Objectives

- Planning
- Team Work

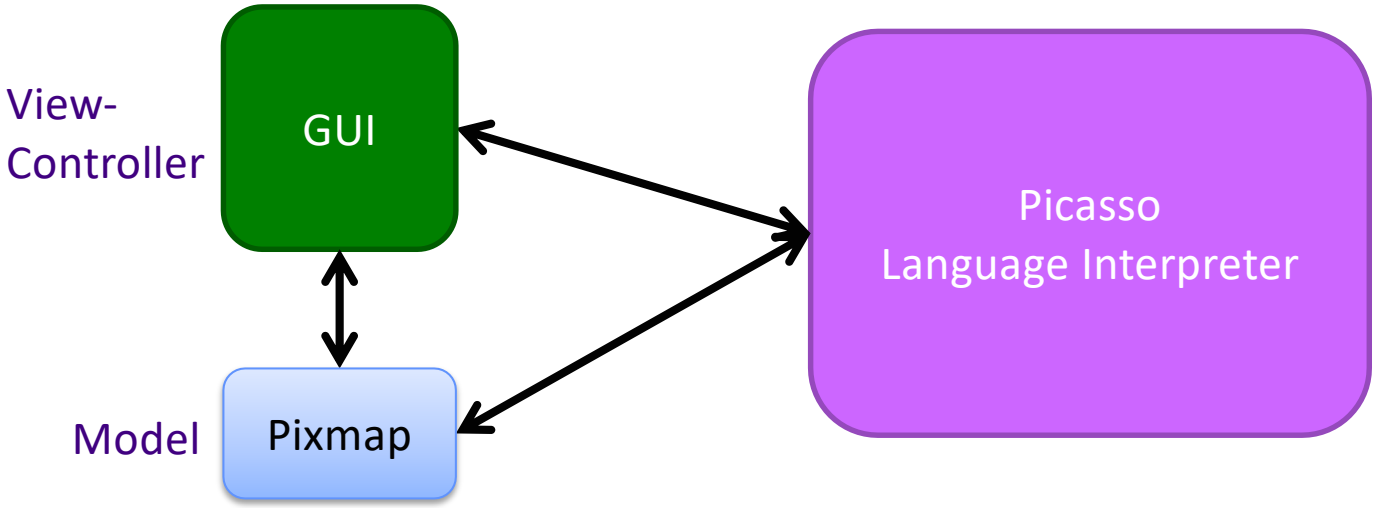
Review: Picasso

- It's okay to be a little intimidated
- Let that motivate you
- But *believe* that you can successfully tackle the project

