Web application frameworks as reusable components

Prstačić, Svebor

University of Zagreb

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Svebor Prstačić

WEB APPLICATION FRAMEWORKS AS REUSABLE COMPONENTS

DOCTORAL THESIS

Zagreb, 2020

University of Zagreb

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Svebor Prstačić

WEB APPLICATION FRAMEWORKS AS REUSABLE COMPONENTS

DOCTORAL THESIS

Supervisor:

Professor Mario Žagar, PhD

Zagreb, 2020

Sveučilište u Zagrebu

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Svebor Prstačić

APLIKACIJSKI OKVIRI WEBA KAO PONOVNO ISKORISTIVE KOMPONENTE

DOKTORSKI RAD

Mentor:

Prof. dr. sc. Mario Žagar

Zagreb, 2020.

This dissertation was prepared at the University of Zagreb, Faculty of Electrical Engineering and Computing, Department of Control and Computer Engineering

Mentor:
Professor Mario Žagar, PhD

Dissertation has: 161 pages

Dissertation number: _____

ABOUT THE SUPERVISOR:

Mario Žagar was born in Kupjak (Gorski kotar) in 1952. He received his dipl.ing., M.Sc.CS and Ph.D CS degrees, all from the University of Zagreb, Faculty of Electrical Engineering and Computing (FER) in 1975, 1978 and 1985 respectively.

From January 1977 he is working at the Department of Control and Computer Engineering at FER and was since involved in scientific projects, educational activities, improvements of educational programs, laboratories and computer equipment. He was visiting researcher at the University of Rostock, Germany (1980), received British Council fellowship (UMIST – Manchester, 1983) and Fulbright fellowship (UCSB – Santa Barbara, 1983/84). In 2002 he was promoted to Tenure Professor of Computing. He participated in numerous domestic and international research projects. He was a member and project leader in five scientific projects financed by Ministry of Science, Education and Sports of the Republic of Croatia. He was also co-leader/leader in Unity Through Knowledge (UKF) project, TEMPUS project (in cooperation with the University of Paderborn, Germany), several projects in cooperation with the Mälardalen University, Västerås, Sweden. M. Žagar is author and co-author of more than a hundred articles and five books on computer architectures, distributed, ubiquitous and pervasive computing, Internet of Things (IoT), Open computing, e- learning and other Internet related technologies.

Prof. Žagar is a senior member of IEEE/CS, honorable president of Croatian Society for Open Systems, member of Croatian Academy of Technical Sciences. He participated in international conference program committees, he is a member of two journal editorial boards is a technical reviewer for various international journals. M. Žagar was awarded a Bronze plaque Josip Lončar (1975), awards INFORMATIKA93 (1993) and INFORMATIKA96 (1996) from the Croatian Informatics Society. He also received the Golden plaque Josip Lončar for the improvement of education, scientific and research work and organization of the Faculty from FER (2002), “Best educator” award, IEEE/CS Croatia Section (2006), Croatian Academy of Sciences and Arts (HAZU) award “Josip Juraj Strossmayer” for the most successful scientific work in Croatian in the field of information science (2007), for e-edition of “Unix and how to utilize it” book. He also received the “Acknowledgment for leading the development of the dynamic FERWeb application” started in 2001 and currently the central FER Web site, first prize for the best e-course at the University of Zagreb (academic year 2009/2010), Open Computing course and several other awards and acknowledgments.

O MENTORU:

Mario Žagar rođen je u Kupjaku (Gorski kotar) 1952. godine. Diplomirao je, magistrirao i doktorirao u polju računarstva na Sveučilištu u Zagrebu Fakultetu elektrotehnike i računarstva (FER), 1975., 1978. odnosno 1985. godine.

Od siječnja 1977. godine radi na Zavodu za automatiku i računalno inženjerstvo FER- a. i od tada je uključen u različite znanstvene projekte, nastavne aktivnosti, poboljšanja nastavnih programa, laboratorija i opreme. Bio je gostujući istraživač na Sveučilištu u Rostocku (Njemačka, 1980. godine), nositelj je stipendija „British Council fellowship“ (UMIST - Manchester, 1983. godine) i „Fulbright fellowship“ (UCSB - Santa Barbara, 1983./1984. godine). Godine 2002. izabran je za redovitog profesora računarstva u trajnom zvanju. Sudjelovao je u brojnim domaćim i međunarodnim istraživačkim projektima. Bio je član i voditelj pet znanstvenih projekata koje financira Ministarstvo znanosti, obrazovanja i sporta Republike Hrvatske. Bio je također suvoditelj/voditelj projekata „Unity Through Knowledge (UKF)“, „TEMPUS“ (u suradnji sa Sveučilištem Paderborn, Njemačka) te više projekata sa Sveučilištem Mälardalen, Västerås, Švedska. M. Žagar autor je i koautor više od stotinu članaka i pet knjiga iz područja računalnih arhitektura, raspodijeljenog, sveprisutnog i prožimajućeg računarstva, Interneta stvari, otvorenog računarstva, e-učenja i drugih Internetu usmjerenih tehnologija. Prof. Žagar je član senior IEEE-a i IEEE CS-a, počasni predsjednik udruge HrOpen (Hrvatske udruge za Otvorene sustave i Internet) te redoviti član Akademije tehničkih znanosti Hrvatske (HATZ). Sudjelovao je u više programskih odbora međunarodnih konferencija, član je dva uredničkа odbora i recenzent u više međunarodnih časopisa. M. Žagar primio je Brončanu plaketu Josip Lončar FER-a 1975. godine, nagrade INFORMATIKA93 i INFORMATIKA96 Hrvatske Informatičke Zajednice (HIZ), najviše priznanje Zlatnu plaketu Josip Lončar za unaprjeđenje nastave, znanstveno-istraživačkog rada i organizacije dodjeljuje FER 2002. godine, nagradu „Najbolji edukator“ Hrvatske Sekcije IEEE-a, 2006. godine, nagradu Hrvatske akademije znanosti i umjetnosti (HAZU) “Josip Juraj Strossmayer” za najuspješniji znanstveni rad u Republici Hrvatskoj u polju informacijskih znanosti, 2007. godine (za e-izdanje knjige „UNIX i kako ga iskoristiti“). Također je dobitnik „Priznanja za vođenje razvoja dinamičke aplikacije FERWeb“ koja je od 2001. godine do danas središnja stranica Weba FER-a, prvu nagradu za najbolji e-kolegij na Sveučilištu u Zagrebu za predmet Otvoreno računarstvo (akademska godina 2009./2010.) i još niz drugih nagrada i priznanja.

ACKNOWLEDGEMENTS

Before you lies a dissertation, a bit overdue in the making and almost neglected at certain periods of time since I started my PhD work.

I owe a great deal of thanks to patience, guidance and professional inspiration that my mentor **Mario** provided, almost from the first day that we met. Mario, you were crucial in finding the direction for my professional growth, as well as finding meaning to technology. Small acts of encouragement and nudges in the right direction at key points in time are things that got me where I am today, and that made it possible to finish this thesis.

To my parents, **Ivana** and **Miroslav**, who set the standard for achievement, perseverance and patience to a reasonably high level, who encouraged me, and without any doubts believed and pushed me to finish this thesis. Thank you.

I was writing this thesis with great support from my girlfriend **Ivona.** There was a period when I spent all my free time in darkened rooms, staring at the screen. Hearing a simple you got this now and then, made me really believe it.

My colleagues **Vlatka**, **Ivana**, and **Siniša**, who tirelessly read what I wrote and pointed at the weird bits, and encouraged me to fix them, thank you.

Thank you to all my other colleagues, especially **Krešimir.** I owe a great debt in knowledge and cooperation. You were always open to debate, full of encouragement, ideas, and at times, the right questions.

And also to everyone else, that listened to my, sometimes endless, meanderings about writing this PhD thesis, who directly or indirectly bolstered my resolve, thank you for listening and sharing that vision, faith, sometimes also doubt, but mostly excitement.

ABSTRACT

Component-based software engineering is one of key disciplines for efficient production of high quality software. As software industry evolves and focus moves to advances in fields like AI, big data or blockchain and crypto technologies, component-based software development remains the key method for efficient development. However pervasive component-based software engineering, there still are limitations to the way certain types of software components are built and assembled. For example, three-tier Web application components that are built adhering to a specific component model, using a certain development framework are only reusable inside other instances of applications adhering to the same component model. WordPress components are usable only inside other WordPress instances, Laravel components are unusable inside applications built using CodeIgniter or Yii frameworks etc.

These limitations stem from the fact that components are tightly coupled with the underlying development framework or platform, that is responsible for providing integration interfaces to connect multiple components. These interfaces provide required data and interface to framework's components that provide services for custom-built components to run. Through such interfaces, frameworks even provide base building blocks to build components. Many similar, but incompatible, critical building blocks are present in different development frameworks, while required component interfaces and data provided are similar. However small, differences exist and tight coupling limits reuse and portability to other components built using different component models.

Ideally, it should be possible to reuse components in whichever environment, just like an LCD display can be used in both the home monitor, car dashboard or a plane cockpit. The goal of this research is to define a technique to lessen that limitation and enable systematic component reuse outside of it's originating component model. The focus is on three-tier Web application components that include multiple code constructs, such as: models, controllers and user interfaces.

To achieve this overall goal of research (i) a model of software framework as complex reusable Web application component is defined; (ii) a method for building Web applications using heterogeneous components based on the proposed software framework model has been proposed; and (iii) a prototype of the software development framework and evaluation of the component model applicability for popular Web applications is tested and validated.

Keywords: component architectures, component model, component reusability, framework as a component

APLIKACIJSKI OKVIRI WEBA KAO PONOVNO ISKORISTIVE KOMPONENTE

Programsko inženjerstvo orijentirano na komponente (PIK) je jedno od ključnih područja računalstva, koje je omogućilo efikasnu proizvodnju sve većeg i kompleksnijeg, ali i kvalitetnog softvera. Industrija softvera napreduje i posljednjih je godina fokus sve više na novim područjima kao što su umjetna inteligencija, analiza podataka (engl. big data), kriptografija i tehnologija povezanih blokova (engl. *blockchain*), a programsko inženjerstvo orijentirano na komponente ostaje osnovni način razvoja softvera u tim područjima.

Usprkos tome, PIK donosi i ograničenje u načinu na koji je moguće graditi komponente i u načinu na koji je komponente moguće sastaviti u kompozitne komponente. Na primjer, komponente troslojne (engl. three-tier) arhitekutre za web aplikacije su uvijek građene za neki komponentni model, korištenjem nekog razvojnog okvira (engl. framework) koji određuje platfomu i tehnologiju. Komponente za sustav WordPress su iskoristive samo unutar instanci sustava WordPress, komponente za razvojni okvir Laravel samo u aplikacijama građenim pomoću razvojnog okvira Laravel. Iz perspektive razvojnih okvira CodeIgniter ili Yii, Laravel i WordPress komponente su nefunkcionalni fragmenti programskog koda.

Ova ograničenja proizlaze iz činjenice da su komponente uvijek čvrsto povezane s razvojnim okvirom pomoću kojeg su nastale, a isto tako i s platformom i tehnologijom koje su korištene za razvoj razvojnog okvira. Ta veza je ostvarena kroz sučelja komponenti pomoću kojih komuniciraj s razvojnim okvirom, osnovnim i sistemskim komponentama, te na taj način dobivaju podatke i upravljačke signale koji su portebni za njihovo izvršavanje. Mnogo je sličnih i međusobno nekompatibilnih sučelja i podataka u različitim razvojnim okvirima, koji služe sličnoj namjeni.

