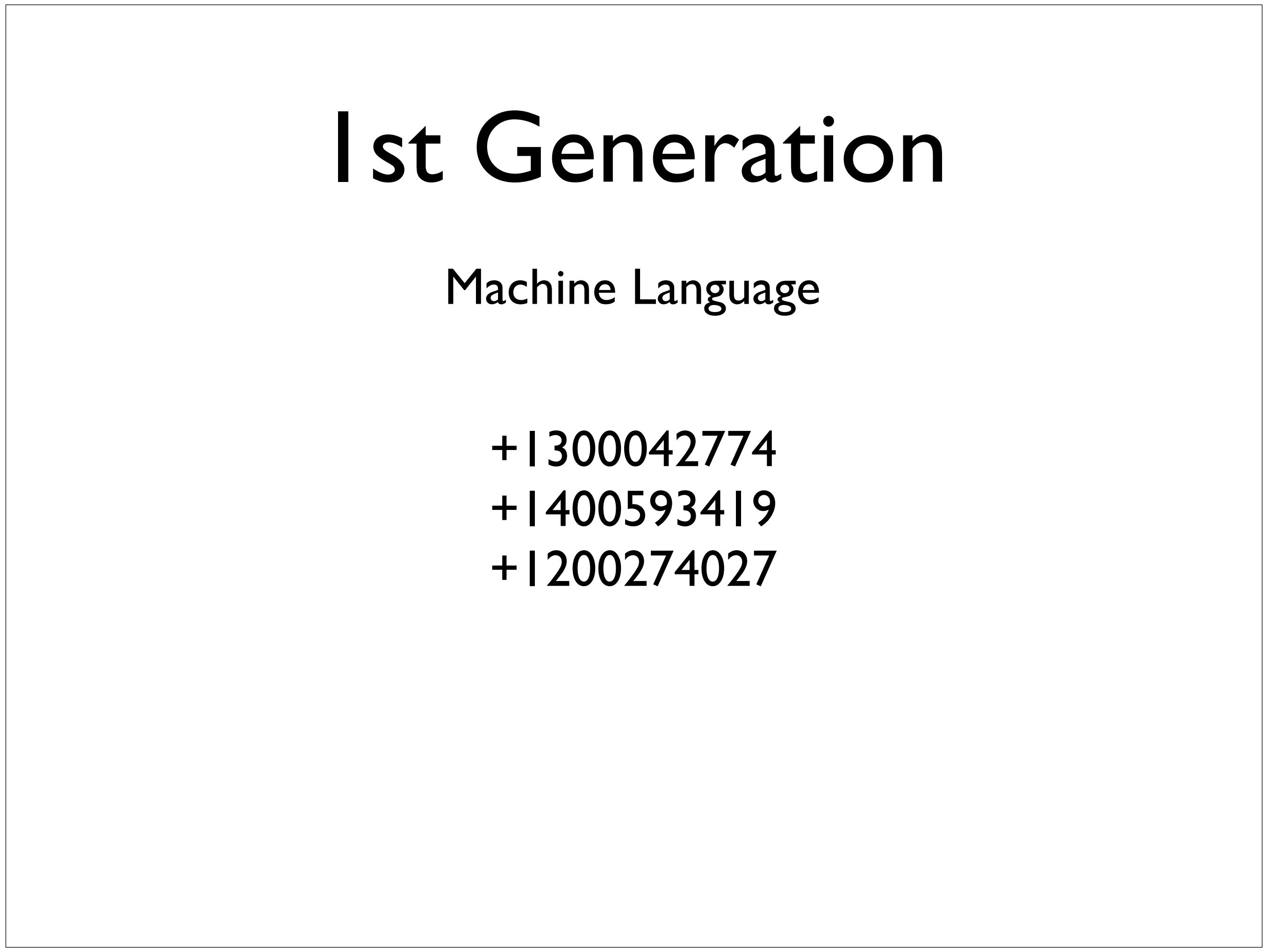
Programming Languages

Remember that computers only understand 1s and 0s. We use a layer of abstraction called a programming language to give the computer instructions.

This way we have a language that both the computer AND
the programmer understands

We direct the computer with programming languages

There are hundreds of computer programming languages out there.



1st Generation

Machine Language

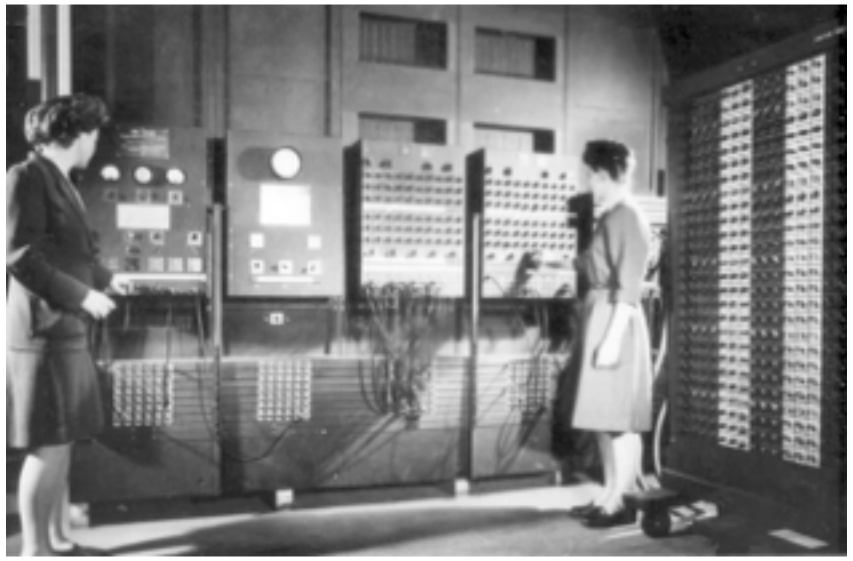
+1300042774
+1400593419
+1200274027

1st gen programming introduced in the 1940s. The instructions were entered in binary or using special machine instructions that manipulated the memory and CPU of the computers. Debugging was very difficult, and you could only write simple programs.