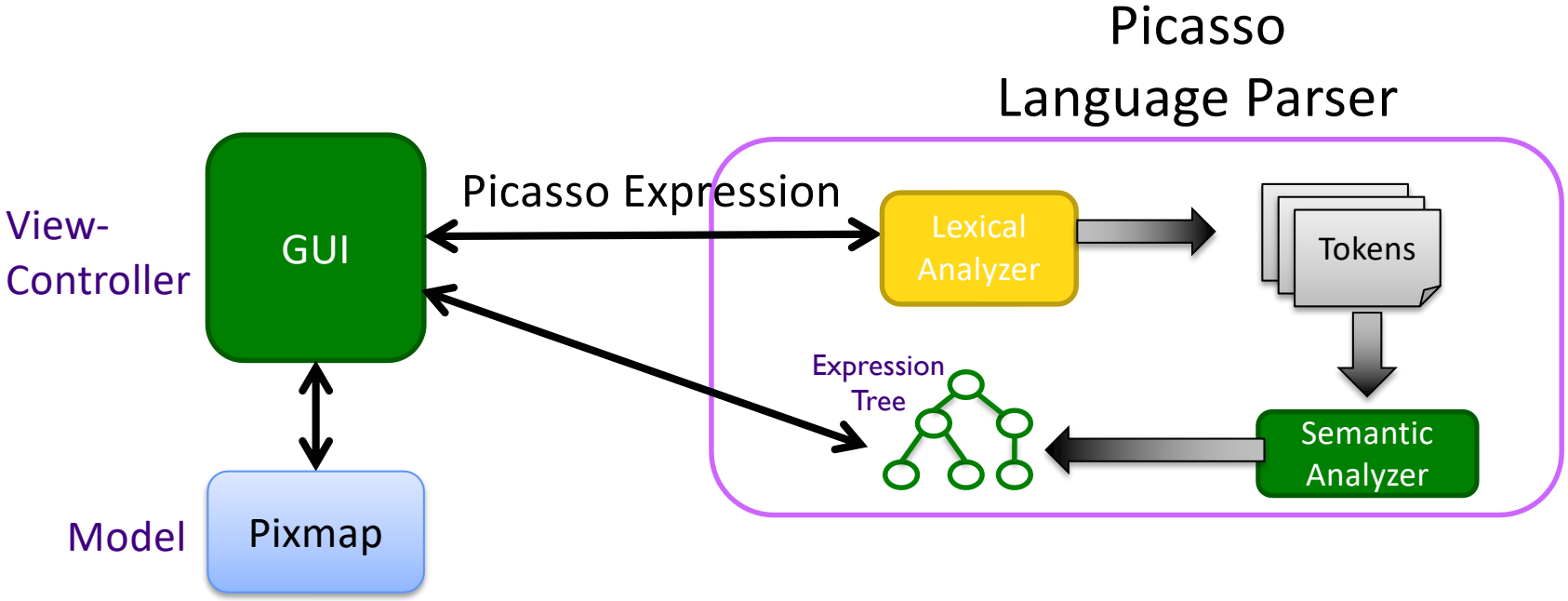
Review

- What is the MVC design pattern? How does it relate to the Picasso project?
- What are the major components of the existing Picasso code base?
- What parts of project need to be completed?
- (Rhetorical) Who are your teammates?

