Objectives

- Compiler Optimizations
- Java vs Python
- Software Development

1

Review

1. What are 3 different ways to categorize Java stream classes?
2. Java provides a bunch of classes that we can combine to get the functionality we want. What are the tradeoffs of that design decision?
3. What are the 3 standard streams? (not Java-specific)
   - How do we refer to those streams in Java?
4. How can we package our Java code for easy distribution?
5. How do we tell Java where to look for classes we want to use in our code?
6. What is compiling vs interpreting?
   - What are examples of compiler optimizations?

2

Review: Stream Categories

1. Categorize based on flow of stream
   1. Input
   2. Output
2. Categorize based on type of data they handle
   1. Bytes: `InputStream OutputStream`
   2. Text: `Reader Writer`
3. Categorize based on their source
   1. Data Source (primary source)
   2. Filtered (another stream)

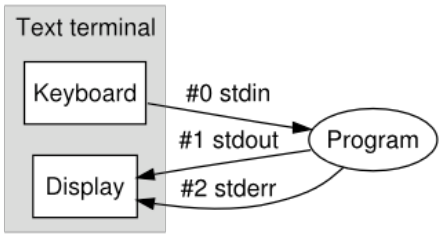
Review: Stream Design Decisions

Combine different types of streams
to get functionality you want

- Alternative: Creating a class for every combination would result in even more classes and a lot of redundant code
  - Consider what is required if some functionality must be updated
  - Tricky for user to pull together various streams BUT also would be hard to find the class you want that has the right combination of functionality

Review: Standard Streams

- Preconnected streams
  - Standard Out: stdout
  - Standard In: stdin
  - *Standard Error: stderr*
    - For error messages and diagnostics
    - In Java: `System.err`



- **Benefit:** separate *output* vs *error* messages
  - Can save outputs in two different files, e.g., error.log vs output.log
  - Eclipse (and other IDEs) differentiates between output (black text) and error (red text)

Review: Jar (Java Archive) Files

- Archives of Java files
- Package code into a neat bundle to distribute
  - Easier, faster to download
  - Easier for others to use
- `jar` command: create, view, and extract Jar files
  - Works similarly to `tar`

```
jar cf myapplication.jar *.class
```

- Run it using `java`

```
java -jar myapplication.jar
```

Review: Classpath

- Tells the compiler or JVM where to look for user-defined classes and packages (jar files)
  - Often when using third-party libraries
- Similar to PYTHONPATH
- Typically know it needs to be set when there are "Class not found" error messages in your code but you have the appropriate import

Review: Compiled vs Interpreted Languages

In pure forms

Compiled

- – Spends a lot of time analyzing and processing the program
- Resulting executable is some form of machine- specific binary code
- Computer hardware interprets (executes) resulting code
- ✓ Program execution is fast
  - Efficient machine/byte code generation
  - Performance gains

Interpreted

- ✓ Relatively little time spent analyzing and processing the program
- Resulting code is some sort of intermediate code
- Another program interprets resulting code
- – Program execution is relatively slow
- ✓ Faster development/prototyping

Compiler Optimization Examples*

- What is the optimization?
  - How does it make the code more efficient?
- For each optimization
  - Should you transform the code yourself to do that optimization?
  - Or, is it something that only the compiler should do?

*Not literally what the code optimizations look like
- Not in Java code but in byte code
- CSCI210 may help illuminate why these decrease runtime

Compiler Optimization Examples

Original:

```
for(int i = 0; i < 10; i++ ) {
        int j = 10;
        System.out.println(i + ", " + j);
}
```

Optimization 1

```
int j = 10;
for(int i = 0; i < 10; i++ ) {
        System.out.println(i + ", " + j);
}
```

Optimization 2

```
for(int i = 0; i < 10; i++ ) {
        System.out.println(i + ", " + 10);
}
```

Compiler Optimization Examples

Original:

```
for( int i = 0; i < 10; i++ ) {
        if( i == 0 ) {
                System.out.println("Do this");
        }
        else {
                System.out.println("Do that");
        }
}
```

Optimization 1

```
System.out.println("Do this");

for( int i = 1; i < 10; i++ ) {
        System.out.println("Do that");
}
```

Optimization 2

```
System.out.println("Do this");
System.out.println("Do that");
System.out.println("Do that");
System.out.println("Do that");
…
```