Na primjer, svaki razvojni okvir za razvoj web aplikacija osigurava uslužne mehanizme (*engl. utilities*) za rješavanje ponavljajućih potreba aplikacije. Između ostaloga, to su mehanizmi za upravljanje sjednicom (*engl. session*) korisnika, pohranjivanje datoteka koje korisnici presnime na poslužitelj kroz aplikaciju, pristup pojedinih komponenti relacijskoj bazi podataka; mehanizmi za povezivanje manjih komponenti na različitim slojevima na primjer, korisničko sučelje, upravljački dio i dio za upravljanje modelima podataka. Na svakom od slojeva aplikacijski okvir implementira vrlo slične mehanizme, ponekada i temeljene na istim tehnologijama. Na primjer, mehanizmi za šabloniranje definicije korisničkog sučelja (*eng.*

templating) kao što su Mustache[1], Smarty[2], Blade[3] mogu biti korišteni u različitim razvojnim okvirima, ali način na koji razvojni okvir kroz njih pruža pristup uobičajenim informacijama koje su potrebne prilikom izgradnje korisničkog sučelja je gotovo uvijek drugačiji. Svaki od mehanizama za šabloniranje pruža mogućnosti za nadogradnju, kako bi aplikacijski okvir u koji su integrirani bio u mogućnosti na jednostavan način dodatnim komponentama omogućiti izgradnju ponavljajućih dijelova sučelja. To mogu biti na primjer komponente kojima aplikacijski okvir omogućava brzo stvaranje linkova ili gumba koji pokreću neku akciju komponente kojoj pripadaju, upravljačke strukture kojima je moguće upravljati koje informacije se prikazuje svim korisnicima, a koje samo prijavljenim korisnicima ili određenim korisnicima.

Takav pristup pomaže u brzoj izgradnji sučelja komponenti za neki aplikacijski okvir, ali tako definirana sučelja nije lako preseliti u drugi razvojni okvir, bez obzira koristi li isti mehanizam za šabloniranje.

Jednaki uzorci se ponavljaju i na drugim slojevima različitih aplikacijskih okvira. Način na koji upravljačke komponente (*engl. controller*) funkcioniraju obično vrlo slične između aplikacijskih okvira, ponekad djeluju i kao da su iste – kada korisnik izvrši neku akciju kroz korisničko sučelje, na primjer klikne link, poziva se metoda upravljačke klase komponente čije je sučelje korišteno. No, način na koji potom komponenta dolazi do ulaznih informacija čije je stvaranje uzrokovao taj klik, kao što su informacije o korisniku, sjednici, razini prava korisnika itd., su specifični za svaki pojedini aplikacijski okvir.

Razlog za to je ustvari vrlo jednostavan. S vremenom se programski jezici razvijaju, a s vremenom se razvijaju i aplikacijski okviri, te nastaju novi. Ponekad s ciljem kako bi ispravili propuste ili poboljšali način na koji je neki stariji aplikacijski okvir rješavao neke od tih uobičajenih problema. No istovremeno, to je ujedno i razlog zašto komponente koje su nastale u različitim aplikacijskim okvirima nisu prenosive u druge razvojne okvire.

U idealnom slučaju bi trebalo biti moguće ponovno koristiti softverske komponente baš kao što je to moguće u drugim inženjerskim područjima, kroz standardizaciju sučelja. LCD zaslon tako može biti iskorišten u automobilu, monitoru u uredu ili avionu. Cilj istraživanja čiji je

---

[1] https://github.com/bobthecow/mustache.php

[2] https://www.smarty.net/

[3] https://github.com/jenssegers/blade

rezultat ova disertacija je definirati način na koji je moguće ublažiti negativne utjecaje postojanja različitih komponentnih modela kroz mehanizam za sistematsko ponovno korištenje komponenti i izvan njihovog komponentnog modela i razvojnog okvira. Fokus je na troslojnim komponentama za web aplikacije koje sadrže više programskih konstrukcija ili podkomponenti na sva tri sloja.

Ove probleme razvoja softvera industrija rješava arhitekturama kao što su *Arhitektura orijentirana na usluge* (*engl. Service-oriented architecture*) i mikroservisima (*engl. microservices*). U jednom i drugom slučaju je cilj zamijeniti komponente ili dijelove komponenti na nekom od nižih slojeva aplikacije web uslugama kojima je moguće pristupiti na uniforman način, putem HTTP protokola. Time je postignuta veća iskoristivost pojedinih komponenti ili funkcionalnosti koje pružaju utoliko da pojedine web usluge nisu nužno korištene samo za web aplikacije, već služe i kao komponente ili dijelovi komponenti mobilnih ili desktop aplikacija.

No, kada pričamo o web aplikacijama, primjena takvih arhitektura je dovela do povećanja kompleksnosti samih aplikacija. Testiranje pojedine web aplikacije, a time i njeno ispravno funkcioniranje ovisi o komponentama koje su potencijalno udaljene logički, fizički i tehnološki. Replikacija takve arhitekture ponekada postaje vrlo teška na laptopu razvojnog inženjera: potrebno je pokrenuti uslugu u javnom oblaku, 10-ak mikroservisa, te ponekad i nekoliko komponenti koje poslužuju i samo korisničko sučelje. Dodatno, i sama korisnička sučelja su postala kompleksnija na način da su to sada aplikacije troslojne arhitekture, često koriste MVC ili MVVM arhitekturne obrasce, pisane u JavaScript programskom jeziku.

Ovakva arhitektura je nužna za razvoj visoko dostupnih i kompleksnih modernih web aplikacija kao što su Facebook, YouTube i slične. No, većina manjih poslovnih aplikacija, te jednostavne web stranice kao što su privatne web stranice, web stranice tvrtki ili manji web dućani postaju vrlo teški i kompleksni za implementaciju. Iz tog razloga, starije monolitne arhitekturne paradigme i dalje odolijevaju, usprkos nedostacima i velikoj količini aplikacijskih okvira kako je ranije navedeno. Da monolitne arhitekture neće tako brzo nestati pokazuje i najpopularniji sustav za izgradnju stranica, je sustav otvorenog koda, monolitne arhitekture – WordPress.

Istraživanje koje je tema ove doktorske disertacije se bavi arhitekturom aplikacijskog okvira za razvoj monolitnih web aplikacija, no s ciljem da to ne bude još samo jedan okvir koji omogućava razvoj komponenti koje je teško iskoristiti izvan njegovih granica, već je cilj

istraživanja aplikacijski okvir kao ponovno iskoristiva komponenta weba. U tom slučaju aplikacijski okvir je i sam komponenta, s dobro definiranim aplikacijskim sučeljima kroz koja omogućava integraciju s proizvoljnom web aplikacijom.

Preduvjet za takvu integraciju je pak identifikacija svih uobičajenih kontekstualnih podataka koje web aplikacije prilikom izvođenja pojedine komponente stvaraju ili koriste. S obzirom na razlike strukture i moguće razlike u interpretaciji tih podataka između različitih web aplikacija, ili između različitih aplikacijskih okvira, aplikacijski okvir kao ponovno iskoristiva komponenta weba mora definirati i mehanizme za prijevod takvih podataka.

Za postizanje ciljeva istraživanja je (i) definiran model programskog okvira kao ponovno iskoristive kompleksne komponente aplikacije weba; (ii) definirana metoda izgradnje aplikacija Weba korištenjem heterogenih komponenti temeljena na predloženom modelu programskog okvira; i (iii) napravljen prototip programskog okvira i izvršena evaluacija primjenjivosti komponentnog modela na primjerima popularnih aplikacija weba.

Model programskog okvira kao ponovno iskoristive kompleksne komponente aplikacije weba

Okviri za razvoj softvera su zbirke građevnih blokova i uslužnih programa koji omogućuju izgradnju komponenata i pružaju kontekst izvršavanja tijekom izvođenja. Aplikacijski okviri su ujedno dio definicije komponentnog modela ali i primjer njegove primjene. Postoje mnogi načini na koje je aplikacijske okvire za razvoj web aplikacija moguće kategorizirati, između ostalog ih je moguće podijeliti u dvije kategorije: (i) okviri opće namjene i (ii) okviri koji su dio aplikacije koja pruža određenu osnovnu funkcionalnost, te proširivost kroz komponente.

Okvir opće namjene, npr. Laravel, je aplikacijski okvir za programski jezik PHP, koji omogućava jednostavan razvoj komponenata i pruža osnovne gradivne blokove za uobičajene funkcionalnosti: usmjeravanje; pristup bazi podataka, prijenos datoteka i upravljanje, itd. Aplikacijski okvir ne pretpostavlja i ne čini ograničenje koje će funkcionalnosti izgrađena aplikacija sadržavati. Kompleksna JavaScript aplikacija, aplikacijsko sučelje ili web usluga, blog, sustav za upravljanje sadržajem itd. Komponente izgrađene pomoću takvih okvira mogu se, uz manje ili više truda, integrirati unutar drugih aplikacija koje su izgrađene pomoću tog istog okvira. Jednostavnost s kojom se to može učiniti obično opada kako složenost komponente raste, jer je tada obično povezanost s ostalim komponentama u izvornoj aplikaciji veća.

Aplikacijski okvir koji je dio aplikacije je sličan, ima istu svrhu i obično također ima sve potrebne osnovne blokove za postizanje zajedničkih funkcionalnosti, ali je na neki način ipak restriktivniji. Omogućuje izgradnju komponenata ili modula koji proširuju osnovnu funkcionalnost aplikacije. Npr. takav okvir omogućava stvaranje nove komponente za sustav upravljanja sadržajem Drupal ili WordPress. Komponente izgrađene pomoću takvih okvira mogu se koristiti samo unutar različitih instanci te iste aplikacije. Same mogućnosti razvoja takvih komponenti su više restriktivne, ali je ponovna iskoristivost unutar drugih instanci obično lakša, jer je način proširenja osnovne funkcionalnosti aplikacije bolje definiran, tj. ograničen.

Obje ove vrste aplikacijskih okvira pružaju načine za izgradnju komponenti koje kao rezultat toga čine aplikaciju. No, jednom kada ove komponente napuste okruženje u kojem se smatraju komponentama, više ne rade. Svi uslužni programi koje okvir pruža moraju biti tamo da bi komponenta nastavila funkcionirati, a način na koji bi programeri komponenti mogli zaobići taj problem jest izgradnjom adaptera - softverskih komponenti koje prevode i prilagođavaju nekompatibilna sučelja. Ovo je vrlo složen pristup jer podrazumijeva osiguravanje da su svi podaci i drugi osnovni građevinski blokovi i osnovne komponente dostupni ili da se barem čine dostupnim komponentama – putem adaptera.

Model aplikacijskog okvira kao ponovno iskoristive složene komponente se temelji na ideji da je moguće zaobići složenost opisanu u prethodnim odlomcima iz perspektive programera svake komponente, ako je aplikacijski okvir dizajnirani i izgrađeni kao komponenta web aplikacije, dizajnirana i izrađena za ponovnu upotrebu [30]. Takav je aplikacijski okvir komponenta i ujedno univerzalni adapter za komponente koje su njegovim korištenjem izgrađene. Kao adapter, uvijek stoji između dvije komponente, bez obzira jesu li te komponente izrađene pomoću građevnih blokova koje on sam pruža, ili je njima povezana "strana" proizvoljna komponenta, proizvoljne aplikacije, tj. proizvoljna aplikacija.

Metoda izgradnje aplikacija Weba korištenjem heterogenih komponenti temeljena na predloženom modelu programskog okvira

Komponentni model zasnovan na okviru kao složenoj komponenti web aplikacije koja se može ponovno upotrijebiti (ili kraće, okvir kao komponenta - FAC) definira metodu za razvoj suvremenih komponenata web aplikacija i način njihove integracije i sastavljanja u cjelovite aplikacije. FAC model uključuje heterogeni i homogeni sastav komponenata kao i dobro

definirani ponovljivi postupak, koji je uvijek isti za bilo koje dvije komponente. Također definira razvoj troslojne komponente koji se temelji na arhitektonskom uzorku MVC-a. Budući da je sastav komponente uvijek isti, razvojni proces i značajke komponenata uvijek su primarni fokus programera, a ne planiranje i razvoj komponente posebno za ponovnu upotrebu.