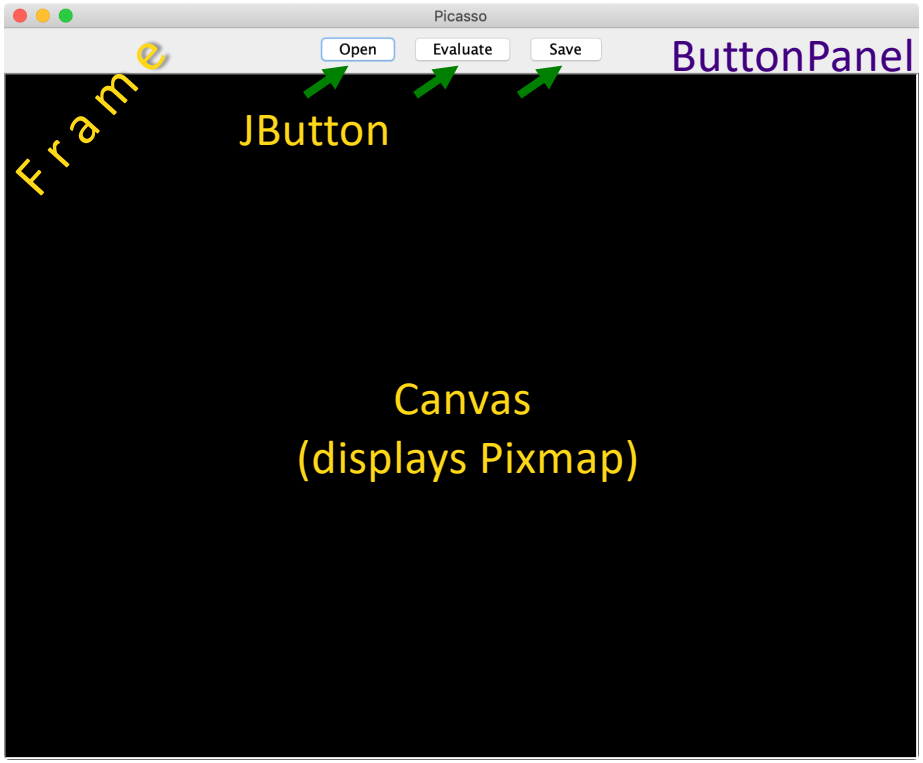
Picasso Architecture



Picasso Architecture



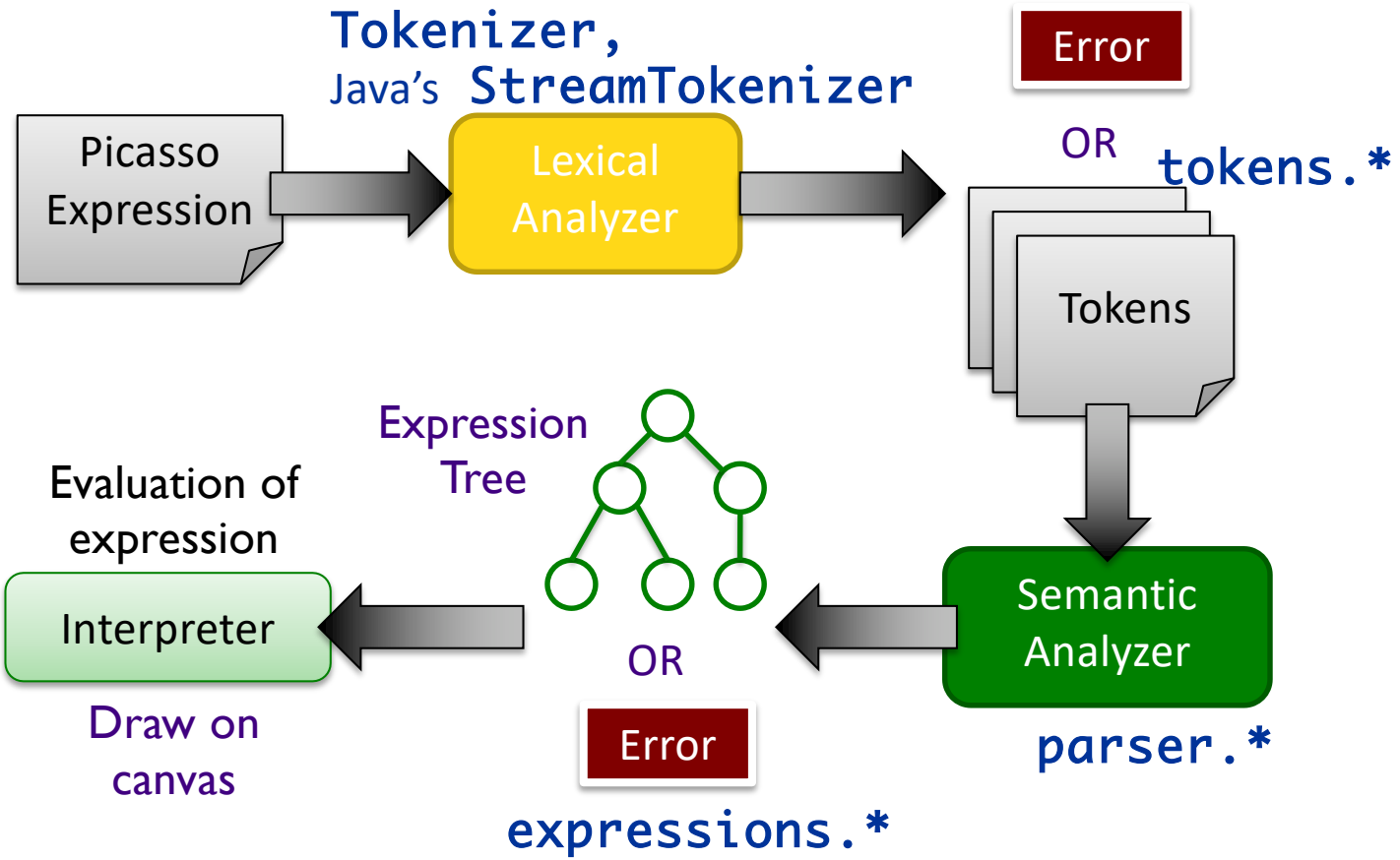
Review: Picasso GUI



Picasso's GUI uses classes from two main Java packages:

- Abstract Windowing Toolkit: java.awt
- Swing: javax.swing

Review: Interpreting the Picasso Language



Code Review: Lexical Analysis

- Process
  - `picasso.parser.Tokenizer`
  - `picasso.parser.tokens.TokenFactory`
- Output:
  - `picasso.parser.tokens.*`

