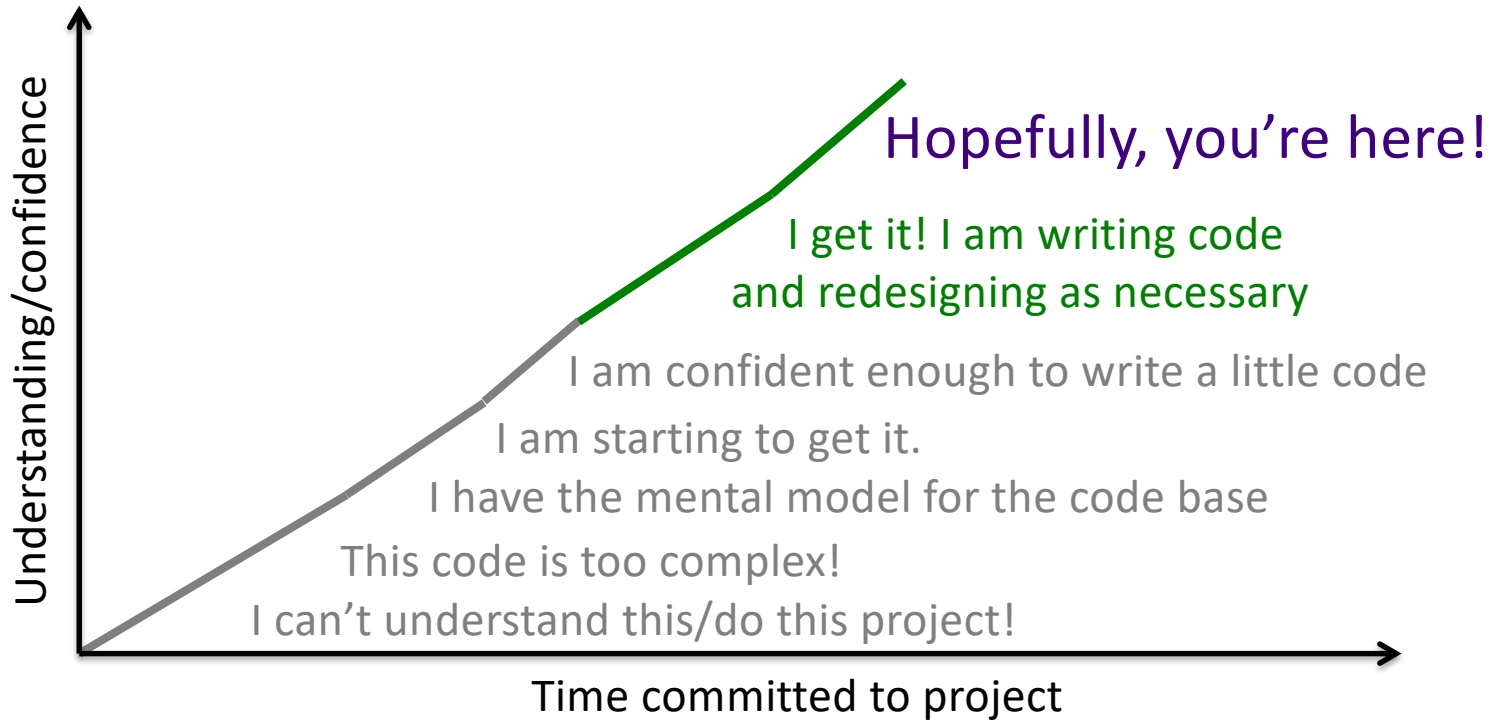
# Objectives

- Picasso!

# Review: Typical Trajectory of Projects




Preliminary deadline retrospective
(starting early, testing)

# Project Deliverables Timeline

| Deliverable | Who | Weight | Done? |
|---|---|---|---|
| Preparation Analysis | Individual | 10% | ✅ |
| Preliminary Implementation | Team | 15% | ✅ |
| Intermediate Implementation | Team | 15% | |
| Final Implementation | Team | 45% | |
| Analysis | Individual | 15% | |

Week 1: Understand code base, analyze/plan project
Week 2: Implement preliminary functionality
Week 3: Implement intermediate functionality
Week 4: Implement final version of application

# Alumnus Note

“The company is very small—only 14 employees, all software developers. Our backend is built in Java, and we use Eclipse and Github, so, in many ways, the development side of things feels like a scaled-up version of the Picasso Project.

Specifically, when I first started, I had a similar experience to that project: trying to make sense of an intimidating amount of existing code, figuring out where and how to inject new functionality, and learning the coding standards and practices already in place. These practices include: "Don't Repeat Yourself", …, a class and the methods within that class should serve a single purpose, bugs should be fixed at the most upstream point, ….”