Some computers weren't even able to store programs, so in order to run the program again, you'd have to feed it to the computer again.

1st Generation

Required intense training and specialized skills.
Fast but error-prone.



(U.S. Army photo from the archives of the ARL Technical Library)

Programmers Betty Jean Jennings (left) and Fran Bilas (right) operate ENIAC's main control panel at the Moore School of Electrical Engineering.

2nd Generation

Assembly Language

```
LOAD BASEPAY
ADD OVERTIMEPAY
STORE GROSSPAY
```

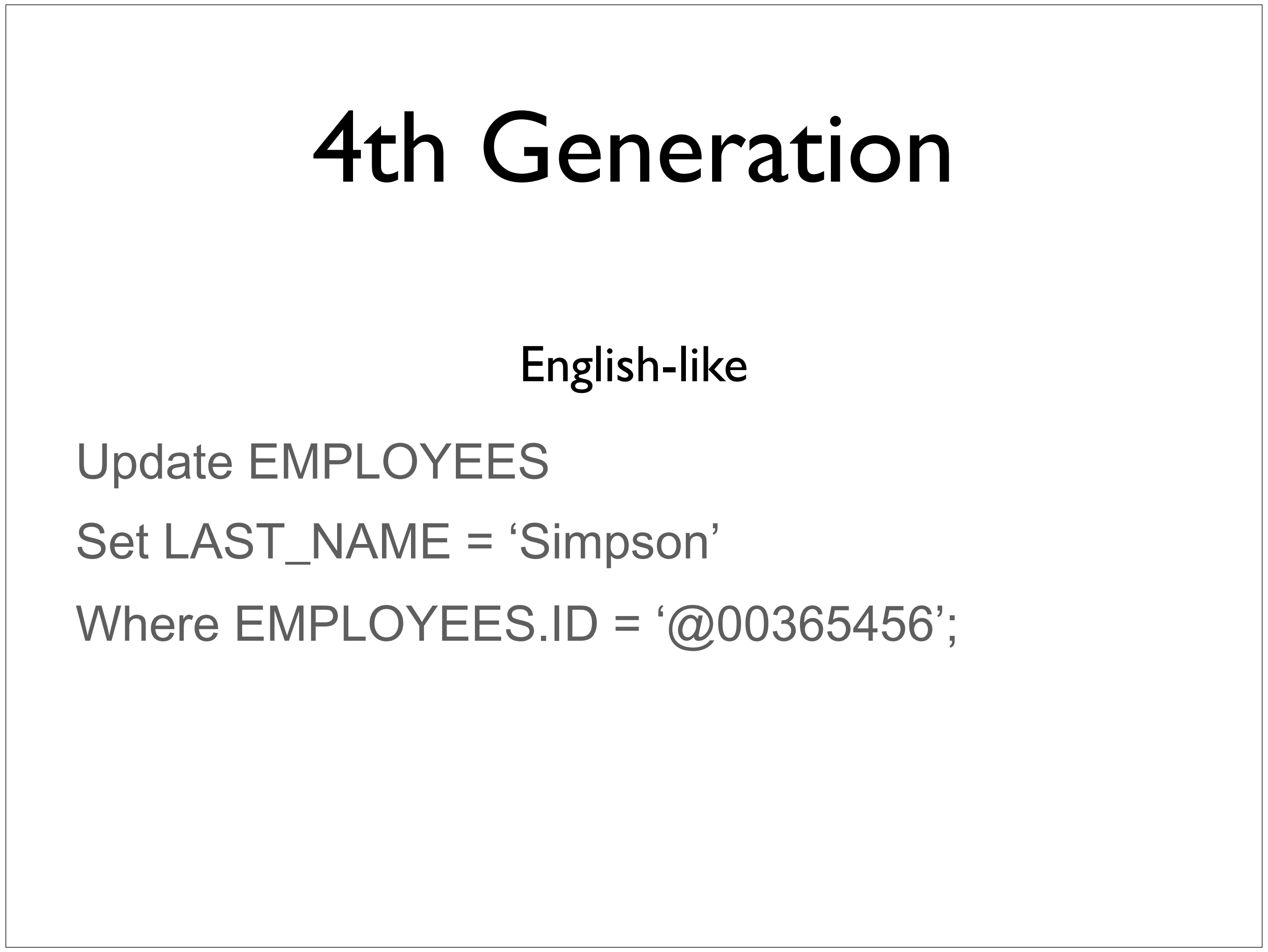
Easier to understand but still relies on
specific computers and processors.   It’s eventually compiled, or translated, down to machine code.

3rd Generation

High-level languages

```
var grossPay = basePay + overtime;
```

Single program statements accomplish
more substantial tasks, and run on more than
one kind of computer. Most of our modern languages fall in this category. C, C++, COBOL, Visual Basic, Java, etc.