Code Review: Semantic Analysis

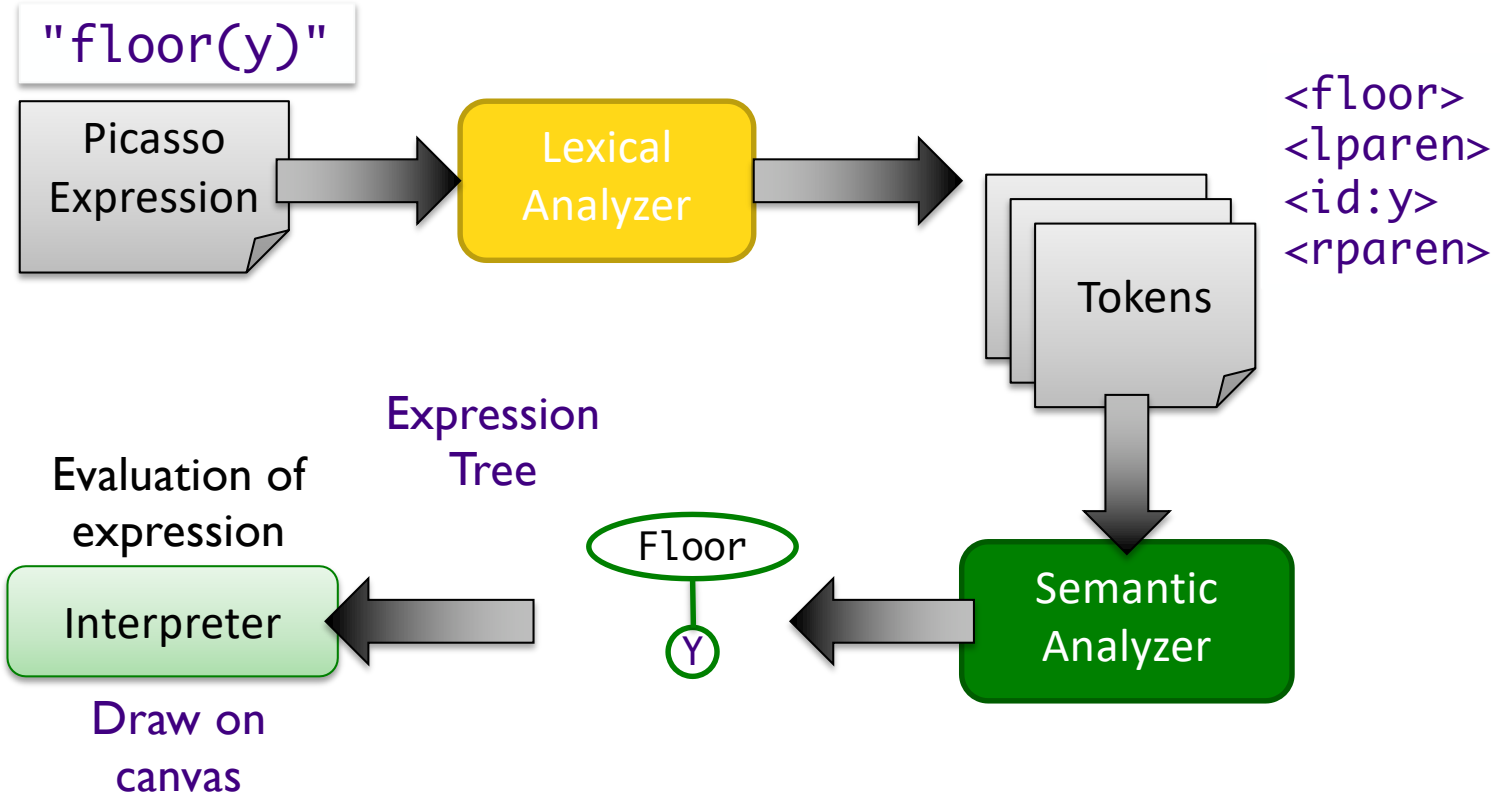
- Process
  - `picasso.parser.ExpressionTreeGenerator`
  - `picasso.parser.SemanticAnalyzer`
  - `picasso.parser.*Analyzer`
- Output
  - `picasso.parser.language.expressions.*`