Da bi to bilo moguće, svaka komponenta mora dobiti kontekst za izvođenje, kao što je na primjer trenutni korisnik, razina korisničkog odobrenja i drugi relevantni podaci od roditeljske komponente. Svaki dio konteksta, osim podataka iz nadređene komponente (koju nadređena komponenta, roditelj, sama osigurava), uvijek pruža platforma, a to su osnovni blokovi i usluge koje pruža razvojni okvir. U web aplikacijama podatke koje koristi ili stvara komponenta roditelj se definiraju kroz naziv i jedinstveni prikaz tih podataka.

Primjer kojim je to moguće objasniti: svaka web aplikacija započinje svoj kontekst izvršenja iz te dvije komponente podataka: naziv je URL, a jedinstveni prikaz podatka je stvarni URL, npr. www.fer.hr. Aplikacija koja se nalazi iza te adrese zatim odlučuje koje će komponente prikazati za navedenu adresu. Te komponente zauzvrat odlučuju koje će podređene komponente one pokrenuti, te dati isti par identifikatora: imena i referenci, temeljenih na podacima kojima one upravljaju. Iz toga je vidljivo da se komponente na neki način spajaju na par (ime podatka, jedinstvena reprezentacija podatka) koje roditelj pruža. Taj par je u istraživanju nazvan kuka (*engl. hook*).

Na primjer, ako www.fer.hr prikazuje članke vijesti i svaki od njih želi koristiti komponentu „Komentari“, par „kuka” koji bi dobila komponenta „Komentari” bi bio (Naziv: članak vijesti; stvarni ID pod kojim je članak pohranjen u relacijskoj bazi).

Stoga su kuke podataka i univerzalna sučelja za ponovnu upotrebu komponenata ključni koncepti za metodu izrade web aplikacija pomoću heterogenih komponenti na temelju predloženog modela aplikacijskog okvira.

Prototip programskog okvira i izvršena evaluacija primjenjivosti komponentnog modela na primjerima popularnih aplikacija weba

Razvijen je prototip aplikacijskog okvira weba kao ponovno iskoristive komponente, koji je iskorišten za procjenu i provjeru valjanosti modela aplikacijskog okvira. U disertaciji je kroz dvije eksperimentalne integracije prototipa FAC-a evaluirana primjenjivosti u stvarnim scenarijima izrade softvera. Kroz eksperimentalne je integracije je izgrađen softver s jasno definiranim funkcionalnostima, od kojih je dio implementiran kroz komponente izgrađene

pomoću FAC-a. Time je pokazano da se na način koji je dobro definiran i koji je moguće ponoviti za različite postojeće web aplikacije, komponente izgrađene pomoću FAC-a mogu integrirati u popularne postojeće web aplikacije koje se temelje na, spram FAC-a, heterogenim komponentnim modelima. Time se poboljšava ponovna upotrebljivost komponenti građenih FAC-om, spram komponenti koje su građene ostalim monolitnim aplikacijskim okvirima. Primjenom metrika za računanje ponovne iskoristivosti programskih komponenti je nad eksperimentalnim integracijama prototipa pokazano da je uz primjenu FAC-a moguće izgraditi zadani softver uz manje programerskog rada i manje je koraka potrebnih za integriranje i ponovnu upotrebu komponenata heterogenim komponentnih modela, te da primjena FAC-a čini ponovnu upotrebu komponenata dobro definiranim ponovljivim postupkom, a da ponovna iskoristivost svake komponente koja je izgrađena korištenjem FAC-a raste s brojem implementiranih, tj. iskorištenih komponenti koje su izgrađene FAC-om.

Ključni pojmovi: arhitekture komponenti, komponentni model, ponovna iskoristivost komponenti, razvojni okvir kao komponenta

CONTENTS

1. Introduction....1

1.1. Research questions....1

1.2. Research contribution....4

1.3. Research methodology....6

1.4. Thesis outline....8

2. Theoretical background....11

2.1. Software engineering sub-disciplines....12

2.1.1. Software requirements....12

2.1.2. Software design....12

2.1.3. Software construction....17

2.1.4. Other sub-disciplines....19

3. Component-based software engineering....21

3.1. Component models....24

3.2. Component lifecycle....27

3.3. Component construction....28

3.4. Component-based software engineering in web application development....30

3.4.1. Model-view-controller architectural pattern....32

3.4.2. An example....32

3.4.2.1. ASP.NET....34

3.4.2.2. ASP.NET MVC 3....34

3.4.2.3. Groovy on grails....35

3.4.2.4. Django framework....35

3.4.2.5. PHP Symfony framework....36

3.4.3. Common component reuse failings....36

3.5. Web application frameworks....37
3.6. Building web application components....39
3.6.1. Building a HelloWorld component using Laravel....40
3.6.2. Building a HelloWorld component using Quilt CMS....41
3.6.3. Building a HelloWorld component using WordPress....43
4. Measuring software components' reusability....45
4.1. Object-oriented software metrics....47
4.1.1. Cohesion and coupling....50
4.1.1.1. Component coupling....50
4.1.1.2. Component cohesion....52
4.2. Component-based software metrics and reusability....54
4.2.1. Measuring component reusability....55
4.2.2. Measuring reusability from complexity....56
4.2.3. Criticality metrics and implications for reusability....57
4.2.4. Reusability through properties of black-box components....58
5. Framework as a component (FAC) component model....61
5.1. Domain and premise....62
5.2. Architecture....63
5.2.1. Architecture overview....64
5.2.2. Universal adapter....66
5.2.3. Hooks....67
5.2.4. Context mappers and integration components....70
5.2.4.1. Database access and persistent storage....70
5.2.4.2. Context mapper....71
5.2.4.3. User data mapper....72

5.2.4.4. Permission mapper....74
5.2.4.5. Configuration mapper....75
5.2.4.6. Event broker....76
5.2.5. Extensions....76
5.2.5.1. The *presentation tier*....78
5.2.5.2. Application logic tier and data tier....79
5.2.5.3. Extension lifecycle....80
5.2.6. Extension manager....82
5.3. Limitations....85
6. Building software using FAC framework....89
6.1. Method for building software using framework as component model....90
6.1.1. Design principles application....92
6.1.2. Integration of FAC and host application....94
6.1.3. Building extensions....95
6.2. FAC framework implementation....97
6.2.1. FAC framework root folder and components....98
6.2.2. FAC framework folder and components....99
6.2.2.1. The framework/lib/core folder....101
6.3. Building software using FAC framework....104
6.3.1. Building software using FAC framework and WordPress....105
6.3.1.1. FAC extension HelloWorld component structure....106
6.3.1.2. WordPress theme....108
6.3.1.3. WordPress plugin....109
6.3.1.4. Implementing WordPress mappers....112
6.3.1.5. Composing software....115

6.3.2. Building with Quilt CMS....118
6.3.2.1. Implementing hooks....118
6.3.2.2. Data change events....119
6.3.2.3. Implementing Quilt mappers....121
7. Evaluation of applicability....127
7.1. Component structure and framework utilities....128
7.1.1. HelloWorld component structure comparison....128
7.2. Framework utilities....129
7.3. Evaluation of reusability....132
7.3.1. FAC component reusability....133
7.3.2. FAC and extensions' component reusability....135
7.3.3. Measuring reusability from complexity....137
7.3.4. Coupling complexity....140
7.3.5. Black-box reusability metrics....142
7.3.6. Object-oriented software metrics....142
7.3.7. Criticality....144
7.4. Performance impact of FAC....145
8. Conclusion....149
Bibliography....151
Biography....158
Životopis....160

1. INTRODUCTION

In software engineering, there is no shortage of component models. The abundant and highly competitive market of emerging and declining tools, programming languages, applications, and frameworks that define ways in which software and software components are built persistently grows and changes. Some component models are specific to a certain application domain, some have been honed to perfection for very specific purposes e.g. big data analysis, deep learning and artificial intelligence, and some are built to provide means for application development for a certain platform. Some component models and tools built around them are focused primarily on the description of components, interfaces of those components and ways in which components are assembled into composite components, whilst leaving the implementation to another set of tools or a programming languages, but most are accompanied by concrete implementations in the form of development frameworks.

The development frameworks are part of the component model – they define how to build, compose and execute components, but are also sets of building blocks to create components. Frameworks provide base components that enable developers to create components that adhere to a given set of rules – the component model. A framework and a resulting applications or platforms built on top of them make component development and reuse predictable and manageable, but only to a degree. Components built using a framework, conforming to a specific component model cannot be composed with components built using a different framework. A framework will tie a component to a certain execution environment, or computing platform: the operating system, programming language and a set of sub-components or libraries used to build components.

Software architectures like *Service-oriented architecture* (SOA) and *microservices* have been invented to lessen software and components' coupling to their platform and make them reusable through standard interfaces. In this thesis a component model is defined that makes reuse of complex components predictable, manageable and possible beyond their component model, through a defined and repeatable integration process.

1.1. Research questions

The overall goal of this research is to provide advancements in development of reusable complex three-tier web application components that include graphical user interfaces, and to

enable their efficient reuse outside of their component model – the execution environment or framework.

To achieve this, current prevailing component models and their properties and methods for building and composing components are analyzed and their common shortcomings identified. To achieve this goal, the following research questions were posed:

Research question 1: *Can a component model be defined that doesn't restrict, but rather simplifies component reuse, through a set of universal interfaces provided by the development framework, that serves as a universal component adapter?*

Making components reusable translates to making components that have simple standardized interfaces. There are two sets of component interfaces: interfaces used to communicate with the underlying platform, and interfaces used for component reuse. If every component had a single standardized set of interfaces – the one for reuse, than it would be easier to reuse that component outside of its native framework or execution environment. In fact, it would mean that a component still is a component, not just inside a strictly defined context. This is hardly achievable for complex web application components, that span multiple application layers. For example, a discussion forum built as a Laravel application, can hardly be used as a WordPress plugin. In fact, it is unreasonable to expect that building such components, or rather applications, using any development framework without them having required interfaces and being coupled to other platform components could be possible. But a software development framework may be defined as a component, that is also a universal adapter for all the components that were built on top of it. Than it is possible to consider components as having just one set of interfaces – the one for reuse. It could then be possible leverage the framework to interface to all the components in a uniform way.

The framework as a component, as a universal component adapter should then have two sets of interfaces – the one for component reuse, and the one that provides and requires interfaces for integration with the underlying application platform, or rather the component that is reusing the framework component. Once the framework integration interfaces are connected, components built on top of the said framework can be reused. And uniformly – using only the dedicated set of interfaces – the one made for simple component reuse.

In terms of existing frameworks, consider for example that Laravel framework can be used as a WordPress plugin. And that as a consequence, every application built using Laravel is a reusable WordPress module, or plugin.

Research question 2: *How can components be defined using the framework as a component component model, and how can they be built and assembled?*

If the *framework as a component* model simplifies reuse of components, how can components be developed on top of it? Would there be any limitations, and if so, would those be acceptable? If components adhering to such a component model have to be assembled, would they also be assembled through the framework as a component, the universal adapter? If not always, then when and why? Because components should be easily combined into composite components and the developer shouldn't have to feel the strain of the *framework as component.*

From the first two questions, the following (third) research question naturally arises.