4th Generation

English-like

```
Update EMPLOYEES
Set LAST_NAME = ‘Simpson’
Where EMPLOYEES.ID = ‘@00365456’;
```

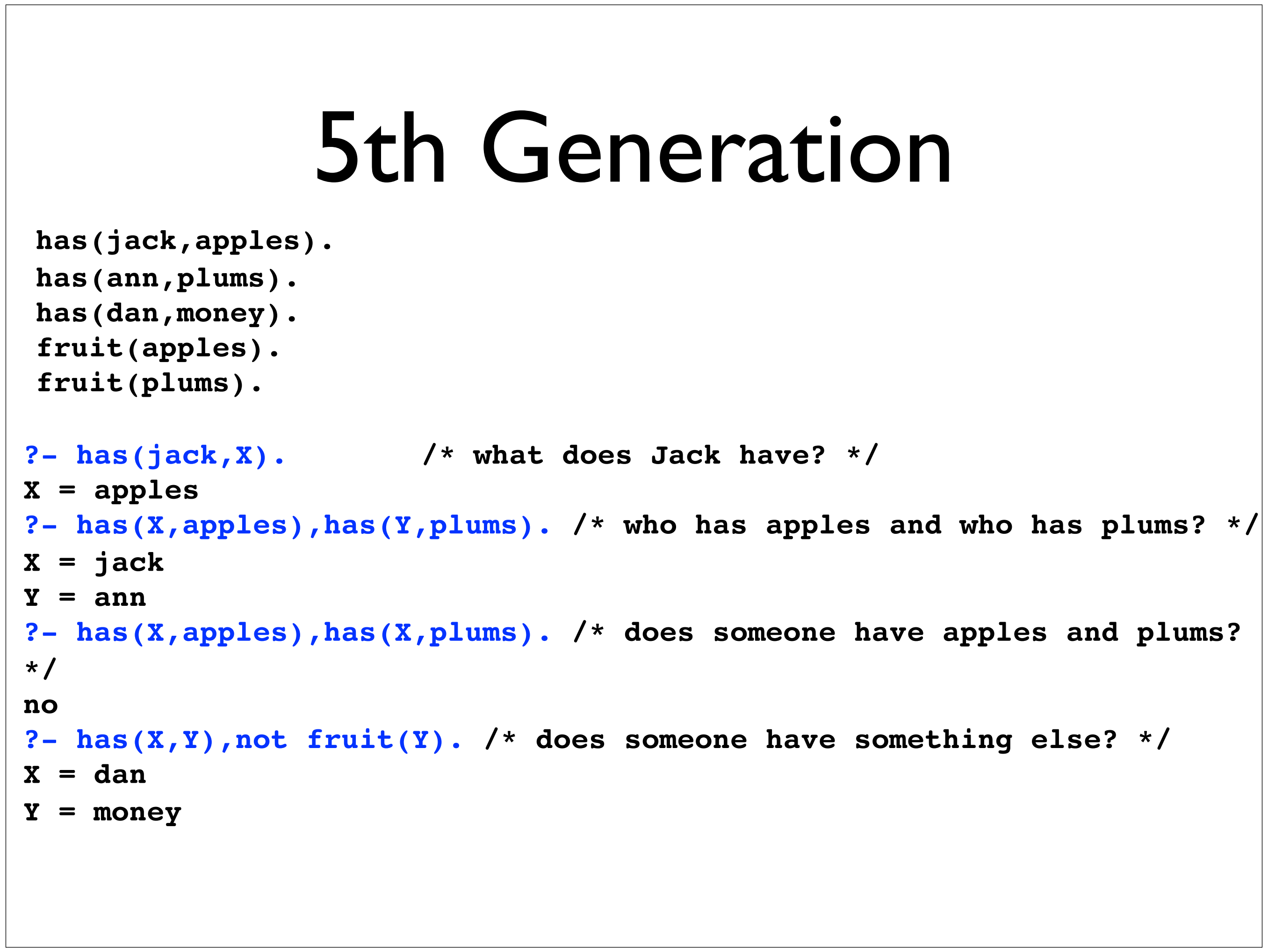
4th gen languages are languages designed to make a problem in a specific domain simple to implement. This reduces development time and cost. However it also requires more specific developer learning.

SQL is probably the best known 4th generation language. CSS could also be classified as a 4th generation language.

5th Generation

Artificial Intelligence, Expert Systems, Robots

5GL languages are not in wide use. Mostly found in research and mathematics, but also in use in AI and decision process. They use constraints.



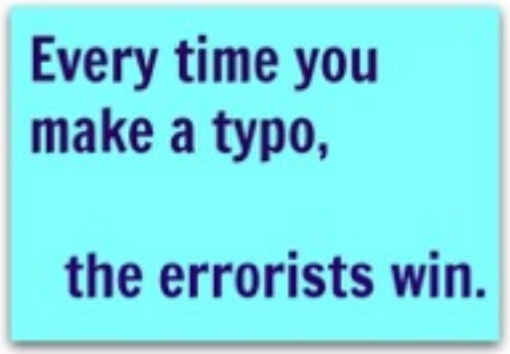
5th Generation

```
 has(jack,apples).
 has(ann,plums).
 has(dan,money).
 fruit(apples).
 fruit(plums).

?- has(jack,X).        /* what does Jack have? */
X = apples
?- has(X,apples),has(Y,plums). /* who has apples and who has plums? */
X = jack
Y = ann
?- has(X,apples),has(X,plums). /* does someone have apples and plums?
*/
no
?- has(X,Y),not fruit(Y). /* does someone have something else? */
X = dan
Y = money
```

Here's an example of a Prolog program. We feed it facts and then we can simply test the facts. We can also introduce rules. Prolog works great for doing things like calculating insurance premiums, because instead of writing an algorithm, we can just feed a bunch of facts into the rules engine and get the answer we need.

