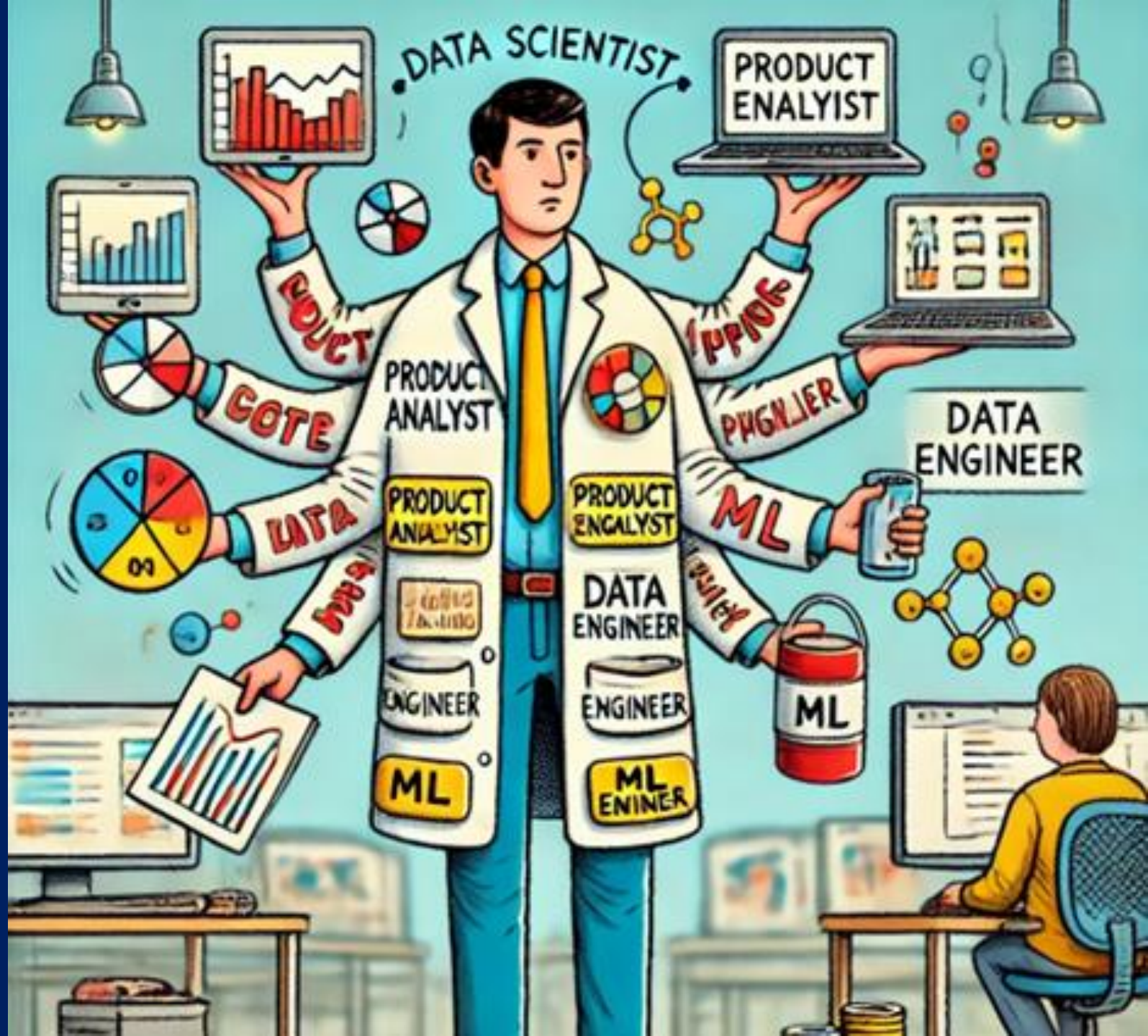
CS 51 Computer Science Principles

Module 3: Data, Internet, Computer and Programming

Unit 1: Creative Development

LECTURE 1 CREATIVE DEVELOPMENT

DR. ERIC CHOU
IEEE SENIOR MEMBER



Welcome to AP Computer Science Principles!

Welcome to AP® Computer Science Principles! Designed as an introductory course, AP CSP is a great way to learn about the fundamentals of coding and the world of programming.

(If you've taken this class before, don't be surprised if some of the content covered here looks different. As of the 2020-21 school year, the curriculum is undergoing major changes.)

Why Take AP CSP?

- **AP CSP** is a great way to learn the basics of coding and gain coding experience, all while getting AP credit at the same time. You don't need any prior experience and your school should provide you with all the technology you'll need.
- AP CSP is also very **open-ended**. The class usually focuses on project-based learning, and the requirements for your final Create Project are pretty flexible. This gives you many opportunities to make something you're passionate about.
- Often times, the class will count as a tech credit locally as well.
- Convinced? Without further ado, let's begin.

Unit 1 Overview

Exam Weighing:

- 10-13% of the AP Exam
- Practically, this translates to about 20 questions on the test.

Innovation

LECTURE 1

Computing Innovations

- Computing innovations, according to the College Board, are innovations that use a program as a key part of their function. Put simply, they wouldn't operate without a computer program making them work. If you can use the word "computer" or "coded" when describing this innovation, it's probably a computing innovation.
- Another way to identify computing innovations is to think about data. Does the innovation you're thinking of collect data and use it when operating? If so, you've probably got a computing innovation on your hands.
- Computing innovations can be both physical and non-physical, and they come in all shapes and sizes.

Examples of Computing Innovations:

Physical

- Self-driving cars
- Smart appliances (fridges/watches/toasters)
- Tablets (Kindles/iPads)
- Smart Phones
- Gaming devices (Nintendo Switch/Xbox)
- Robots (Roombas, for instance)

Examples of Computing Innovations:

Non-Physical

- Picture Editing Software (Photoshop/Adobe Lightroom)
- Word Processors (Word/Pages/Google Docs)
- Communication platforms (email/text messaging/video conferences)
- Digital video games (Dark Souls/Minecraft/Super Mario Kart)
- Applications (iPhone Apps)
- Even some concepts, like e-commerce or social networking, count

Collaboration

LECTURE 2

Collaboration in Computer Science

- While a lot of code-writing is independent by nature, the computer science field has a lot more collaboration in it than you'd think. Programmers of all sorts have to work with coworkers and bosses when dealing with large projects. They also have to work with their clients to make sure what they're coding meets client needs.

Collaboration in Computer Science

Different people have different backgrounds, perspectives and ways of thinking. Here are some ways **such diversity is helpful** when creating a computing innovation:

- More hands working on a project can sometimes get it done faster than one person can alone, (despite what the results of your last group project might indicate 😂).
- Discoveries can be made thanks to the multiple perspectives on deck.
- Biases can be avoided during the development process, creating a more inclusive innovation.
- Working with users and clients specifically during the development process can ensure that the finished product is one everyone is happy with, saving both time and energy.

Collaboration Between Users and Developers

- During the creation of a computing innovation, users and developers will communicate with each other. For example, video games will have testers that check the product for bugs and report them to the developers. Often, this conversation begins even before the product is made—some companies will conduct market research to determine what features would be best to include in their newest innovations.
- However, communication doesn't stop there! Even after the product is released, developers will often ask for feedback and offer areas for users to report any problems they may have.

Computing Developments that Foster Collaboration

Collaboration between programmers isn't a new concept. The computer science field has several models designed to foster collaboration, such as pair programming.

Pair Programming

Driver/Navigator

- **Pair programming is a programming model where two people share one computer.** *One person codes while the other person oversees the work, and the two often switch places.*
- At the same time, the internet makes collaboration between developers easier. You can see a version of this in your own life: most people today use Google Docs or Slides to work on shared projects.
- **Github** and **Bitbucket** are famous examples of collaborative development websites.

Ways to be a Good Team Player!

- In the AP CSP class, there are times where you'll have to collaborate with others to work on projects in class. The final Create project also gives you the option to work with another person during the development phases of the project.

Ways to be a Good Team Player!

Here are some tips, AP CSP-style, to make your collaborative team the most successful it can be!

- Communicate kindly and often.
- Practice consensus building within your team by listening to every member within it and taking their perspectives into consideration.
- Create norms such as establishing team roles or policies to help mediate any conflicts that might arise.
- A team project is a team effort, and that means everyone in the group needs to have a say. Sometimes, that might mean compromise.

Program Function and Purpose

LECTURE 3

Programs and Code Segments… with Cake

- A **program** is a collection of instructions that a computing device executes, or carries out. It's like a recipe that a computer follows. Programs run on **inputs** and **outputs** and exhibit specific behaviors.
- Within a program, you have **code segments**. A code segment is a smaller collection of statements that are part of a program. It's like the part of a cake recipe that tells you how to make the frosting.

Programs and Code Segments... with Cake

- Each code segment is made up of **statements**, or individual instructions. It's like a line in your recipe that says, "Crack three eggs into the bowl."
- So, a computer program is made up of code segments, which are made up of statements.
- Programs are also known as **software** or **applications**.

Describing Programs

- A **program** can be described in a broad sense by what it does.
  - My adding program lets the user add two numbers of their choice.
- You can also go more in depth by describing how the program's statements accomplish this goal.
  - My adding program asks the user to input two numbers, then adds them together and displays the final sum.

