CS 326
Design and Style

Stephen Freund

1

Modular Design

- Module: Any design unit in software
- Modular design focusses:
  - what modules are defined
  - what their specifications are
  - how they relate to each other
- Not the implementations of the modules
  - Each module respects other modules' abstraction barriers

Ideals of Modular Software

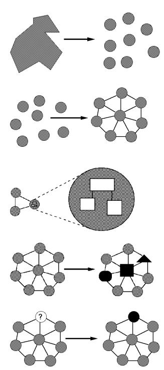
Decomposable – can be broken down into modules to reduce complexity and allow teamwork

Composable – "Having divided to conquer, we must reunite to rule [M. Jackson]."

Understandable – one module can be examined, reasoned about, developed, etc. in isolation

Continuity – a small change in the requirements should affect a small number of modules

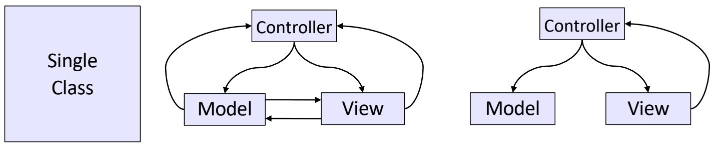
Isolation – an error in one module should be as contained as possible



General Design Issues

- **Cohesion**: how well components fit together to form something that is self-contained, independent, and with a single, well-defined purpose.
- **Coupling:** how much dependency there is between components
- **Decrease coupling. Increase cohesion.**
  - Each method does one thing well.
  - Each module represents a single abstraction.

Cohesion and Coupling



"god class" *Strongly coupled* *Weakly coupled*

Method Cohesion

- Methods should do one thing well:
  - Compute a single value
  - Observe or mutate, don't do both
  - Don't print as a side effect of some other operation
- Don't limit future uses of the method by having it do multiple, not-necessarily-related things
- Avoid:
  - long parameter lists
  - "flag" parameters (symptom of poor cohesion)

Properties

- A variable should be a property if and only if:
  - It is part of the inherent internal state of the object
  - It has a value that retains meaning throughout the object's life
  - Its state must persist past the end of any one public method
- Computed properties
  - connect abstract state to concrete variables
  - do minor book-keeping
  - don't over-do it

Method vs Computed Property?

```
public struct FacialExpression {
  ...

  let eyes: Eyes
  let mouth: Mouth

  public func happier() -> FacialExpression {
    return FacialExpression(eyes: eyes, mouth: mouth.happier())
  }

  // vs:

  var sadder: FacialExpression {
    return FacialExpression(eyes: eyes, mouth: mouth.sadder)
  }

}
```

Initializers

- Object should be completely initialized after initializer is done
  - The rep invariant should hold
  - Shouldn't need to call other methods to “finish” initialization
- Use optional initializers if failures may occur

```
class Double {
   init?(_ str : String) {
     if (str is not valid) {
       return nil
     } else { ... }
   }
```

```
if let d = Double("  ") {
  …
}
```

Names

- Follow conventions of language you are using
- https://swift.org/documentation/api-design-guidelines/#naming

Good Names

- Class names: generally nouns
  - Beware "verb + er" names, e.g. **`Manager`**, **`Scheduler`**, **`ShapeDisplayer`**
- Interface/protocol names often –able/-ible adjectives: **`Iterable`**, **`Comparable`**, …

Good Names

- Property/Method names: noun or verb phrases
  - Nouns for properties: **`count`**, **`totalSales`**
  - Nouns/Adjectives for observers: **`distance(to:)`**, **`successor()`**, ~~**`pointIsInside(_:_:)`**~~, **`inside(_:of:)`**
  - Verbs for mutators: **`print()`**, **`sort()`**, **`append(_:)`**
- Choose affirmative, positive names over negative ones
  - **`isSafe`** not **`isUnsafe`**
  - **`isEmpty`** not **`hasNoElements`**