Syntax

refers to rules of the programming language


Every time you
make a typo,

the errorists win.

English syntax is things like grammar, spelling, punctuation.

Syntax Errors

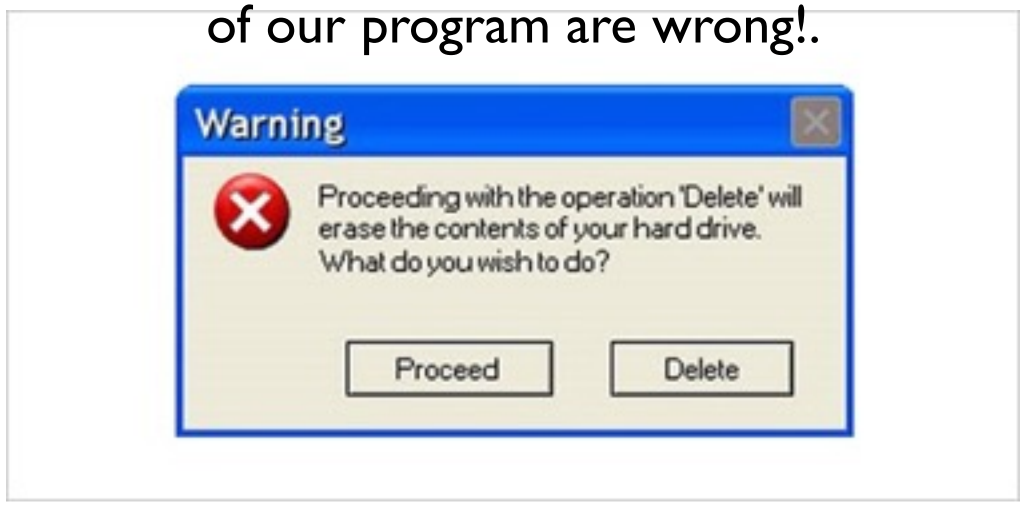
must be correct in order for the program to run

Check for syntax errors using development tools
or peer reviews

Syntax errors are where a lot of programmers who are new to the language spend a lot of their time. Development tools like the JavaScript web console are great for finding your errors. But sometimes it’s hard to see the problem if you stare at it too long. You might ask someone else to help you find the error.

Logic Errors

The steps,, decisions, or repeating processes of our program are wrong!.


Warning

Proceeding with the operation 'Delete' will erase the contents of your hard drive. What do you wish to do?

Now, syntax errors are going to prevent our program from even running. But we can still have a program that doesn't work due to logic errors. Maybe it's a label, maybe it's that we've really goofed something up.

Writing a program

- Key the program into a text editor or development environment
  - The instructions you enter are displayed on the screen as you type
  - You save the file to the hard drive
  - Then you run the program.

The CPU does not understand programming languages!

The programming language instructions must be translated into machine language that the CPU can understand.

Interpreters

- Interpreters are programs that must be in main memory before loading the program to be executed
- They translate each instruction written in the programming language into machine language a line at a time. The translated instruction is passed to the CPU where it is executed.

Interpreters

- Advantages
  - Done in one step, can be real-time
- Disadvantages
  - One line at a time
  - Program must be reinterpreted each time the program is run.

Compilers

- Compilers are programs that create an independent, executable file of the program to be executed
- They go through the code you write (source code) and translate (compile) the instruction statements into assembler-level code called object code that can be executed

Compilers

- Advantages
  - "Executable" code runs very fast because it is not interpreted line-by-line
  - The executable file of the program can be run over and over without being recompiled
- Disadvantages
  - Compilation of the source code (explain source code) takes several steps (but only needs to be done once unless the source code is changed)

