Assignment II: Calculator

Ziele

In diesem Assignment erweitern Sie Ihren Taschenrechner, so dass er eine “Variable” als Eingabe erlaubt und eine Undo-Funktion unterstützt. Ebenfalls bereiten Sie den Taschenrechner für weitere Funktionalität vor.

Stellen Sie sicher, dass Sie die Tipps Sektion weiter unten lesen!

Stellen Sie ebenfalls sicher, die Testat Sektion weiter unten zu lesen, um alle Anforderungen für ein Testat zu erfüllen.

Material

- Sie müssen erfolgreich Assignment I beendet haben. Dieses Assignment baut darauf auf.

Aufgaben

1. Ändern, entfernen oder fügen Sie keine public API (d.h. nicht `private` `func`s und `var`s) hinzu aus Ihrem `CalculatorBrain` aus Assignemt I, außer `undo()` und wie in Aufgabe 3 und 4 spezifiziert. Verwenden Sie weiterhin `Dictionary<String,Operation>` als die primäre interne Datenstruktur des `CalculatorBrain` zum Ausführen von Operationen.
2. Ihr UI sollte immer synchron zu Ihrem Model sein (das `CalculatorBrain`)
3. Fügen Sie die Fähigkeit hinzu, dass Ihr CalculatorBrain als Eingabe *Variablen* erlaubt. Manchen Sie dies, indem Sie die fokgende API im `CalculatorBrain` implementieren...

   ```
   func setOperand(variable named: String)
   ```

   Dies muss exakt tun, was wir uns darunter vorstellen würden: es gibt eine "Variable" als Operand ein (z.B. gibt `setOperand(variable: "x")` die Variable mit dem Namen `x` sein). Setzen des Operanden auf `x` und dann das Ausführen der Operation `cos` bedeutet, `cos(x)` ist in Ihrem `CalculatorBrain`.
4. Da Sie nun erlauben Variablen als Operanden einzugeben, fügen Sie eine Methode zum evaluieren des `CalculatorBrain` hinzu (d.h. das Ergebnis berechnen) durch Substitution von Werten für diese Variablen, welche in einem bereitgestellten `Dictionaty` zur Verfügung gestellt sind...

   ```
   func evaluate(using variables: Dictionary<String,Double>? = nil)
         -> (result: Double?, isPending: Bool, description: String)
   ```

   Beachten Sie, dass dies ein Optional `Dictionary` (mit `String`s als Keys und `Double`s als Values) als Parameter annimmt und der Parameter per Default `nil` ist, wenn dieser Parameter nicht übergeben wird, wenn die Methode aufgerufen wird. Beachten Sie auch, dass die Methode ein Tuple zurück gibt (wobei das erste Element ein Optional `Double` ist). Die Methode ist nicht `mutating` und Ihnen ist es nicht erlaubt dies dazu zu machen. Wenn eine Variable, die als Operand gesetzt wurde, nicht im `Dictionary` gefunden wird, nehmen Sie deren Wert als Null an.
5. Wir haben die `result`, `description` und `resultIsPending` `var`s als non-`private` API in Assignment I angelegt. Dies bedeutet, wir sind einen Vertrag eingegangen sie weiter zu unterstützen, obwohl wir nun ein neues Feature (Variablen) in diesem Assignment hinzufügen, was sie praktisch irrelevant macht. Was wir eigentlich machen sollten ist dies als *deprecated* zu behandeln (in Xcode treffen Sie auf viele deprecated iOS API). Zum jetzigen Zeitpunkt behalten wir die alten `result`, `description` und `resultIsPending` `var`s und implementieren jede davon durch Aufruf von `evaluate` mit dem Parameter nil (d.h. sie geben die Rückgabe unter der Annahme dass der Wert von einer Variable Null ist). Verwenden Sie jedoch keine dieser `var`s irgendwo in Ihrem Code in diesem Assignment. Verwenden Sie stattdessen `evaluate`.
6. Verwenden Sie nirgends die Swift Typen `Any` oder `AnyObject` in Ihrer Lösung.
7. Fügen Sie zwei neue Buttons zu Ihrem UI hinzu: `→M` und `M`. Opfern Sie dafür keinen der Operations Buttons aus Assignment I, um dies hinzuzufügen (Sie dürfen aber gerne noch mehr Operation Buttons hinzufügen). Diese Buttons setzen oder rufen eine Variable (get/set) im `CaluculatorBrain` mit dem Namen `M`.
   a. `→M` ruft `evaluate` in Ihrem Model auf mit einem `Dictionary`, welches einen einzelnen Eintrag hat dessen Key `M` ist und dessen Value der aktuelle Wert des `display` ist, dann das `display` updated um das `result` zu zeigen welches von `evaluate` zurückgegeben wird. Bis dieser Button (oder der Clear Button) noch mal gedrückt wird, sollte dieses gleiche `Dictionary` jedes mal verwendet werden, wenn `evaluate` aufgerufen wird.
   a. `→M` führt **nicht** `setOperand` aus.
   a. Tippen von `M` sollte `setOperand(variable: "M")` im `brain` ausführen und dann das result des Aufrufs von `evaluate` im `display` zeigen.
   a. `→M` und `M` sind beides Controller Mechanismen, nicht Model Mechanismen (obwohl sie beide die Model Mechanik von Variablen nutzen).
   a. Dies sind keine sehr guten "Memory" Buttons für unseren Calculator, können aber verwendet werden um zu testen ob unsere Variablen Funktion, die im Model implementiert ist richtig funktioniert.