Research question 3: *How can it be proved that components built are more easily reusable?*

Reusability of components is a common issue in all software engineering fields, and researchers have tried to develop various methods and metrics for measuring reusability of software components. Because of the complexity of the issue, there is no single approach that can give the answer on how reusable a component is, for example on a scale 1 – 10, but it is possible to apply various techniques to get meaningful results and indicators of component reusability in various cases.

Because of this, this research will answer whether it is possible to measure the effort required to reuse a number of components built using the framework as a component model, in unrelated applications. For example, WordPress and Laravel, compared to the effort required to write a component for one and then modify it for reuse in the second application.

In other words, is it worth integrating the framework as a component and then simply reusing all the components built on top of it, rather than building components for multiple platforms or applications?

1.2. Research contribution

Research questions, posed in the previous section, yielded the following research contribution, divided into three parts.

Research contribution 1 (RC1): *Model of software framework as complex reusable Web application component*.

Software development frameworks are collections of building blocks and supporting software utilities that enable building of components, and provide execution context during runtime. They are both a part of the definition of a component model and the implementation of it. Among many ways in which web application development frameworks can be categorized, it is also possible to assign them to two categories: (i) general-purpose frameworks and (ii) frameworks that are part of an application that provides some base functionality and extendability through components.

A general purpose framework, e. g. Laravel, is a PHP framework that enables easy development of components and provides base building blocks for common functionalities: routing; database access, file uploads and management, etc. It doesn't make a presumption what the application will be. A single page application, an API, a blog, a content management system. It can be anything. Components built using such frameworks can be integrated inside other applications that are built using the same framework. Ease with which it can be done usually drops as complexity of a component rises, and when integration of the component with other components in the original application is higher.

A framework that is a part of the application is similar, has the same purpose and usually also has all the required base building blocks to achieve common functionalities, but is more restrictive. It is there to enable building of components or modules for the base application functionality. E.g. create a new module for the Drupal content management system, or a new plugin for WordPress. Components built using such frameworks can only be used inside instances of that same application.

Both these types of frameworks provide ways to build components that as a result, make up an application. Once these components leave the environment in which they are considered components, they no longer work. All the utilities that the framework provides have to be there in order for the component to continue functioning, and the way in which component

developers could circumvent this problem is by building adapters – software components that translate and adapt incompatible component or framework interfaces. This is a very complex approach because it entails making sure all the data and other base building blocks and base components are reachable, or seem reachable, to components through the adapter.

Model of a software framework as complex reusable web application component is based on the idea that it is possible to circumvent the complexity described in previous paragraphs from the perspective of each components' developer, if development frameworks are designed and built as application components, designed and built for reuse [30]. The framework is thus a component and also a universal adapter for components that are built on top of it. It always stands in between two components, no matter if those components are built using building blocks it provides, or whether it is interfaced by a "foreign" arbitrary component of an arbitrary application. This is the first research contribution of this dissertation.

Research contribution 2 (RC2): *A method of building Web applications using heterogeneous components based on the proposed software framework model*.

The component model based on a framework as a complex reusable web application component (or shorter, framework as a component – FAC) defines a method to develop modern web application components and a way to integrate and assemble them into whole applications. FAC model includes heterogeneous and homogeneous component composition as well-defined repeatable process, that is always the same for any two components. It also defines three-tier component development that is based on MVC architectural pattern. Since component composition is always the same, the development process and component features are always the primary focus of the developer, rather than planning and development of a component specifically for reuse.

To make this possible, each component needs to receive the execution context, such as the current user, user permission level, and other relevant data from the parent component. Every part of the context, except the data from the parent component (which the parent component provides itself) is always provided by the platform, the base building blocks and services the development framework provides. A standardized format for the parent component's data is needed. In web applications hat data is defined through two components: the name and a unique representation of that data.

Every web application starts its execution context from such data, the name is *URL* and the unique representation is the actual URL, e.g. *www.fer.hr*. The application serving that URL then decides what components to render. These components in turn decide which child components to render, and can feed them the same *name-identifier* pair to reference. From that it is visible that components, in a way, hook onto these two components of the data a parent provides. These are called these *hooks*, and *hook data*.

For example, if www.fer.hr was showing news articles, and each of these wanted to use a "Comments" component, the hook data it would provide would be: *news article* – as the name and the actual id of the article.

Data hooks and universal interfaces for component reuse are thus key concepts for the method of building Web applications using heterogeneous components based on the proposed software framework model (FAC).

Research contribution 3 (RC3): *Prototype of the software development framework and evaluation of the component model applicability for popular Web applications*.

The prototype of FAC has been developed that can be used to evaluate and validate the model (RC1) and all the depending methods (RC2) and their applicability in real-world scenarios. This thesis shows that components built using FAC can be integrated into popular existing web applications, that are based on different component models. There are gains in component reusability, less development work and fewer steps required to integrate and reuse components across heterogeneous component models. The FAC approach makes reusing components a uniform process, with no need for a developer to examine a component in detail before being able to connect it to the rest of the system. As a consequence, it is possible to measure improved reusability using common software component reusability metrics.

1.3. Research methodology

Research has been conducted according to the following steps:

1. Investigation of state-of-the-art and state-of-the-practice in component-based software engineering for Web application development
2. Definition of a concrete research problem, followed by an in-depth literature review and definition of a research goal.

3. An iterative process of: (i) development of a theoretical research result, (ii) implementation of a prototype development framework adhering to the FAC component model, (iii) evaluation of the method using the prototype FAC.
4. Validation of the method using a metrics-based approach, a case-study and a set of tests.

Existing methods were investigated to develop reusable three-tier web application components, that can be used across different component models, i.e. using different development frameworks or for different applications. Is there e.g. a way to build a component that works in both WordPress and Drupal, that writes structured data into the database and references some common WordPress data and also, without separate integration modification, that same data in Drupal? This could be user details, permissions, site or page data. No such research or methods were found, and the only solution that researchers were addressing were microservices or service-oriented architectures in general. That is a completely different architecture and approach, and doesn't include reusing components that have multiple layers, and it especially doesn't include components with graphical, or any other for that matter, user interfaces. Since the approach of microservices, or service-oriented architectures, requires a complete application rewrite and redesign, and in addition implementation of components that expose microservices to the end user through graphical user interfaces, which might also need to be implemented using different component models, it doesn't satisfy the criteria that was defined for research.

There were similar approaches and techniques in other domains, devised for different purposes, application domains and always addressing more low-level components than our research is about. Most notable approaches of all are Java beans, CORBA, and Microsoft's COM. These target reuse of parts of the user interface, but the developer is supposed to connect them to other components that connect those to components that implement business logic, provide data etc. Once all this is connected, simple reuse is again impossible.

Consequently, research was approached in the following way: investigate the state-of-art approaches (i); based on existing methods and solutions define the research goal and problem that would be based on recent advances and methods of component-based software engineering (ii); and go through the iterative process of developing, prototyping tools and models, evaluation (iii) and in the end testing their applicability in real-world applications

through a case study. The real-world case study was done using the actual instance of FAC framework prototype and a number of components. The FAC prototype was with different host applications, thus testing functional correctness of extensions and data integrity of data that those extensions manipulate. Finally, component reusability was measured through the application of known software and component metrics (iv).

1.4. Thesis outline

This thesis is divided into 8 chapters. The first chapter introduces the PhD topic, research methodology, research questions and the research contribution.

Chapter 2, the theoretical background on software engineering is introduced, which provides one of two pillars which will be used to define the method of building software using FAC (RC2).

Chapter 3, component-based software engineering is introduced. This is a more in-depth look at how software can be built by applying component-based methods. This is another pillar for defining the method of building software using FAC (RC3), and building the prototype of FAC (RC3).

Theoretical background on component models is also introduced in this chapter, contributing to body of knowledge required to define the Model of software framework as complex reusable Web application component (RC1).

Chapter 4 introduces software and component and component reusability metrics and concepts, that will make it possible to evaluate component reusability that results from the application of FAC to composing components and building software.

Chapter 5 introduces the Model of software framework as complex reusable Web application component (RC1) in detail.

Chapter 6 goes into detail on the implementation of the prototype of the software development framework and evaluation of the component model applicability for popular Web applications (RC3).

It also describes in detail how software can be built, through two example implementations of two different host applications. Implementations then yield a method of building Web

applications using heterogeneous components based on the proposed software framework model (RC2).

Chapter 7 applies the software and component reusability metrics to evaluate the prototype and method for building software using FAC.

2. THEORETICAL BACKGROUND

Component based software engineering practices emerged in the 1960's. The decade when the boundaries between software and hardware were getting more and more visible and with rising complexity of both software and hardware, the need for a more efficient and formal development disciplines became unavoidable and needed. It is in this decade that NATO science committee organized the first conference on software engineering in Germany, to discuss the challenges of creating more complex and reliable software.

It is in 1968 that Doug McIlroy, then head of the Bell Labs Computing Sciences Research Center, at the NATO conference, said: "*My thesis is that the software industry is weakly founded, and that weakness is the absence of a software components sub-industry*.". It wasn't by accident that McIlroy was one of the authors of the first Unix operating systems, and the inventor of the Unix pipes, a system that made it possible to use Unix command-line programs as components that could exchange text data and through that chaining, work as a system of reusable components.

Today, software engineering is a complex field, that is comprised of many sub-disciplines [1]. One of the main goals of software engineering is to reduce complexity of the real world, employ computers to work for us. Since human beings are not good at holding large amounts of data in memory, especially complex data in short-term working memory, reduction of complexity is also the goal that spans all the sub-disciplines of software engineering. Software is created only by reducing complexity and dividing a complex real-world problem into smaller manageable problems engineers can tackle. Sub-disciplines of software engineering that are employed, are analyzed in great detail in a book IEEE Software Engineering Body of Knowledge [46]. There are 15 chapters, each focusing on one of the sub-disciplines. While each is equally important, only a few are shortly introduced, directly relevant for this dissertation topic, especially as an introduction to methods of building software using FAC, in chapter 6.

2.1. Software engineering sub-disciplines

2.1.1. Software requirements

Every software development project begins with **software requirements** definition. In this first phase, software engineers, among others, try to define user stories, functional, non-functional and other types of requirements for the software, so that it will fulfill a real-world requirement, or solve a real-world problem.

Software requirements are changed and updated as the software is built, so it is a process that lasts throughout the whole lifecycle of the software. Although the term is "Software requirements", it often encapsulates other requirements tied to the software or the software development or management process.

Non-functional properties may implicitly describe how a low-level feature should be built, e.g. visual feedback to the user should be shorter than one second. This would probably cause many design decisions and technical solutions. To reiterate the continuity of software requirements process, such a requirement may be proven very difficult to achieve, and might even be changed in the future.

Product or process requirements might also be placed upon the software. For example, it might be required that the development team develops software using the SCRUM methodology. Other process or even documentation and administration requirements might be placed upon the software if for example specific funding by the government is used. Another product requirement might be "Users will only be able to log-in using Open ID protocol".

Some requirements might emerge from other integrated software, or simple truths about the real-world. For example, a user interface on the phone must be adapted to small screens, since phones are – small.

2.1.2. Software design

Software design are software lifecycle activities, that transform software requirements into software blueprints, that developers will build. Software design is done by software architects – software engineers that define the architecture of the whole software system. This usually entails identifying components that can be reused, programs and development frameworks that will be used, operating systems and platforms that will run the resulting application. Both

functional and non-functional requirements influence the resulting architecture. Naturally, the more existing software is identified that can be used to achieve the desired requirements, the shorter and simpler consequent phases will be. This, more broad approach, to software design is called *software architectural design*.

Software design activities can also be more detailed, focused on defining how smaller parts of the software will work. For example, how a component architecture would look internally, what algorithms will be used for component methods, which interfaces will be built etc. These are called *software detailed design*, and the goal is to simplify programming of each part or component of the software.