Choosing a Programming Language

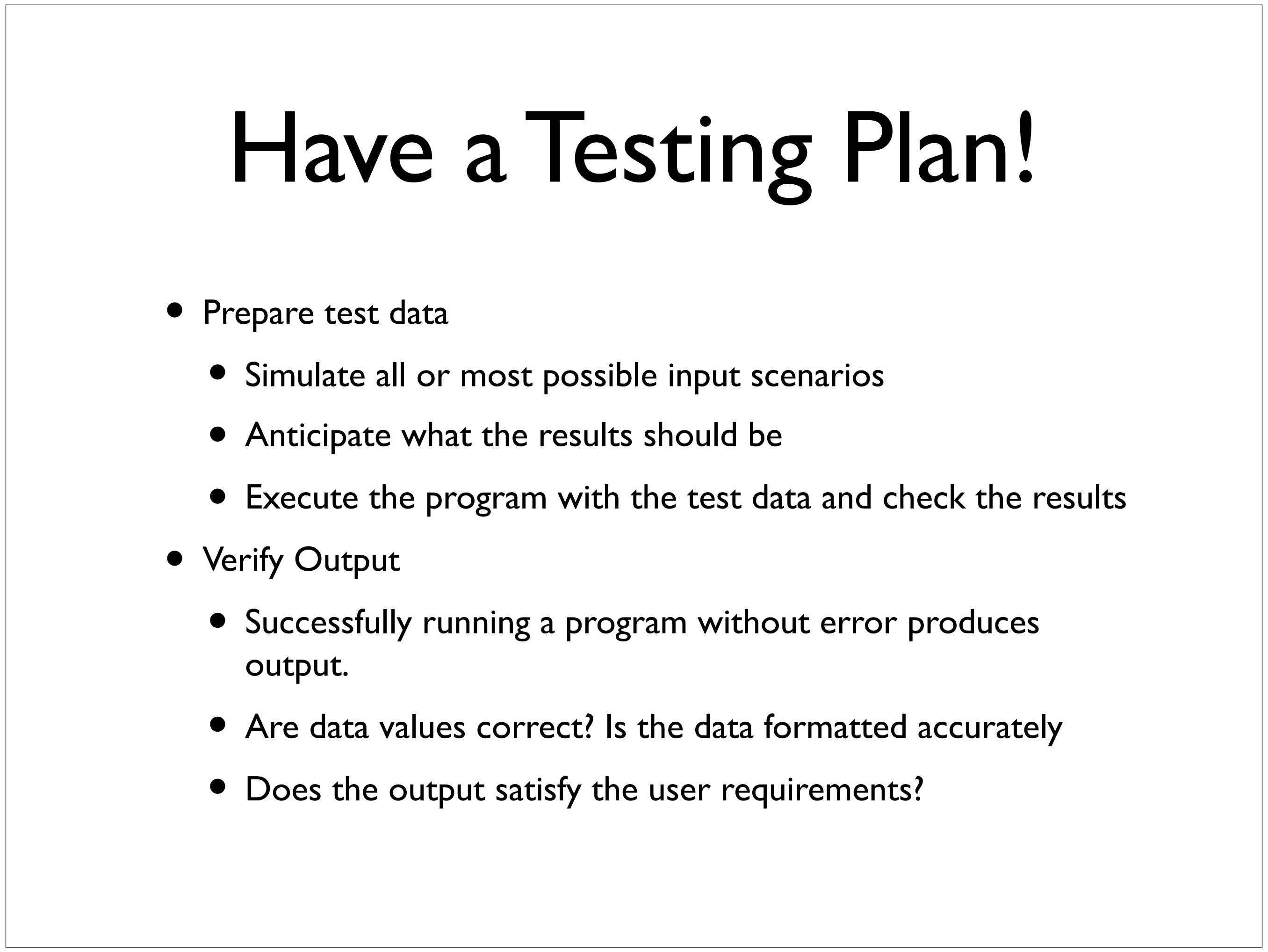
- Right way
  - Pick a programming language that solves the problem at hand
- Wrong way
  - Use the language you know to solve every problem you encounter

Programming Languages are Tools

A good craftsman knows how to use many tools. We’re working with JavaScript in this class because its an easy language to learn. We can build web apps, mobile apps, desktop apps with JavaScript. It’s a language we can use for lots of things. But it doesn’t do every one of those things well. There’s not a single programming language out there that does everything well. There’s no golden hammer. And I can tell you that till I’m blue in the face but you’ll need to learn more than one or two programming languages well before you realize that.

4. Program Testing

- Test to ensure that the program works as it should and the end result meets original specifications.
- The real goal is to reduce as much as possible the amount of errors through good design
  - But, rarely does a program work perfectly the first time!
  - So – test to uncover errors...fix them, and test again, and again...until there are no errors left! TEST, TEST, TEST!



Have a Testing Plan!

- Prepare test data
  - Simulate all or most possible input scenarios
  - Anticipate what the results should be
  - Execute the program with the test data and check the results
- Verify Output
  - Successfully running a program without error produces output.
  - Are data values correct? Is the data formatted accurately
  - Does the output satisfy the user requirements?

- Have a test plan that outlines all of the processes and situations that should be tested.

Test Plan??

- Inputs: “Homer”
- Expected Result:“Hello, Homer!”
- Actual Result:

Test Plan!

- Inputs: “Homer”
- Expected Result: “Hello, Homer!”
- Actual Result:

Syntax Error

```
alert'Hello 'name

Exception: missing ; before statement
```

Occur when any of your program instructions violate the syntax rules of the programming language. These will be caught at the interpretation or compilation stage

That’s like punctuation, spelling, and grammar in the English language.

Test Plan!

- Inputs: “Homer”
- Expected Result: “Hello, Homer!”
- Actual Result: “Hello Homer”

So we run our test plan again and now it works

Logic errors

```
alert("Hello" + name);
```

- Inputs: “Homer”
- Expected Result: “Hello, Homer!”
- Actual Result: “Hello Homer”

Occur because the logic designed and coded does NOT process the data in the manner desired (i.e., you don't get the output you desired or the program aborts because it did something illegal like access an out-of-bounds area of memory – dividing by zero)

5. Program Documentation

- What is the program to accomplish?
- How does it do that?
- How will you remember why you did what you did 6 months from now?