Code Review: Evaluation

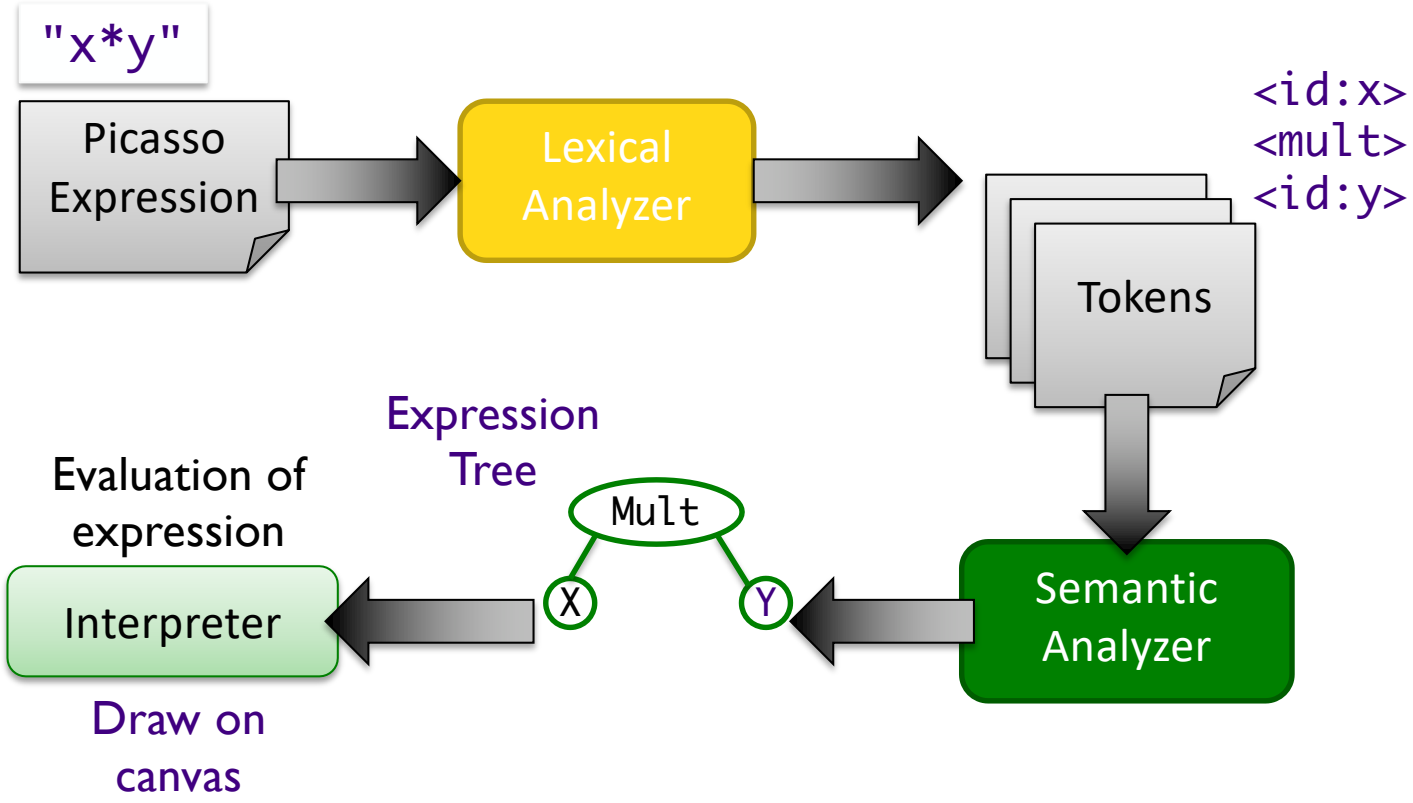
- Process
  - `picasso.parser.language.ExpressionTreeNode`
- Output:
  - `picasso.parser.language.expressions.RGBColor`