Bad Names

- Bad:
  - **count, flag, status, label, check, value, pointer**
  - names starting with my…
- Describe what is being counted, what the "flag" indicates, etc. Phrases are fine!
  - **numberOfStudents, isCourseFull, calculatePayroll, validateWebForm,** …

Bad Names

- Avoid non-standard/ambiguous abbreviations: **calc, disp, oper, acc, clr, ctrller, btn,** …
- Short names in local contexts are good:
  - Good:

```
for i in 0..<size { items[i] = 0 }
```

  - Bad:

```
for theLoopCounter in 0..<size {
 theCollectionItems[theLoopCounter] = 0
}
```

Class Design Ideals

- **Cohesion**
- **Coupling**
- **Completeness**: Every class should present a complete interface
- **Consistency**: In names, param/returns, ordering, and behavior

Completeness

- Include **important** methods to make a class easy to use
- Counterexamples:
  - A mutable collection with **add** but no **remove**
  - A tool object with a **setHighlighted** method but no **setUnhighlighted** method
  - **Date** class with no date-arithmetic operations

Completeness

- Objects that have a natural ordering should implement `Comparable` protocol (`==` and `<`)
- Objects that you test for equality, store in other structures, or use as keys in map should implement:
  - `Equatable` protocol (`==`) , or
  - `Hashable` protocol (`==` and `hashValue`)
- Most objects should implement `CustomStringConvertible (description)`

http://www.cs.williams.edu/~freund/cs326/GraphADT/RGB.swift

But…

- **Don't** include everything you can think of
  - If you include it, you're stuck with it forever…
  - …even if almost nobody ever uses it
- Tricky balancing act
  - Include what's useful, but don't make things overly complicated
  - You can always add it later if you really need it

Consistency

- A class should have
  - Consistent names, parameters/returns, ordering, and behavior
  - Use similar naming; accept parameters in the same order
- Counterexamples:
  - `setFirst(index: Int, value: String)`
  - `setLast(value: String, index: Int)`
  - In Java: `String.length(), array.length, Vector.size()`

Open-Closed Principle

- **Big Idea:** Software entities should be open for extension, but closed for modification.
- Add features by adding new classes or reusing existing ones in new ways
- Don't add features by modifying existing classes
  - Existing code works and changing it can introduce bugs and errors.
  - Classes can become over-specialized.

Documenting a Class

- External: /** ... */ or ///
  - Classes, structs, properties, methods.
  - What clients need to know (Spec!)
  - Specific enough to exclude unacceptable implementations
  - General enough to allow for all correct implementations
- Internal: /* ... */ or //
  - Inside method bodies
  - What developer needs to know
  - How code is implemented
  - Invariants, internal pre/post conditions
  - Design rationale

RatNum Source

RatNum Docs

Other Random Items

- Enum with only 2 values better than Bool:
  - `oven.set(temp: 200, units: true)`
  - `oven.set(temp: 200, units: Temperature.celsius)`
- Don't use Strings to represent non-text data
  - `struct Point { x,y : Int }` vs `"(3,4)"`
- MVC!
- Don't put print statements in your core classes
  - Not `func printDescription() {...}`
  - Use `var description : String {...}`

Closing Thoughts on Design

- Always remember your reader
  - Clients
  - Other programmers
- What do they need to know?
  - Clients: How to use it
  - Implementers: How it works, **why it was done this way**
- Re-read style and design advice regularly
  - Pragmatic Programmer Readings!
- Practice. It will become more natural...
- But always look for better ways to do things!

Choosing types – some hints

Numbers: Favor `int` and `long` for most numeric computations

EJ Tip #48: Avoid `float` and `double` if exact answers are required

Classic example: Money (round-off is bad here)

Strings are often overused since much data is read as text

Independence of Views

- MVC!
- Don't put print statements in your core classes
  - Locks your code into a text representation
- Instead, have your core classes return data that can be displayed by the view classes
  - Bad: `func printMyself() {...}`
  - Good: `var description : String {...}`