Program Inputs

- Programs run on inputs, which are pieces of data that a computer takes in and processes. These inputs can be directly submitted by the user or they can come from other programs.
- Inputs can be...
  - **Auditory**, such as spoken words or music notes
  - **Visual**, such as photos or videos
  - **Tactile**, such as strokes of a keyboard
  - **Text**, such as words or numbers

Program Events

- In order for a program to receive inputs, an **event** needs to happen. An event is simply an action that gives a program data to respond to.
- For example, pressing the left arrow causes a video game's program to make an avatar go left when before this it was running right. Pressing the key is the event in this case.
- Events cause programs to change how they're running. In the previous example, pressing the left key caused a change. If the arrow hadn't been pressed, the avatar would've kept running right.

Examples of common events:

- clicking a button
- moving a mouse
- starting a program
- pressing a key
- running down a timer

Event-Driven Programming

- A lot of modern software is **event-driven**, which means that it's designed to respond to events in order to run. Let's go back to our adding machine example. You start the program, an event, and press the enter key to input numbers into the program, another event. The program is designed to respond based on these events, so it's considered event driven.
- On a larger scale, your phone's software is event-driven as a whole because it responds primarily to your taps, clicks and swipes.

Program Outputs

- If program inputs are data that a computer takes in, **program outputs** are the data that the computer returns. They can be in any of the formats that inputs can be. In the adding machine example, the program output is the final sum that the program returns.

🔗 **CPU, Memory, Input & Output by Khan Academy and Code.org**

Program Behavior

- When you run a program, you expect it to perform in a certain way. This is known as a program's **behavior**. Usually, program behaviors are defined by how a program will respond to a user interacting with it.
- It's important to think of what behavior you want your program to have before you code it. In order to figure that out, you'll need to know what the purpose of your program is.

Computing Innovations and Purpose

- The **purpose** of a computing innovation is what it's designed to do.
- **Solving problems** is one of the major purposes of computing innovations. For example, text messaging and video-chat platforms solve the problem of needing to communicate quickly over long distances. A computing innovation can also be a form of **creative expression**, such as making a video game.

Computing Innovations and Purpose

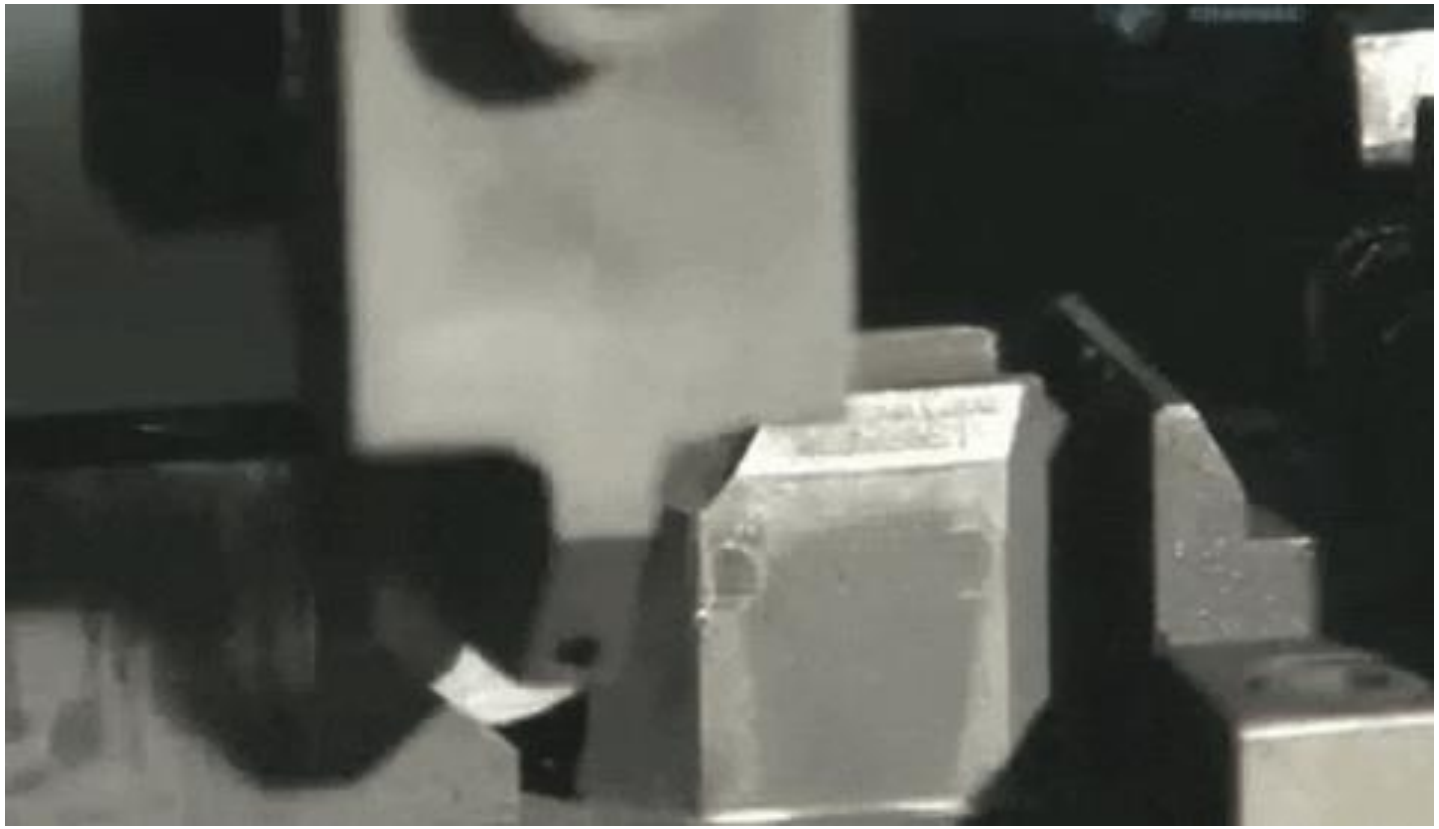
- Having a clear purpose for the computing innovation you want to make is like having a good thesis for an essay. It focuses your work and establishes what your goals are. In short, it strengthens your ability to develop your creation.
- So, how do you develop a clear purpose for your program? The answer to that lies in the development process.

