Objectives

- Software Development
- Testing
- Collaboration

1

Review

1. What are differences between compiled and interpreted languages?
   - What are the tradeoffs in compiling?
2. Compare and contrast Java and Python
   - Characteristics
   - Benefits of each
3. True or False. If the compiler is finding/applying optimizations to your code, you are writing your code poorly.
4. What are two models of the software development process?
   - What are their benefits? Limitations?
5. What are prototypes?

2

Review:

In pure forms

Compiled vs Interpreted Languages

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

Review: Compiler Tradeoffs

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

Review: 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

5

Review: Language Comparison

Java

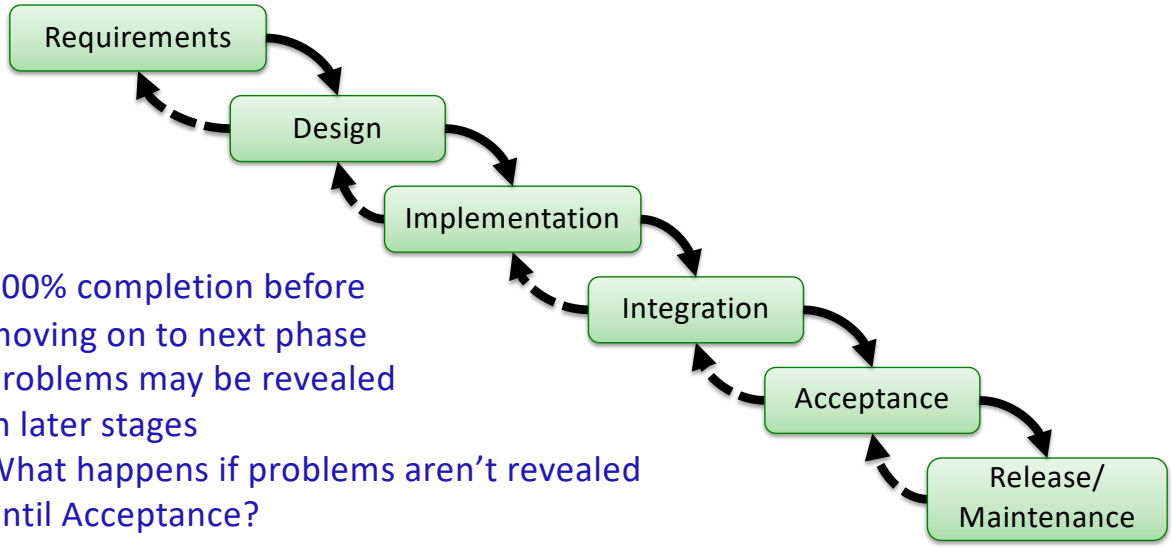
- Entirely Object-oriented*
  - Functional programming mimicked through using just static methods within a class
- Statically, strongly typed
- Compiled

Python

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

6

Review: Waterfall Model

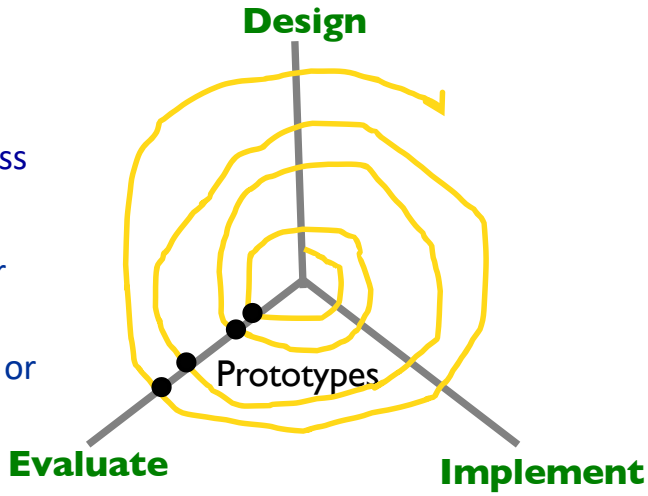


- 100% completion before moving on to next phase
- Problems may be revealed in later stages
- What happens if problems aren't revealed until Acceptance?