Compiler Optimization Examples

Original:

```
public void f(int i) {
    a[0] = i + 0;
    a[1] = i * 0;
    a[2] = i - i;
    a[3] = 1 + i + 1;
}
```

Optimization 1

```
public void f(int i) {
  a[0] = i;
  a[1] = 0;
  a[2] = 0;
  a[3] = i + 2;
}
```

Compiler Optimization Examples

Original:

```
public void f(int i) {
    a[0] = i + 0;
    a[1] = i * 0;
    a[2] = i - i;
    a[3] = 1 + i + 1;
}
```

- Why is the code written like this? It seems silly!
- Likely after some previous optimizations
  - Ex: know variable is a constant

Optimization 1

```
public void f(int i) {
  a[0] = i;
  a[1] = 0;
  a[2] = 0;
  a[3] = i + 2;
}
```


Compiler Optimization Examples

Original:

```
int add(int x, int y) {
    return x + y;
}

int sub(int x, int y) {
    return add(x, -y);
}
```

add method stays the same

Optimization 1

```
int sub(int x, int y) {
    return x + -y;
}
```

Optimization 2

```
int sub(int x, int y) {
    return x - y;
}
```

Compiler Optimization Examples

```
class Parent {
       void final f() {
           System.out.println("f");
       }
}
```

```
for( Parent p : parentArray ) {
       p.f(); // check p's actual type at runtime
                // and execute its method f
}
```

Optimization:

```
for( Parent p : parentArray ) {
       System.out.println("f");
}
```


Compiler Tradeoffs

- Upfront costs
  - Searching for optimizations
  - Make optimizations
    - Typically not Big-O efficiency improvements (unless program is written really inefficient)
- Improved runtime
  - Expect executed many more times than compiled

Should You Apply the Optimization?

- Your priority: keeping code abstract to make it easier to change
- If you can apply the optimization without making the code harder to change, you should do it

17

LANGUAGE COMPARISON

18

Language Comparison

Java

Python

1) Focus on their characteristics (just the facts, not tradeoffs)
2) Pros and cons, preferences

Language Comparison

Java

- Entirely Object-oriented*
  - Procedural
  - Functional - newer
- Statically, strongly typed
- Compiled

Python

- Object-oriented
  - Also procedural and functional programming
- Dynamically, strongly typed
- Interpreted

Pros and cons of using each?

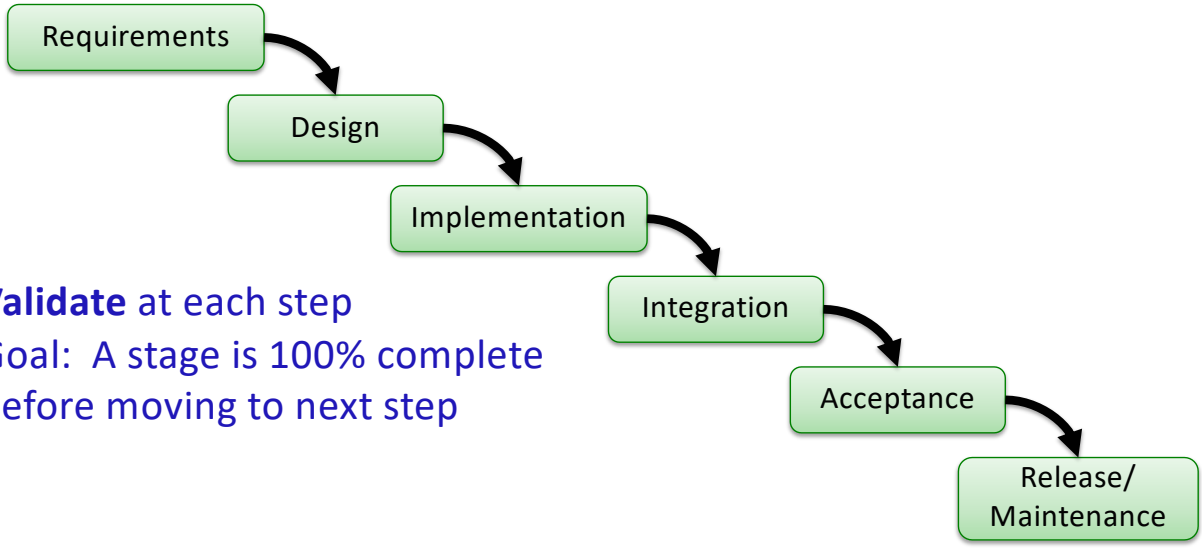
Rest of the semester

- Shift from learning Java, specifically, to learning how to develop software (abstractly) with Java as our implementation/example
- Why Java?
  - Popular language
  - Many frameworks and tools for Java
  - Java's structure allows for strict adherence to design techniques
- Just a start on Java
  - You'll need to continue learning more Java on your own