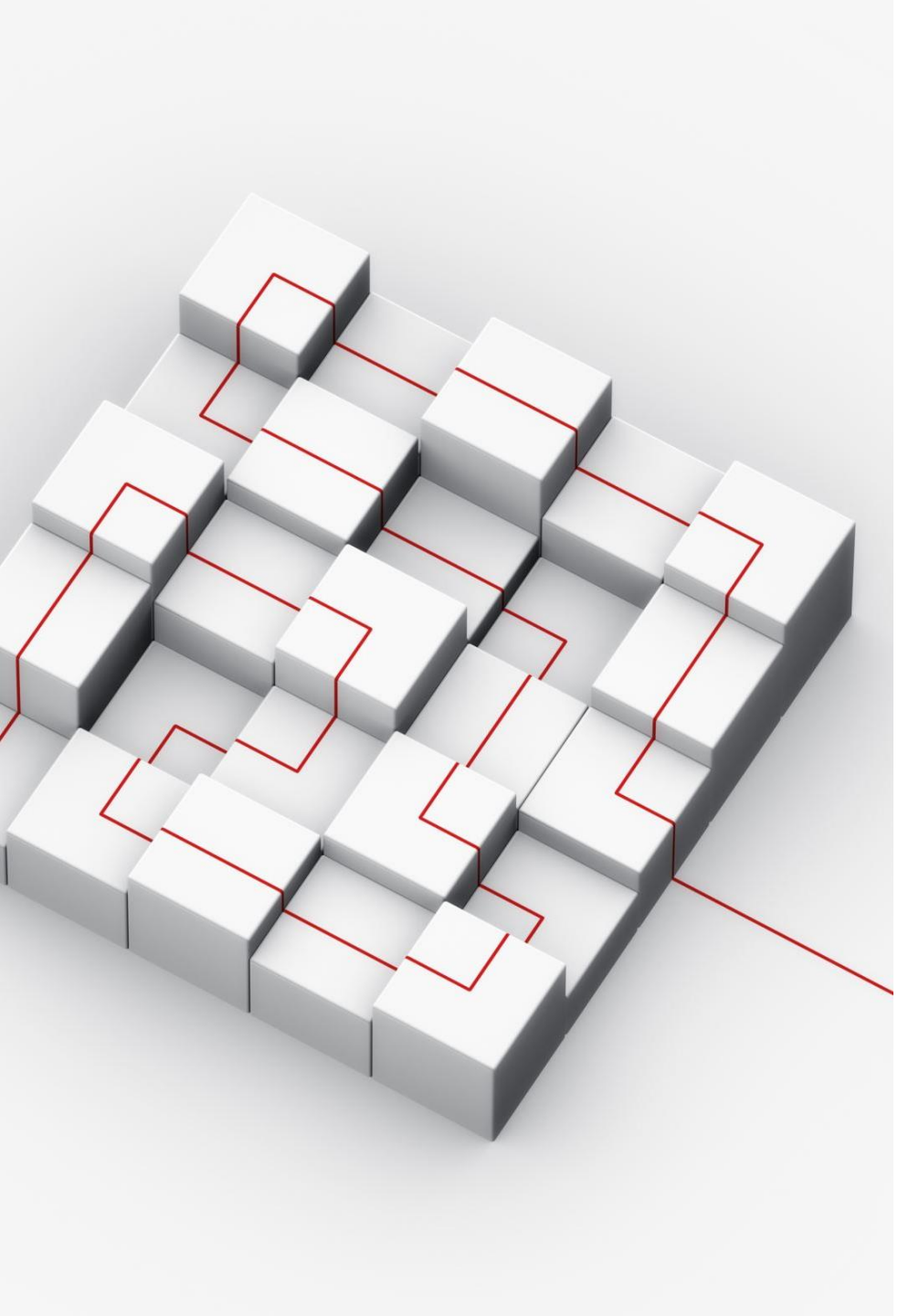
Program Design and Development

LECTURE 4

Development Process

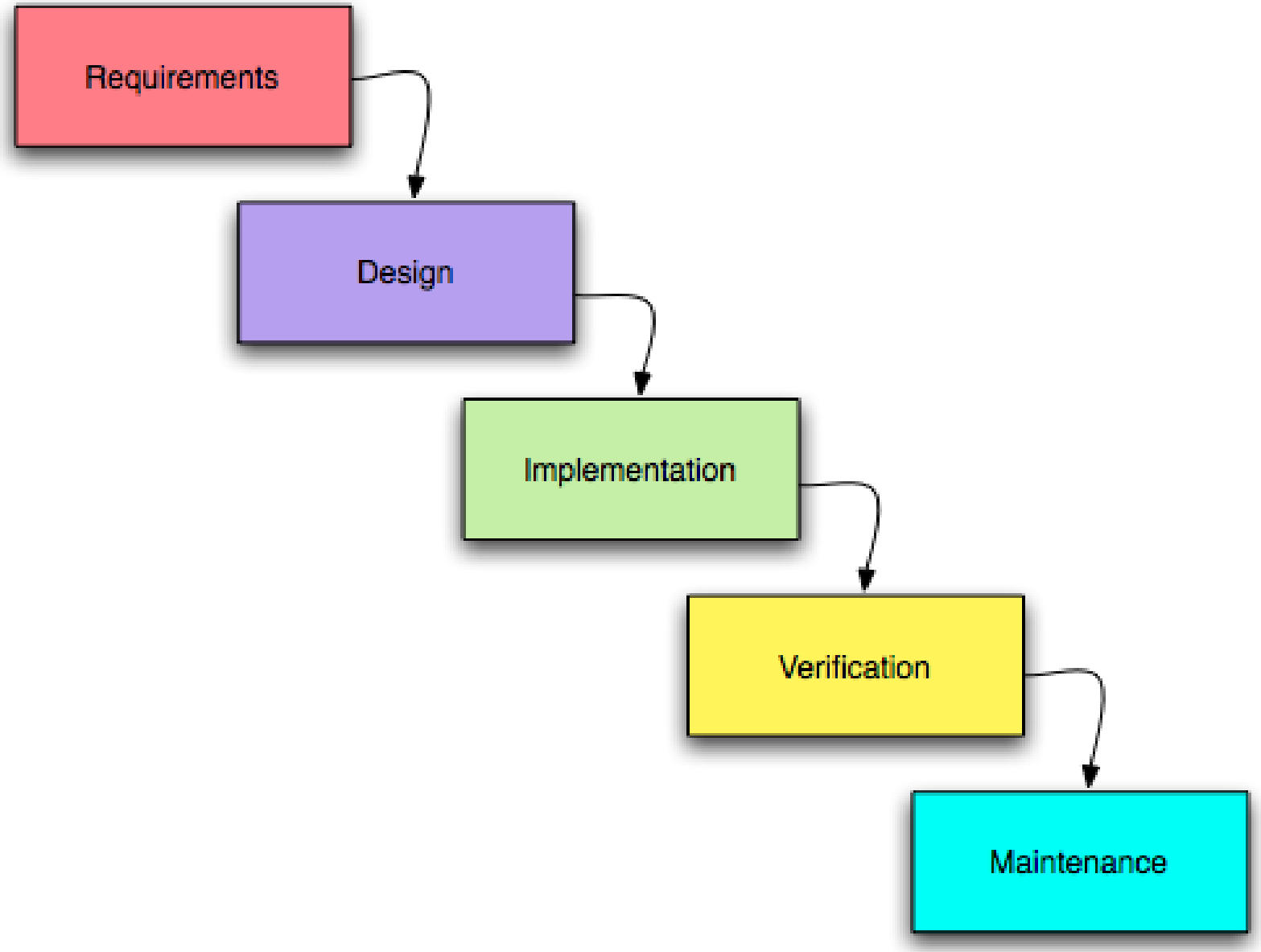
The development process for a program is how it's made. It consists of the steps it takes to go from planning to programming and beyond!



