Реализације омотача за OpenSSL библиотеку у програмском језику Свифт

Бранко Поповић

Математички факултет, Универзитет у Београду

Ментор:

проф. др Владимир Филиповић

Саржај

1 Увод

У овом раду се бавимо системом *OpenSSL* у контексту програмског језика Свифт. У оквиру рада је реализован *CryptoLab*, омотач криптографског дела система *OpenSSL* написаног у програмском језику Свифт за *iOS* платформу [1,2].

Идеја за пројекат се јавила у току практичног рада са криптографским функцијама, када је била неопходна комуникација између различитих мобилних платформи. Примећено је да би било изузетно корисно на свакој платформи имати исто језгро, тј. исту имплементацију криптографских функција које се користе. Главна предност исте имплементације језгра је што је тиме исти начин рада осигуран на свакој платофрми па се самим тим елиминише добар део потенцијалних багова.

Са друге стране, развој мобилних платформи и техника имплементације су веома живи и актуелни трендови. Свака нова верзија платформе може да донесе значајне промене, као и нове стандарде којима је неопходно прилагодити се у огромној мери. Имајући то у виду квалитетан програмски кôд је онај који поштује све актуелне стандарде платформе којој је намењен.

Основна мотивација за овакав избор теме је управо недостатак решења које ће на савремен начин да омогући коришћење криптографских функција погодних за међуплатформску комуникацију.

У даљем раду биће описана имплементација решења за:

1. Проблем јединственог језгра функционалности и имплементације у случају употребе криптографских алата и механизама у међуплатформској комуникацији.
2. Могућност примене савремених концепата програмирања у криптографске сврхе.

Решење је имплементирано у виду омотача око криптографског дела система *OpenSSL* за *iOS* платформу коришћењем актуелне верзије програмског језика Свифт. Језгро *OpenSSL-a* пружа компатабилност између више различитих платформи, док актуелна верзија програмског језика Свифт чини да се омогући примена свих савремених концепата које овај језик доноси.

Целокупни пројекат је отвореног кôда и може се наћи на адреси: https://github.com/thebrankoo/CryptoLab. Даљи развој ће се базирати на захтевима заједнице и њеном учествовању у директној имплементацији нових карактеристика.

2 Програмски језик Свифт

Свифт је програмски језик развијен од стране компаније *Apple* [3,4]. Прва верзија је објављена у јуну 2014. године и имплементиран је као језик који обухвата све концепте савременог развоја софтвера са циљем да у будућности што више замени првобитни језик за *Apple* платформе, *Objective-C*. Актуелна верзија језика је 3.1 и у њој је написан омотач представљен у овом раду.

Сам језик се заснива на програмским језицима *C* и *Objective-C*. Управо је због тога могуће да се из Свифта користе функције написане у та два језика. Поред традиционалних основа, Свифт усваја и савремене концепте програмирања.

Једна од битних карактеристика програмског језика Свифт је програмирање оријентисано према протоколима, концепт који је званично уведен у верзији 2.0. Предност омотача описаног у овом раду, у односу на слична решења, је што имплементира и користи могућности Свифт протокола. Сходно томе, у наставку ће бити описане основе ПОП приступа.

2.1 Програмирање оријентисано према протоколима (ПОП)

На годишњој конфернецији компаније *Apple* (WWDC) одржаној 2015. године, одржано је предавање у коме је званично представљен и објашњен концепт ПОП-а [5].

ПОП је уведен као савремен одговор на проблеме који се јављају приликом наслеђивања класа и рада са њима, а који ће бити детаљније објашњени у наставку. Такође је увођењем ПОП-а стављен акценат на побољшање предвидивости, продуктивности и перформансама.

2.1.1 Свифт протоколи

Протокол дефинише шематски план метода, атрибута и осталих елемената програма са сврхом обезбеђивања одређене функционалности.

```
/**
Protocol with default hash function implementation
*/
public protocol HashingFunction {
        /**
        Hash data

        - parameter dataToHash: Data to hash

        - returns: Hashed data
        */
        func hash(data dataToHash: Data) -> Data

        /**
        Updates current data with new data

        - parameter data: New data
        */
        func update(withData data: Data)

        /**
        Finishes hashing of all data added with update function

        - returns: Hashed data
        */
        func finishBlock() -> Data
}
```

Слика 1: Пример дефиниције протокола

Протоколе могу да усвајају класе, структуре, енумератори и њихово задужење је да пруже конкретне имплементације свих захтева који су наведени у протоколу.

Протоколи такође могу да садрже и екстензије у којима се реализује подразумевана имплементација захтева или додају нове функционалности.

```
extension HashingFunction {
        public func hash(data dataToHash: Data) -> Data {
                return (self as! CoreHashUser).coreHash.hash(data: dataToHash)
        }

        public func update(withData data: Data) {
                return (self as! CoreHashUser).coreHash.update(withData: data)
        }

        public func finishBlock() -> Data {
                return (self as! CoreHashUser).coreHash.finishBlock()
        }
}
```

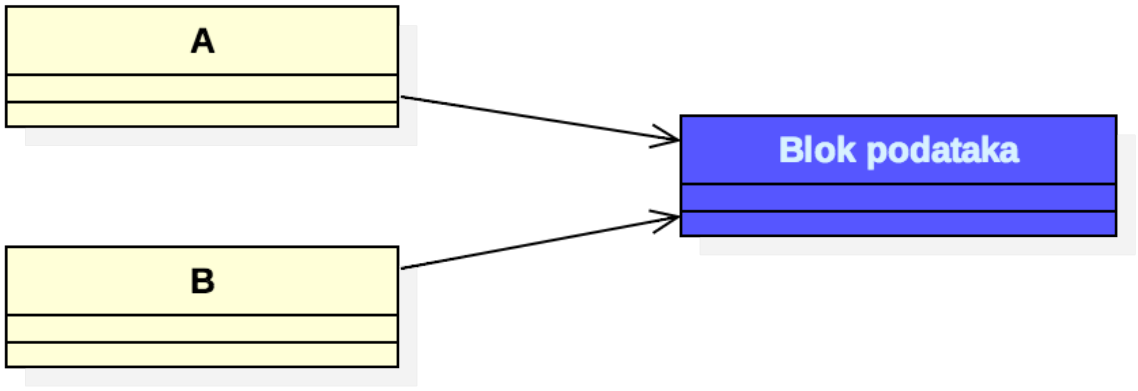
Слика 2 Дефинисање екстензије протокола

2.2 Проблеми са класама

Проблеми са класама су се експлицитно адресирали на годишњој конфернецији компаније *Apple* (WWDC) чиме се и званично дао повод за увођење концепта ПОП. Последица проблема који се јављају у раду са класама је да тренд програмирања без стања (*енг. stateless*) све више расте. Управо због тога функционално програмирање добија све више на популарности. Свифт подржава и функционалну парадигму али ПОП представља специфичан вид решења за све оно што је дефинисано као недостатак објектно оријентисаних класа. У даљем тексту следи опис тих недостатака.

2.2.1 Имплицитно дељење

Имплицитно дељење се јавља када се два или више објеката обраћају истом блоку података или објекту (*Слика 3*).



Слика 3: Имплицитно дељење

Основни проблеми имплицитног дељења:

1. Прављење великог броја копирајућих конструктора
2. Услов трке (енг. *Race Condition*)
3. Подизање сложености програма које резултује баговима

Прављење великог броја копирајућих конструктора се обично јавља када се препозна учесталост имплицитног дељења. На тај начин се покушава са контролисањем стања програма. Као резултат се врло лако може десити нарушавање прегледости и квалитета кôда као и већа подложост баговима.

Услов трке се јавља као последица имплицитног дељења када се два или више различита објеката обрађају истим подацима при чему, када један објекат промени податке, остали ни на који начин нису обавештени о промени. Проблем се јавља када неки од осталих објеката хоће да приступи подацима, а они нису у очекиваном стању.

Подизање сложености је логична последица која следи из претходна два проблема. Да би се имплицитно дељење контролисало, улаже се доста напора и прибегава различитим механизмима и решењима која могу бити изузетно гломазна. На пример, често се прибегава механизму *катанца (енг. lock)* чиме се доприноси неефикасности као и подложности баговима, а могуће је јављање *мртвих петљи (енг. Deadlocks)*.

2.2.2 Наслеђивање класа

Наслеђивање класа може да постане сувише обавезујуће. Следећи проблеми су јако чести:

1. Једноструко наслеђивање
2. Непотребно наслеђивање једне класе
3. Терет иницијализације
4. Немогућност ретроактивног моделовања

Једноструко наслеђивање је могућност наслеђивања само једне класе. То може бити проблем уколико желимо да нека класа искористи карактеристике више различитих класа. Са друге стране у Свифту једна класа може да усвоји произвољан број протокола, без нарочитог подизања комплексности кôда.