Beispiele...

`9 + M = √` ⇛ description ist `√(9+M)`, display ist `3` da `M` nicht gesetzt ist (also 0.0)

`7 →M` ⇛ display zeigt nun `4` (die Wurzel von 16), description ist noch immer `√(9+M)`

`+ 14 =` ⇛ display zeigt nun `18`, description ist nun `√(9+M)+14`

8. Zeigen Sie den Wert von `M` (wenn gesetzt) irgendwo in Ihrem UI.
9. Stellen Sie sicher, dass Ihr `C` Button aus Assignment I richtig in diesem Assignment funktioniert. Zusätzlich sollte er das `Dictionary` nicht verwerfen, welches für die `M` Variable verwendet wurde (es sollte `M` *nicht* auf Null oder einen anderen Wert setzen, nur das `Dictionary` nicht mehr nutzen bis `→M` wieder gedrückt wird). Dies erlaubt Ihnen den Fall einer nicht gesetzten Variable zu testen.
10. Fügen Sie einen Undo Button zu Ihrer Calculator hinzu. Vielleicht haben Sie freiwillig schon einen "Backspace" Button in Assignment I hinzugefügt. Hier ist ein Backspace und ein tatsächliches Undo in einem einzigen Button gemeint. Wenn der User dabei ist eine Zahl einzugeben, sollte sich das Undo Button wie ein Backspace verhalten. Wenn der User nicht dabei ist eine Zahl einzugeben, soll die letzte Aktion die im `CalculatorBrain` durchgeführt wurde rückgängig gemacht werden. Machen Sie nicht das speichern von `M`'s Werten Rückgängig (machen Sie aber das Setzen einer Variable als Operand rückgängig).