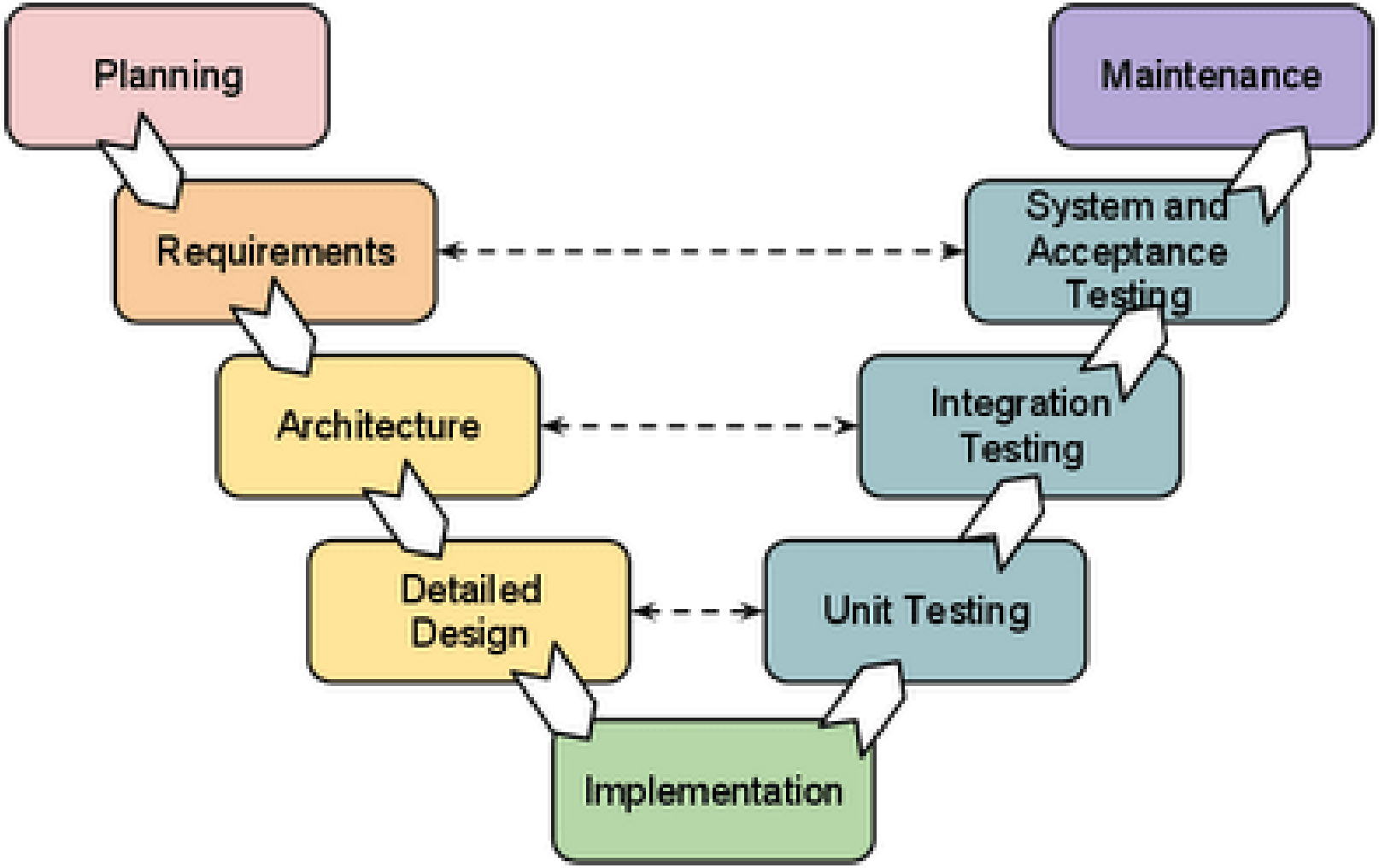


Types of Development Processes

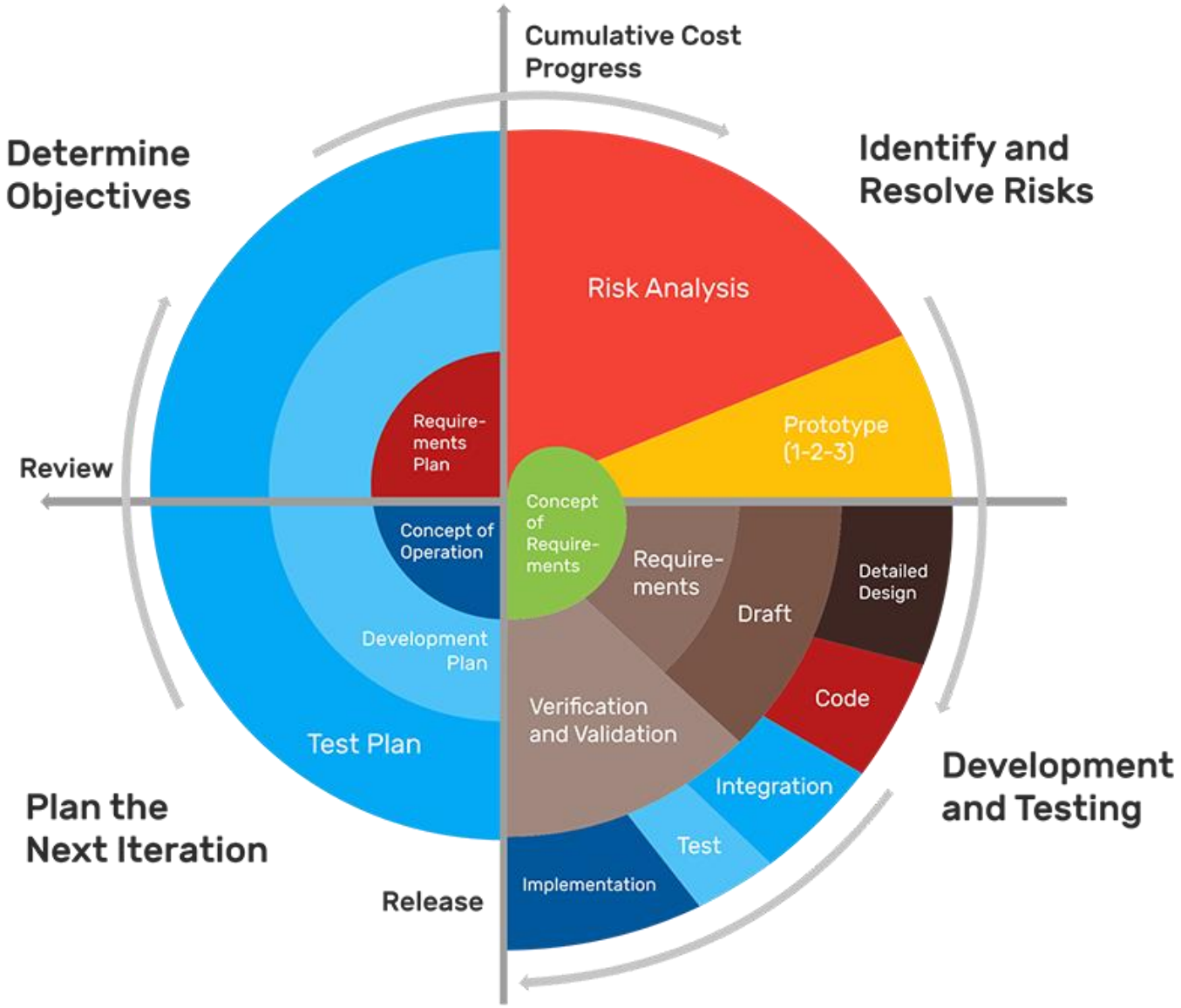
- Traditionally, program development follows a set path. It is orderly and intentional. One of the most well-known frameworks for developing software is known as the **waterfall** development model. It's characterized by a step-by-step process, where one step flows naturally into another.



Waterfall model


Planning
Maintenance
Requirements
System and Acceptance Testing
Architecture
Integration Testing
Detailed Design
Unit Testing
Implementation

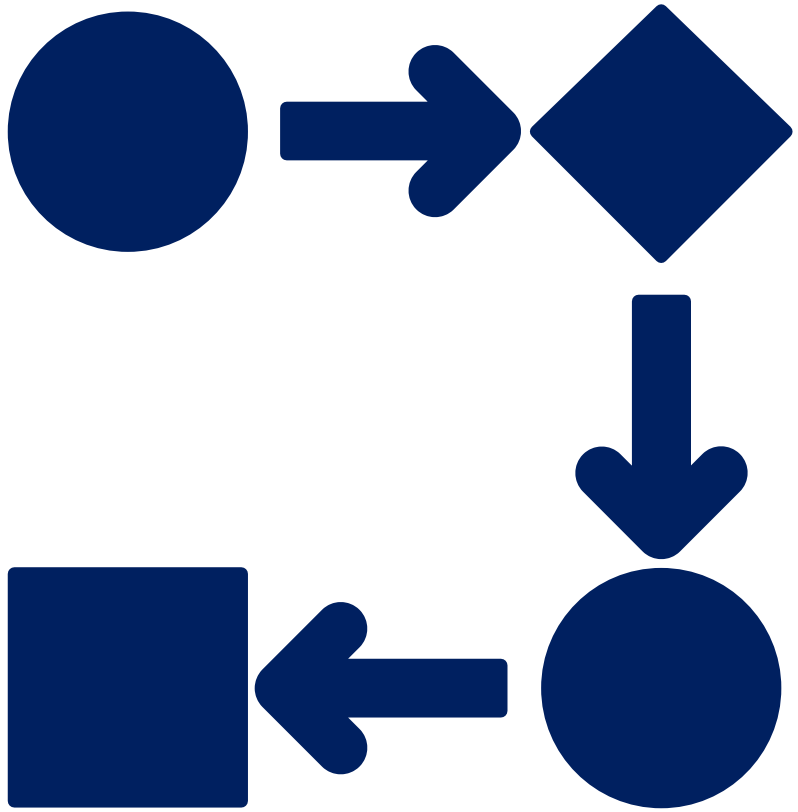
V model



Spiral model

Types of Development Processes

- Over the years, new methods of creating programs were created, such as the Agile and DevOps development methodologies. These methods can offer a more flexible approach to development, and help foster collaboration between developers and users. There can also be variations within methods. For example, programmers have created variations, or modifications, on the waterfall development model over the years.
- Some development processes aren't intentional. They're exploratory in nature, and the programmer experiments as they go along. This can happen when a program doesn't have specific guidelines behind it or when developers are under time constraints.

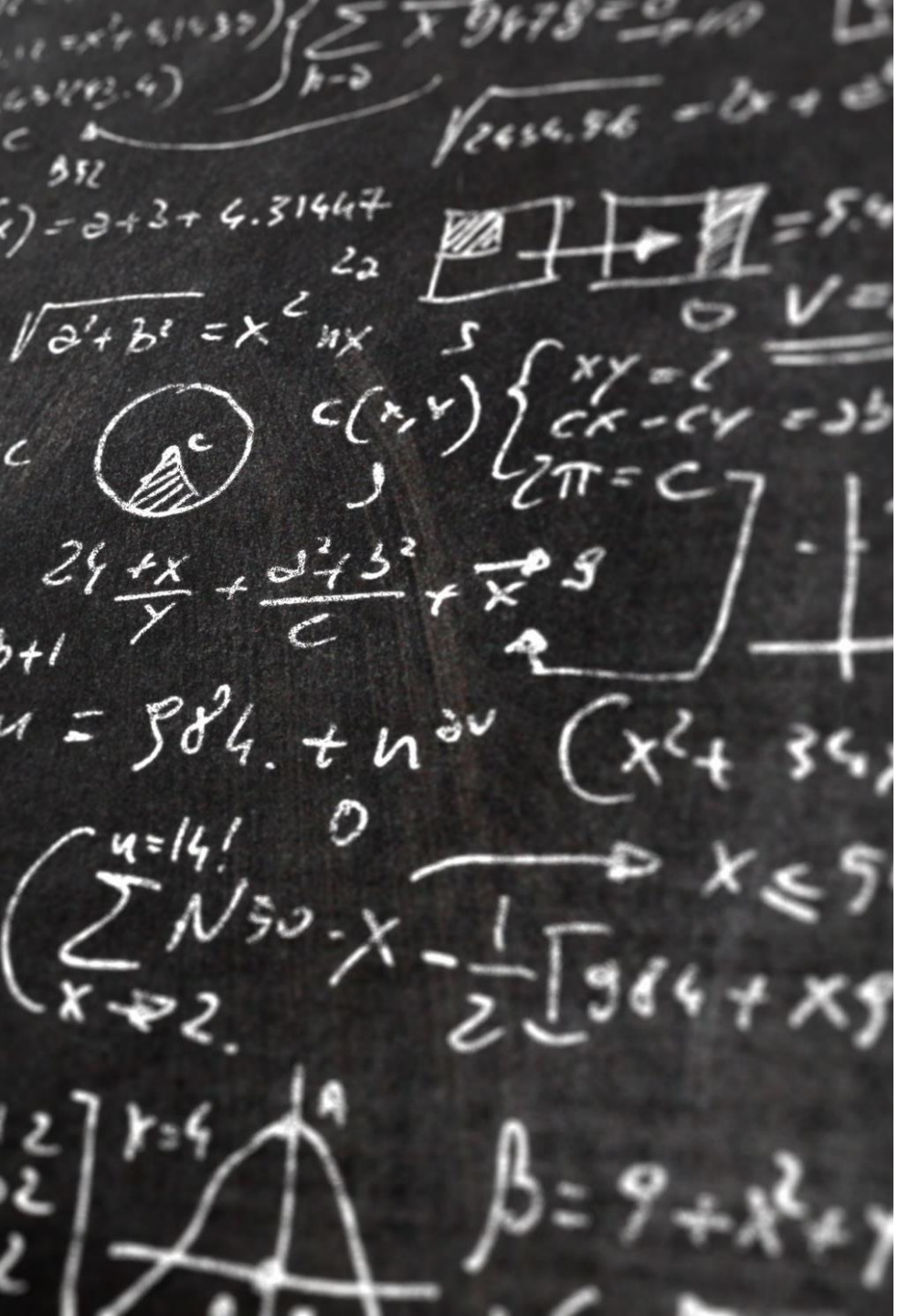


Iterative and Incremental Development Processes

- When you think of "iteration," think of repetition. In an **iterative development process**, programmers develop working prototypes of their program and go back through the stages of their development method to make their programs better.
- In an **incremental development process**, programmers will break the program they're working on into smaller pieces and make sure that each piece works before adding it to the whole.

Iterative and Incremental Development Processes

- Programs can be developed using both of these models, or any combination of the two. For example, you can take a code segment of a larger program and revise it until you finish developing it completely, then move on to another part of the program.
- You're using an iterative process by revising that code segment to completion, but an incremental one for the program at large.