7

Review: 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



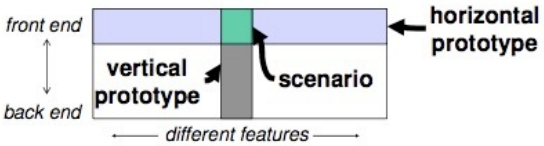
Radial dimension: cost

[Boehm 86]

8

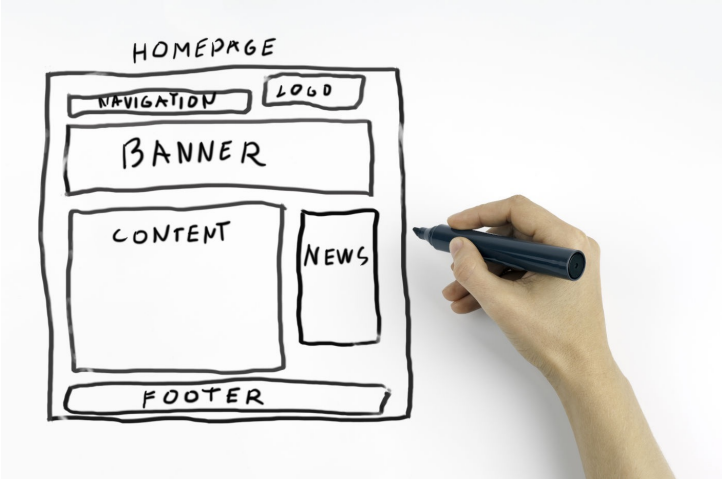
Prototypes: Fidelity

- 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

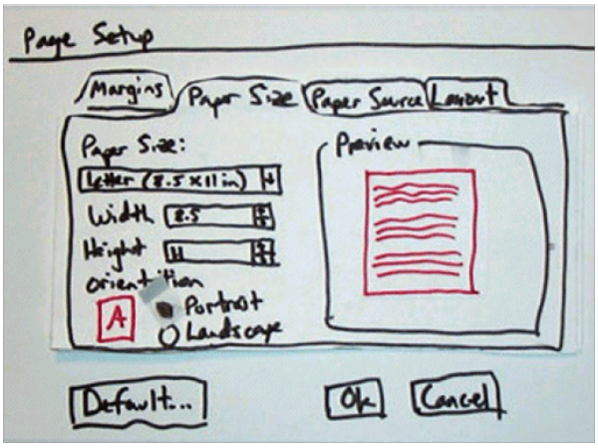
- Media: HTML (non-interactive), PowerPoint, Video
- Examples: Mockups, Wizard of Oz

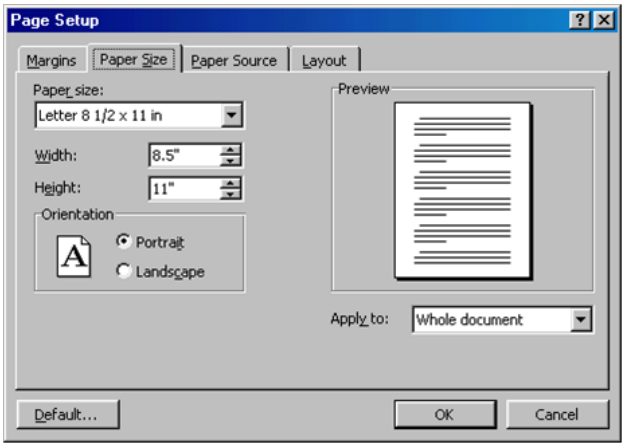
Virtual Peer for Autistic Children

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

11

Comparing Low-Fidelity and High-Fidelity Prototypes

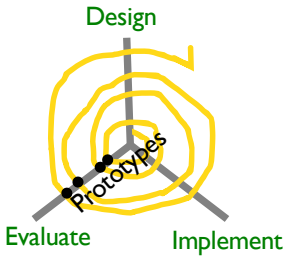




How do they differ in the kinds of things you can test and get feedback about?