# Review

1. What is a design pattern?
   - What design patterns have we discussed?
     - What problems do they solve?
   - What design patterns are used in the Picasso project? (could vary by team)
2. Why do we need to convert the input to postfix?
3. Draw the stacks for
   - x+y
   - x+floor(y)*x
   - (x+floor(y))*x
   - var = x + floor(y) * x
   - imageWrap("foo.jpg", x, y+y)

1. What is our git workflow?
2. What is a merge conflict? How do you resolve it?

# Review: Design Pattern

General reusable solution to a commonly occurring problem in software design

- Not a finished design that can be transformed directly into code
- Description or *template* for how to solve a problem that can be used in many different situations
  - "Experience reuse", rather than code reuse

# Design Pattern: **Strategy**

- Defines a family of algorithms, encapsulates each one, and makes them interchangeable
- Allows algorithm/behavior to vary independently of clients that use it
  - Allows behavior changes at runtime
- Design Principle:

Favor ***composition*** over inheritance

# Stack Rules

- The last operation performed is at the top
- For binary operators, the right side is next on the stack, then left

# Draw the Stacks

| Infix | Postfix |
|---|---|
| x+y | x y + |
| x+floor(y)*x | x y floor x * + |
| (x+floor(y))*x | x y floor + x * |
| var = x + floor(y) * x | var x y floor x * + = |
| imageWrap("foo.jpg”, x, y+y) | imageToken x y y + imageWrap |

Using shorthand:

x is IdentifierToken: x

y is IdentifierToken: y

# Merge Conflict

- Occurs when competing changes to the same lines in a file
  - Git doesn't know how to resolve the merge
- Resolving: manually edit the conflicted file to what you want to keep in the merge
  - Could be in GitHub interface or locally
  - Explain your fix in comment
- Check resulting branch still works as expected

# Towards Intermediate Deliverable

- Set up to report errors to users
  - Currently: in the printed output but users aren't going to see that
  - Helpful errors → translated for users
- Opening a file that contains an expression
- Handling new operations
  - Order of operations
  - Assignment statement
- Functions with multiple arguments, image names
- Extensions

# Suggestions

- Check the FAQ for your questions
- Create unit tests, when possible/appropriate
  - Sometimes, you need to *make* it possible
  - Run using coverage tool to see what is (and isn't) covered.
- Draw things (e.g., stacks, trees) out on paper
- Trace through the code

# Project Goals

- Everyone contributes significantly to the project
  - Has at least one part where they can say “I made this!”
- Everyone understands the code and its design
  - All of it.  Well, 90% of it, at least at a high level
- Everyone feels valued as a team member

# Contributing to the Team

- Always some concern that your grade is based on lines of code written
  - Not all lines of code are equal
  - Number of lines of code is not a good indicator of work or quality of code
- Variety of opportunities to contribute to the team

# Tip: Compare Binary Operators

- Likely need to implement the equals method in various classes (e.g., Addition, Subtraction, …)
- Stop after you've written and tested two
- Compare the methods
  - Is there a code smell? Refactor!

# Tip: Error Handling

- Don’t do too much translation too soon
- Can mask *your programming* errors (that aren’t user error errors)
  - Example: NullPointerException

# Final Implementation: Documentation

- You leave, I'm still here, trying to use [grade] your code
- Documentation
  - Extensions aren't always obvious
  - State in README
- Javadocs: Purpose of Java classes
  - Update comments
  - Auto-generated daily
  - Can be seen on the project web site

# Deliverables: Tagging

- While given code had compiler errors because of using test-driven development, there should be no compilation errors in deliverables' tagged versions
  - None for final version
  - For intermediate, okay if you have clearly marked test classes for test-driven development

# Secondary Goals

- You're going to figure out that your final design isn't perfect—maybe not even good!
  - Fix more critical and/or smaller things
    - Refactoring!
  - Note larger things
    - analysis/post-mortem due at end of finals week

Good judgment comes from experience.
How do you get experience?
**Bad judgment** works every time.

# Final Project: Project Analysis - Individual

- Understand teammates' design/code/parts
  - *At least* at a high level
- Contents: Description, Planning, Status, Code Analysis, Collaboration, Future Work
  - Complete specification online

# Project Planning

- Review project specifications
- Know what tasks are left
  - Intermediate deadline provides direction, but there are a variety of other tasks that can be implemented.
- Be agile!

# Looking Ahead

- Wednesday: Course Retrospective
- Friday: Intermediate Deadline, Demo
- Finals Week
  - Wednesday: Final Implementation Deadline
  - Friday - noon: Final Analysis