Tipps

1. Ausgenommen das Verbot in Aufgabe 1, können Sie die `private` Implementierung des `CalculatorBrain`s ändern wie Sie wollen (nur nicht die public API). Die Hauptmodifikation, die Sie durchführen müssen ist, sich die Reihenfolge der Eingabe von Operanden und Operationen in das brain zu merken, da Sie in der Lage sein müssen diese neu zu evaluieren, mit beliebigen Variablenwerten (und Sie müssen auch in der Lage sein die Sequenz rückgängig zu machen, Schritt für Schritt).
2. Da `evaluate` nicht `mutating` ist (und es auch nicht sein darf), wird sie (und kann auch nicht) durch side-effecting `var`s im `CalculatorBrain` (wie `accumulator`) implementiert. Tatsächlich muss `accumulator` kein Property im `CalculatorBrain` `struct` mehr sein.
3. Vergessen Sie nicht, dass Methoden in Swift in anderen Methoden verschachtelt werden können. Dies ist ein Tipp, keine Voraussetzung.
4. Ob Sie es glauben oder nicht, ausser `evaluate` können Sie wahrscheinlich jede andere public Methode und `var` (inkl. `clear` und `undo`) in einer Zeile Code implementieren. Es macht duraus Sinn, dass `evaluate` etwas komplizierter als alles andere ist, da die Auswertung der Kern der `CalculatorBrain` Funktionalität ist.
5. Vielleicht machen Sie sich nun Sorgen um die Performance Auswirkung, da `result`, `description` und `resultIsPending` nun alle ständig `evaluate` aufrufen. Zuvor wurde das Ergebnis von evaluate "gecached" in den private `var`s (wie `accumulator`) von `CalculatorBrain`. Natürlich deprecaten wir diese sowieso, aber selbst wenn wir das nicht tun würden, und diese nicht irgendwo ständig in einer Schleife aufgerufen werden würden, sind die Performance Charakteristiken dieser `var`s wahrscheinlich irrelevant, da sie relativ selten im vergleich zu viel teureren Operationen wie das Zeichnen on Screen sind. Etwas zu optimieren, das keine Performance Probleme verursacht zu Lasten von verschleiern von Code ist schlechtes Design. Siehe Donald Knuth's *The Art of Computer Programming* und *A Box Darkly*.
6. Aufgabe 3, 4 und 5 sind Model Aufgaben. Sie haben nichts mit dem UI zu tun (d.h. Ihr Controller und View). Sie sollten nicht eine Zeile Code in irgendeiner Datei ändern, außer der, die Ihr `CalculatorBrain` beinhaltet um diese drei Aufgaben zu lösen.
7. Schlagen Sie das Wort deprecate in einem Wörterbuch nach, wenn Sie dies noch nie in der Verbindung mit einer API gehört haben. Es bedeutet "to express disapproval of". Es bedeutet dass Sie anderen Programmierern sagen, diese API nicht mehr zu nutzen, da Sie diese in naher Zukunft wahrscheinlich ent-

fernen. Bei der Durchsicht der iOS Doumentation treffen Sie auf viele deprecated Methoden. Nun wissen Sie was Apple tun muss, wenn sie public API entfernen wollen, die in der Vergangenheit eingeführt wurde.

8. Aufgabe 7 und 8 sind Controller/View Aufgaben. Es sind keinerlei Änderungen in `CalculatorBrain` nötig hierfür nötig. Wenn zum Beispiel irgendwo `M` in Ihrem `CalculatorBrain` Code auftaucht, dann haben Sie das MVC Pattern verletzt. Die M Funktionalität ist ein reines UI Feature, welches einfach nur die generische Variablen-Verarbeitungsfunktion des `CalculatorBrain` Models nutzt.
9. Aufgabe 9 und 10 verlangen wahrscheinlich die Implementierung in beiden, Model und Controller. Seien Sie sich im klaren darüber was Sie wohin implementieren. Erinnern Sie sich daran, dass Ihr Model UI-unabhängig ist. Erinnern Sie sich auch daran, dass Ihr Controller nur die non-private API des Models kennt (d.h. er kann nichts über die interne Model-Implementierung wissen).
10. Ohne Undo, ist es nur möglich *numerische* Werte für `M` einzugeben. Wenn Sie versuchen irgendetwas anderes für den Wert von M zu speichern (wie z.B. π oder 23÷2), wird dieser Ausdruck zum aktiven neuen Ausdruck im Calculator und ersetzt damit den Ausdruck, den Sie ursprünglich auswerten wollten. Wenn Sie aber erstmal Undo haben, können Sie zum Vorherigen Ausdruck zurückgehen, nachdem Sie `M` gesetzt haben. Zum Beispiel, geben Sie `M cos`, dann π, dann →`M`, dann Undo (um π wieder loszuwerden) ... dann zeigt Ihr Calculator den Wert von `cos(M)`, was `-1` sein sollte.
11. Ihr Model ist nun nicht mehr nur ein `CalculatorBrain`. Ihr Model besteht nun aus zwei unterschiedlichen und komplett verschiedenen structs: ein `CalculatorBrain` und ein `Dictionary` (welches die Werte von `M` beinhaltet). Das ist vollkommen zulässig. Es gibt keine Regel die besagt, dass Ihr Model nur eine einzige Datenstruktur sein muss.
12. Der Rahmen des Assignments ist ähnlich zum vergangenen Assignment (d.h., wenn Sie mehr als 100 Zeilen Code brauchen, dann sind Sie auf dem falschen Lösungsweg).