12

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

How to Implement an Effective Solution

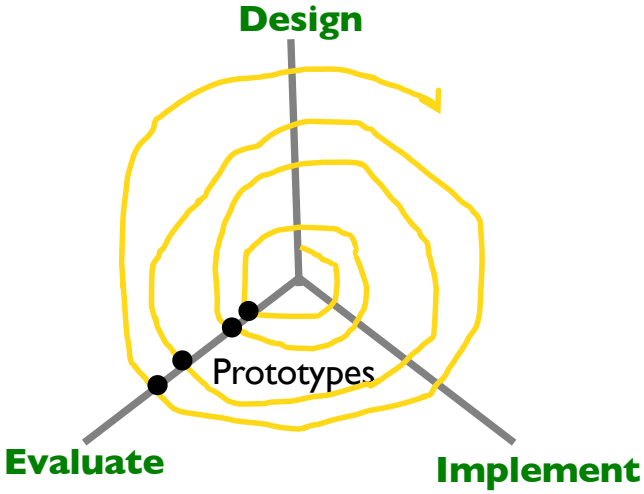
1. Understand the problem
2. Understand external constraints
3. Design an effective solution to the problem
4. While designing the solution, design some ***tests*** to verify that the problem is solved (and remains solved)
5. Code the effective solution to the problem
6. Teach other team members about your solution to the problem

How to Implement an Effective Solution

1. Understand the problem (interact with ***people***)
2. Understand external constraints (interact with ***people***)
3. Design an effective solution to the problem
4. While designing the solution, design some ***tests*** to verify that the problem is solved (and remains solved)
5. Code the effective solution to the problem
6. Teach other team members about your solution to the problem (interact with ***people***)

Agile Development: Breaking Down Further

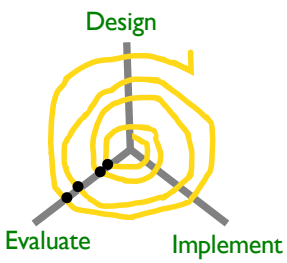
- Project's development process: Spiral Model
- What does this look like day to day?
  - Agile development is a common implementation



Radial dimension: cost

[Boehm 86]

Iterative Development Steps



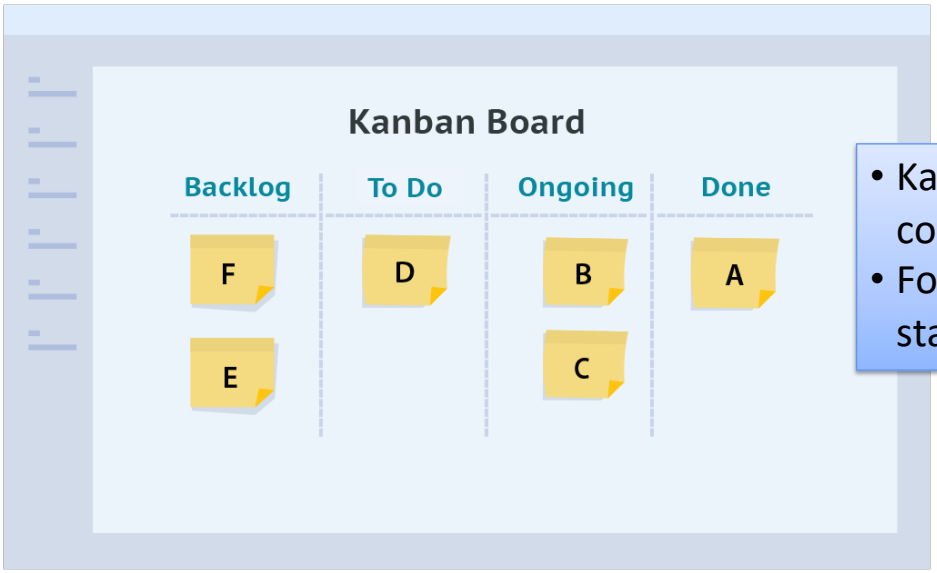
1. Design a {method, class, package}
2. Implement the {method, class, package}
3. Test the {method, class, package}
4. Fix the {method, class, package}
5. Deploy the {method, class, package}
6. Get feedback – from team lead or customer
   - Probably will require modifications to design
   - May even need to rollback a previous version
7. Repeat, building up