Супер класе, тј. класе које су „веома важне” и које су беспотребно наслеђиване велики број пута, се често јављају као резултат „лењости” програмера. Уместо да се започне са прављењем потпуно нове класе, део функционалности супер класе се искоришћава наслеђивањем што на крају као последицу има читав ланац зависности класа. Врло је вероватно да цела ситуација створи огроман хаос када кôд нарасте.

Терет иницијализације обавезује класу да позива специфичне конструкторе надкласе као и да дефинише атрибуте који јој нису нужно неопходни па тако цео процес може доста да

смањи ефикасност извршавања програма. Протоколи не обавезују на иницијализацију а свако дефинисање атрибута наведних у протоколу се извршава веома ефикасно.

Немогућност ретроактивног моделовања смањује флексибилност класа. Протоколи нуде могућност ретроактивног моделовања где се могу додавати нове функционалности класама чијем кôду немамо директан приступ или које не желимо из неког разлога директно да мењамо. Преко екстензија се могу дефинисати и подразумеване имплементације метода дефинисаних у протоколу.

2.2.3 Губљење типа надкласе (*Lost type relationship*)

Овај проблем се јавља приликом мешања интерфејса и имплементације. Дешава се да је за имплемнтацију методе базне класе неопходно урадити експлицитно пребацивање на конкретан тип у свакој подкласи (енг. *downcast*).

У примеру који следи (*Слика 4*), поткласа *Number* имплементира методу надкласе *Ordered.* Класа *Number,* у имплементацији методе надкласе, ради само са објектима типа *Number.* Стога је неопходно да се уради *downcast.* Пошто метода класе *Number,* као аргумент очекује објекат типа *Ordered,* јавиће се проблем када год се као аргумент пошаље нека друга потклаcа класе *Ordered* (нпр. *Label*).

```
var str = "Hello, playground"

class Ordered {
    func precedes(other: Ordered) -> Bool { fatalError("Implement me!") }
}

class Label: Ordered {var text String = "" ...}

class Number: Ordered {
    var value: Double = 0
    override func precedes(other: Ordered) -> Bool {
        return value < (other as! Number).value
    }
}
```

Слика 4: Губљење типа надкласе

2.3 Решење засновано на концепту ПОП

Концепт ПОП нуди решење за наведне проблеме са класама. Његове главне карактеристике су то што:

1. Може да се користи и над типовима података који се преносе по вредности (структуре, енумератори), а не само над типовима који се преносе по референци (класе)
2. Омогућује ретроактивно моделовање (додавање нових функционалности типовима кôд којих немамо приступ изворном кôду)
3. Нема терета иницијализације
4. Пружа јасану шему шта треба бити имплементирано

Могућност употребе протокола над типовима које се преносе по вредности пружа механизам спречавања проблема имплицитног дељења који се јавља код класа. Протоколи дају потпуно нову функцију структурама и енумераторима који добијају ширу могућност примене јер их је могуће користити скоро као класе. Став да већина багова настаје услед погрешног стања је заправо идеја функционалног програмирања а Свифт структуре и енумератори (нарочито структуре) се све више користе како би се стекла потпуна контрола над порменама стања програма.

Ретроактивно моделовања у Свифту се реализује путем екстензија. Уколико користимо неку класу или протокол чији је кôд затворен или из неког другог разлога не постоји могућност њиховог директног мењања и проширивања функционалности, коришћењем екстензија управо то може да се постигне.

На слици 2 се види екстензија протокола *HashingFunction.* Екстензијом се додаје нова функционалност која на крају резултује тиме да се дефинисане методе могу позвати над било којим објектом који имплементира *HashingFunction* протокол. Екстензије нису резервисане искључиво за протоколе, него се могу применити и над класама.

Терет иницијализације је потпуно уклоњен. Појава уланчавања конструктора код класичног наслеђивања је готово неизбежна (нпр. звање конструктора надкласе). Ланчање конструктора свакако представља терет за програмера, а и сам софтвер је неефикаснији зато што троши одређено време на уланчану иницијализацију. Протоколи овде знатно олакшавају ствари и не обавезују бављење било каквим конструктором који се не тиче конкретног објекта. Пошто не постоји специфична хијерархија класа и сваки објекат посебно дефинише како ће да израчуна атрибуте наведене у протоколу, терет иницијализације не постоји и извршавање је ефикасно као било који позив *C* функције.

Јасна шема имплементације обавезује објекте да имплементирају методе и атрибуте наведене у протоколу и самим тим се обезбеђује јединствен интерфејс ка специфичним имплементацијама.

3 Преглед функционалности *CryptoLab* омотача

3.1 Хеш функције

Криптографска хеш функција је функција која произвољан низ битова пресликава у низ битова фиксне дужине [6]. Резултат хеш функције се назива хеш кôдом. Хеш функције се веома често користе у рачунарству а разлика између криптографских и стандардних хеш функција је та што криптографске хеш функције морају да буду једносмерне, тј. да се резултујући хеш кôд не може трансформисати назад у почетну вредност.

У пракси једносмерност није довољна особина коју морају да поседују криптографске хеш функције. Оне такође треба да буду отпорне на колизије што би значило да не постоје, или макар да је тешко наћи, две различите ниске битова чија хеш функција као резултат даје идентичан хеш кôд. Да би се избегла колизија, дужина хеш кôдова би требало да буде макар 128 битова у случајевима када велика сигурност није неопходна, док је препоручена минимална величина 160 битова.

Ниска битова коју криптографска хеш функција обрађује би требало да буде неограничене величине. У пракси је тешко дефинисати функцију, која ради на бесконачном домену, па се обично дефинишу компресионе функције које битове дужине s мапирају у битове дужене n где је $s > n$ да би се потом вршило уланчавање у циљу стварања функције над бесконачним доменом. Популаран алгоритам за уланчавање је Меркле–Дамгардов алгоритам (видети [6]).

Основни принцип дизајнирања хеш функције је ефекат лавине где мала промена код улазних података резултује велику и непредвидиву промену у излазним подацима. Овај принцип је карактеристика *MD4* групе алгоритама. Сви *MD4* алгоритми су итеративни.

Најзначајнији алгоритми који припадају овој групи, а уједно и имплементирани у овом раду, су:

- md5
- sha1
- sha224
- sha256
- sha384

- sha512

3.2 Шифровање

Шифровања је процес где се подаци из свог изворног стања трансформишу у шифровано стање [7]. Процес обрнут од шифврования се зове дешифровање. Постоје две врсте алгоритама за шифровање: симетрични и асиметрични алгоритми. Симетрични алгоритми користе исти кључ за шифровање и дешифровања, док асиметрични користе пар кључева од чега је један приватни и обично служи за дешифровање, а други је јавни и служи за шифровање.

Имплементирани су следећи алгоритми за шифровање:

- AES
- RSA
- Blowfish

3.3 Аутентификација порука и размена кључева

Криптографски хеш кôдом се може проверити да ли су подаци мењани, рачунањем кôда на почетку слања података и након пријема истих [8]. Израчунати хеш кôдови би требало да се поклапају. Хеш код у овом случају би такође требало да буде заштићен да не би дошло до његовог нежељеног мењања. Ово се обезбеђује коришћењем кôда за аутентикацују порука, *MAC*–а. Овај механизам се заснива на дељеном тајном кључу који је неопходан за крајњу верификацију. Наравно, овим процесом се не обезбеђује поверљивост података него само њихов интегритет.

Имплементирани алгоритми за аутентикацију порука:

- DSA
- RSA
- HMAC

Алгоритми за шифровање користе кључеве за податке које хоћемо да шифрујемо. Кључеви могу да буду приватни или јавни. Да би две стране размениле шифорване поруке и успешно их дешифровале, неопходно је да безбедно размене одговарајуће кључеве.

Имплементиран је алгоритам за размену кључева [9]:

- Diffie-Hellman

4 Постојећа решења

Постојећа решења се грубо могу поделити у следеће групе:

1. Омотачи око *Apple*-ове *CommonCrypto* библиотеке написани у Свифту [10]
2. Решења од стране трећих лица (енг. *third party*) са директном имплементацијом криптографских функција у Свифту
3. Разни омотачи написани у језику *Objective-C*

Омотачи око Apple-ове CommonCrypto библиотеке написани у Свифту за циљ имају, слично као пројекат реализован у оквиру овог рада, олакшавање криптографије у међуплатформској комуникацији, али раде само на *Apple* платформама (*iOS, macOS*). Иако оваква решења пружају потпуну сигурност што се тиче поузданости функционалног језгра, потпуно су неупотребљива ван *Apple* платформи. Тачније, немогуће је да се направи омотач *CommonCrypto* библиотеке који би радио на платформи која не припада *Apple* породици.

