CS 5044
Object-Oriented Programming with Java

Q&A Session

Enumerated types

- Java has a special `enum` type to represent a set of unique fixed values
  - The enum is declared as (and acts as) a special kind of class

```
public enum Gear { PARK, REVERSE, NEUTRAL, DRIVE; }
```

    - Values are arbitrary and human-readable identifiers, in all caps by convention
    - Specify values in your code by type-dot-value: `Gear.PARK` or `Gear.DRIVE`
    - Declaration and assignment is similar to a primitive: `Gear myGear = Gear.NEUTRAL;`
  - Each enum value is actually an object, but values should be compared via `==` or `!=`

```
if (myGear == Gear.PARK) { /*...*/ }
```

    - Note that an enum reference may be `null`
  - What's the point? Why enum? Because type-safety is enforced early, by the compiler
    - Constant values were common before Java 5 (and still in other languages) but...

```
public static final String PARK = "park";          // better to use enum instead of this
if (oldGear.equals(PARK)) { newGear = "Park"; }    // oops! wrong but allowed

public static final int PARK = 1;                  // better to use enum instead of this
if (oldGear == PARK) { newGear = -99; }            // oops! wrong but allowed
```

    - You could create a near-equivalent of `enum` prior to Java 5, but not very easily
- We'll use enumerated types in Project for `Gear` and `LogEntry` values
  - Above is everything you need to know about `enum` types for Project 2