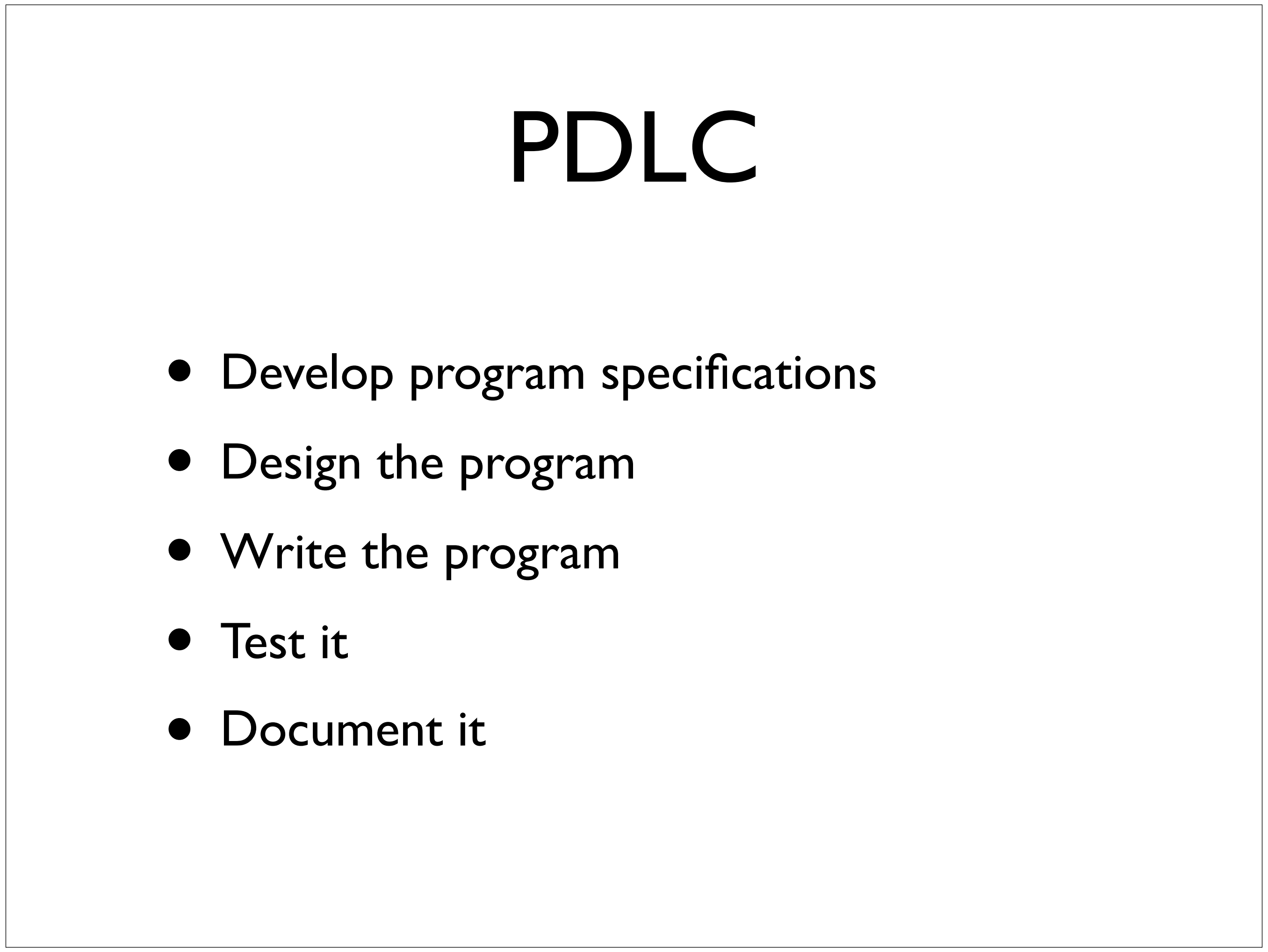
Internal Documentation

- Standard practice among employers
- Comments are placed within the program source code
- Explains the data and processes being used
- Enhance readability of the program for other programmers
- Important because you often maintain other programmers' programs!

External Documentation

- User documentation
  - Instructions and training manuals for the people using your program
- System documentation
  - Operating instructions, data formats, error messages and their meanings, etc...
  - Design documents, like the flowcharts you prepared
  - Data dictionary (descriptions of all the data used)

That’s a lot to remember!



PDLC

- Develop program specifications
- Design the program
- Write the program
- Test it
- Document it

So this is the “Waterfall” method that’s used over and over, and it’s bad.

Beware BDUF

Big Design Up Front causes projects
to fail miserably.

BIg Design Up Front causes projects to fail. People spend months doing all of the design trying to come up with every single feature the system will ever have, and you can’t do that.