Популаран представник овог типа је *IDZSwiftCommonCrypto.*

Решења од стране трећих лица (енг. *third party*) са директном имплементацијом криптографских функција у Свифту су у великој мери заступљена. Њихова главна карактеристика је имплементација криптографских функција од „нуле” или коришћење већ готовог језгра где је непознато или није наведено ко је заправо извршио имплементацију. Криптографија представља изузетно осетљиво поље, где најмање грешке у реализацији основа могу довести до огромних проблема и злоупотреба. Мана овог типа решења је што се не може гарантовати поузданост криптографске функционалности, док је предност у томе што је употреба обично доста једноставна и интерфејси ка кориснику су прилично јасни. Популаран представник овог типа је *CryptoSwift.*

Разни омотачи написани у језику Objective-C су најстарији тип решења. Обухватају типове решења из две горе наведене групе, само што је комплетна реализација у језику *Objective-C.* У зависност и од тога шта чини језгро система, омотачи ове врсте имају све

предности и мане одговарајућих верзија у Свифту. Поред тога, треба имати у виду да *Objective-C* све више застарева, а софтверу написаном у Свифту природно више одговара омотач написан у Свифту него у језику *Objective-C.* Овакви омотачи не могу да пруже пун комфор Свифт порограмеру као и примену свих специфичности Свифт програмирања.

Популаран представник овог типа је *MIHCrypto.*

4.1 Основне карактеристике *CryptoLab* омотача

Омотач представљен у овом раду покушава да искористи најбоље карактеристике већ постојећих решења. Циљ је да се користи поуздана основа у виду *OpenSSL*–а у комбинацији са једноставним итерфејсом ка кориснику.

Основне карактеристике овог омотача су:

1. Комплетна имплементација у програмском језику Свифт
2. Интуитивна употреба
3. Омотач око библиотеке *OpenSSL*
4. Омогућена одговарајућа надрградња типова *Data* и *String*
5. Покривеност тестовима јединица кôда

Имплементација у програмском језику Свифт – Програмски језик Свифт, као нова технологија, све више мења првобитни језик коришћен за *Apple* платформе, *Objective-C*. Овај омотач је имплементиран комплетно у Свифту корошћењем свих званичних стандарда и препорука. Тиме се обезбеђује удобност коришћена свих предности које Свифт технологија доноси.

Интуитивна употреба је један од основиних захтева на који је требало одговорити у имплементацији овог пројекта. Стога је прилагођавање релативно компликованих и неудобних интерфејса *OpenSSL-a* био основни циљ.

```
do {
    //encryption
    cryptorEnc = try AESCipher(key: key16Byte, iv: genericIV, blockMode: .cbc)
    let encrypted = try cryptorEnc.encrypt(data: testData)

    //decryption
    cryptorDec = try AESCipher(key: key16Byte, iv: genericIV, blockMode: .cbc)
    let decrypted = try cryptorDec.decrypt(data: encrypted)
}
catch let err {
    // handle error
}
```

Слика 5: Пример коришћења шифровања и дешифровања

Омотач око библиотеке OpenSSL – *OpenSSL* је један од најпоузданијих система за мрежне протоколе и енкрипцију и ужива поверење великог броја људи. Библиотека је поуздана и користи се на свим платформама. Омотач око ове библиотеке користи сву њену логику и самим тим садржи сву поузданост и применљивост.

Одговарајућа надоградња типова Data и String – Интуитивна употреба је још више поспешена имплементирањем одговарјућих екстензија за типова података над којима се најчешће примењују криптографске функције. Тиме се омогућава изузетна брза и ефикасна употреба омотача.

```
let testData = "Test string".data(using: .utf8)!
let hashCodeFromData = testData.md5()
let hashCodeFromString = "Test string".md5()
```

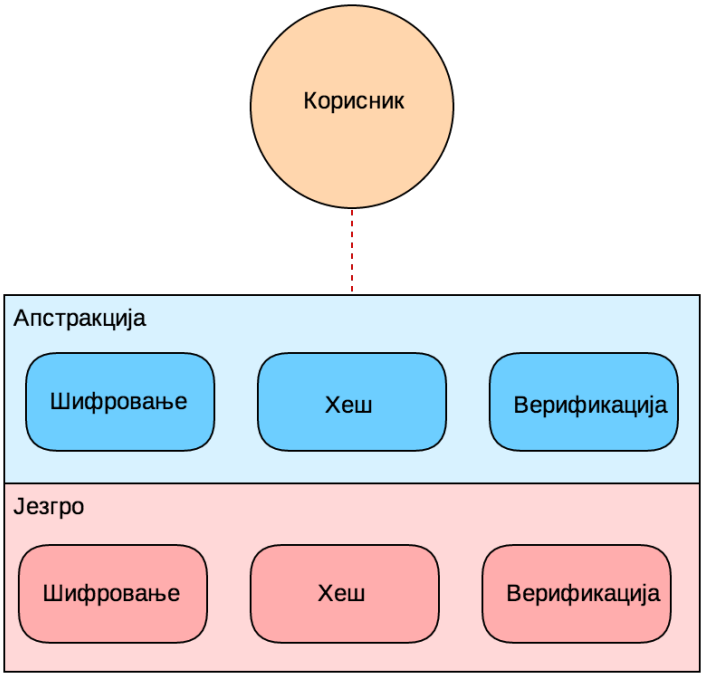
Слика 6: Рачунање md5 хеш кôда директно над типовима String и Data

Кôд покривен тестовима – Тестови јединица кôда су написани за сваку функционалност омотача. Тиме се обезбеђује провера да криптографске функције раде како би требало. Тестови такође могу да буду корисни, као примери основиних случајева употребе, новим корисницима као и онима који желе да допринесу даљем развоју.

5 Преглед архитектуре

Омотач је подељен на три архитектуралне целине и то су:

- Целина која се бави шифровањем/дешифровањем
- Целина која се бави потписивањем и верификацијом порука
- Целина која се бави рачунањем хеш функција



Слика 7 Општи преглед архитектуре

Целине су међусобно независне и свака представља врсту засебног система. Основни циљ приликом дизајна ових целина је било раздвајање апстракције од имплементације. Пошто свака компонента у свом језгру имплементира методе *OpenSSL-a*, било је неопходно да се кориснику пружи једноставан интерфејс за све функције језгра, као и да се даље ажурирање оба дела учини што једноставнијим и независнијим.

Због тога је сличан принцип коришћен у свим целинама и он се заснива на обрасцу за пројектовање *Мост*, чија је основна сврха раздвајање имплементације од апстракције.

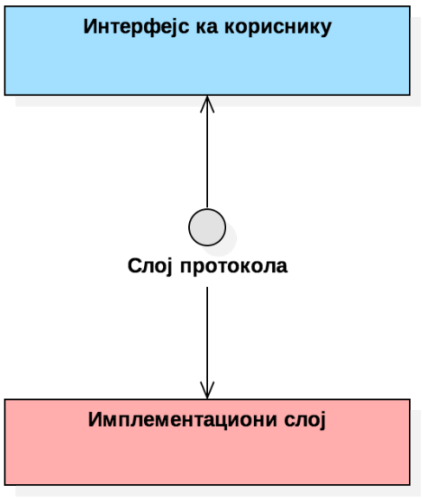
Свака архитектурална целина, која је наведена, се састоји из следећих слојева (*Слика 9*):

- Имплементациони слој
- Слој протокола
- Интерфејс ка кориснику

Имплементациони слој чини језгро сваке архитектуралне целине и бави се конкретном иплементацијом. У овом слоју се директно позивају и користе функције *OpenSSL*. Ово је једини слој који комуницира са *OpenSSL*–ом и који уопште зна да *OpenSSL* постоји. Имплементациони слој се састоји од више класа које су међусобно независне. Било која класа овог слоја не зна за постојање осталих класа. Тиме је функционалност строго локализована и одговорности се ни у ком случају не деле. Сав посао који се обавља у одређеној класи зависи искључиво од ње.

Слој протокола се састоји искључиво од протокола неопходних да се обезбеде све предности ПОП приступа као и да се повежу имплементациони слој и слој интерфејса ка кориснику.

Итерфејс ка кориснику је једини слој који је директно изложен крајњем кориснику. Овде је акценат на једноставним позивима метода и обезбеђивању интуитивног коришћења целокупног система. Овај слој посредно, преко слоја протокола, комуницира са имплементационим слојем и прилагођава његове функционалности спољном свету.



Слика 8 Општа структура слојева

5.1 Образац за пројектовање *Мост*

У развоју софтвера обрасци за пројектовање су решења за честе проблеме у дизајну софтвера, а која се могу изнова примењивати. Образац за пројектовање не пружа готово решење које се може директно преточити у програмски кôд, већ пружа шаблон за решавање проблема који се може искористити у више различитих ситуација. Обрасци са пројектовање се деле на следеће групе: градивни обрасци, структурни обрасци и обрасци понашања [11].