ArrayList: our first collection class

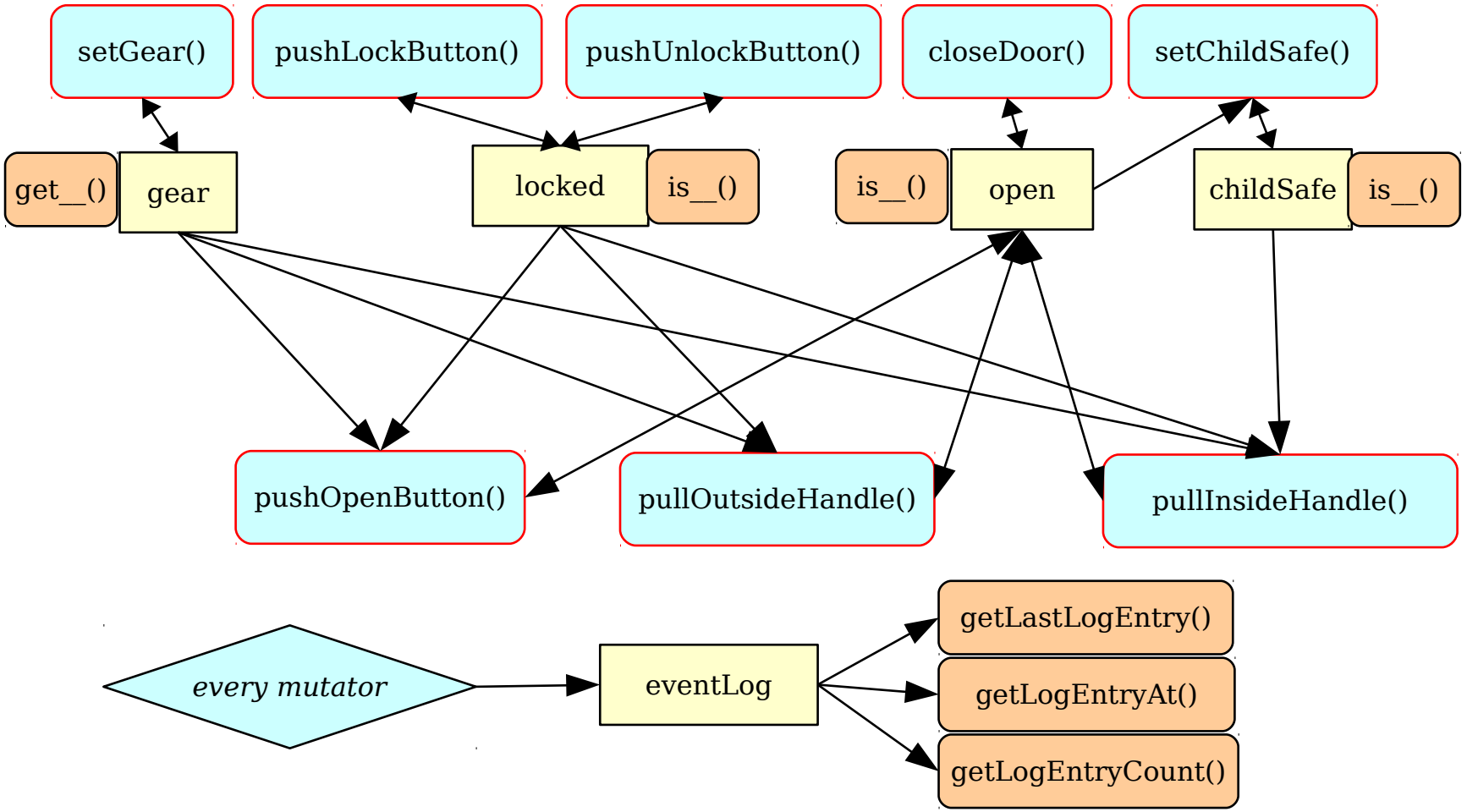
- We'll cover this in detail in two weeks, along with arrays, but for now...
- Project 2 introduces some basic concepts
  - Declaring and defining as a field:
    - ```
      private final List<LogEntry> eventLog; // declare a list collection of LogEntry values
      // Declaring as "final" means we must assign it exactly once (keeps us from forgetting!)
      ```
    - ```
      eventLog = new ArrayList<LogEntry>(); // define a new empty instance in the constructor
      // Forgetting to define a "new" collection would result in a NullPointerException later
      ```
  - Populating the list:
    - ```
      eventLog.add(LogEntry.DOOR_LOCKED); // append one element to the end of the list
      ```
  - Querying the list:
    - ```
      int logSize = eventLog.size(); // get the number of elements
      ```
    - ```
      LogEntry value = eventLog.get(0); // fetch the element at index location 0
      ```
      - Locations are numbered sequentially, starting from index 0 as the first element
      - The last element (if any) is always at index: `(eventLog.size() - 1)`
      - `ArrayIndexOutOfBoundsException` means you called get() with an invalid index
        - » Be sure to avoid this, by always validating the index *before* calling get()
- Above is everything you need to know about `ArrayList` for Project 2
      ```