The Drawing Board

- The beginning steps for most development processes are the planning phases. Lots of work goes into a program even before anyone opens a code editor.

Investigation and Reflection

- This is the game-plan stage of the development process. The goal of this investigation is to make the goal of the programmers as clear as possible.
- In this stage, programmers establish what their purpose is and the problem they're trying to solve. They figure out what their program will need to do, and also what their program will need in order to function properly.

Investigation and Reflection

- They'll often have program specifications to help them with this task. **Program specifications** are descriptions of the goals of the program agreed on by both the programmers and the clients.
- In order to get a clear picture during this stage, programmers have to consult with many sources. If their program is for a client, the client will be consulted to determine what they want. If they're working in a group, they may have to consult each other to make sure all parts of their program will work together. They may also have to do external research to discover what sorts of programs are similar to the one they're trying to make.

Ways Programmers Investigate:

- collecting data through communication channels, such as surveys
- user testing
- conducting interviews with clients to assess what their needs are
- direct observation of the project in action if the project has already been through one or two loops

In an iterative process, this stage can also include a reflection aspect on ways the project could improve or why something isn't working. With every repetition, the questions to investigate and reflect on change, becoming more specific over time.

Designing Code

The design phase of the development phase details how to accomplish the goals of the program.

This design phase may include:

- brainstorming
- planning and story-boarding
- organizing the program into modules and functional components
- creating diagrams that represent the layouts of the user interface
- developing a testing strategy for the program

Program Requirements and Specifications

- Through investigation, programmers are able to discover their program's requirements. Program requirements describe how a program works. What should a user be able to do? What does a user need to provide for the program to work?
- Program requirements also include what's needed on the programmer's end (ex: the program must be compatible with multiple browsers) as well as what tests need to be run to confirm that all these requirements are in place.

Program Requirements and Specifications

- From here, programmers are free to start building, prototyping, and testing their programs to their hearts' contents.
- While they're writing their programs, they'll need to record what they're doing. They'll do this through the process of writing **program documentation**.

Program Documentation

- **Program documentation**, at its simplest, is a description of how something in your program works. Sometimes, this description can include how your program was developed, such as why you chose one method of coding over another. The documentation could describe a code segment, event, procedure, or even the whole program itself.

Program Documentation

- Why is documentation used? Programmers document a program in order to break it down and explain it. They need to do this for a variety of reasons. Programs are complex things, and when you're dealing with pages and pages of code, documentation helps you keep things straight. They also foster collaboration. Someone other than the original programmer may someday need to work with this code, and documentation helps explain what's going on.
- Think of documentation as notes for your code in the same way you'd take notes for a textbook.
- With this in mind, let's turn to a common form of program documentation: comments.

Comments

- One of the most common forms of program documentation is known as comments. Comments are documentation written directly into the program itself.

```
            default='Y',
            )
    global_scale_setting = FloatProperty(
            name="Scale",
            min=0.01, max=1000.0,
            default=1.0,
            )

def execute(self, context):

    # get the folder
    folder_path = (os.path.dirname(self.filepath))

    # get objects selected in the viewport
    viewport_selection = bpy.context.selected_objects

    # get export objects
    obj_export_list = viewport_selection
    if self.use_selection_setting == False:
        obj_export_list = [i for i in bpy.context.scene.objects]

    # deselect all objects
    bpy.ops.object.select_all(action='DESELECT')

    for item in obj_export_list:
        item.select = True
        if item.type == 'MESH':
            file_path = os.path.join(folder_path, "{}.obj".format(item.name))
            bpy.ops.export_scene.obj(filepath=file_path, use_selection=True,
                                     axis_forward=self.axis_forward_setting,
                                     axis_up=self.axis_up_setting,
                                     use_animation=self.use_animation_setting,
                                     use_mesh_modifiers=self.use_mesh_modifiers_setting,
                                     use_edges=self.use_edges_setting,
                                     use_smooth_groups=self.use_smooth_groups_setting,
                                     use_smooth_groups_bitflags=self.use_smooth_groups_bitflags_setting,
                                     use_normals=self.use_normals_setting,
                                     use_uvs=self.use_uvs_setting,
                                     use_materials_setting,
```

eC Learning Channel

Comments

- In the above image, the **grey portions of text** with the number symbol (#) in front of the words are comments. Each comment describes what the code below it is intended to do.
- Comments are often marked by some sort of symbol, like the number symbol, that tells the computer to disregard the text when running the program.

❗ While most of the programming languages you'll be working with in this class will support comments, there are some that won't. In this case, you'll need other documentation methods.

Sourcing Your Work

- As you travel further and further on the path of computer science study, you may come across a situation where you're using code that someone else helped make. Maybe you and a friend were working together on the Create project. Maybe you had a group project. Maybe you found this really nice code segment that does exactly what you need...
- Not so fast, buster! There are some things you need to do before using code that other people helped create.
- **First**, check with your teacher: are you allowed to use this code?
- **Next**, you need to get the user's permission, if possible.

Sourcing Your Work

- If not, code is often licensed in a way that will tell you what your usage permissions are. Verify that you have permission to use this code.
- **Finally**, you need to cite your sources. Just like when writing an essay or making a PowerPoint, it's important to cite your sources and acknowledge any work that you didn't make. Fortunately, this can be done within your program's documentation.
- Overall, your citation should include who originally wrote the code and where the code came from.
- There are plenty of online guides about how to cite code, and you're in good hands! **Here's a comprehensive one to get you started.**

Identifying and Correcting Errors

LECTURE 5

Common Errors

Logic Errors: A mistake in a program that causes unexpected behavior. For example, a program outputs a result different from the expected value, like saying that three times three equals thirty-three.

Syntax Errors: A syntax error occurs when the rules of the programming language aren't followed. Any typos in your code, such as forgetting to close a parentheses or spelling a variable wrong, will cause an error. Fortunately, many code editors will point out syntax errors, making them one of the easier errors to solve.

Run-Time Errors: An error that occurs when the program is running. You'll be able to start your program if you have a run-time error, but something will go wrong when you're trying to use it. Logic errors are examples of run-time errors.

Overflow Errors: An error that occurs when a computer tries to handle a number that's outside of its defined range of values. Usually, this means that your computer's trying to handle a number too big for it.

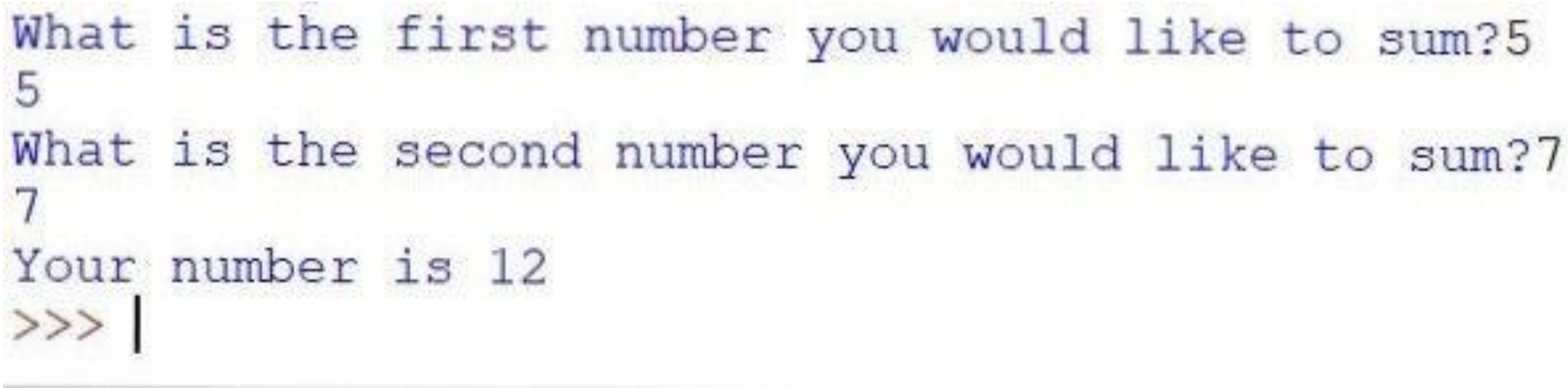
Error Fixes

Here are some effective ways to identify, find and correct errors.

- Testing different inputs, and testing at different points in your program's runtime.
    - Good testing procedures include testing with valid and invalid data. You want to make sure your program works with valid data and knows what to do if invalid data is imputed.
    - You should also test **boundary cases,** or values on the edge of program limits. This is where errors are likely to happen. For example, if your program only accepts values less than 9, you would check 8 and 10 as boundary cases.
- Hand tracing, or manually tracking what values your variables have as your program goes along. Many hand tracing methods use charts to organize these values.

Error Fixes

- Using visualizations, or visual representations of your code. Try plugging in some code into **this visualization tool** to see how it works!
  - Although they're helpful, you won't need to know how to use visualization tools for the AP test or to write your Create program.
- Using debuggers, or programs designed specially to test for bugs
- Adding extra print statements to your code to make sure that the "invisible" steps (the ones that aren't shown as output) are working properly.



Adding extra print statements is also a common way to test code. In this example, the numbers 5 and 7 are printed in order to make sure that the input system is working properly.

Error-Hunting Example

Give this problem a try!

`val` appears in the list `myList`. The procedure does not work as intended.

```
Line 1: PROCEDURE countNumOccurences(myList, val)
Line 2: {
Line 3:     FOR EACH item IN myList
Line 4:     {
Line 5:         count ← 0
Line 6:         IF(item = val)
Line 7:         {
Line 8:             count ← count + 1
Line 9:         }
Line 10:    }
Line 11:    RETURN(count)
Line 12:}
```

Which of the following changes can be made so that the procedure will work as intended?