One of key issues that software design deals with is packaging, decomposition and organization of software components. In other words, software design always entails, among concerns like performance, security, reliability, usability..., the component model for the software. Engineers thus follow the seven software design principles:

- **Abstraction** is a process through which only relevant properties of an object are identified for observation and analysis. There are two key concepts for abstraction in software design. The first is by parametrization, which means defining names of parameters and data types that a function takes, and the second is by abstraction by specification. Abstraction by specification means defining what a part of code or a module does. It can entail outlining functions in pseudocode, sketching-up classes or objects in UML, or defining what data will be collected and what properties data entities might need to have. Abstraction by specification is thus divided into: procedural abstraction, data abstraction and control (iteration) abstraction. Through abstraction, only the most relevant information that servers the purpose of the object is extracted, while the rest is ignored.;
- **Coupling and cohesion** – planning for low coupling between modules, and high cohesion of modules will greatly improve the architecture of future software. The higher cohesion, and the lower coupling is, the more reusable modules (software components) will be. Parts of the software, its modules or the system, will more easily be replaced or changed in future software construction activities. A more detailed overview what coupling and cohesion are and how they affect software is given in section 4.1.1.;

- **Decomposition and modularization** are concepts tied to coupling and cohesion concerns in design activities, the aim of decomposition and modularization is to separate functionalities into separate modules or components of the software, plan their interactions and interfaces for communication.;
- **Encapsulation and information hiding** – just like classes hide their internal functioning, through private properties, on a grander scale the goal of encapsulation and information hiding is to make sure all module or component internals are hidden from other parts of software. Or rather, it should be exposed only in a certain way – through component interfaces or functions.
- **Separation of interface and implementation** – tied to encapsulation, separation of interface and implementation makes sure users of a component aren't exposed to its internal specifics. The public interface definition is thus a separate design concern, from encapsulation.
- **Sufficiency, completeness and primitiveness** – software, and components that will be built will be sufficient and complete if all the needed abstractions are defined, for functional requirements that are defined. Primitiveness is satisfied if design is based on known design patterns, that are easy to implement.
- **Separation of concerns** – just like concerns are separated in software implementation, they should be separated when designing software architecture [47]. The most common is the 4+1 architectural view model, defined by P. Kruchten [48]. The system should be designed with 4+1 use cases (views) in mind: *development view* – the blueprint for the developer, like UML of components, classes, packages; *logical view* – descriptions of functionalities for end-users, description of how the software achieves functional requirements; *physical view* – the topological view of components, connections between components; *process view* – describes processes of parts of software, how they communicate; *scenarios* – examples of how parts of the system will work and communicate for certain test use cases.

There are key issues, that software design must address. At first, a good software design must encompass both the technical details and visuals of software. From the technical point of view, it is important for software to have a satisfying performance, that it scales when needed. It should also be secure and reliable. The usability aspect can be observed through all the said

properties, with the addition of intuitive graphical (or other) user interfaces. One of key issues software design must deal with is decomposition, organization and packaging of software components.

Other key issues that software has to deal with, or problems it needs to solve, don't directly contribute to the software problem domain, but rather the properties of the system and functioning of the software. These issues are called software design aspects [2], and are are equally important to domain problems. Those are usually properties that affect the performance or semantics of components in systemic ways.

Software design aspects

Concurrency – designing software by decomposing it into threads, processes, tasks, and dealing with synchronization, scheduling etc.

Control and handling of events – designing software so that it has efficient data flow, and reactivity to data changes and other occurrences in the system.

Data persistence is another important design aspect, for an application that doesn't efficiently handle storing the data that the user creates will hardly be of any use.

Distribution of components also needs to be well designed so that software components that reside on different hardware, use networks to communicate, do so efficiently.

Error and exception handling is another systemic aspect of software design. Software should always work in exceptional conditions and, it should report those conditions and eventual errors, so that they can be fixed.

Designing ways in which software interactions for users are programmed, the way these concerns are separated from the actual implementations, is an **aspect of interaction and presentation**. This is not the aspect of specifying the actual user interface, but rather patterns to implement it, such as Model-View-Controller approach.

Structure and architecture

After aspects are defined, they will directly and indirectly influence structure of the software, which is "the set of structures needed to reason about the system, which comprise software elements" [68]. In essence, software structure is part of design that specifies abstractions. Some structural abstractions are made to deal with architectural design, like architectural

styles, and some with detailed design like design patterns, for example MVC, architectural viewpoints. To be able to reason about structure and architecture of FAC, key architectural concepts are introduced.

Architectural structures and viewpoints

There are different viewpoints which can be used to reason about design and structure of a software system – behavioral, functional, structural or data modeling. One doesn't exclude the other, and many can be used in parallel to reason and design the software. For example, structure of modules, mapped to behaviors with data models.

Architectural styles

Architectural styles define specialization of elements and relations, and constraints on how they can be used. Architectural styles define high-level properties of software, for example, software layers, distributed systems, interactive systems, interpreted systems etc.

Architectural styles can also be mixed, for example, web applications are distributed systems, that are interactive though user interfaces, whose structure can be split to at least three layers.

Design patterns

Design patterns are common solutions to common repeating problems for various architectural styles. For example, it is possible to apply the MVC pattern to achieve interactivity for software.

Architecture design decisions

During software design, which is a creative process, software designers have to make decisions that influence many properties of software and the software development process itself. Thus, it is important to view the software design from the decision making perspective, not just as an activity of software engineering.

Families of programs and frameworks

One of key results of architectural design are families of programs or frameworks – software whose components may be ported and integrated into different separate instances, for Example, WordPress websites with different plugins installed. This is also one of main

methods of building software by component reuse, and one of main limitations for reusability of components, one that FAC is designed to better.

2.1.3. Software construction

Software construction is linked to all other activities and phases of the software lifecycle, that are a part of all other software engineering sub-disciplines. But, most tightly, software construction is linked to software design and software testing. In essence, software design is the input for software construction activities, which create input for software testing activities.

The most important deliverable of software construction is code – software constructs such as files, software components and implementation of architectures, but is not limited to it. Documentation and test cases are also outputs of software construction.

Software construction includes both constructing software from existing components, and developing new software components, or other smaller, simpler code constructs, that can be integrated to provide a larger whole. The more detailed software design, the simpler the task of software developers becomes, that transform design into working software more quickly. The more roughly defined the design, the more additional software abstractions will be created during software construction, and the more additional documentation will be created.

When constructing from existing components, software developers sometimes create additional components, called adapters, to simplify integration or interaction of two components, or to satisfy functional or non-functional requirements.

Software project management is also related to software construction activities, and in practice has proven to present considerable challenges [3].

Software construction fundamentals can be divided into:

- Minimizing complexity,
- anticipating change,
- constructing for verification,
- reuse
- and standards in construction.

Minimizing complexity is fundamental to software construction because humans cannot hold large complex data structures and algorithms in mind, or computer terms, working memory. So, engineers create abstractions that will divide the problem into smaller programmatic entities to minimize complexity,

Software changes with time, and anticipating those changes will influence construction decisions. Software components help with building software with anticipated change, because a component-based architecture enables easier replacement of parts of software.

Which also helps with **constructing software for reuse**. Building software that is reusable will speed-up all future software construction activities that can benefit from available reusable components. Systemic reuse can also considerably raise software quality, productivity and speed of construction [3], [30]. **Constructing software with reuse**, or in some literature referred to as build by reuse, is the process of analyzing software design, and achieving software purpose through choosing and composing existing components, described in more detail in chapter 3.

Standards in construction make it possible to achieve software construction project's objectives, like cost, efficiency, quality, code readability and reliability etc. Examples of standards in construction are:

- communication methods – standardized documents, decision workflows, communication channels…,
- programming languages,
- coding standards – file, variable, class… naming conventions, indentation and code formatting…,
- platforms,
- tools.

Standards can be external or internal to the development team, institution or company. External standards are most often industry or open standards that are applied to certain aspects of technology, for example ISO standards, XML, SQL, etc.

2.1.4. Other sub-disciplines

Once a complex system is deployed, it is usually configured in a specific way – the platform, the operating system, the various integrated components. Every single property, and why it is set has to be documented – why and when was it set. Also, every change has to be documented. This is a Sub-discipline of software engineering, called **Software configuration management**. Purpose of software configuration management is to understand how a software, after it has been deployed, changed with time, and how changes were made through configuration through time.

Other software engineering sub-disciplines, that deal with software engineering economics, software development teams' organization, computer science and mathematics theory, business and engineering process management, etc. will not be considered or applied in this PhD, thus will not be extensively introduced here. Though, it is important to note that in real-life scenarios, when producing software, one has to consider each and every of those sub-disciplines if successful software production is to be expected.

3. COMPONENT-BASED SOFTWARE ENGINEERING

Software is inherently complex [44]. Software engineers are in business of converting ideas, concepts and processes into working abstractions presented by algorithms and abstracted data structures. This means that there is often a large gap of understanding between humans that directly or indirectly define software requirements and humans that understand computers and translate software requirements into code. Abstractions, component models, underlying architectures and platforms are almost exclusively something a software engineer will understand. The effect that this has on software development processes, quality and reliability, and software development projects is hindering. Non-technical managers have a hard time understanding how long, why and what it takes to build software. On the other hand, software engineers often have a hard time realistically predicting the complexity and time it will take to achieve the desired software functionality – to create the product. In general, the complexity of abstractions of real-world ideas into software is the main reason why many projects fail, or considerably break deadlines [44], [45].

The industry thus came up with the use of software components, and ideas like *commercial off the shelf* (COTS) components, to solve these problems. Well tested components, used in multiple products, well documented and with predictable behavior provide a well rounded set of properties that both software engineers and businesses value. Splitting functionalities, abstractions, processes into pieces that are manageable, interchangeable and not too complex on their own makes the process of software development more measurable in both time and effort, and makes it possible for multiple engineers to work on different parts of the system simultaneously. It also makes replacement, development or upgrade of existing components, or parts of a software system, easier. This means that development and maintenance can be better predicted, planned, and executed, and it also makes it possible to automate tests of separate components and the whole system.

Considering other engineering disciplines, like construction or vehicle manufacturing, in contrast to software engineering, many of the pieces of the final product are visible and intuitively recognizable to users, non-engineers. Consider a bridge. A bridge will not be secure or usable without solid foundation and strong columns that support it. Every component of the bridge is a solid object, built using widely available materials and

components, intuitively understandable without the need to understand low-level technical intricacies like e.g. material elasticity, load limits or physics and material properties needed to build the structure. In a way, this simplicity and intuitiveness of reuse should be the goal of component-based software engineering. And yet, after more than 50 years of software development, the industry is not ready to provide software construction with the same level of ease of component composition and reuse as competing engineering fields. Again, a much higher level of abstraction than that of competing engineering fields is the culprit.

To define a software component, one has to consider the platform and the application domain in which the component exists [6]. Overall performance of the system is influenced by individual performance of its components. If one component fails, the whole system can fail. Therefore, one has to consider all the functional and non-functional requirements of the system when designing and building each component. Component definition also stems from the domain in which it is to be used. It also depends on the current development context, i.e. it depends on the actual work a certain software engineer is doing at a certain time: when creating e.g. a development framework, one has to think very low-level. In that case, components might be classes, namespaces, functions or other data or code structures, even variables. When developing a user-oriented Web application, components are almost always defined as more complex code constructs, containing multiple classes, namespaces, spanning different layers of code of the application [18], [19], [20], [21], [23], [24], because many of the low-level concerns are already addressed by the underlying framework.

Certain application domains, like web applications which is the focus of this dissertation, have the same or similar definition of what a component is. Referenced literature shows that Web applications are most commonly constructed from components built using a model-view-controller [24] design pattern, which enables separation of code that handles user interface and presentation to the user, from code that processes user actions and executes those actions on models that manipulate actual data stored in databases.