- Displayed in `PixMap` on `Canvas`

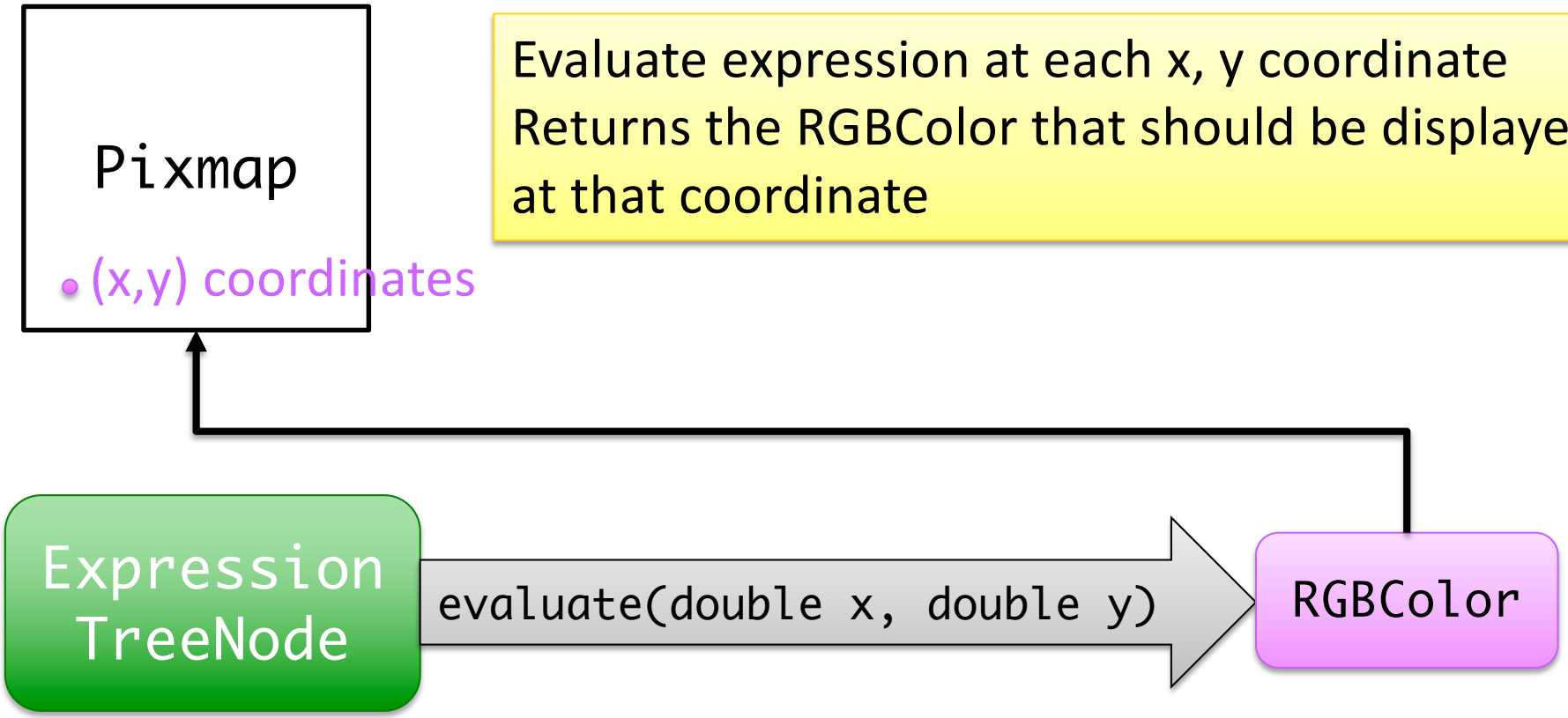
Review: Interpreting the Picasso Language



Interpreting the Picasso Language



Evaluator: Expression Evaluation



What Steps Need To Be Completed?

- Model: Images
  - API
  - State
- GUI
  - Expression user interface (interactive)
  - Open expression files (batch)
  - Call Picasso parser/interpreter
  - Error handling
- Picasso interpreter
  - Parse expressions (functions, operations, variables, …)
    - **Handle errors** appropriately
  - Evaluate expressions
    - Manipulate canvas appropriately
- Extensions
- **TESTING!**

What Classes are Dependent on Each Other?

- How tightly coupled are they?

Dependencies

- Interpreter classes (tokens, analyzer, expression) are very dependent on each other
- Need to hook GUI to Interpreter
- Need to hook Image/Canvas to GUI and Interpreter
- Can test without other pieces but easier and more satisfying to see results displayed