(A) Changing line 6 to `IF(item = count)`

(B) Changing line 6 to `IF(myList[item] = val)`

(C) Moving the statement in line 5 so that it appears between lines 2 and 3

(D) Moving the statement in line 11 so that it appears between lines 9 and 10

Answer

- The answer is C: **Moving the statement in line 5 so that it appears between lines 2 and 3.**
- First, let's take a look at the program's purpose: returning the number of times the value val appears in a list. For example, if val equaled 5 and 5 showed up twice in the list imputed, the program would return 2.
- If you're having trouble on a code question, try rewriting the code to the side. This can help you see the code lines in a different light. Just make sure to pay attention to details such as spacing when you rewrite.

Answer

- Rewritten without the brackets, the code looks like this:

PROCEDURE countNumOccurences(myList, val) FOR EACH item IN myList count ← 0 IF(item = val) count ← count + 1 RETURN(count)

- Let's make a test list [5, 2, 3, 5] and set val equal to 5 to see what the error might be. The program should return a value of 2.
- Starting from the beginning, we set the count to 0. The first number in the list equals the value of val, so count now equals 1.

Answer

- Because we're in a loop (which we'll talk more about in Big Idea 3), you're going to go back through the loop for the second item in the list, which is 2. We set count to zero again, and...
- Wait, that isn't supposed to happen! The error in this code is that every time the loop resets, count also resets to zero.
- How would we fix that? By taking count ← 0 outside of the loop. We need it to be before the loop starts, so we have the count variable to work with, but it can't be inside the loop itself.
- The loop begins in line 3, so putting count ← 0 between lines 2 and 3 works perfectly.

The Need For Programming Languages

LECTURE 6

Vocabulary

- **Algorithm** - A precise sequence of instructions for processes that can be executed by a computer
- **Programming Language** - derived from the human need to concisely give instructions to a machine
- **Ambiguity** - refer either to something (such as a word) which has multiple meanings, or to a more general state of uncertainty.

Programming

- One of the most important things you can do in programming starts well before you write any code.

It's about how you think.

- "Everybody should learn how to program a computer . . ., because it teaches you to think." -Steve Jobs

Why do we have programming languages? What is a language for?

1. way of thinking -- way of expressing algorithms
2. languages from the user's point of view
3. abstraction of virtual machine -- way of specifying what you want
4. the hardware to do without getting down into the bits
5. languages from the implementor's point of view

The Art of Language Design

- Evolution
- Ease of Implementation
- Special Purposes
- Standardization
- Personal Preference
- Open Source
- Expressive Power
- Excellent Compiler
- Ease of Use for Novice
- Economics/Patronage/Inertia

declarative	
functional	Lisp/Scheme, ML, Haskell
dataflow	Id, Val
logic, constraint-based	Prolog, spreadsheets, SQL
imperative	
von Neumann	C, Ada, Fortran, …
object-oriented	Smalltalk, Eiffel, Java, …
scripting	Perl, Python, PHP, …