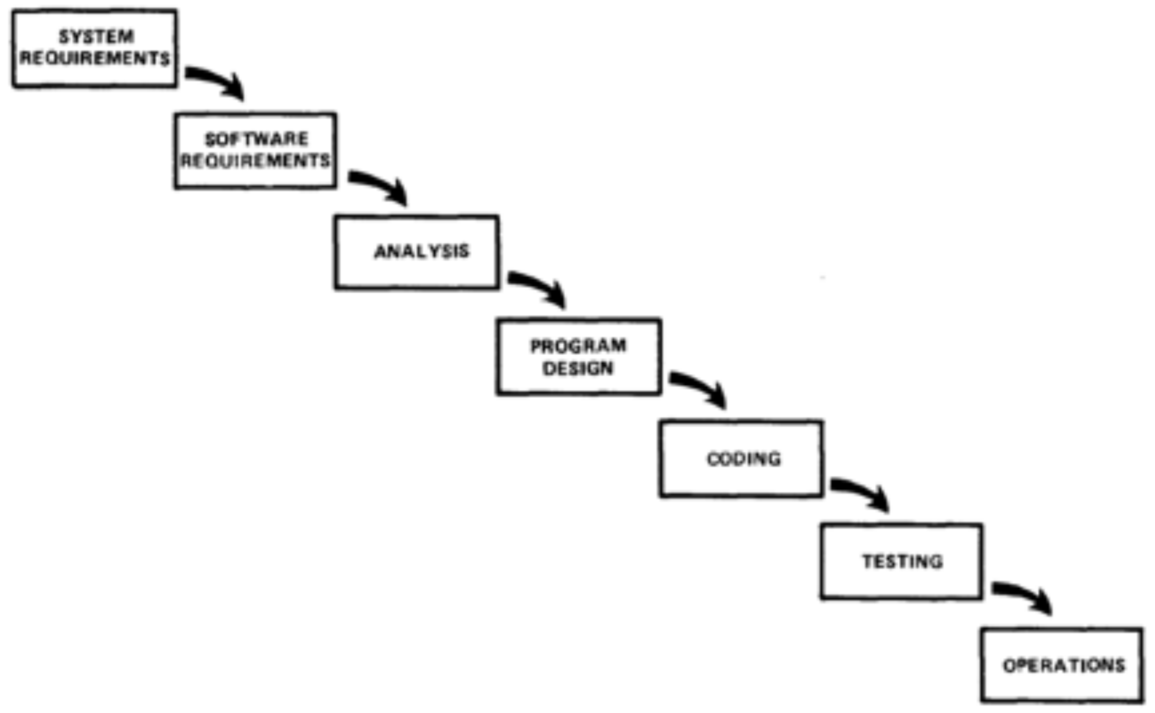


Figure 2. Implementation steps to develop a large computer program for delivery to a customer.

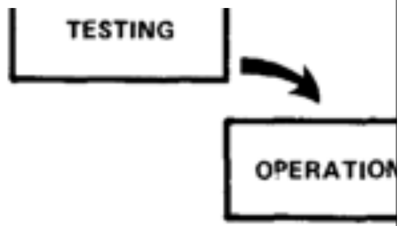
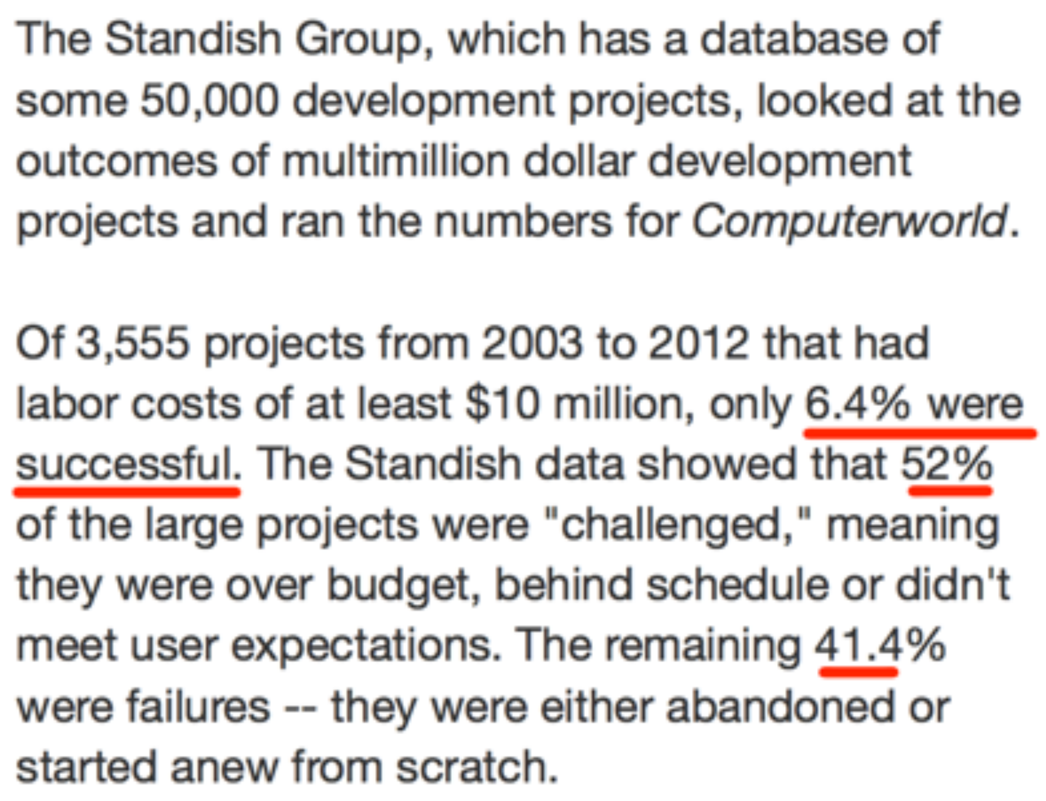


Figure 2. Implementation steps to develop a large computer program for delivery to a customer.

I believe in this concept, but the implementation described above is risky and invites failure. The



The Standish Group, which has a database of some 50,000 development projects, looked at the outcomes of multimillion dollar development projects and ran the numbers for *Computerworld*.

Of 3,555 projects from 2003 to 2012 that had labor costs of at least $10 million, only 6.4% were successful. The Standish data showed that 52% of the large projects were "challenged," meaning they were over budget, behind schedule or didn't meet user expectations. The remaining 41.4% were failures -- they were either abandoned or started anew from scratch.

http://www.computerworld.com/s/article/9243396/Healthcare.gov_website_didn_t_have_a_chance_in_hell_?source=rss_keyword_edpicks

Requirements change!