Project 2: Requirements review

- Operational rules constrain the system:
  - The following requests always succeed (but don't necessarily change any state):
    - Lock, unlock, or close the door
    - Change the gear
  - Changing the child-safe feature succeeds only if the door is already open
  - A request to open the door -- via Button or Outside -- succeeds only if:
    - Gear is in park, AND
    - Door is unlocked
  - A request to open the door -- via Inside -- succeeds only if:
    - Child-safe feature is disengaged, AND
    - Open request via Button or Outside would succeed
- Please be sure to review *all* of the `LogEntry` comments before you start!
  - The precedence for the "no action" section is in the order of appearance in the file:
    - ```
      OPEN_REFUSED_CHILD_SAFE,
      OPEN_REFUSED_GEAR,
      OPEN_REFUSED_LOCK,
      CHILD_SAFE_CHANGE_REFUSED,
      NO_ACTION_TAKEN;
      ```
      ```

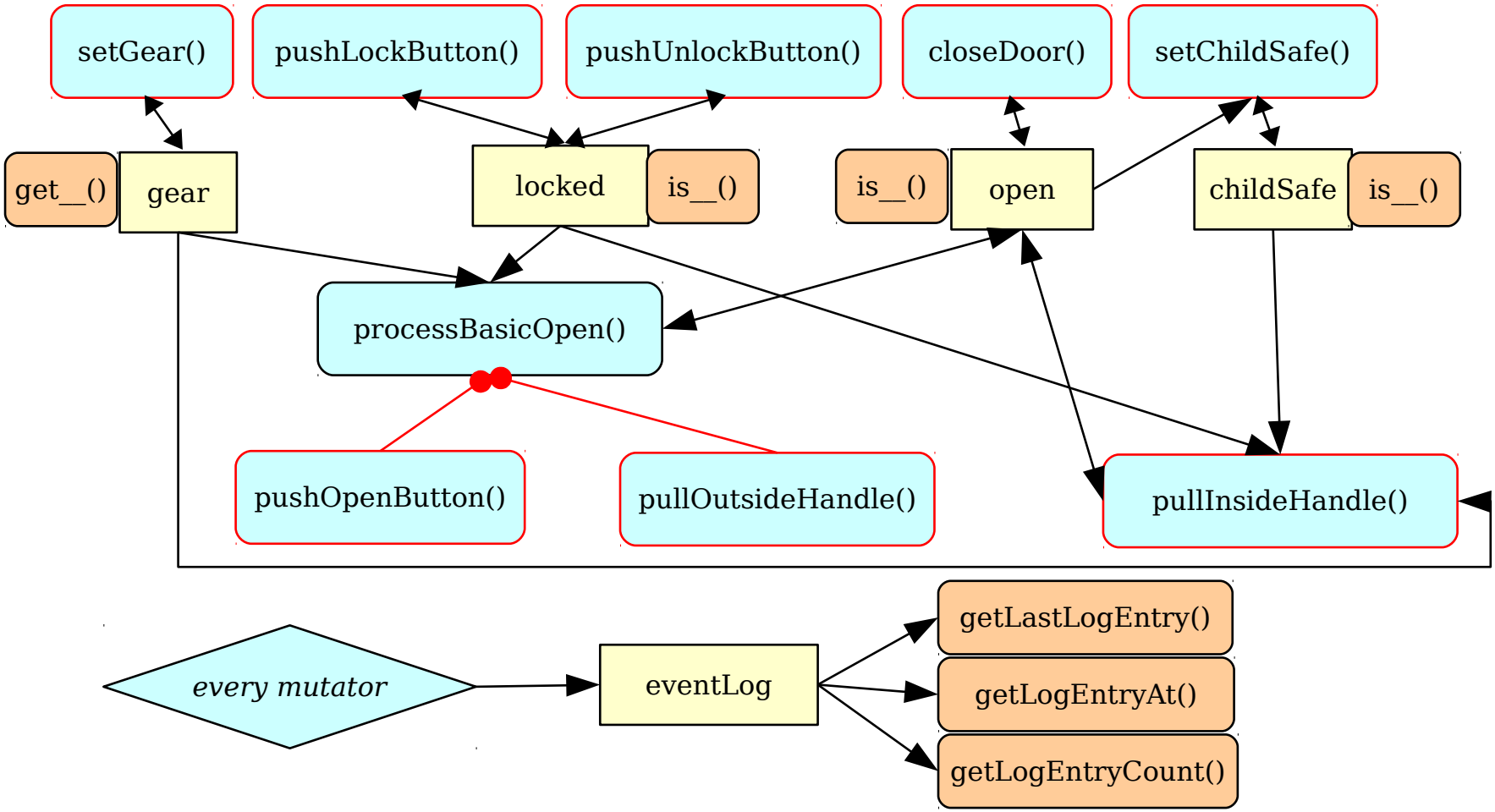
Project 2: Dependency model

- Start with a basic model of the operational rules (constructor not shown)