Computer Scientist Group Language as …

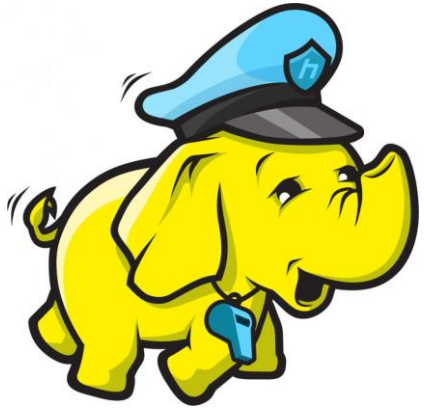
Network (Web-server) Languages

Desktop (.exe) Languages

Mobile (App) Languages

Markup/Data Languages

Database Languages



Number and Data Processing Languages

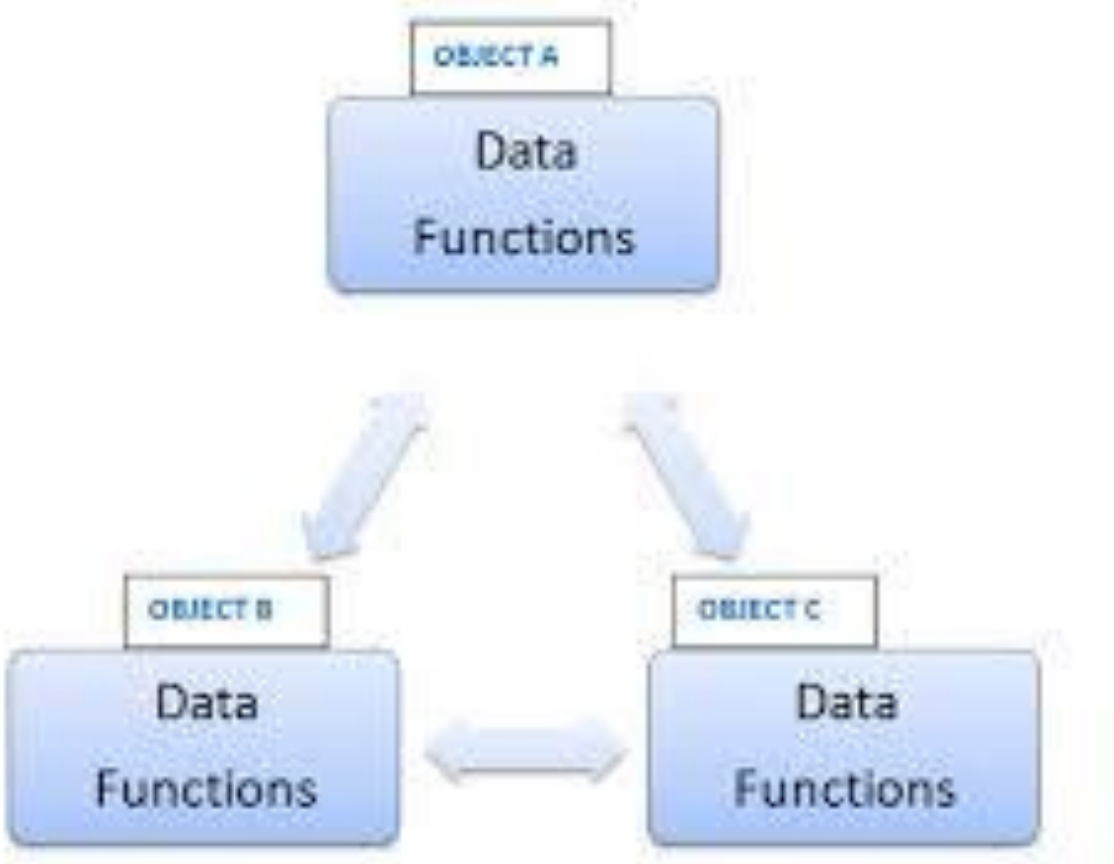
Hardware Description Languages

HSPICE®

Electronics Languages

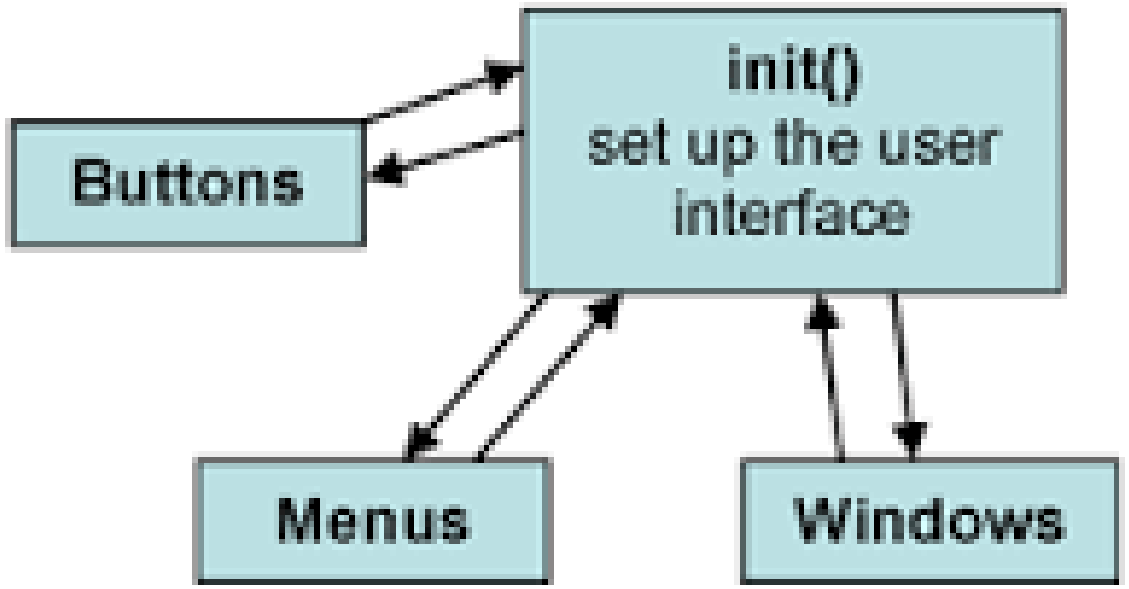
Programming Paradigm

LECTURE 7



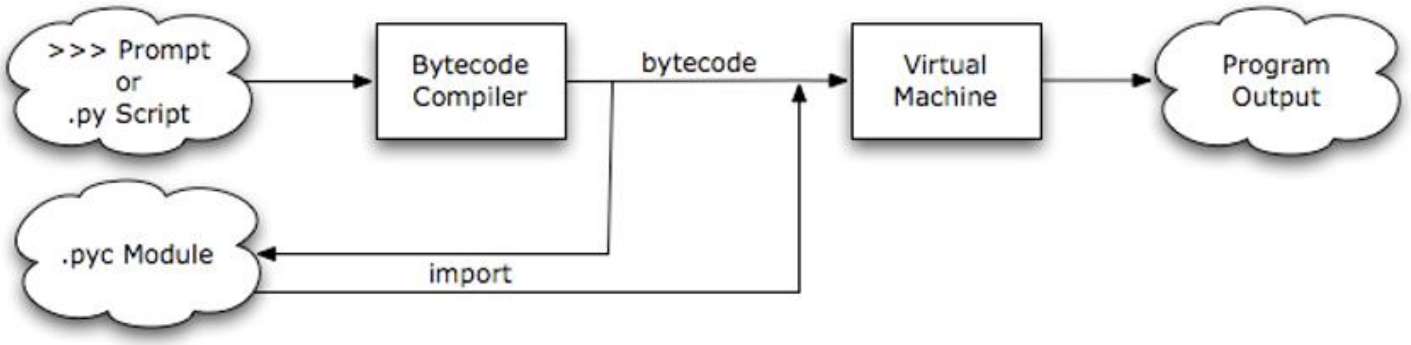
Imperative Languages

Object-Oriented Programming



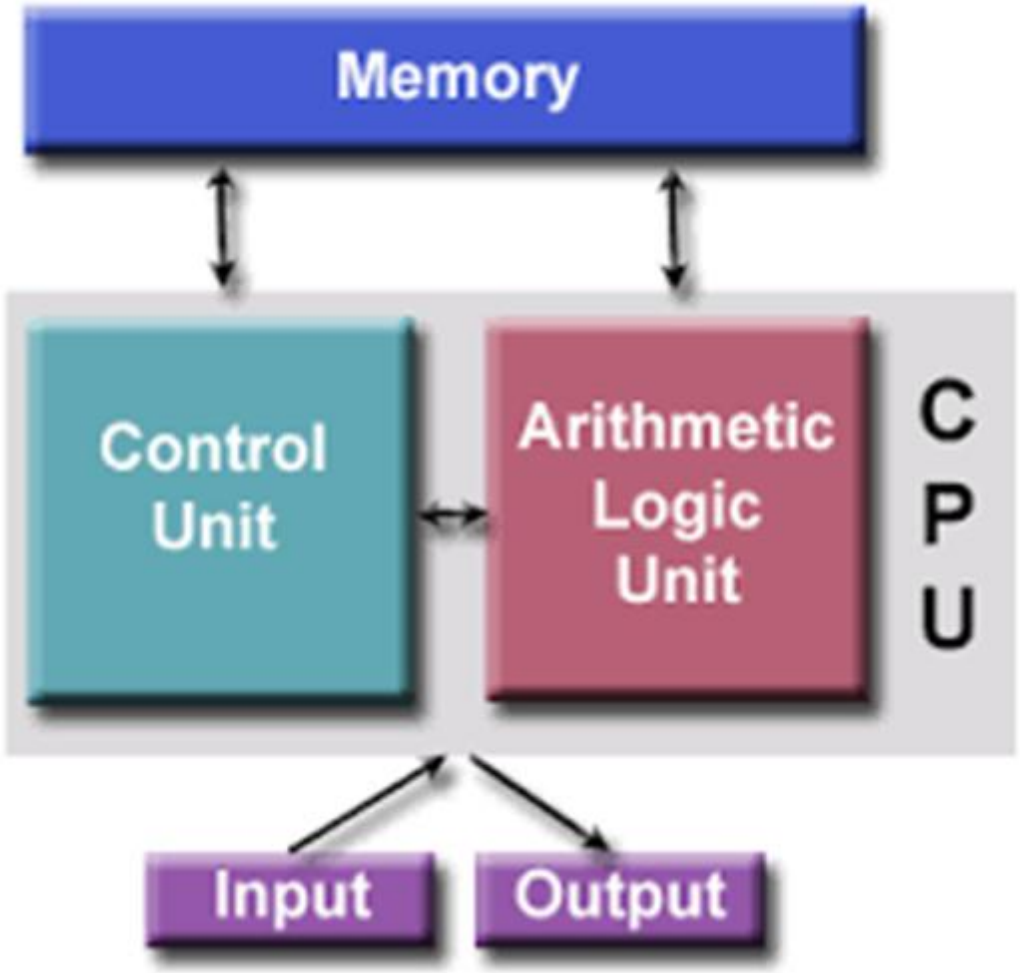
Imperative Languages

Event-Driven Programming



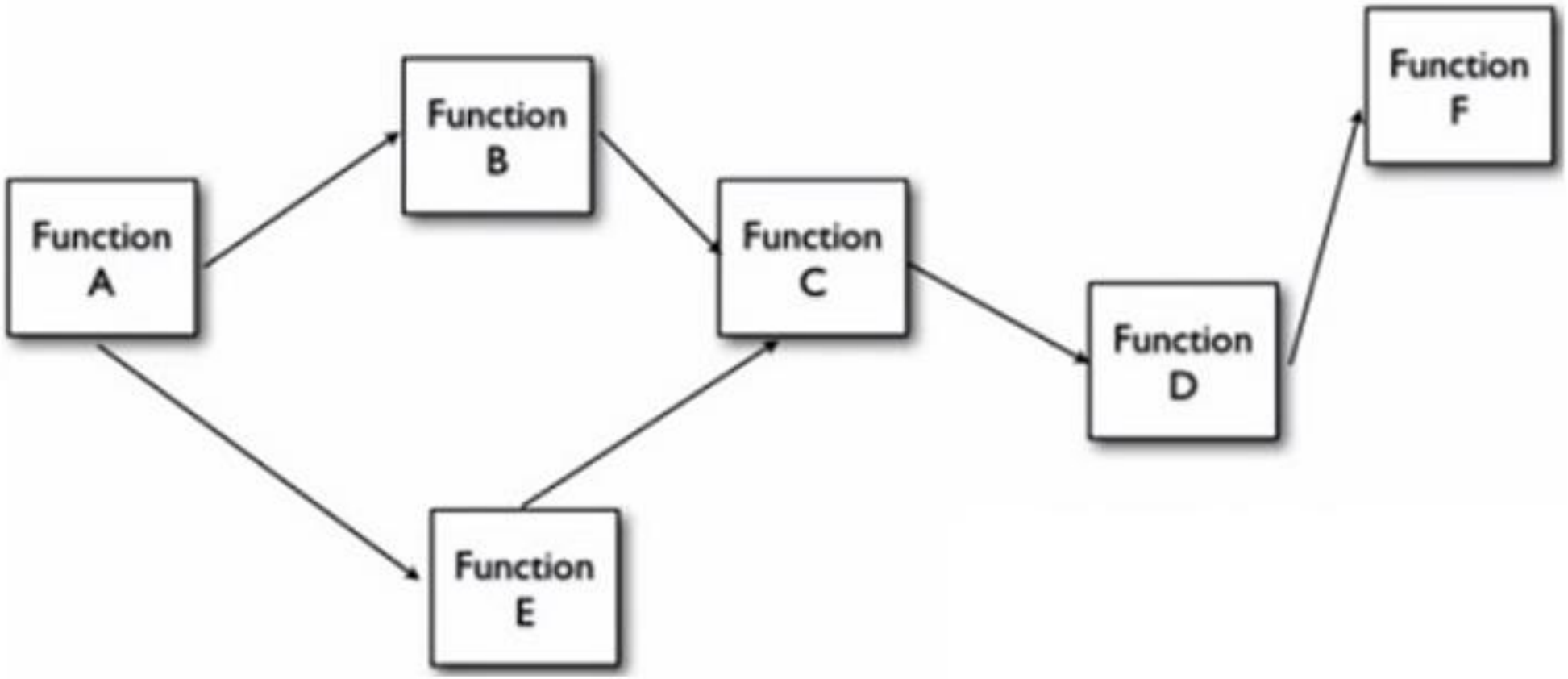
Imperative Languages

Scripting Programming



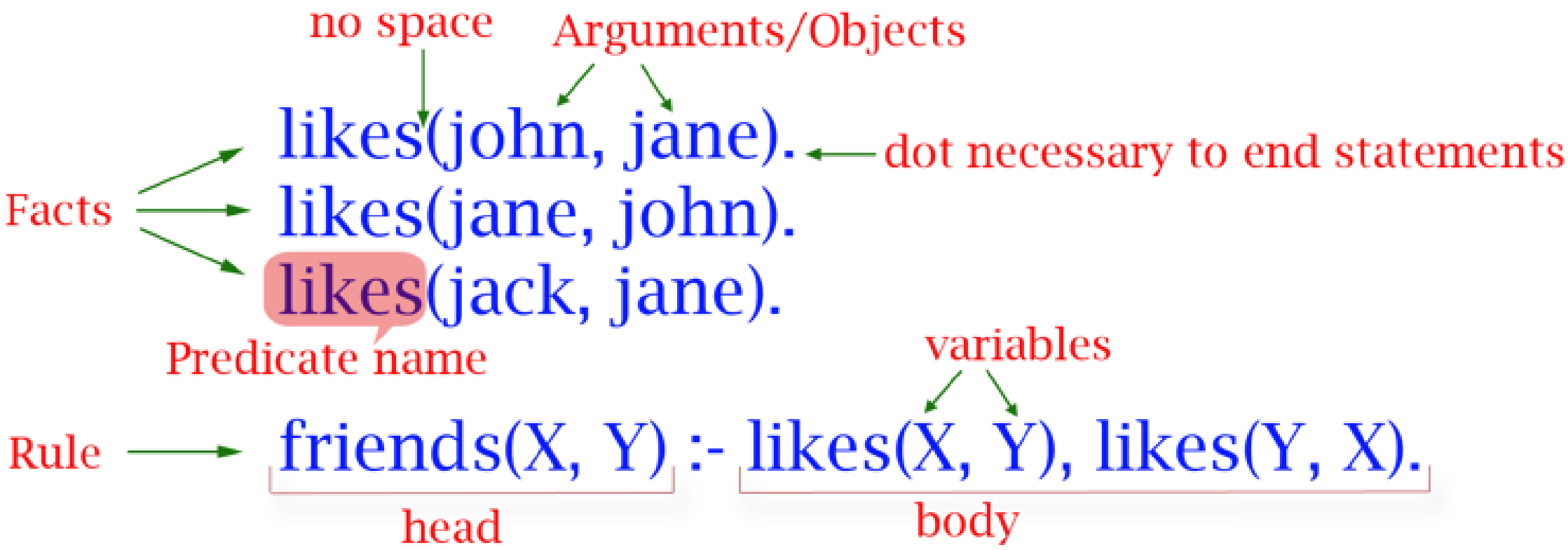
Imperative Languages

Von Neumann Programming



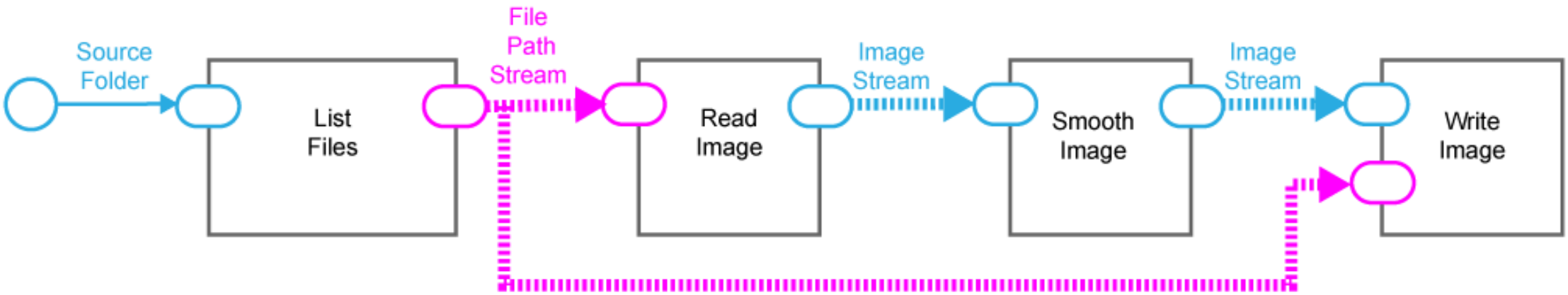
Declarative Programming

Functional Programming



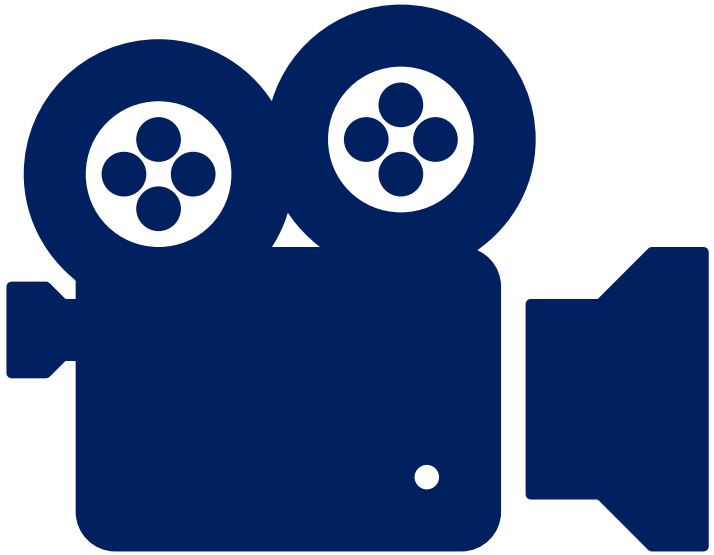
Declarative Programming

Logic Programming



Declarative Programming

Data flow Programming



You should Learn to Program

VIDEO

Video

- If you did not watch this video in Lesson 1, watch it here:
- **Video:**

You Should Learn to Program: Christian Genco at TEDxSMU - Video

Get Started for Programming

- We're going to launch into an activity right now that reveals an important principle of programming.
- Try your best, and we'll discuss at the end.

Language Struct -> Programming Project

