Object-oriented design unit:

- Test 1 retrospective (Monday)
- OO design goals (**Today**)
- Class extension (Friday)
- More about class extension; refactoring (next week Monday)

Today:

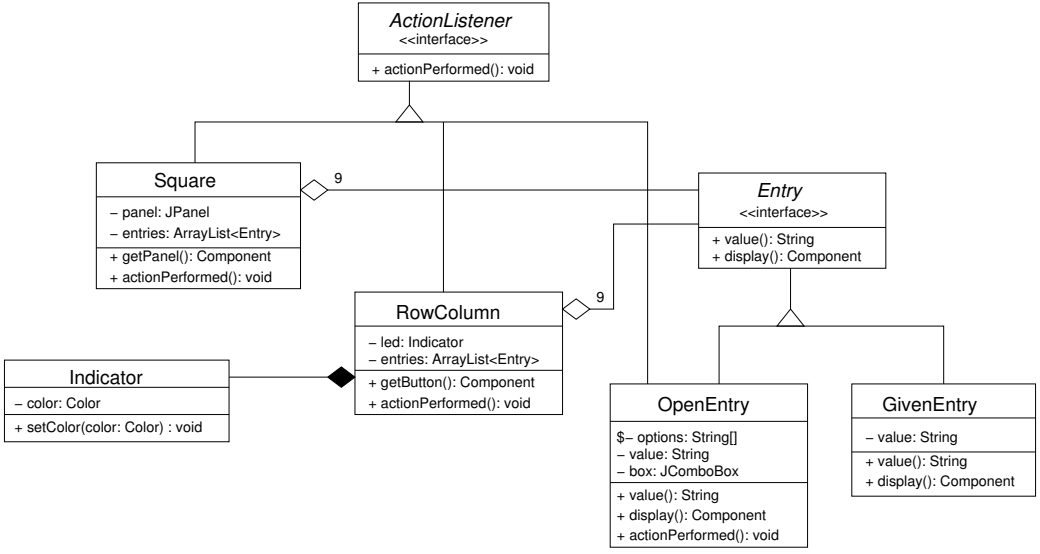
- Test 1 vocab retake
- UML
- Design principles
- Anticipating class extension

The exposition of OO design principles are drawn from Barnes and Kölling, *Objects First with Java*, Pearson, 2012.

Acquaintance. The objects know each other as peers. One object is not considered a component to the other. We show this as a plain line, although if the reference is one-directional, we decorate it with an arrowhead.

Aggregation. The object of one class is considered a component to the other class (the other class is an *aggregate* class), but not something that is exclusively owned. Shown with open diamond.

Composition. The object of one class is considered an exclusive component of the other class. Shown with filled diamond.


ActionListener
<<interface>>
+ actionPerformed(): void
Square
– panel: JPanel
– entries: ArrayList<Entry>
+ getPanel(): Component
+ actionPerformed(): void
9
Entry
<<interface>>
+ value(): String
+ display(): Component
RowColumn
– led: Indicator
– entries: ArrayList<Entry>
+ getButton(): Component
+ actionPerformed(): void
9
Indicator
– color: Color
+ setColor(color: Color) : void
OpenEntry
$– options: String[]
– value: String
– box: JComboBox
+ value(): String
+ display(): Component
+ actionPerformed(): void
GivenEntry
– value: String
+ value(): String
+ display(): Component

Design goals

- Code that is easily understood, reused, and changed; code that is less likely to contain errors.
- Where should this information go? Where should this functionality go? Who should know whom?
- *Coupling* is the interconnectedness and interdependence among classes.
  - *Tight coupling* is when two classes depend on each other's implementation. Thus a change in one class's implementation affects the other.
  - *Loose coupling* is when classes communicate through well-defined and stable interfaces and thus don't depend (much) on each other's implementation.
- *Encapsulation* is the hiding of implementation details of a module from the rest of the system. "Only information about *what* a class can do should be visible to the outside, no about *how* it does it. If no other class knows how our information is stored, then we can easily change how it is stored without breaking other classes" (Barnes and Kölling, pg 207).

Design goals (continued)

- *Responsibility-driven design* is the principle that each class is "responsible for handling its own data" (Barnes and Kölling, pg 212). Data storage and manipulation should be in the the same place.
- *Reuse.* Reusing code refers to adapting classes (with instance variables and methods) for a new purpose. *White box reuse* is when the re-user can observe the what is inside the structure being reused. *Black box reuse* is when the re-user knows only what the reused structure does, not how it does it.
- *Cohesion* refers to how well a "unit of code" maps to a logical task. "A cohesive method is responsible for one, and only one, well-defined task. . . A cohesive class represents one well-defined entity" (Barnes and Kölling, pg 219–220).

Coming up:

- **Due Thurs, Feb 28 1:15pm.** *Take Canvas quiz on UML and OO design terms after class on Wednesday. (No prelab reading but read the SPECIFICATION file in your repository.)*
- **Due Fri, Feb 27.** *Do Project 3, "Homemade Linked-list Map."*
- **Due Fri, Mar 6.** *Do Project 4, "Text-based adventure game."*