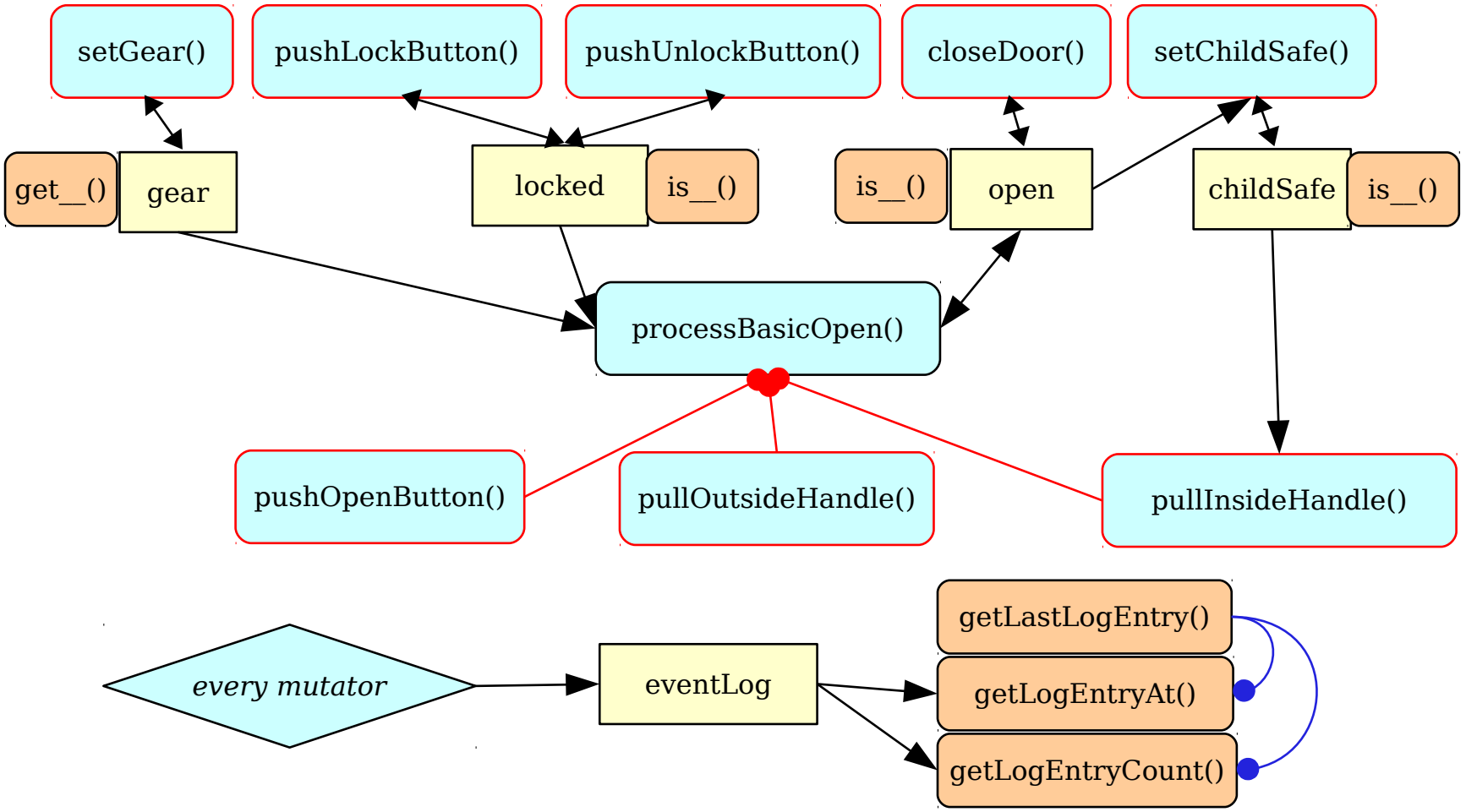
Project 2: Dependency model

- Consider centralizing any redundancies into helper methods



Project 2: Dependency model

- We can even go a bit further to leverage our own methods; this is the final model:



Project 2 (and beyond) Web-CAT score overview

- There are now **three** components to your Web-CAT "correctness/testing" score:
  - Results from Running Your Tests:
    - Percentage of your own JUnit tests that your code passed
      - Indicates how well your code passed your own JUnit tests
      - Feedback is very similar to that of JUnit in Eclipse for your own test methods
  - Code Coverage from Your Tests:
    - Percentage of your code covered by your own JUnit tests
      - Indicates how comprehensively you tested your own code
      - Feedback is very similar to that of Emma in Eclipse for your own test methods
      - Source code lines not fully exercised by your test cases will be highlighted in pink
  - Estimate of Problem Coverage:
    - Percentage of instructor-generated "reference" JUnit tests that your code passed
      - Indicates how well your code actually met the requirements
      - Several "hints" will be displayed to help you identify failed assertions
- Style checking (Checkstyle and PMD) is graded, as before, as a separate category
  - Click the file name in the File Details section
    - Source code lines with style issues are highlighted in red
  - JUnit test files are NOT subject to style checking (but please still use Ctrl-Shift-F)