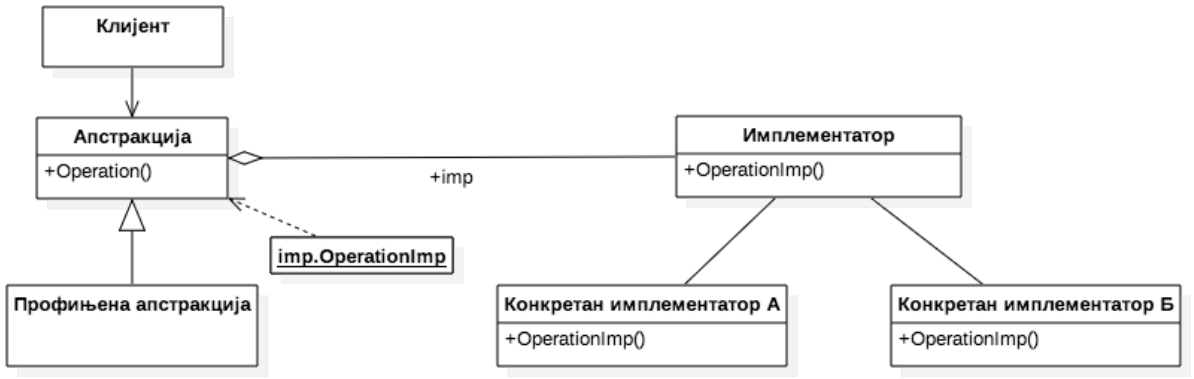
Структурни обрасци описују начине на које класе и објекти могу међусобно да се комбинују и граде структуре. Пажња се обраћа на структуру класа и објеката и на функционалности које та структура доноси а не на конкретну имплементацију и интерфејсе.

Групу структурних образаца чине следећи обрасци за пројектовање: *адаптер (енг. Adapter)* , *мост (енг. Bridge)*, *састав (енг. Composite)*, *декоратер (енг. Decorator)*, *фасада (енг. Facade)*, *мува (енг. Flyweight)*, *прокси (енг. Proxy)*.

Мост је структурни образац за пројектовање и служи да би се одвојила апстракција од имплементације чиме би се, на крају, стекла могућност да се оба дела независно мењају.

Када апстракција има неколико могућих имплементација, ситуација се контролише тако што се користи наслеђивање. Апстрактна класа дефинише интерфејсе док се конкретне подкласе баве имплементацијом. У неким случајевима ово решење није довољно флексибилно зато што је имплентација подкласе везана за апстракцију надкласе и свака промена у једном од то два дела, захтева промену у другом делу.

Мост образац решава овај проблем тако што прави одвојене хиерархије наслеђивања за апстрастракцију и имплементацију. Однос између апстрактних наткласа тих хиерархија заправо чини *мост* између два дела.



Слика 9 Образац за пројектовање Мост

Мост образац има следеће карактеристике:

- Одваја имплементацију од апстракције чиме се повећава флексибилност
- Унапређује процес наслеђивања зато што свака целина има одвојену хиерархију којој припада
- Јасно подељени слојеви скривају детаље имплементације од клијената

5.2 Примена обрасца за пројектовања *Мост*

У претходним поглављима је описан концепт ПОП са свим предностима које доноси као и његово поређење са стандардним решењима са класама. Свифт имплементација обрасца *Мост* је управо заснована на принципу ПОП тако да се сви делови имплементације представљају на одговарајући начин ПОП-а.

5.2.1 Пример имплементације у програмском језику Свифт

Улогу апстрактних класа у Свифту преузимају протоколи. Комплетан пример имплементације је приказан на слици 10 и биће детаљније објашњен.

```
protocol Switch {
    var appliance: Appliance {get set}
    func turnOn()
}

protocol Appliance {
    func run()
}

class RemoteControl: Switch {
    var appliance: Appliance

    func turnOn() {
        self.appliance.run()
    }

    init(appliance: Appliance) {
        self.appliance = appliance
    }
}

class TV: Appliance {
    func run() {
        print("tv turned on")
    }
}

class VacuumCleaner: Appliance {
    func run() {
        print("vaccum ckeaber turned on")
    }
}
```

Слика 10 Пример обрасца Мост

Класа *RemoteControl* је задужена за апстракцију док класе *TV* и *VacuumCleaner* пружају конкретну имплементацију.

Да би се апстракција одвојила од имплементације, направљена су два протокола: *Switch,* који делује са стране апстракције, и *Appliance,* који се бави имплементацијом.

Протокол *Appliance* обавезује сваког ко га усвоји да имплементира методу *run()*, што и чине имплементационе класе (*TV* и *VacuumCleaner*) тако да свака имплементациона класа има своју специфичну имплементацију методе *run()*.

Протокол *Switch* обавезује сваког ко га усвоји да имплементира методу *turnOn()* као и да дефинише промењљиву *appliance*, објекат који усваја *Appliance* протокол. Обратимо пажњу управо на *appliance* промењљиву зато што она представља мост између апстракције и имплементације.

RemoteControl усваја *Switch* протокол па самим тим имплементира методу *turnOn()* која је отворена ка клијенту и служи као интерфејс. Која конкретна имплементација се користи када се позове *turnOn()* зависи од тога који је објекат инстанциран у промењљивој *appliance.*

Пример употребе ове имплементације се може видети на слици 11:

```
var tvRemoteControl = RemoteControl(appliance: TV())
tvRemoteControl.turnOn()

var fancyVacuumCleanerRemoteControl = RemoteControl(appliance: VacuumCleaner()
fancyVacuumCleanerRemoteControl.turnOn()
```

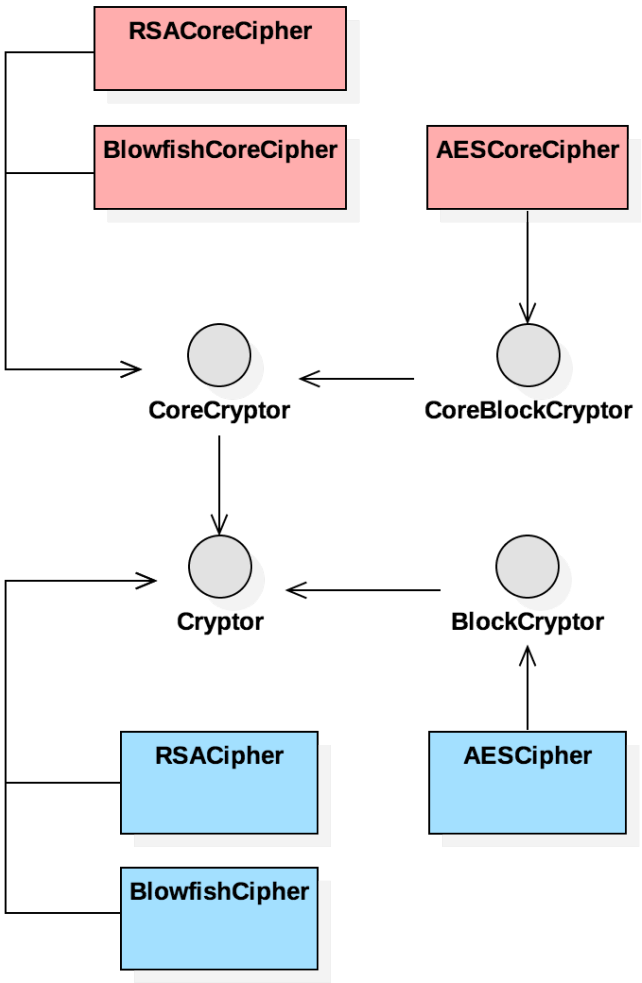
Слика 11 Употреба Мост обрасца

Приликом иницијализације објекта класе *RemoteControl* се иницијализује конкретан објекат класе која усваја *Appliance* протокол. Тиме се постиже да метода *turnOn() "*зна" како да се понаша када буде била позвана.

Имплементација узорка *Мост* у овом раду није потпуно идентнична наведеном примеру, него је адаптирана потребама омотача. Поред одређених модификација основни циљ је остао исти, а то је раздвајање имплементације од апстракције. Следи објашњење конкретне имплементације узорка за пројектовање *Мост* у омотачу.

5.3 Детаљан преглед дела за шифровање

Детаљну употребу обрасца *Мост* у омотачу ћемо видети у примеру сегмента који се бави шифровањем.



Слика 12 Структура дела за шифровање

Класе које су (*Слика 12*) црвене боје су имплементационе, плаве класе имплементирају апстракцију док је сиви слој између апстракције и имплементације слој протокола који заправо представља мост.