An example might be log-in component in a content management system. A log-in component would include a user interface layer that provides username and password entry, a controller that would receive data entered by the user and then execute the required actions on models. In this case, models would e.g. represent a user, user’s roles etc. If the data was retrieved from

the database, it would usually be done through a database access layer (or sometimes called database middleware [31]) - Illustration 1.

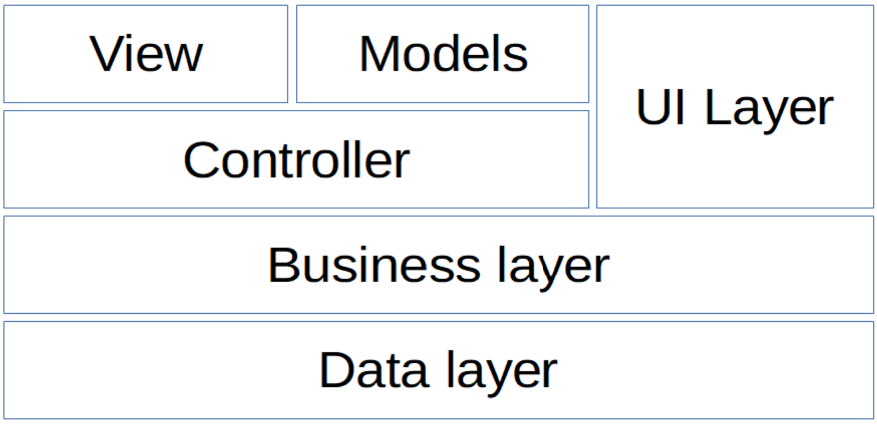

Illustration 1: Model-View-Controller

MVC pattern is additionally described in section 3.4.1. Here, it is only important to note that each of the layers consists of a set of components that make up a single complex (MVC) web application component. A view consists of many smaller UI components, such as buttons, links, forms, etc. In the same way, a controller might employ an array of smaller components or libraries to process user input before it is translated into actions on models or one of the models or components of the business layer. Illustration 1 shows that models are part of the UI layer. But the model is also a part of the business logic, so models are also contained within the business layer. Each MVC component may have its own parts in the business layer, or use other available services or components from it, that persist the data to the database, or some other data storage (the data layer).

It is clear that MVC components are complex code constructs, made up of many smaller components spread out through different layers. Complex components like these can be perceived as applications themselves [13]. Components like these can only run when used in an application environment, a stack of software (a platform), family of software that they were built for [2], [12], [18], [19], [20]...

Rules that define what is required and provided, how the component it is executed, how the underlying platform is constructed and how it runs, how components are assembled and built are part of a component model. The following section provides a more detailed overview of component models.

3.1. Component models

The software engineering industry and community has developed and employed many different component models over the last decade. Each of the models in some way improves our ability to use software components effectively, and does it in a different way – by trying to provide a different, and hopefully better definition of what a component is, and isn't, specific to a certain application domain, requirements of that domain, requirements or properties of the system, specific to a certain technological platform. In the literature component models are often described in a similar way, as a set of rules, standards and conventions for developing or designing software using components and components themselves, e.g. [7]: "*A component model defines a set of standards for component implementation, naming, interoperability, customization, composition, evolution and deployment*".

Extensive research on component models and their classification has been done by I. Crnković et al. [11]. Authors have tried to classify different component models through enumeration of concepts employed. Different models prioritize various aspects of software and component lifecycle, component composition and construction methodologies. Components and component systems that are built by component composition, conform to the component model. There are exceptions, sometimes composite components don't conform to the component model, i.e., two components can be composed using standard provided interfaces defined by the model, but a composed component no longer provides those same interfaces, thus, doesn't conform to the component model.

Some components models, especially the ones that are technology-specific like Java Beans or DCOM, define both the platform which executes the components, and provides both conventions and building blocks (development frameworks) of components.

Others component models focus on the design phase of the software lifecycle, like UML [37]. Such component models are platform agnostic, bat can provide tools to transform designed components into executable software entities.

Although different, definition of properties of components and the component-based system is pervasive to all component models. This defines various functional and extra-functional properties of components, how components bind and interact among themselves, how they

interact with the platform, how they can be assembled and composed into a system or composite components, sometimes in the literature also called assemblies.

Component models also define which interfaces a component should have so that other components and the platform can communicate with it. When a component interface is connected to another component's interface, the components are bound, but it isn't possible to just connect any two components using two interfaces. Generally, there are two types of interfaces: required and provided interfaces. Provided interfaces most often depend on the component design and specification, while required interfaces are interfaces through which a component receives data or signals from the platform. For example, a greatly simplified analogy for a provided interface would be the "accelerate" interface of a car engine, while required interfaces would be the gasoline and air input pipes.

One of the best ways to visually represent components is UML. Although it can also be considered a component model focusing on the design phase of the component lifecycle it is also for practical purposes a design tool. In UML notation, a component is shown by a rectangle. Required interfaces are represented by open receptacles – halves of circle, and a single provided interface is represented by a full circle. From the example shown in Illustration 2, it is intuitively understandable how these two can be used to represent connected interfaces – the circle goes inside the receptacle. It also intuitively signals the viewer that an empty receptacle means the component will not work unless connected.

Illustration 2: Simple component representation in UML

This representation does not show other properties or internals of the component, but they can be ignored for general design-time considerations, that UML is used.

So finally, to formally define a software component model, one has to define a component. A component is represented with a letter **C**. **C** is an entity consisting of two sets, properties **P** and interfaces **I**:

C = <**I**, **P**>, where I = {i, i2, i3,… }and P = {p1, p2, p3, … }.

Depending on the component model, these can be interfaces and properties of a class, multiple classes, or collections of various code constructs. Whatever the case, the definition stands. Only the methods to determine a component's properties or identifying what interfaces of the component are would change according to the component model.

A component model also includes a description of the platform, or in case of technology-specific models, provides the platform for component execution and also provides methodologies and interfaces for component binding – for both inter-component binding and component-platform binding. Such a union is called a component-based system, and formally I. Crnković et al. [11] define it as:

CBS = < **P**, **C**, **B** >

Where P is a system platform; C is a set of components and B is a set of bindings, visualized in Illustration 3 ([11]). An important property of the component-based system that this definition implies, is that there is no component-based system without a platform. In essence, this means that a component-based system does not exist without the platform, and that all the components don't exist without the platform. The platform being a specific family of applications that employ their own component model, or applications that are built using a certain technology, like Oracle's Java Beans that will execute on the Java platform, or Microsoft's ASP.NET that will execute on the Internet Information Services Web server, loaded up with the .NET framework, etc. That is the exact reason why so many component models exist today, and since the definition depends on the platform, why there are many definitions of what a component is and how bindings between the components are made.

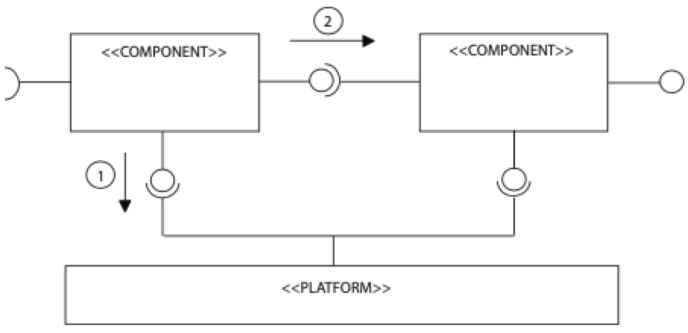

Illustration 3: Component-based system

3.2. Component lifecycle

A component lifecycle resembles that of a typical software product lifecycle, shown on Illustration 4.

A component lifecycle can be viewed as a set of tools and services that a component model provides to various component-focused activities during the lifecycle of a component [33].

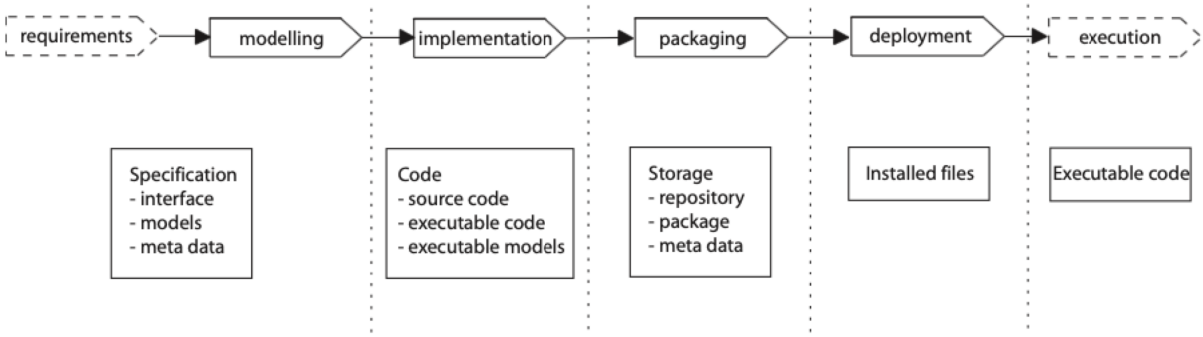

Illustration 4: Component lifecycle [11]

The modeling stage

During the modeling stage a component is described, a blueprint documentation is made that a component developer will transform into code. Models are used for either architectural description of systems, or for defining state transitions, functional aspects of the system or component.

Implementation stage

Component models may also be responsible for supporting production of code. In most cases, this is done through frameworks built as a part of a component model description. In the implementation stage, designed abstractions and desired functionality, states and transitions are transformed into code constructs such as classes, functions, libraries etc.

Packaging stage

Packaging a component transforms it into a portable format – that can be copied and deployed into another instance of the compatible system. This is usually done by creating compressed archives or folders that contain all the component code and data. Depending on the technology, code may be compiled. Usually, components built using frameworks in

interpreted programming languages contain source code, while components of programming languages that have to be compiled are packaged compiled.

Deployment stage is the moment in which a packaged component is copied to a system that will execute it.

3.3. Component construction

Component construction activities are not much different from software construction. Indeed, building a component is building a reusable software entity. In fact all the activities that lead up to construction, introduced in section 2.1, should be equally performed. The difference to software construction in general is tied to the nature of components, and it is only in the focus on building for reuse, building software that conforms to a certain component model.

The rest of software construction fundamentals are the same: minimizing complexity, anticipating change, constructing for verification and standards in construction.

To minimize complexity, components can be divided into sub-components. Well designed and constructed components will also be configurable. The ability to configure and replace sub-components will contribute to changeability of components.

Standards in construction are essential to building components effectively, since they usually entail standards that contribute to practice that directly contributes or enables reusability. These can be standard platform, programming language, coding standards, design and architectural pattern application etc. [33]. Conforming to a standard architecture of the component model, having clear purpose of the component defined, with interfaces and data types that contribute to that purpose.