Agile Development Framework: Scrum

- Product owner creates prioritized wish list: *a product backlog*
- Team works in a *sprint*, usually 2-4 weeks
  - During planning, team picks a subset of wish list, *a sprint backlog*, and decides how to implement those pieces
  - Daily Scrum: team meets daily to assess its progress
    - ScrumMaster keeps the team focused on its goal
  - At end of sprint, work should be potentially shippable:
    - ready to hand to a customer, put on a store shelf, or show to a stakeholder
  - The sprint ends with a sprint review and retrospective
- Repeat sprint

`https://www.scrumalliance.org/why-scrum`

Tools to Help: Kanban Board



- Kanban is continuous, fluid.
- Focus on short start to finish time

 https://www.digite.com/kanban/what-is-kanban/

Summary: Agile Development Processes

- Goal: Effective software development
  - Planning with frequent feedback
  - Iterative, continuous improvement
- Involves teams, frequent communication
  - Teams: development, quality & assurance, …
- Lots of variations – often company- or team-specific

SOFTWARE TESTING PROCESS

21

A Bad Role Model

http://imgur.com/HBSbn

22

Microsoft Windows Vista Testing

- Beyond their internal testing …
  - 5 million people beta tested
  - 60+ years of performance testing
  - 1 Billion+ Office 2007 sessions
- Still, users found correctness, stability, robustness, and security bugs

Type 1 Bugs: Compile-Time

- Syntax errors
  - Missing semicolon, parentheses
- Compiler notifies of error
- Cheap, easy to fix

Type 2 Bugs: Run-Time

- Usually logic errors
- Expensive to locate, fix

Aside: Objections to "Bug" Terminology

- "Bug"
  - Sounds like it's just an annoyance
    - Can simply swat away
  - Minimizes potential problems
  - Hides programmer's responsibility
- Alternative terms
  - **Defect**
  - **Fault**

Types of Testing

(Non-Exhaustive)

- Black-box testing
- White-box testing
- Non-functional testing
- Acceptance testing

Ideas about or definitions of any of these?

Types of Testing

(Non-Exhaustive)

- Black-box testing
  - Test *functionality* (e.g., the calculator)
  - No knowledge of the code
  - Examples of testing: boundary values
- White-box testing
  - Have access to code
  - **Goal**: execute *all* code
- Non-functional testing
  - Performance testing
  - Usability testing (HCI)
  - Security testing
  - Internationalization, localization
- Acceptance testing
  - Customer tests to decide if they accept the product

Discussion: Your Testing Process

- How do you test?
- Categorize what you test/look for
- Are you a good tester? Why or why not?
  - What do you do well?
  - What do you need to get better at?

Common Bad Development Approaches

- Run the code. Did it do what you expect? <shrug/>
- Identify bug. Fix the bug on the test case that revealed the error. Don't test the other cases.
  - Similar: made a change to code (famous last words: "it shouldn't affect anything") and don't retest
- Random (only) testing

Software Testing Issues

- How should you test? How often?
  - Code may change frequently
  - Code may depend on others' code
  - A lot of code to validate
- How do you know that an output is correct?
  - Complex output
  - Human judgment?
- What caused a code failure?