Lernziele

Hier eine unvollständige Liste von Konzepten, für die Sie in diesem Assignment Erfahrung sammeln bzw. lernen sollen.

1. `Array`
2. Werte Semantic
3. Tuples
4. Default Paramater Werte
5. Setzen von Dictionary Werten
6. Verschiedene Swift Sprachen-Features
7. MVC

Für alle Testate in diesem Semester ist das Ziel qualitativ hochwertigen Code zu schreiben, der ohne Warnings und Errors baut. Ebenfalls sollte die so entstandene App auf Funktion und Fehler gestestet werden und dies iterativ so lange, bis alle Fehler beseitigt sind und die App so funktioniert, wie gedacht.

Hier sind die am meisten vorkommenden Gründe, weshalb Sie vielleicht kein Testat bekommen:

- Projekt baut nicht
- Eine oder mehrere Aufgaben wurden nicht zufriedenstellend gelöst
- Ein fundamentales Konzept wurde nicht verstanden
- Code ist optisch mangelhaft und schwer zu lesen (z.B. ist das Einrücken nicht konsistent, etc.)
- Ihre Lösung ist schwer (oder unmöglich) für jemanden zu lesen, durch mangelhafte Kommentare, schlechte Variablen/Methoden Namen, schlechte Lösungsstruktur, zu lange Methoden, etc.
- Die Swift API Design Guidelines wurden nicht eingehalten

Lernziele

Für alle Testate in diesem Semester ist das Ziel qualitativ hochwertigen Code zu schreiben, der ohne Warnings und Errors baut. Ebenfalls sollte die so entstandene App auf Funktion und Fehler gestestet werden und dies iterativ so lange, bis alle Fehler beseitigt sind und die App so funktioniert, wie gedacht.

Hier sind die am meisten vorkommenden Gründe, weshalb Sie vielleicht kein Testat bekommen:

- Projekt baut nicht
- projekt baut nicht ohne Warnings
- Eine oder mehrere Aufgaben wurden nicht zufriedenstellend gelöst
- Ein fundamentales Konzept wurde nicht verstanden
- Code ist optisch mangelhaft und schwer zu lesen (z.B. ist das Einrücken nicht konsistent, etc.)
- Programm kann zum Absturz gebraucht werden (z.B. wurde ein Optional das `nil` ist mit `!` unwrapped)
- Ihre Lösung ist schwer (oder unmöglich) für jemanden zu lesen, durch mangelhafte Kommentare, schlechte Variablen/Methoden Namen, schlechte Lösungsstruktur, zu lange Methoden, etc.
- UI ist chaotisch, Dinge sollten ausgerichtet sein und angemessene Abstände haben um "nett auszusehen"
- Private API wurde nicht korrekt abgegrenzt
- Die Grenzen von MVC wurden verletzt
- Die Swift API Design Guidelines wurden nicht eingehalten

Häufig fragen Studenten wie viele Kommentare im Code nötig sind. Die Antwort ist einfach. Der Code muss einfach und vollständig lesbar sein. Sie können davon ausgehen, dass die iOS API und der Code aus der Vorlesung bekannt sind. Sie sollten nicht davon ausgehen, dass bereits eine Lösung für die Aufgabe bekannt ist.