"There is no single development, in either technology or in management technique, that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity."
– Fred Brooks

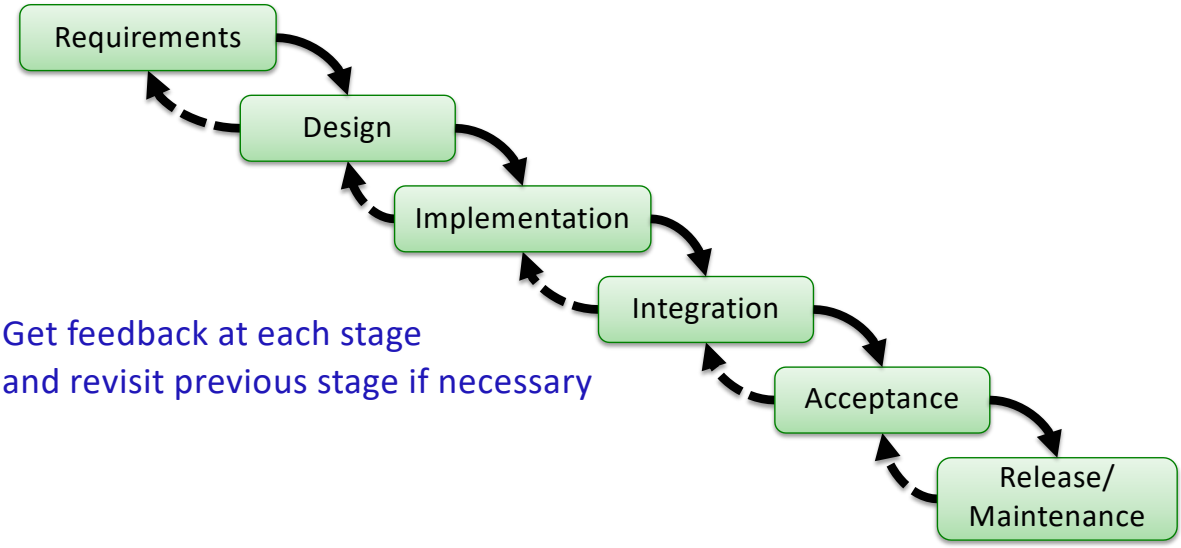
SOFTWARE DEVELOPMENT

Traditional Software Engineering Process: Waterfall Model



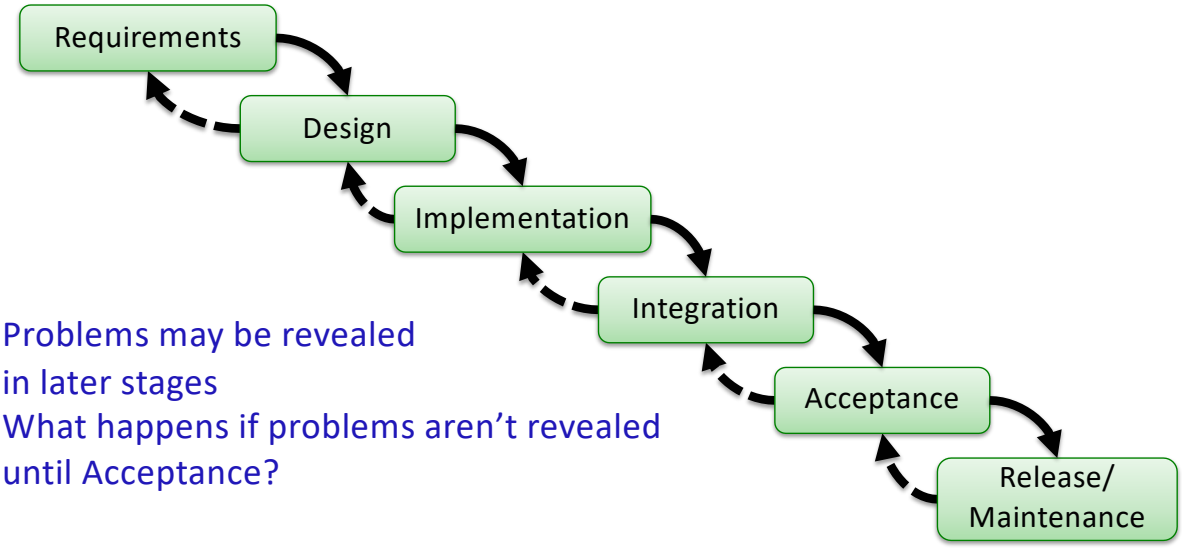
Validate at each step
Goal: A stage is 100% complete before moving to next step