- So long as there are multiple ways to interpret language, we cannot have perfect precision.
- If we rigorously define the meaning of each command we use, then we can avoid misinterpretation and confidently express algorithms.
- This is different from the way we normally think and talk, and it might even take a while to get comfortable with communicating in this way.

Imagination -> Reality

- **Ambiguity** in human language led to issues or at least difficulty in creating the arrangements.
- We need to create a well-defined set of commands that all parties can agree upon for expressing the steps of a task.
- We need a **programming language**.

Abstraction -> Programming Language

- Today we saw how human language may not always be precise enough to express algorithms, even for something as simple as building a small LEGO arrangement.
- The improvements you have suggested actually create a new kind of language for expressing algorithms, which we as computer scientists call a **programming language**.
- In the coming unit we are going to learn a lot more about how we can use programming languages to express our ideas as **algorithms**, **build new things**, and **solve problems**.

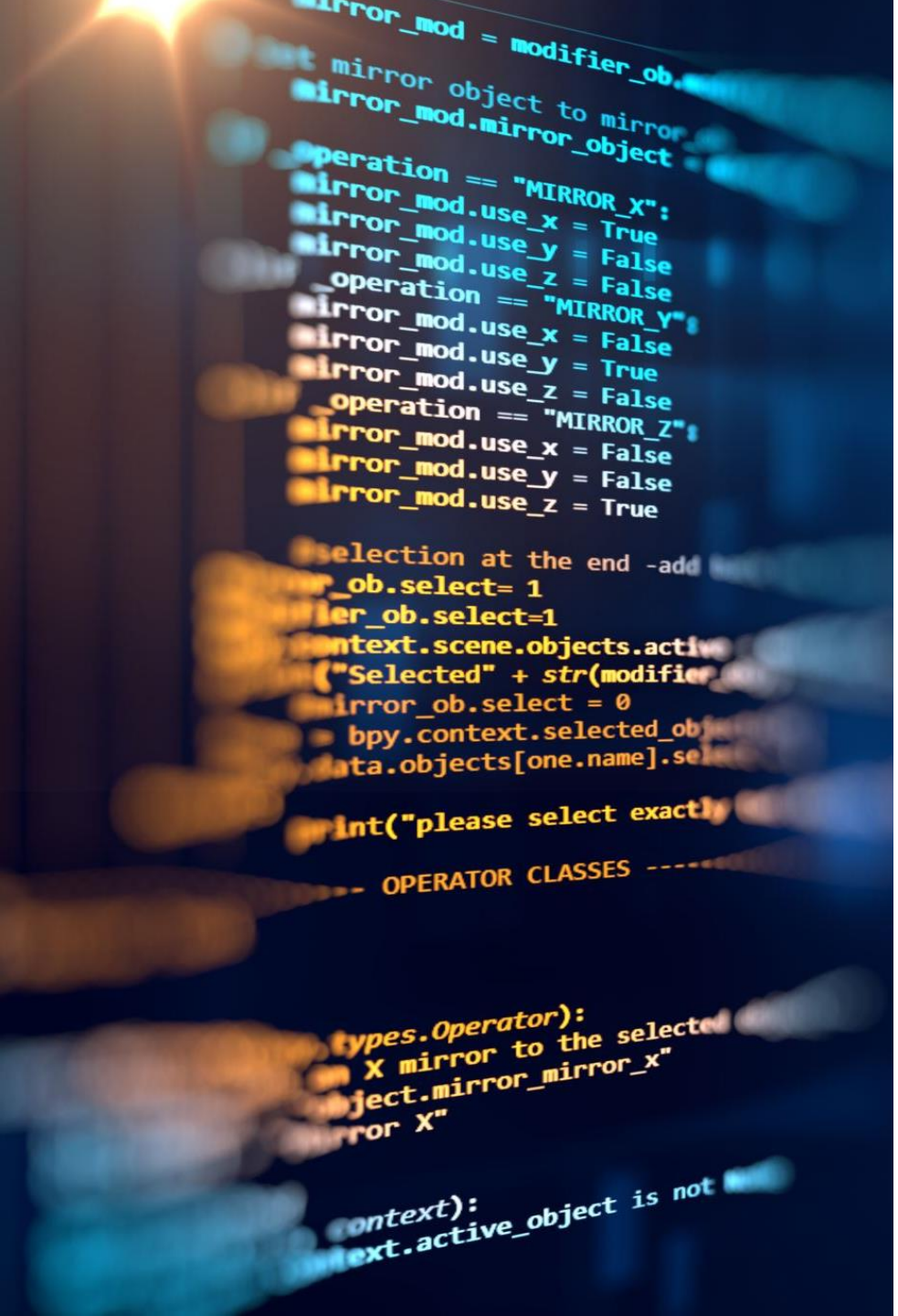
Programming

Building Block -> LEGO Project

Imagination -> Reality

Abstraction -> Implementation

Language Struct -> Programs



Code Studio

- Finish up in Code Studio with Assessments and Reflections