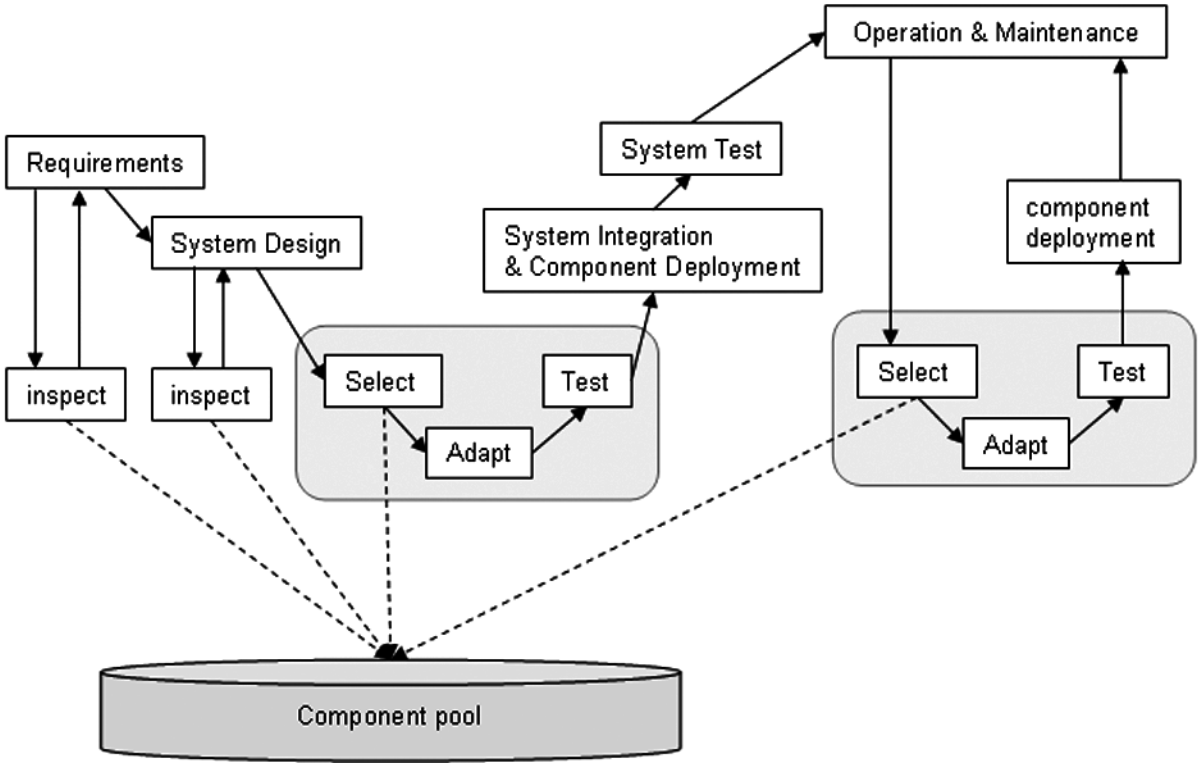

Illustration 5: V development process [33]

Once the component itself is built, construction activities will have yielded additional **extra-functional properties** of the component. These are properties of a component that contributes to its description, such as performance, complexity, capacity etc. These, are usually achieved during testing and will, along with the documentation describing functional properties of the component, be of essential value for the builders of the system that have to reuse the component.

The V process says that during creation of requirements for a specific software and during the design phase, engineers should inspect available components, and choose which ones should be reused. Extra-functional properties are of great value in this phase. Component construction can be a parallel or preceding activity to system construction. The most usual way to describe system and component construction is the v development process model [33], illustration 5.

The ones that are selected should then be adapted (if possible), and tested if they indeed satisfy the requirements of the system. If there are no available components that satisfy the requirements, then new ones should be built.

Once the components are integrated, the whole system should be tested if requirements are met. Once the result is satisfactory, the built software system will become operational.

During software system operation, requirements might be changed, components may prove unsatisfactory, and revision of system architecture, or replacement of components may be required. The process of finding or building suitable components is repeated in the same ways as when the software system was being built. This is also visualized in illustration 5.

3.4. Component-based software engineering in web application development

By analyzing how websites are built, especially using open source tools and content management systems, since those systems and component repositories are available for inspection online, there are examples of components that had to be developed multiple times, or integrated into multiple systems. An example might be CiviCRM[4]. Analysis of CiviCRM documentation shows separate adapter components have been built for different systems [26]. The alternative to that is building separate components for different component models, or different software families, that implement a solution's domain functionality that has already been created or modified at least once for a different component model, software family, system or platform.

The simplest examples of this can be found if one takes time to explore the Web, searching most popular Web search providers with for example, a term "Photo gallery", followed by a name of a web application development framework will always yield many results, linking to various components or packages of components that should provide the searched for functionality. Of course, there are dedicated Web applications that may provide that same functionality, and these applications often provide various ways and levels of integration, for example, Flickr[5], Google photos[6], or open source solutions like the Gallery project[7].

[4]CiviCRM, an open source customer relationship management web application for non-profits

[5]www.flickr.com, online photo sharing platform by Yahoo!

[6]photos.google.com, online photo sharing platform by Google

[7]galleryproject.org, open source self-hosted photo sharing web application

Opposed to the component-based approach, dedicated applications are completely decoupled from the rest of the applications or services a company or individual might use. That is a good thing as long as the gallery application has mature APIs that encapsulate all the functionalities. This approach is the basis of Service-oriented architecture (SOA). SOA isn't about giving up on using software components, it is rather a way in which software engineers were able to standardize ways of communication between services – components in SOA. However, building everything as services is not the right approach. Using services has more processing and communication overhead than using components. Services usually communicate over the Internet using a HTTP protocol, are stateless just like Web applications, and thus need to authenticate clients on each request. This makes them inappropriate for rendering parts of a Web page, which is usually done by software components.

Additionally, let's consider an arbitrary applications that needs to communicate with the gallery. Achieving this communication and full integration includes use of multiple additional components that provide various integration points, checking user authentication, user permissions, quotas, photo and gallery creation, photo and gallery search and retrieval. For the sake of the example *Flickr* is considered as the gallery application with which an arbitrary web application is integrated. *Flickr* provides various integration possibilities through the use of APIs that are accessible as web services. To integrate it in an arbitrary application, integration components have to be implemented in that specific application. So, an arbitrary web application has to handle and mediate user requests to *Flickr*. This means a complete user interface has to be created along with components that connect to *Flickr*.

It is possible to group all the components: classes or functions that are responsible for communication with *Flickr* into a library and use it in other applications one might want to integrate with *Flickr*. But components would have to be created for each new family of applications, or rather for each component model a certain application uses. It would be required to create a user interface, a controller and integrate and reuse the *Flickr* library. When built, all these components will provide the same basic functionality, yet they will be incompatible and different in order to conform to different component models and architectural requirements of their respective platforms. The component-based software engineering approach would ideally mean the integration component is created just once and then used everywhere, without repeating the work that had already been done for some other application or component model.

3.4.1. Model-view-controller architectural pattern

Today, modern web applications are built using many different frameworks, that implement similar component models, based on the model-view-controller (MVC) architectural pattern [24] for implementation of domain components. In MVC architecture, domain components are complex software entities, consisting of various code constructs like classes, functions, user interface templates. All the code that contributes to a component belongs to one of the three layers: model, view or controller.

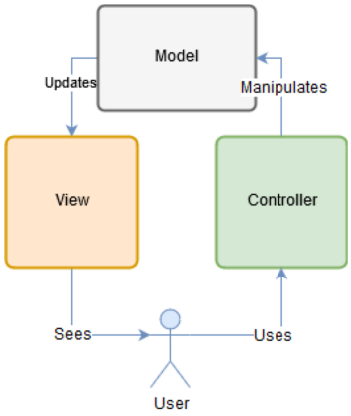

Illustration 6: MVC

MVC pattern defines the way in which the component's code is structured and it promotes separation of concerns in code: data retrieval and manipulation; user input processing; data and user interface visualization.

The user controls the application, or rather a certain component, through the *view* that implicitly exposes the *controller* of the component, for the data that is visible to the user. This data is usually a *model* that has been exposed through the view.

So, the view is populated by the data of the model that the controller and requested. The model itself usually accesses the data through a database abstraction layer, that can sometimes even create entity class instances – models. While the model instances usually represent database entities they also include business logic and methods that simplify the task of manipulating models for the Controller.

Once the data is retrieved and model instances created, the controller then passes the data on to the view which again exposes it to the user, possibly in the form of controls that could again trigger new actions by the controller. The controller always includes a default method, or action, that fires when there is no explicit user action. In these cases it retrieves a default data-set of model instances and presents them to the user through the default view. An example of this would be loading up a title page of a web application.

3.4.2. An example

To further elaborate on the component-based software engineering for the web, architecture of what an architecture of a photo gallery application might look like is inspected. Let us suppose that the application requirements are as follows:

The user should be able to log in, and when logged in she should be able to create photo galleries and upload photos to them. It should be possible to upload an infinite number of galleries. When she is not logged-in, she should be able to view the galleries.

The title page of the application should show the list of galleries. When the user clicks a certain gallery, another page should load, showing thumbnails of photos in the gallery. If the user clicks a photo, it should be loaded by the web browser.

Let us suppose the application was implemented using one of the popular Web application frameworks, for example Laravel. Just like many other Web development frameworks [14], components in Laravel are built using a model-view-controller architectural pattern.

To build it, three **models have to be defined**, the user, gallery and photo. Each of those representing a relational entity, a row in the database table. User can be the owner (creator) of the gallery and the owner (uploader) of the photo in a gallery.

After the models have been defined, and the underlying database created, one would continue with **setting-up routing**. The main route "/" would be setup, to load a view that lists all the galleries. In Laravel, that is done by specifying which controller and controller action to invoke. The controller then loads up the models, defines which view to render, and feeds the data (models) to the view.

The **view** would then just iterate through the received data and render the list of galleries and create links to the route that shows the photos of the gallery, for example "/gallery/1". Where one is the unique identifier that a particular library would have assigned by the database. That route would again specify the controller action, and so on.

This principle and architecture repeats throughout all the popular web application frameworks available today [14], for example: ASP.NET and ASP.NET MVC 3 [29], Groovy on Grails [20], Django Framework (Python) [21], Symfony framework …

The following sections will briefly introduce the frameworks and component models that they define. These frameworks employ the MVC architectural pattern. This shows that the architecture and implementation of the example would be done in a similar way.

3.4.2.1. ASP.NET

ASP.NET provides a few different component models to build Web application components, one of them is Web forms and ASP pages, that are a paradigm to separate application presentation from application logic similarly to MVC [24], [28], and make Web application development similar to the event based desktop Windows programming.

ASP.NET and Web forms provide a fast and simple way to reuse components of varying complexity: data access, user interface components, common application logic and combinations of those. However, the is no way to easily integrate architectural unit type of components. For example, a more complex application, such as .NET Nuke CMS, provides its own array of interfaces to create architectural unit type of application components [29]. Components for such applications are not easily reusable inside other applications, regardless being built with the same underlying framework. Additionally, more problems can arise if a component is built using another component model than the application uses, for example ASP.NET MVC (explained in the following section). Integrating such a component would require additional programming work to both the application which would reuse it, and the component itself.

3.4.2.2. ASP.NET MVC 3

ASP.NET MVC 3 [12], [16] is the newest component model available in ASP.NET. It provides an alternative way to define Web application components: a “Razor” template engine, which strongly resembles Smarty [13], for PHP, both in syntax and the way it is used. Razor also provides “HTML helpers” which are a way to provide functionality that Smarty plugins [13] provide. Consequently, various components can be nested inside each other so that they combine various views of various models. For each reuse case however, one has to know what parameters a component expects, or in other words, a component can define an arbitrary required interface. This provides flexibility when developing components, but also has adverse side effects to component reuse.. For example, reusing components might have to do the work of preparing data that a reused component expects, or a reused component could be implemented in a way to integrate with a specific reusing component, making the reused component hardly reusable for other arbitrary components.

3.4.2.3. Groovy on grails

Groovy on grails [20] is a modern and versatile Web application framework that provides Web development using a dynamic language (Groovy) and runs on a Java virtual machine. This combination provides easy use of all the power of Java and Java components, and flexibility of Groovy. Groovy provides a MVC pattern for development and a strong Object-relational mapping (ORM) database access layer. Classes that define data entities are called domain classes, and controllers are called domain controllers. Groovy on grails defines its own template language to define views.