Feedback in Waterfall Model



- Get feedback at each stage and revisit previous stage if necessary

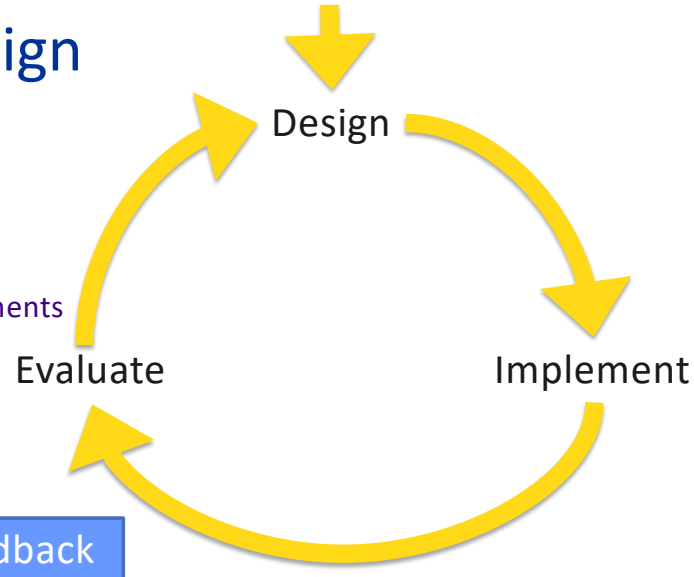
Feedback in Waterfall Model



- Problems may be revealed in later stages
- What happens if problems aren't revealed until Acceptance?

27

Iterative Design



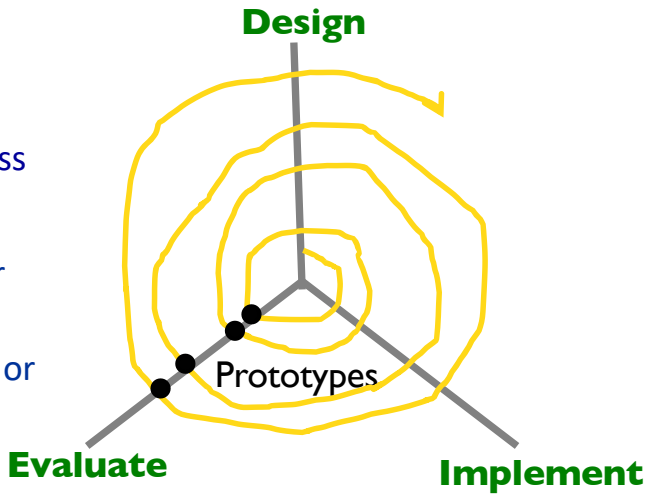
Get feedback/requirements from users/clients

Goals: Frequent feedback
→Identify problems early
→Higher quality product

28

Spiral Model

- Idea: smaller prototypes to test/fix/throw away
  - Finding problems early costs less
- In general…
  - Break functionality into smaller pieces
  - Implement most depended-on or highest-priority features first



Radial dimension: cost

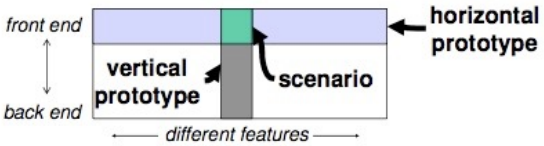
[Boehm 86]

Prototypes Overview

- Sample of application
  - Often: Demonstrate one part/purpose
    - Focus on one thing, not the whole thing
- Purpose/Dimensions
  - Functionality
  - Interaction
  - Implementation