Имплементациони слој чине класе *RSACoreCipher*, *BlowfishCoreCipher* и *AESCoreCipher*. Класе комуницирају директно са *OpenSSL*–ом и позивају његове функције. Такође ”не знају” једна за другу тако да су потпуно независне.

```
private func encrypt(data dataToEncode: Data, rsaKey: UnsafeMutablePointer<RSA>) throws -> Data {
        let rsaStruct =  UnsafeMutablePointer(rsaKey)

        let dataPointer = (dataToEncode as NSData).bytes.bindMemory(to: UInt8.self, capacity: dataToEncode.count)
        let dataSize = dataToEncode.count

        let encryptedPointer = UnsafeMutablePointer<UInt8>.allocate(capacity: Int(RSA_size(rsaStruct)))

        let encryptedSize = RSA_public_encrypt(Int32(dataSize), dataPointer, encryptedPointer, rsaStruct, padding)

        if encryptedSize == -1 {
                throw CipherError.cipherProcessFail(reason: CipherErrorReason.cipherEncryption)
        }

        return Data(bytes: UnsafePointer<UInt8>(encryptedPointer), count: Int(encryptedSize))
}
```

Слика 13 Пример методе имплементационог слоја

Интерфејс ка кориснику пружају класе *RSACipher*, *BlowfishCipher* и *AESCipher* и њихов посао је да на прегледан и интуитиван начин учине цео омотач употребљивим. Овде наилазимо на прву разлику у односу на пример наведен у претходном поглављу (*Слика 10*).

На пример из претходног поглавља се може видети да се увек креира један исти објекат класе *RemoteControl* који у свом конструктору прима објекат чију ће имплементацију да зове (*TV* или *VacuumCleaner*). Уколико бисмо идентичан принцип применили на омотач, постојала би класа која би се звала *Cipher* и која би у конструктору примала одговарајући објекат за *RSA*, *AES* или *Blowfish* алгоритам. Тај начин, због саме прегледности и интуитивности, не одговара концепту омотача. Имајући у виду да сваки алгоритам има специфична подешавања, позив таквог објекта би изгледао не баш тако јасно (*Слика 14*).

```
let blowfishEnc = Cipher(cipher: BlowfishCipher(key: key, iv: genericIV, encryptionMode: .ecb))
```

Слика 14 Пример позива немодификованог обрасца Мост

Промена је настала тако што се нису све класе објединиле кроз једну класу, него сваки тип алгоритма има своју класу која је мостом повезана за имплементационим слојем. У случају дела за криптовање класе *RSACipher*, *BlowfishCipher* и *AESCipher* су одвојене и засебно одржавају везу са одговарајућом имплементацијом. На примеру испод (Слика 15) се види имплементација *RSACipher* класе.

Имплементација је изузетно јасна и прегледна тако да се кориснику ствара чиста слика како се користи овај систем. Линија обележена са *// 1* (*Слика 15*) показује дефинисање промењиве која врши улогу моста. Са *// 2, // 3* и *// 4* су обележени конструктори где се дефинише конкретан објекат који представља мост. Исти принцип је коришћен и за класе *BlowfishCipher* и *AESCipher*.

```
public class RSACipher: NSObject, Cryptor {

    // 1
    public var coreCryptor: CoreCryptor

    public var privateKey: String? {
        return (coreCryptor as? RSACoreCipher)?.privateKey
    }

    public var publicKey: String? {
        return (coreCryptor as? RSACoreCipher)?.publicKey
    }

    // 2
    public init(padding: RSAPadding = .none) {
        coreCryptor = RSACoreCipher(padding: padding)
        super.init()
    }

    // 3
    public init(publicKey: Data, padding: RSAPadding = .none) {
        coreCryptor = RSACoreCipher(publicKey: publicKey, padding: padding)
        super.init()
    }

    // 4
    public init(publicKey: Data, privateKey: Data, padding: RSAPadding = .none) {
        coreCryptor = RSACoreCipher(publicKey: publicKey, privateKey: privateKey, padding: padding)
        super.init()
    }
}
```

Слика 15 Пример слоја апстракције

Слој протокола чине протоколи *CoreCryptor, Cryptor, CoreBlockCryptor* и *BlockCryptor.* За разлику од имплементационог слоја и интерфејса ка кориснику (који одговарају

имплементацији описаној у претходном поглављу), слој протокола је прилагођен на следећи начин. *CoreCryptor* је основни протокол имплементационог нивоа и он обавезује све класе који га усвајају да имплементирају одговарајуће функције за шифровање и дешифровање (*Слика 16*).

```
public protocol CoreCryptor {
    /**
    Encrypts data and throws error if needed

    - parameter toEncrypt: Data to encrypt
    */
    func encrypt(data toEncrypt: Data) throws -> Data

    /**
    Decrypts data and throws error if needed

    - parameter toDecrypt: Data to encrypt
    */
    func decrypt(data toDecrypt: Data) throws -> Data
}
```

Слика 16 Дефинисање CoreCryptor протокола

CoreBlockCryptor је протокол који усваја протокол *CoreCryptor* и намењен је класама које пружају могућност шифровања блокова података. Класа која имплементира овај протокол би требало да имплементира све методе дефинисане у *CoreCryptor*–у, као и оне наведене у *CoreBlockCryptor* протоколу. Кроз овакву врсту хијерархије даље настављамо да градимо мост.

На страни слоја интерфејса ка кориснику се налазе протоколи *Cryptor (Слика 17)* и *BlockCryptor* који су повезани на исти начин као и горе описани *CoreCryptor* и *CoreBlockCryptor*.

```
public protocol Cryptor {
    /**
    Internal object
    */
    var coreCryptor: CoreCryptor {get}
}
```

Слика 17 Дефиниција Cryptor протокола

CoreCryptor протокол дефинише промењиву која ствара мост. Та промељива може да буде било који објекат који имплементира *CoreCryptor* протокол.

```
public protocol CoreBlockCryptor: CoreCryptor {
    /**
    Inits new block encryption and throws error if needed

    - parameter key: Encryption key
    - parameter iv: Initialization vector
    */
    func initEncryption(withKey key: Data, andIV iv: Data) throws

    /**
    Updates new block encryption and throws error if needed

    - parameter data: Data block to update encryption with
    - parameter iv: Initialization vector
    */
    func updateEncryption(data toUpdate: Data) throws

    /**
    Finishes block encryption and throws error if needed

    - returns: Encrypted data
    */
    func finishEncryption() throws -> Data

    /**
    Inits new block decryption and throws error if needed

    - parameter key: Decryption key
    - parameter iv: Initialization vector
    */
    func initDecryption(withKey key: Data, andIV iv: Data) throws

    /**
    Updates new block encryption and throws error if needed

    - parameter key: Data block to update decryption with
    */
    func updateDecryption(withData data: Data) throws

    /**
    Finishes block decryption and throws error if needed

    - returns: Decrypted data
    */
    func finishDecryption() throws -> Data
}
```

Слика 18 CoreBlockCryptor протокол

Овде веза између протокола блиских имплементационом слоју постаје корисна. Пошто *CoreBlockCryptor* усваја *CoreCryptor,* сваки објекат који имплементира *CoreBlockCryptor* је уједно и *CoreCryptor* протокол, тако да промењиву која представља мост могу да чине обејкти који се баве обичим и блоковским шифорвањем.

Коначно, мост се реализује преко екстензије *Cryptor* протокола (*Слика 19*).

```
extension Cryptor {
    /**
    Encrypts data.

    - parameter dataToEncrypt: Data to encrypt

    - returns: Encrypted data
    */
    public func encrypt(data dataToEncrypt: Data) throws -> Data {

        do {
            let encrypted = try self.coreCryptor.encrypt(data: dataToEncrypt)
            return encrypted
        }
        catch let error {
            throw error
        }
    }

    /**
    Decrypts data.

    - parameter dataToDecrypt: Data to decrypt

    - returns: Decrypted data
    */
    public func decrypt(data dataToDecrypt: Data) throws -> Data {
        do {
            let decrypted = try self.coreCryptor.decrypt(data: dataToDecrypt)
            return decrypted
        }
        catch let error {
            throw error
        }
    }
}
```