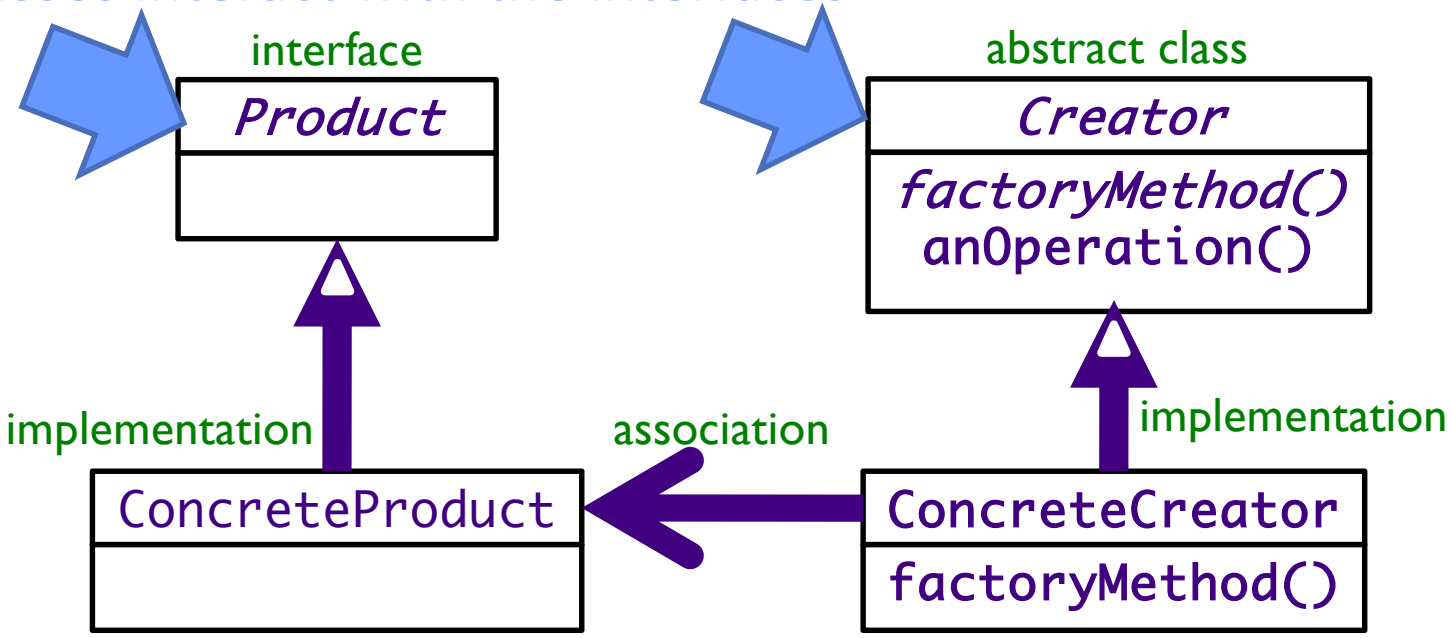
FACTORY DESIGN PATTERN

Design Pattern: **Factory Methods**

- Allows creating objects without specifying exact (concrete) class of created object
- Often used to refer to any method whose main purpose is creating objects
- How it works:
  1. Define a method for creating objects
  2. Child classes override method to specify the derived type of product that will be created

Factory Method Pattern

Client classes interact with the interfaces



Dependency Inversion Principle

Depend upon Abstractions

"Inversion" from the way you think

Using Reflection in Java

- *Reflection* allows us to create objects of a class using the *name* of the class
- Example adapted from MutantMaker:

```
public static void initMutantMaker() {
      mutants = new Mutant[numMutants];
      mutants[0] = new Wolverine();
      for (int i = 1; i < numMutants; i++) {
            Class<?> mutantClass;
            try {
                  mutantClass = Class.forName("mutants.Mutant"+ i);
                  mutants[i] = (Mutant)
                        mutantClass.getDeclaredConstructor().newInstance();
            } catch (Exception e) {
                        e.printStackTrace();
            }
      }
}
```

Using Reflection in Java

- Can create objects of a class through the *name* of the class
- Used in `SemanticAnalyzer`
  - Gets list of functions
    - Read from `conf/functions.conf`
  - Maps a token to the class responsible for parsing that type of token
  - When `SemanticAnalyzer` sees that token, calls the respective analyzer to parse
  - Example: `FloorToken` maps to the `FloorAnalyzer`
    - `FloorAnalyzer` pops the `Floor` token off the stack and then parses the (one) parameter for the *floor* function

Process of Adding Cosine Function to the Picasso Language

(in given code)

- Add function name to `functions.conf`
- Create a *token* for the cosine function
  - Same prefix as new function, e.g., `CosToken.java`
- Create a *semantic analyzer* for the function with same prefix as function, e.g., `CosAnalyzer.java`
  - `Analyzer` class implements `SemanticAnalyzerInterface`, returns an instance of `ExpressionTreeNode`
- Create a child of `ExpressionTreeNode` for function: `Cosine.java`