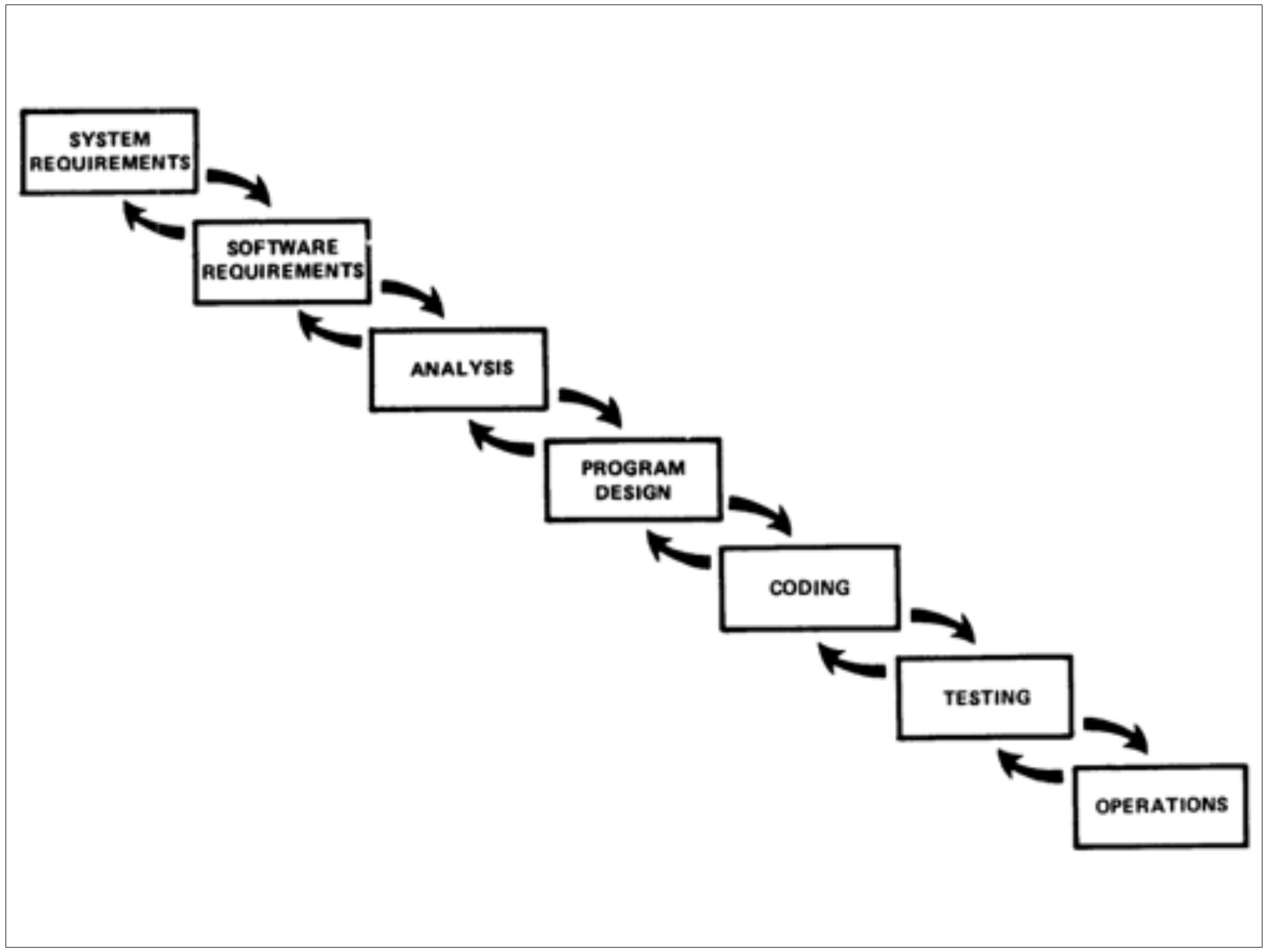
If you have a big system, your requirements may change by the time you write it.

People will leave the company, people will change their minds. You need a better approach.

Iterations!

Do the steps of the PDLC in small iterations, for each feature.

This is a more “agile” approach.



And if you look at Royce's paper on software development, the next page shows this diagram and discusses the idea of iterative processes.

Iteration

- Develop feature specifications
- Design how the feature will be coded
  - Design with test plans up front
- Write the feature
  - Test it
  - Document it
- Repeat!

You need to break the programs down into small pieces, and tackle these using a variation of the PDLC that's more "agile". Do the testing and coding and documentation at the same time. This is really the only way that can scale for building maintainable code.

There is a more formal process called Test-Driven Development where you write programs that test your actual programs. But we'll save that for later.

Our Process

- Gather requirements for the small program
- Design program
    - Test plans
    - Pseudocode
    - Flowcharts
- Code program
    - Verify tests

This is the process that we’ll use when developing programs in this class. You’ll get some requirements and you’ll make sure you have whatever you need. Then you’ll go build something. You’ll look at test plans and determine what your program’s expected output should be. Then you’ll come up with pseudocode and flowcharts to design the code you’ll write. Then you'll go and write your code, verifying your tests as you go.

Wrap up

- How programs work
- Syntax and Logic
- PDLC (waterfall) vs PDLC in iterations
- Program design