Слика 19 Екстензија Cryptor протокола

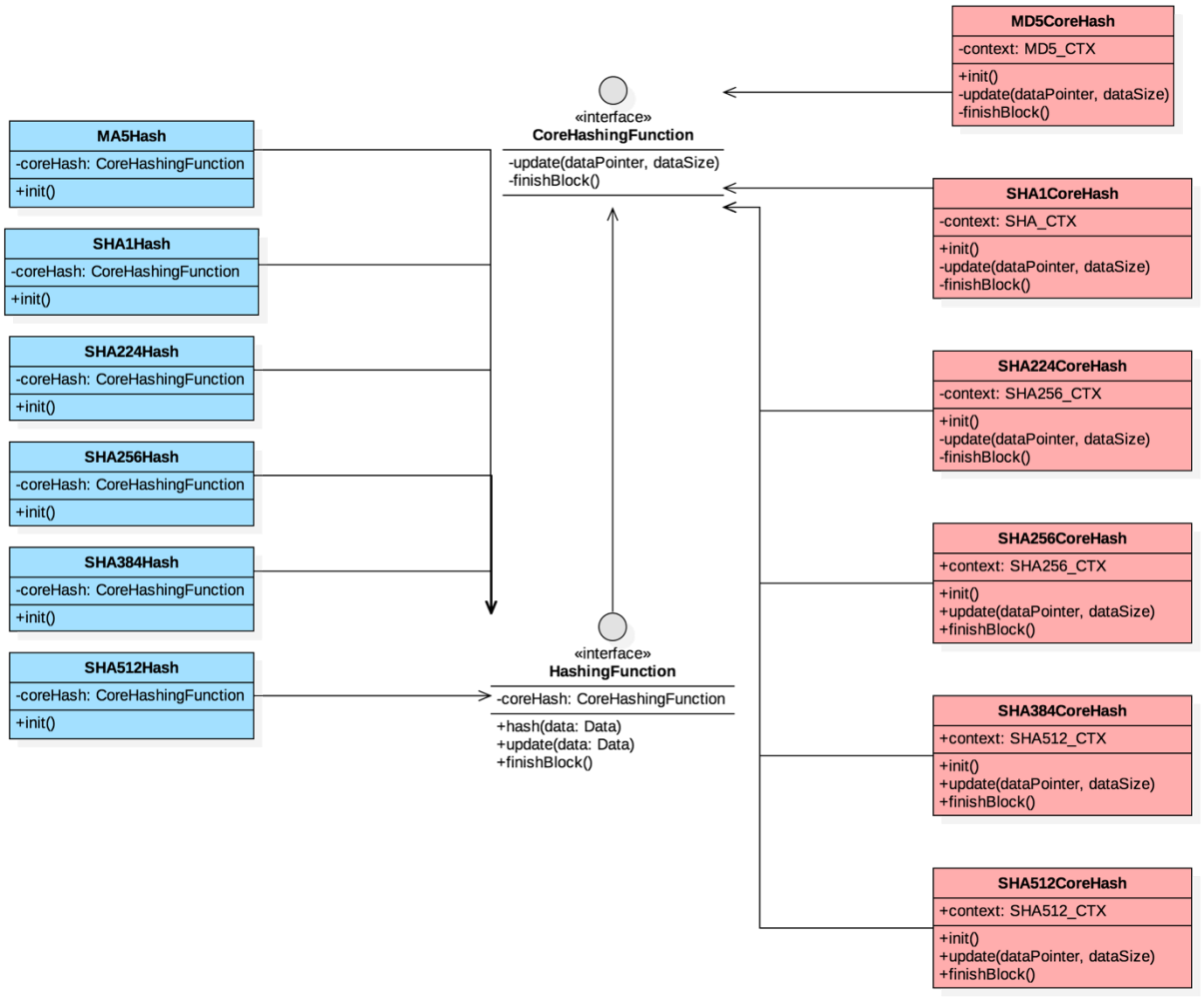
Приметимо да се у дефиницији овог протокола наводи само промењљива која представља мост. То значи да свака класа која имплементира овај протокол мора да дефинише промењљиву која има улогу моста. Методе дефинисане у екстензији се нигде не појављују, осим у екстензији протокола. Овим путем постижемо јединствену имплементацију за све типове шифровања. Који ће алгоритам за шифровање да се зове, зависи искључиво од тога који је објекат са улогом моста дефинисан.

5.4 Преглед осталих делова

У претходној секцији је детаљно описана организација дела за шифровање. На потпуно исти начин образац *Мост* је искоришћен и у деловима за рачунање хеш кôдова и верификацију порука па стога ови сегменти неће бити толико детаљно описани, него ће бити урађен основни структурни преглед.

5.4.1 Део за рачунање хеш функција

Детаљнији приказ организације дела за рачунање хеш функција види се на слици 18. Организација овог дела, за разлику од дела за шифровање, не садржи хијерархију у слоју протокола, него слојеви имплементације и апстракције имају тачно по један протокол који имплементирају.



Слика 20 Преглед система за хеширање

Црвеном бојом су обележене класе које су имплементационе а плаве класе се имплементирају апстракцију. Слој интерфејса је сиве боје. (*Слика 20*)

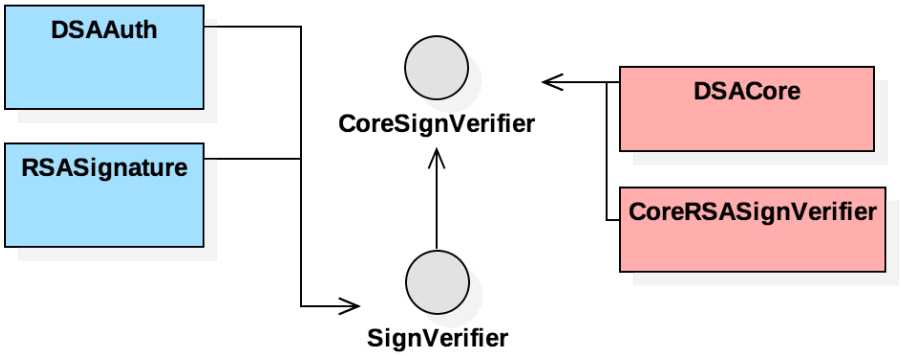
Имплементационе класе усвајају *CoreHashingFunction* протокол чиме се обавезују да ће да имплементирају методу за ажурирање актуелног блока података чији хеш кôд желимо да израчунамо као и методу за завршетак ажурирања и рачунање хеш кôда.

Класе апстракције усвајају *HashingFunction* протокол и обавезују се да ће да дефинишу *coreHash* променљиву која представља мост и која може да буде било који објекат који усваја *CoreHashingFunction* протокол. Поред тога, *HashingFunction* протокол пружа

имплементације метода за рачунање хеш кôда блокова података као и нормално рачунање хеш кôдова, чиме је постигнута потпуна флексибилиност.

5.4.2 Верификација порука

Део за верификацију порука је најједноставнији и функционише на исти начин као и претходно описани делови. (*Слика 21*)



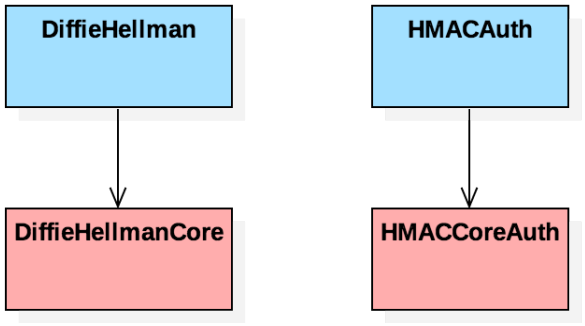
Слика 21 Преглед верификације порука

У овом случају слој протокола чине *CoreSignVerifier* и *SignVerifier* протоколи. Класе имплементације (*DSACore* и *CoreRSASignVerifier*) усвајају *CoreSignVerifier* протокол. Класе апстракције имплементирају *SignVerifier* протокол, који, поред одговарајућих метода, дефинише и променљиву са улогом моста. Променљива са улогом моста може да буде било који објекат који усваја *CoreSignVerifier* протокол.

5.4.3 Делови који нису према обрасцу за пројектовање *Мост*

Изнад су наведене три архитектуралне целине од којих се састоји овај омотач. Поред њих постоје и функционалности (рачунање HMAC-а и Diffie-Hellman размена кључева) које се нису подвеле ни под једну архитектуралну целину, једноставно зато што нуде специфичну групу функционалности коју не деле са остатком система. Организација овог дела система је прилично линеарна (*Слика 22*). Класе имплементације комуницирају

са *OpenSSL*—ом, док класе које су отворене ка кориснику држе референцу на одговарајућу имплементациону класу и позивају њене методе.



Слика 22 DiffieHellman и HMAC преглед

Класе *DiffieHellman* и *HMACAuth* представљају интерфејс ка кориснику и позивају функције класа *DiffieHellmanCore,* односно *HMACAuthCore.*

```
/**
Computes shared secret with public key.

- parameter publicKey: Public key of other client.
*/
public func computeSharedSecret(withPublicKey publicKey: Data) -> Data? {
    return dhCore.computeKey(withPublicKey: publicKey)
}
```

Слика 23 Пример методе у DiffieHellman класи

Методом приказаном на слици 23 се заправо само зове имплементација класе *DiffieHellmanCore* која је приказана на слици 24.

```
func computeKey(withPublicKey pk: Data) -> Data? {
    if let dhKey = dhKey {
        let size = DH_size(dhKey)
        var computedKey = Data.makeUInt8EmptyArray(ofSize: Int(size))

        var bnG = BN_new()
        let gData = pk.makeInt8DataPointer()
        let gPointer = UnsafePointer<Int8>(gData)
        BN_dec2bn(&bnG, gPointer)

        DH_compute_key(&computedKey, bnG, dhKey)
        return Data(bytes: computedKey)
    }
    return nil
}
```

Слика 24 Пример методе из класе DiffieHellmanCore

У дизајну ових класа акценат је стављен на једноставности. Јасно се раздваја део који је задужен за имплементацију и део који је отворен ка кориснику. Пошто немамо више могућих имплементација истих функција као ни више различитих класа које деле сличне или исте функционалности, није било потребе за подизањем архитектуралне сложености.

Ипак, уколико се у будућности овај део буде развијао и постајао комплекснији, због иницијалне јасне раздвојености слојева, биће погодно применити одговарајући узорак за пројектовање без велике количине рефакторисања кôда.

