Patterns in Cocoa: a deeper understanding

This course introduces the student to concepts in programming design patterns by exploring five common Cocoa patterns. Students are assumed to have familiarity with development in Objective-C using XCode.

6 Stages:

Stage 1: Introduction and Overview (How the course will be useful to you)

This stage presents the motivation for learning about design patterns (in terms of developer productivity) and lists the patterns that will be covered.

- Cocoa Design Patterns - framing the discussion (Stage 1.1 - Framing The Discussion)
    a. Why learn design patterns? They are pervasive in the iOS development environment, and it's better to work with them than against them.
    b. Onces you almost certainly seen them already.
    c. Apple uses them extensively in their own frameworks and assumes their use in how it structures new project templates.
    d. Tools we'll be using to explore the design patterns (sample apps, LLDB, etc…)
    e. It's important to use them appropriately (they can be "double-edged" blades)
    f. Design Patterns as part of your software development vocabulary
        - a concise way to communicating an approach to a problem.
        - References appear in blogs, online discussion & Apple's documentation
- What is a design pattern?
    a. Conceptual tools for solving problems
    b. How they differ from *programming paradigms*
    c. Their agnostic nature - E.g. how MVC applies to the problem of structuring applications that support a GUI, regardless of language.
    d. They have different *granularities* or *scopes*, i.e. some address small specific problems(fine-granularity/small-scope e.g. alloc-init), others structure applications at the highest level (coarse-granularity/large-scope, e.g. MVC)
    e. Designed to promote familiar software *virtues*, e.g. decoupling, "don't-repeat-yourself" (a.k.a. DRY)
- Inventory of the design patterns we will be examining
    a. Alloc-init: show example: code reuse and flexibility are it's strengths
    b. Singletons, show code & graphic, we'll look at existing Cocoa examples and creating our own
    c. KVC/KVO - simple code example for KVC - KVO gets a graphic
    d. Notifications - simple code example and graphic, also gets a graphic
    e. MVC - Show XCode file list of MVC structured code, then show graphic

Stage 2: Pattern 1: Two-Stage Object Creation

- Alloc and Init

    a. Simple example. How it works. Step through example Circle code[alloc_init_1 project] in LLDB and examine data structures: “po self”, “p *self”, “p _cmd”, explain “_isa”
- Designated initializers / hierarchy of init methods / misc. details
    a. How it works with inheritance (step through second simple example [alloc_init_2_project]). Why “self = [super init]”?
    b. Designated initializers
    c. Alloc/Init approach compared with ”class methods” approach to object creation (problems with scaling, especially with inheritance)
    d. (id) vs (instancetype)
- Stars and planets circle example - Basically, it’s creating circles in different sizes and colors. Lots of small white ones of uniform size, various custom ones with specific sizes and colors. Write init methods to make life easy.

Stage 3: Pattern 2: Singletons (They’re not evil)

- How it works and why it’s useful (motion graphic, same one from stage 1)
    a. Simple example (basic +sharedInstance implementation)
    b. Refine code example (add dispatch_once(). Explain why)
- Examples in Cocoa
    a. Examples: UIApplication, NSUserDefaults, UIDevice, NSFileManager
- More refinements
    a. Idiot proofing (+hiddenAlloc)
- Pros & Cons
    a. Pros:
        - models unique services accurately
        - decoupling (no reference passing)
        - simplicity
    b. Dangers/Cons:
        - Overuse
        - unit testing
        - global variables/state

Stage 4: Pattern 3: KVC & KVO

- KVC
    a. How it works: a standard dictionary interface for all objects
        - Simple example
        - Fallback process
            - accessor methods: -<key>, -get<Key>
            - methods: -_<key>, -_get<Key>, -_set<Key>
            - instance variables: <key> or _<key>
            - -valueForUndefinedKey:, setValue:forUndefinedKey:
    b. Advanced ideas

        - runtime key generation, like runtime selector generation (e.g. value for key:[NSString stringWithFormat:@"numberView%d])
        - query runtime for properties, class_copyPropertyList()
        - talking any object (even if you don't have a declared interface),
        - significance of having a universal interface
- Keypaths
    a. Sorta like file-paths
    b. Simple Example of object traversal
    c. Use in Core Data
    d. KeyPath collection operators (examples: @avg, @max)
        - Show key path version of averaging/sorting vs. conventional approach
- Key-Value-Observing
    a. Explanation and put in terms of terms of decoupling
    b. Simple example
        - explain *options* and *context* parameters
        - importance of calling [super observeValue…]
        - extend example to show -keyPathsForValuesAffecting<Key>

Stage 5: Pattern 4: Notifications

- How they work (motion graphic from stage 1)
- Example code
- NSNotification center/queues etc..
- Compare with KVO and delegation as a decoupling strategy

Stage 6: Pattern 5: MVC

- What is MVC
    a. Explanation of the problem it solves (motion graphics)
    b. How it's done in Cocoa (simple example)
- A useful pattern, not "The Law"
    a. Examples where MVC is not optimal: e.g. games (look at Spritekit)
- Variants: MVVM (Model-View-ViewModel), MVP(Model-View-Presenter)