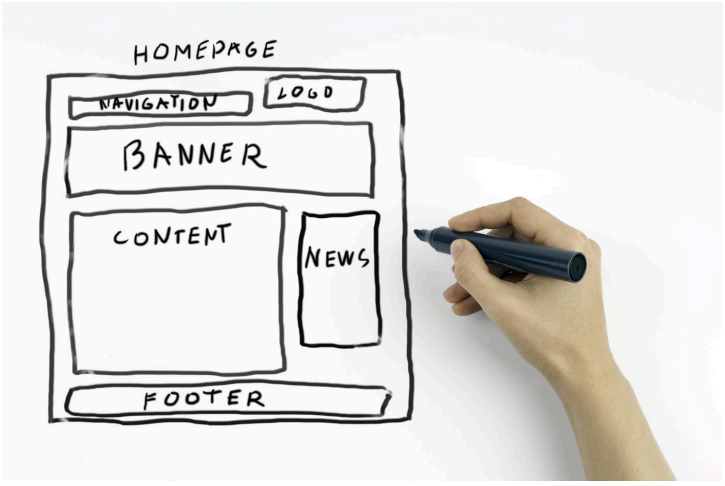
Prototypes: Fidelity

- How similar to finished product
- Low fidelity: omits details
- High fidelity: closer to finished project
- Multi-dimensional
  - Breadth: % of features covered
    - Low-breadth: Only enough features for certain tasks
  - Depth: degree of functionality
    - Low-depth: Limited choices, canned responses, no error handling



From Nielsen, *Usability Engineering*

Low Fidelity Prototypes



- Media: Paper, White board
- Examples: storyboard, sketches, flipbook, flow diagram

High Fidelity Prototypes

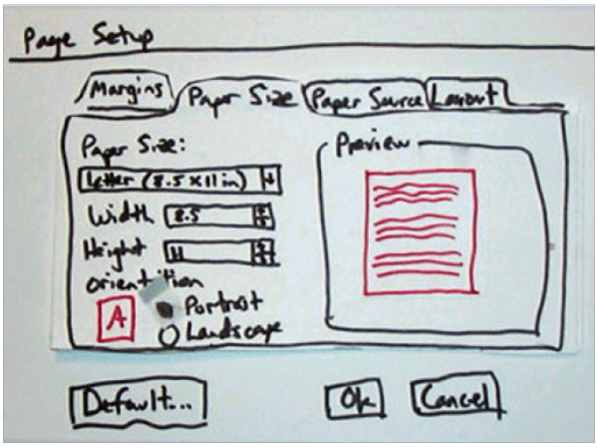
- Media: HTML (non-interactive), PowerPoint, Video
- Examples: Mockups, Wizard of Oz

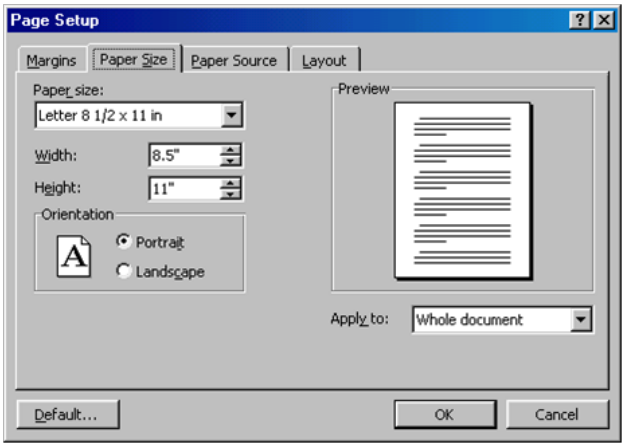
Virtual Peer for Autistic Children

http://articulab.hcii.cs.cmu.edu/

33

Comparing Low-Fidelity and High-Fidelity Prototypes

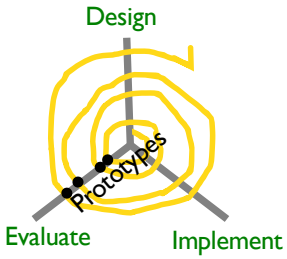




How do they differ in the kinds of things you can test and get feedback about?

34

Summary: Spiral Model/Iterative Design Model Benefits



- Builds in getting feedback from client
  - Demo prototypes or working versions of [parts of] application
  - Clients' requirements may change
  - Clients' requirements may be ambiguous or were misinterpreted
- Makes project development more ***agile***
  - Goal: find problems early
  - Easier to throw away cheaper early prototypes
  - Adjust/adapt to changes

Looking Ahead

- Read slides about testing, JUnit before Friday's class
  - Canvas quiz
- Goal: Hands-on lab in class on Friday
- Thursday – cancel office hours; email me with questions or to make an appointment