6 Примери употребе

У овом делу ће бити наведени конкретни примери употребе овог омотача. Идеја је да се покаже једноставност употребе целокупног система

6.1 Шифровање и дешифровање

6.1.1 RSA

RSA алгоритам је асиметричан алгоритам за шифровање и дешифровање порука. Асиметричан је зато што користи два различита кључа. Овај принцип се такође зове и *криптографија јавног кључа* зато што је један од кључева доступан свима а други мора да остане тајан. Корисник RSA алгоритма дели производ два велика проста броја, заједно са помоћним вредностима, као свој јавни кључ. Прости бројеви морају да остану тајни. Свако са јавним кључем може да шифрује поруку али дешифровање може да изврши само онај ко има просте бројеве (за детаље видети [12]).

У овомо примеру можемо да видимо како функционише шифровање и дешифровање коришћењем *RSA* алгоритма. Овај пример за циљ има симулацију реалног случаја употребе:

Имамо две стране које хоће да размене поруку. Једна страна се бави шифровањем помоћу јавног кључа друге стране, док друга страна врши дешифровање уз помоћ свог приватног кључа.

```
// Корак 1
let testData = Data(bytes: [0x50, 0xb7, 0x73, 0xc8, 0x42, 0x1e, 0x3d, 0x1a, 0x5e,
0xc4, 0x48, 0x50, 0x80, 0x03, 0x03, 0x66])


// Корак 2
let decryptor = RSACipher(padding: .pkcs1)


// Корак 3
guard let pubK = decryptor.publicKey?.data(using: .utf8) else {
```

```
        return
}


// Корак 4
let encryptor = RSACipher(publicKey: pubK, padding: .pkcs1)


// Корак 5
do {
        let encryptedData = try encryptor.encrypt(data: testData)
        let decryptedData = try decryptor.decrypt(data: encryptedData)
}
catch let err {
        //handle error
}
```

Прво дефинишемо промењиву са подацима који се шифрују (// **1**) затим

дефинишемо обејкат који ће бити задужен за дешифровање (// **2**).

Потом следе кораци:

Корак 3

Извлачимо јавни кључ из објекта који је задужен за дешифровање

Корак 4

Дефинишемо обејкат који ће бити задужен за шифровање.

Корак 5

На једноставан начин се врши прво шифровање почетних података па онда дешифровање шифрованих података

6.1.2 Blowfish

Blowfish алгоритам за шифровање са симетричним кључем, што значи да се исти кључ користи за шифровање и дешифровање. Овај алгоритам је прилично безбедан пошто још увек није пронађен ефикасан начин да се открије садржај енкриптоване поруке (за детаље видети [13]).

Слично као и у *RSA* примеру, овде се може видети како функционише *Blowfish* алгоритам у шифровању између два клијента.

```
// Корак 1
let key  = "secret".data(using: .utf8)!
let genericIV =  "12345678".data(using: .utf8)!
let testData = "12345678".data(using: .utf8)!


// Корак 2
let blowfishEnc = BlowfishCipher(key: key, iv: genericIV, encryptionMode: .ecb)
let blowfishDec = BlowfishCipher(key: key, iv: genericIV, encryptionMode: .ecb)


// Корак 3
do {
        let encrypted = try blowfishEnc.encrypt(data: testData)
        let decrypted = try blowfishDec.decrypt(data: encrypted)
}
catch let err {
        //handle error
}
```

Корак 1

Дефинишемо податке неопходне за шифровње: кључ, иницализациони вектор и податке

које шифрујемо.

Корак 2

Дефинишемо објекте који се баве шифровање односно дешифровањем.

Корак 3

На крају се врши само шифровање а потом и дешифровањем.

6.1.3 AES

AES је спецификација за шифровање електронских података коју је написао Национални институт Сједињених Америчких Држава за стандарде и технологију, 2001. године. Овај алгоритам се заснива на Ријндел алгоритму који су развила два белгијска научника. Ријндел заправо представља породицу алгоритама за шифровање са другачијим кључевима и величином блокова (за више детаља видети [14]).

AES алгоритам за шифровање подржава, како обичние податаке, тако и блокове података.

```
do {


// Корак 1

let testData = "test string".data(using: .utf8)

let key16Byte =  Data(bytes: [0x5f, 0xf5, 0x0c, 0x5b, 0x60, 0x96, 0x84, 0xa2, 0x35,
0xd5, 0xc5, 0xbf, 0x24, 0x69, 0x40, 0x8a])

let genericIV =  Data(bytes: [0x4f, 0x83, 0x51, 0xae, 0x1c, 0x48, 0xf4, 0x81, 0x65,
0xf8, 0x1b, 0x53, 0x3d, 0xd6, 0xd9, 0x1f])


// Корак 2

let cryptorEnc = try AESCipher(key: key16Byte, iv: genericIV, blockMode: .cbc)

let encrypted = try cryptorEnc.encrypt(data: testData)


// Корак 3
```

```
let cryptorDec = try AESCipher(key: key16Byte, iv: genericIV, blockMode: .cbc)
let decrypted = try cryptorDec.decrypt(data: encrypted)


}
catch let err {
        //handle error
}
```

Корак 1

Дефинишемо податке неопходне за шифровње: кључ, иницализациони вектор и податке које шифрујемо.

Корак 2

Дефинишемо објекат који је задужен за шифровање па се шифровање потом и врши.

Корак 3

Дефинишемо објекат који је задужен за дешифровање па се дешифровање потом и врши.

Следећи пример показује шифровање блокова података.

```
// Корак 1
let block1 = "block1 ".data(using: .utf8)!
let block2 = "block2 ".data(using: .utf8)!
let block3 = "block3".data(using: .utf8)!


do {
        // Корак 2
        let aes = try AESCipher(key: key32Byte, iv: genericIV, blockMode: .ctr)
        // Корак 3
```

```
        try aes.updateEncryption(withDataBlock: block1)
        try aes.updateEncryption(withDataBlock: block2)
        try aes.updateEncryption(withDataBlock: block3)
        let encryptedData = try aes.finishEncryption()
}
catch let err { //handle error }
```

Корак 1

Дефинишемо блокове података

Корак 2

Дефинишемо објекат за шифровање

Корак 3

Дефинисане блокове појединачно шифрујемо а потом завршавамо шифровање где добијамо коначан резултат.

6.2 Рачунање хеш кôдова

Рачунање хеш кôдова [6] ће бити представљено кроз пример *MD5* хеш функције. Основни пример даје једноставан хеш кôд. У овом примеру се прво (Корак 1) дефинишу подаци чији хеш кôд хоћемо да израчунамо. Након тога (Корак 2) дефинишемо нови *MD5* објекат и рачунамо хеш кôд.

```
// Корак 1
let testData = "Test string".data(using: .utf8)!
// Корак 2
let hashFunc = MD5Hash()
let hashCode = hashFunc.hash(data: testData)
```

Хеш кôд можемо да израчунамо на још једноставнији начин:

```
// Корак 1
let testData = "Test string".data(using: .utf8)
// Корак 2
let hashCodeFromData = testData.md5()
// Корак 3
let hashCodeFromString = "Test string".md5()
```

На почетку се (Корак 1), као у претходном примеру, дефинишу подаци да би се одмах над самим типом података (Корак 2) позвала *md5()* функција која рачуна хеш кôд. Након тога (Корак 3) се рачуна на сличан начин хеш кôд над ниском карактера тј. типом *String*.

6.3 Аутентификација порука

6.3.1 RSA аутентификација

```
// Корак 1
let testData = Data(bytes: [0x50, 0xb7, 0x73, 0xc8, 0x42, 0x1e, 0x3d, 0x1a, 0x5e,
0xc4, 0x48, 0x50, 0x80, 0x03, 0x03, 0x66])

// Корак 2
let signer = RSACipher()
let signature = signer.sign(data: testData, type: .md5)

// Корак 3
let verify = signer.verify(data: testData, signature: signature!, type: .md5)
```

RSA аутентификација ради тако што се над дефинисим подацима (Корак 1) примена функција *sign()*(Корак 2). Након тога се методом *verify()*(Корак 3) врши провера потписа.

6.3.2 DSA аутентификација

Исти принцип функционисања има и *DSA* аутентификација:

```
let dsa = DSAAuth()

let signature = dsa.sign(data: toSign)!

let verified = dsa.verify(signature: signature, digest: toSign)
```

6.4 HMAC кôд

HMAC кôд се рачуна позивом одговарајуће функције над типом података:

```
let testData = "some test data".data(using: .utf8)!

let key = "some test key".data(using: .utf8)!


let hmac = HMACAuth(key: key, hashFunction: .md5)


let code = hmac.authenticationCode(forData: testData)
```

и путем екстензија над одговарајућим типовима:

```
let codeFromData = testData.hmacAuthCode(withKey: key, hashFunction: .md5)

let codeFromString =

"some test string".hmacAuthCode(withKey: key, hashFunction: .md5)
```

6.5 Diffie–Hellman размена кључева

Diffie-Hellman је метода за безбедну размену криптографских кључева преко јавног канала (за више детаља видети [15]).