➥ Need a *systematic*, *automated*, *repeatable* approach

Some Approaches to Testing Methods

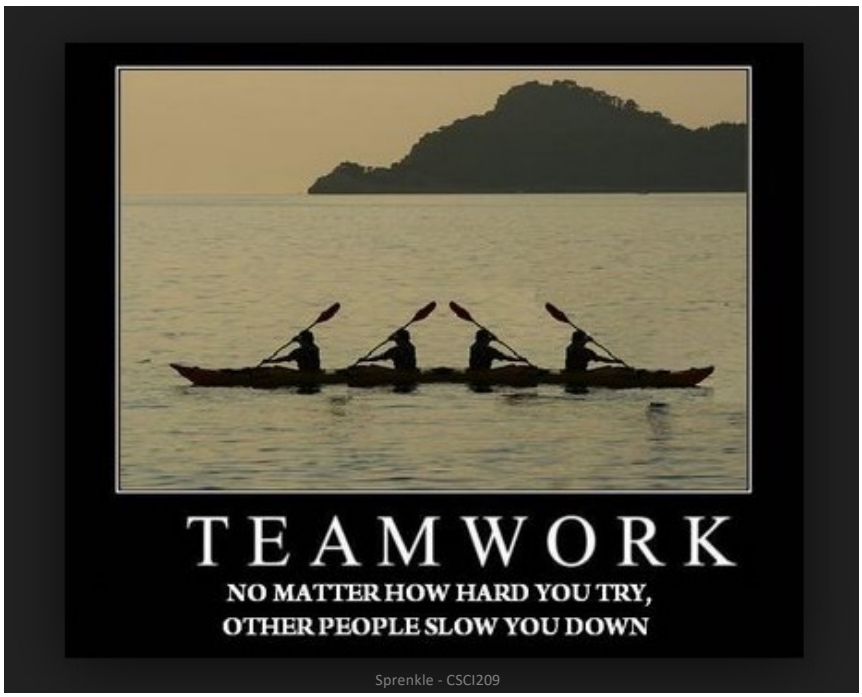
- Typical case
  - Test typical values of input/parameters
- Boundary conditions
  - Test at boundaries of input/parameters
  - Many faults live "in corners"
- Parameter validation
  - Verify that parameter and object bounds are documented and checked
  - Example: pre-condition that parameter isn't null

➥ All black-box testing approaches

Looking Ahead: Team Projects

COLLABORATION

33



34

35

Think about Team (Group) Projects

- Why did some work well?
- Why were some disasters?

36

Teams Work Best When They are **Interdependent**

- In code terms, we want *loose coupling*
  - Depend on each other but don't depend on their details
- Consider
  - Are you allowing your team to truly be interdependent?
  - Who might be you be ignoring?
  - Who might be allowing themselves to feel inadequate?
  - How do you show appreciation for each other and yourself?

Collaboration: Team Project

- Version Control does not eliminate need for communication
  - Process becomes much more difficult if developers do not communicate
- Keep the version to be graded in `main` branch
- Before picking up again, **pull** the repository
  - Get others' changes

Collaboration: Workflow – Seeking Feedback

1. Create a branch for your work
   - Commit periodically
   - Write descriptive comments so your team members know what you did and why
2. Push your branch
3. Open a ***Pull Request*** on your branch
   - Discuss and review potential changes – can still update
   - You can tag your teammates to let them know that you've completed your work
4. Merge pull request into `main` branch

Collaboration: Workflow

1. Create a branch for your work
   - Commit periodically
   - Write descriptive comments so your team members know what you did and why
2. Switch to `main`
3. Pull `main` branch
4. Merge your branch into the `main` branch
   - Handle merge conflicts
   - Commit
5. Push `main` branch

First Team Project – Unit Testing!

- Released Monday
- Create your own teams of 3, with team names
  - Email me by Monday at 5 p.m., CCing your teammates with your team members and team name
  - I can help with matching

Collaboration: Team Project

- Need to talk about the solution
- Discuss your plan, e.g.,
  - Your system for testing to make sure that you test everything
  - Your assumptions
  - Organization of test cases
  - Naming
  - Division of labor
- Maintain planning documents too
  - in GitHub or elsewhere

Looking Ahead

- Testing – Canvas reading quiz
- Make teams for upcoming testing project
  - Email me
- Monday – bring laptop or can use lab machine
  - Clone the lab project – wait on email from me