Name/prefix must match for all but ETN

Process of Adding Cosine Function to the Picasso Language

(in given code)

- Add function name to `functions.conf`
- Create a *token* for the cosine function
  - Same prefix as new function, e.g., `CosToken.java`
- Create a *semantic analyzer* for the function with same prefix as function, e.g., `CosAnalyzer.java`
  - `Analyzer` class implements `SemanticAnalyzerInterface`, returns an instance of `ExpressionTreeNo`
- Create a child of `ExpressionTreeNode` for function: `Cosine.java`

Using Java *reflection* to map tokens to analyzers. (How would we do this otherwise?)

Picasso Code: ReferenceForExpressionEvaluations

This implementation (from the "old" version of the code) is **different** from what we will have in our code. **But, it is a helpful reference.**

```
…
PLUS {
     public RGBColor evaluate(RGBColor left, RGBColor right) {
          double red = left.getRed() + right.getRed();
          double green = left.getGreen() + right.getGreen();
          double blue = left.getBlue() + right.getBlue();
          return new RGBColor(red, green, blue);
     }
},
…
```

What are `left` and `right` referring to?

Extensions

- Extensions could affect your code design
  - Where could change → abstraction
- When does your team need to decide?
  - Technically, not until the final implementation deadline
    - But, see above

Planning for Preliminary Implementation

- Goal is to have you do enough that you'll see issues with an initial design you create and adjust
- Implementation requirement (see project description page for more)
  - Input an expression interactively that includes at least one binary operator and display an image from the resulting expression
  - Tag the version in Git
- Requirement involves a lot of different pieces
  - Don't go too far in breadth, more depth
    - See design issues sooner
      - "We need method/functionality X in class Y"
- Invest in your team
  - If you understand, help your teammates understand

Planning: Tasks/Steps

- Testing: focus on methods' input (parameters), what is returned
- Think about iterative development
  - Not recommended: write all the tokens/parsers/expressions first
  - What is an appropriate process for this project?
- Decide on APIs where there are dependencies
  - Parameters and what is returned

Team Collaboration/Planning

- An hour of thinking/design will save hours of coding
- Given code is not perfect code
  - (Most code is not perfect code)
  - You can change code but make sure you understand it first
- Design GUI on paper/white board first before trying to implement
- You can write some tests first!
  - Helps to frame your implementation

Planning: Division of Tasks

- Work in subgroups?
- Consider how not to be loosely coupled
  - Reminder: Use git branches!
- Consider best # of people per part
  - Likely will keep changing as work gets done and you learn your design
- Not recommended: Person X does all the testing
  - Perhaps pair people up to write tests for each other

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?

Review: Collaboration

- What is our workflow in Git when collaborating?
- What did you like about how your team worked together on previous project?
  - What didn't you like?

Review: Collaboration: Workflow – Seeking Feedback

1. Pull to get the most recent changes to the repository
2. Create a branch from `main` for your work
   - Commit periodically
   - Write descriptive comments so your team members know what you did and why
3. Push your branch
4. On GitHub, 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
5. Merge pull request into `main` branch
6. In Eclipse, pull `main`
   - Merge into your branch or create a new branch from main

Collaboration Models

Good

- Team physically (or over Zoom or similar) works all together or in subteams
- Division of labor is clear
  - Keep track of tasks, what has been completed in a document
  - Agree on team deadlines
- Good, frequent communication
  - Be a sounding board for your teammate even if you don't understand everything they are working on

Bad

- Multiple people trying to do the same task
  - Overwriting each other's code
- Everyone working in the main branch
- Make a plan as a team, then someone goes rogue
- Asking for help too late

Student Questions

- Any code we shouldn't change?
  - Don't change anything until you understand it well
  - There is likely code that you won't change but depends on your extensions
- What if our design isn't perfect?
  - It won't be
  - BUT try to get it to pretty good, especially before the intermediate deadline

Implementation/Code Questions?

Secondary Goals

- You're going to figure out that your final design isn't perfect—maybe not even good!
  - Fix smaller and/or more critical 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.

Looking Ahead

- Friday after Thanksgiving, preliminary implementation deadline
  - Demo in class