Овај пример алгоритма за размену кључева симулира размену између два клијента.

```
// Корак 1
let dhClient1 = DiffieHellman(primeLength: 512)


// Корак 2
let dhClient2 = DiffieHellman(p: dhClient1.p!, g: dhClient1.g!)


// Корак 3
let sharedSecret1 =
dhClient1.computeSharedSecret(withPublicKey:dhClient2.publicKey!.data(using: .utf8)!)


// Корак 4
let sharedSecret2 =
dhClient2.computeSharedSecret(withPublicKey:dhClient1.publicKey!.data(using: .utf8)!)
```

Корак 1 дефинише објекат који генерише параметре за размену кључева. Даље се у кораку 2 користе параметри изегерисани у првом кораку да други клијент конструише одговарајући објекат. Кораци 3 и 4 рачунају кључ који се размењује.

7 Систем отвореног кôда и његова употреба

Развој заснован на дељењу и сарадњи у циљу унапређења изворног кôда је стар колико и сам развој софтвера. У касним деведесетим годинама интересовање за концепт отвореног кôда значајно расте са све већим прихватањем Линукс платформе (*енг. Linux*) од стране јавности и објављивањем изворног кôда итернет прегледача Нетскејп (*енг. Netscape*). Појам отвореног кôда и званично прераста у орагнизацију оснивањем Иницијативе за отворени кôд (*енг. Open source initiative, у даљем тексту ИОК*) која има улогу регулаторног и едукационог тела за сва питања која се тичу отвореног кôда [16,17].

ИОК је званично дефинисао појам отвореног кôда, тј. све услове које програмски кôд мора да задовољи да би био проглашен отвореним. Да би се кôд сматрао отвореним није довољно само да је дозвољен приступ изворном кôду,него је потребно да усвоји лиценцу која испуњава следеће услове

Слободно поновно дељење: Лиценца не сме икога да ограничава што се тиче продавања или предаје софтвера као компоненте софтверске дистрибуције која садржи програме из више различитих извора. Лиценца не сме да захтева накнаду за такву продају

Изворни кôд: Програм мора да укључује изворни кôд и мора да дозволи дељење како изворног кôда тако и компајлиране верзије. Уколико верзија производа није подељена са изворним кôдом, мора бити наведен очигледно означен начин за преузимање изворног кôда уз разумну цену за репродукцију, пожељно је довлачење са интернета бесплатно. Изворни кôд мора да буде у облику у коме програмер жели да модификује програм. Изворни кôдови који намерно крију детаље нису дозвољени. Прелазни облици попут резултата предпроцесора или преводиоца нису дозвољени.

Изведени програм: Лиценца мора да дозволи измене и стварање програма који су изведени из оригиналног кôда а којима мора да буде омогућено дељење под истим условима као и оригиналан софтвер.

Није дозвољена дискриминација према особама и групама: Лиценца не сме да дискримише било коју особу или групу особа.

Није дозвољена дискриминација против домена примене: Лиценца не сме да забрани било коме да искористи програм у специфичном домену. На пример, не сме да се забрани да програм буде искоришћен у предузетништву или у генетским истраживањима.

Лиценца дељења: Права која обезбеђује лиценца морају да важе за све са којима се програм дели без условљавања са додатним лиценцама.

Лиценца не сме да буде специфична за производ: Права које важе за одређени програм не смеју да зависе од тога да ли је тај програм део одређене софтверске дистрибуције. Уколико је програм издвојен из дистрибуције и искоришћен или подељен у скалду са условима лиценце, све они са којима се програм поделио морају да имају иста права као она која су гарантована оригиналном софтверском дистрибуцијом.

Лиценца не сме да ограничава други софтвер: Лиценца не сме да ограничава остале софтвере који се деле заједно са лиценцираним софтвером. Ово је уведено да би се обезбедила слобода дистрибутера софтвера, као и да би се максимизовала доступност производа који усвајају лиценцу отвореног кôда.

Лиценца мора бити неутрална што се тиче технологије: Ниједна одредба лиценце не сме бити заснована на одређеној технологији или стилу интерфејса.

Овај пројекат је отвореног кôда и може се пронаћи на адреси https://github.com/thebrankoo/CryptoLab.

Као такав, његов даљи развој је планиран у складу са захтевима корисника. Такође је изузетно пожељан допринос корисника у виду пријављивања багова, имплементације нових функционалности као и исправљање постојећих багова.

Сама природа отвореног кôда дозвољава свакоме да модификује тренутну верзију и прилагођава је својим потребама. Уколико пак неко жели да та промена уђе у званичну верзију *CryptoLab*–а, мораће да испоштује одређену процедуру. Циљ процедуре је провера квалитета и коректности ажурираног кôда као и праћење имплементације нових функционалности. Следећи кораци треба да буду предузети у циљу модификације тренутне верзије *CryptoLab*–а:

1. Отворити нови *предмет* (енг. *issue)* на *Github*-у за додавање жељене функционалности или изабрати већ постојећи
2. Извршити *рачвање* репозиторијума (енг. *fork*)
3. Документовати промене и покрити их тестовима јединица кôда
4. Послати *захтев за довлачење* (енг. *pull request*)

Ова процедура обезбеђује квалитет и поузданост даљег ширења система као и детаљан увид у све његове специфичности.

8 Закључак

Криптографија је област која је увек актуелна а нарочито данас када се рачунарски системи и њихова примена све више заснивају на различитим видовима мрежне и међуплатформске комуникације. Такође се у савременом развоју софтвера траже решења која су брзо применљива и чији су резултати одмах видљиви. Имајући у виду комплексност и осетљивост криптографских алата као и већ наведене захтеве савременог развоја софтвера, јавља се потреба за решењима која ће два света спојити у један: комплексне криптографске функције представити на начин који је брз и лак за коришћење крајњем кориснику.

Овај рад је управо представио једно такво решење. Решење које лако може да нађе реалну примену и даље да се шири и обликује сходно захтевима корисника. Поред тога, сагледане су све предности и мане већ постојећих, сличних, решења тако да омотач представљен у овом раду искоришћава постојеће предности а решава акутелне проблеме.

Праћењем препоручених технолошких стандарда је обезбеђена интуитивност и једноставност употребе, употребом концепта ПОП су на ефикасан начин имплементиране карактеристике традиционалног објектно-оријентисаног програмирања, имплементацијом образаца за дизајн је обезбеђена архитектурална ефикасност и флексибилност, коришћењем *OpenSSL*—а стабилно функционално језгро а дељењем кôда даља перспектива и могућност развоја.

Литература

1. OpenSSL. Cryptography and SSL/TLS Toolkit. https://www.openssl.org/ (accessed December 2016).
2. iOS Technology Overview. About the iOS Technologies. https://developer.apple.com/library/content/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/Introduction/Introduction.html (accessed May 2017).
3. The Swift programming language. About Swift. https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language (accessed May 2017).
4. Apple Inc. *The Swift Programming Language – Swift 3 Edition.* https://itunes.apple.com/us/book/the-swift-programming-language-swift-3-1/id881256329?mt=11 (accessed May 2017).
5. Protocol-Oriented Programming in Swift. https://developer.apple.com/videos/play/wwdc2015/408/ (accessed May 2017).
6. Alfred Menezes, Paul van Oorschot, and Scott Vanstone. *Handbook of Applied Cryptography*. chapter 9, pages 321 – 333. CRC Press, 1996.
7. Alfred Menezes, Paul van Oorschot, and Scott Vanstone. *Handbook of Applied Cryptography*. chapter 7-8, pages 223 – 312. CRC Press, 1996.
8. Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography (2nd edition)*. pages 158 – 164. CRC Press, 2014.
9. Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography (2nd edition).* pages 359 – 372. CRC Press, 2014.
10. Apple Security. https://developer.apple.com/security/ (accessed May 2017).
11. Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.
12. Burt Kaliski. *PKCS #1: RSA Encryption Version 1.5*. https://tools.ietf.org/html/rfc2313 (accessed August 2017.).

13. Bruce Schneier. *Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)*. https://www.schneier.com/academic/archives/1994/09/description_of_a_new.html (accessed August 2017.)
14. Joan Daemen, Vincent Rijmen. *The Design of Rijndael, AES – The Advanced Encryption Standard*. Springer – Verlag, 2001. https://pdfs.semanticscholar.org/d440/7ce703cc42e2578a09f9352e686fc47775da.pdf (accessed August 2017.).
15. Whitfield Diffie, Martin E. Hellman. *New Directions in Cryptography*. https://ee.stanford.edu/~hellman/publications/24.pdf (accessed August 2017).
16. Open Source Initiative. https://opensource.org/ (accessed May 2017).
17. Andrew M. St. Laurent. *Understanding Open Source and Free Software Licensing*. O'Reilly, 2004.