Domain classes, controllers and views can be packed as plug-ins, which makes them easily portable to other Groovy on grails applications or projects. Integration of plug-ins consists of configuration editing – what controller should be invoked for which URL, but interactions between components have to be implemented individually. So, again, reusing an architectural unit component requires programming, and more than just a few lines of code, especially if different components' data has to reference each other.

3.4.2.4. Django framework

Django is a Python Web development framework that aims to provide great component reuse capabilities [21]. In theory, each component developed using Django is an application that can be easily reused alongside other applications – thus forming a greater whole application. Django provides a very flexible and powerful ORM. When a developer creates a model – Python class, the framework is capable of creating the database schema, and even provide the administrator's user interface to manipulate data that the model represents.

But reusability of all the Django applications strongly depends on their implementation. The developer has to balance between creating applications that are big and monolithic or too small to provide a unique functionality. Only those well balanced can be easily reused.

For example, a reusable application should provide signals for other applications' models and injection points in its views. This introduces complexity because there exists no convention in which the reusable applications communicate inside the framework, and this communication depends on implementation of each of the components and relies almost completely on the experience and effort of the developer. Even then, models often have to be extended with

properties that include references to models of the reusable applications [22], which decreases reuse inefficiency.

3.4.2.5. PHP Symfony framework

In Symfony [23] components that provide some reusable application functionality are called bundles. Each bundle is actually a separate application that can be executed by the Symfony framework. Symfony also uses a MVC pattern as its architectural pattern for bundles, so each bundle consists of models which are called entities, controllers and static files.

Other artifacts that can be packaged into a bundle are static files like view templates, configuration files, CSS and JavaScript files. So, reuse of a bundle consists of installing its files, and writing a few lines of configuration that tell the framework to use it. Interaction between various bundles or their MVC components can be achieved only if the components themselves provide communication interfaces, In many cases, the entities (models) also have to be modified to reference each other across bundles, which again introduces a level of inefficiency in component (bundle) reuse.

3.4.3. Common component reuse failings

Consider a web application that provides article publishing through one of its components. Such a component would define a database model to store articles, a few classes to handle the data, user interfaces and a controller to handle user actions. If commenting functionalities were added, to be able to attach the photo gallery from section 3.4.2, to each of the articles, there are few options available. This applies to all the component models of the previously mentioned frameworks.

The first option is to create a commenting component that integrates tightly with the news component – at the database level, comments reference a specific news article, and at the presentation level – the news component simply invokes rendering of the comments component for each article. The same would be done for the gallery component. But only if the articles application was built using Laravel. This solution is relatively simple, but works only if all the components are built using Laravel. This solution also causes tight coupling of articles, gallery and comments components, since their models would directly reference each other to be able to perform the required functionalities.

The second option is to create a generic comments component that has a mechanism to provide comments for more than just the news component. The commenting component can thus provide a set of required interfaces and handle the data abstraction itself. This is inefficient because it involves implementing generalization and functionalities for reuse inside the component itself. Further component reuse or nesting would be questionable. The same would have to be done with the galleries component. That should be a part of the framework's job. Additionally, reusability is further hindered by the fact that the component will only function inside a specific framework or an application that it was built for.

3.5. Web application frameworks

Web application frameworks are both a part of the definition of the component model, as well as implementations of a certain model using a specific technology.

Web application frameworks that are built for three-tier web application development, or rather development of components of three-tiered web applications, consist of base components and utilities for every of the three tiers [23], [25], [21], [18], etc..

Storage providers are components that enable programmatic use of server-side disk storage, access to remote or cloud file and data storage services etc. is supported by all frameworks, while Laravel has the most elaborate storage provider system of all, supporting both local, network and cloud providers. Other frameworks simply provide components with a path to which to save files.

Database access is the most common requirement of applications and components. Having a database access layer is mandatory for every framework. Using a database abstraction layer for database access is preferred because it removes the requirement of using a specific database management system.

Object-relational mapping is usually a layer on top of the database access, or database abstraction, that makes it convenient for components and component builders to use objects to manipulate data in a relational database. This is done through model objects, whose class is mapped to a certain table in the database, and properties are mapped to columns of that table. Changes and actions on these objects may be written to the database – automatically, or explicitly when a save method of such an object is called.

Model definition and organization comes with object-relational mapping. Usually, for each three-tier component in the system, the framework defines a naming convention and place where model classes should be saved, how they should be named, which makes it possible for the framework to find and load them when needed.

Component composition of three-tier components is usually handled through a strictly designed interface that is used by the framework. This makes it possible to easily load or unload additional components and execute them when necessary. For example, WordPress plugins have an interface that makes it possible for WordPress to invoke and render them when needed. This means that every component is required to have a certain set of methods or properties that will be recognized and used by the underlying system (framework). This makes **run-time composition** possible.

Design-time composition of components means the framework has a way to achieve component composition without tightly coupling components (specifying sub-components directly in parent components), but rather through an external file for example. Such design-time composition is made possible by the same rules that make run-time composition possible.

Packaging of components is an important aspect of a framework. How easily reusable a component is, also depends on how portable it is, and that depends on how easy it is to create a component package. In general, a component will be easier to package if all the sub-components, or code constructs, are saved in the same place. In that case, an archive can be created, and the component can be copied and downloaded over the internet etc.

Component packaging includes file types, directory structure and metadata included within a single component.

Front-end development provided by frameworks usually entails templating and ways in which front-end sub-components may be reused or built. Templating and organized client-side component delivery and composition can speed up repetitive tasks. For example, creating forms that will post data back to the component that generated it.

Localization makes it possible to write applications that are translatable to multiple languages. When dealing with components, it usually entails providing a way to define string names or keys, and their translations in various languages.

Authentication and authorization features are provided so that parts of application or content are visible only to users that have logged in, and depending on roles and permissions a certain logged-in user has.

Authentication utilities might also provide support for various back-ends, such as third-party login services such as Google or Facebook, various two-factor authentication services etc.

Session handling is supported by the underlying platform, the programming language, but it may also be supported in a more elaborate way by the framework, pre-filled with certain application and framework-specific data.

3.6. Building web application components

Web applications are built using web application development frameworks that implement a certain component model, using specific technologies. Consequently, web applications are built by reuse of frameworks, their base components, and reuse of custom-built components.

The custom-built components are built by using framework's base components and framework utilities. All the software engineering disciplines introduced in section 2.1 are applied throughout the development lifecycle of each of the custom components, which satisfies a set of requirements.

To demonstrate example component structure, HelloWorld components are described in the following sub-sections. These are simple components that are used throughout the industry to introduce a component model and the framework. From a HelloWorld component, an engineer will understand the basic component structure and lifecycle.

A HelloWorld component functionality is simple, it writes a "hello world" string to the screen, for which reason it is called a "Hello world" component.

WordPress and Laravel are chosen as for being de-facto industry standards for three-tier web applications. Laravel is the most popular PHP-based web application framework and WordPress is the most popular web content management system with many available components, both free and commercial.

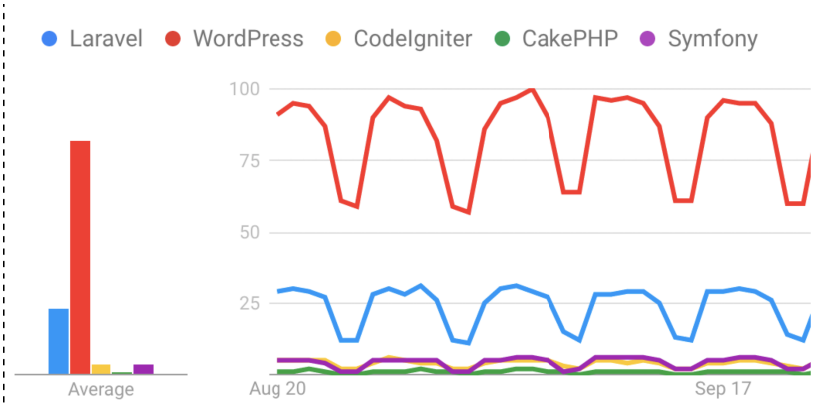

Illustration 7: Popular web application frameworks in August and September of 2019

Popularity of *WordPress* and *Laravel* is observable through Google trends[8], shown in illustration 7. The Y-axis of the graph represents relative popularity of a search term (in our case a framework), while the X-axis represents time. A huge gap in popularity between *WordPress* and other frameworks is visible, and a huge gap between *Laravel* and competing frameworks such as *Symfony*, *CakePHP* and *CodeIgniter*.

FAC is also compared to Quilt CMS because it is a custom content management system, built using a custom framework, thus both "standard" and custom web applications are compared.

3.6.1. Building a HelloWorld component using Laravel

Laravel applications are built by specifying which *controllers* and their methods are invoked for certain *application routes*. User actions are performed through a UI, built through *Blade templates*[9]. In Laravel, controllers are built using *controller classes*, whose methods receive user input from configured routes. Controllers usually use *models* – which are classes that also provide object-relational mapping, which means they can be used to persist data into the database.

Model classes' namespaces and folder organization. Their placement inside the Laravel's *app folder* is completely outside the component model of Laravel, left to software engineer. They

[8] Google trends provides keyword-related data including search volume index and geographical information about search engine users.

[9] Blade templating engine for Laravel, https://laravel.com/docs/master/blade

may be placed directly, or inside sub-folders, should the engineer decide to organize models into namespaces.

Laravel component model requires the software engineer to put all the enumerated code constructs inside scattered folders inside the Laravel application structure.

This is also true for resources such as string translations, CSS and JavaScript files. Translation resources should be placed inside Laravel's *resources/lang* folder, JavaScript and CSS files should be placed into Laravel's *resources/js* and *resources/css* folders, while the component model makes no enforcement of their naming, further organization etc.

To create a component, a controller class is created in the path: *app/Http/Controllers/HelloController.php*.

In contains only one method that invokes rendering of a view called "hello":

```
public function index(Request $request){
      return view('hello');
}
```

The method is invoked when the user visits the application's route */hello*. This is achieved through route definition in the *routes/web.php* file:

```
Route::get('hello', 'HelloController@index');
```

To make the string that is printed out reachable through the localization mechanism of Laravel, a file *resources/lang/en/hello.php* is created, containing only the hello message string:

```
<?php return ['hello'=>'Hello world!'];
```

Finally, the view named "hello" is created with a Hello world message, in the *resources/views/hello.blade.php* file, containing only:

```
{{ trans('hello') }}
```

3.6.2. Building a HelloWorld component using Quilt CMS

To build a Quilt CMS component (portlet), a folder for a component is first created, then all the related subcomponents and code constructs are put inside it. Each portlet is represented by a single class that extends a base component class, called *BaseModule*. This class serves as a controller for the component, and will handle inputs that users submit through views.

In Quilt CMS, software engineers do not have to take care of routing, because the framework does that automatically during run-time.

Views are built using *Smarty* templating, similar to FAC, and resources like language translations, CSS and JavaScript files should be placed inside the portlet folder in sub-folders: *lang*, *css* and *js*.

Model classes, that can also be object-relational mappers, just like in Laravel, should be placed inside the portlet folder, inside the *classes.cms* sub-folder, and should be named inside the portlet's namespace. For example, a *HelloWorld* portlet with a model class *Hello* will have a namespace *Portlets\HelloWorld*, and the model class absolute name will be *Portlets\HelloWorld\Hello*.

To create a Hello world component in Quilt, a portlet was created. Inside Quilt's portlets folder, another folder called *hello_world* is made, containing the file hello_world_portlet.php, along with folders *templates* and *lang*.

Implementation is very simple, the hello_world_portlet.php contains only a class extended from the vase *portlet* class called V1BaseModule:

```
<?php
class portlet_hello_world extends V1BaseModule { public function
defaultaction($params) {
      return v1result([], “hello.tpl”);
}}
```

When a user visits a page that has the built Hello world portlet, Quilt will execute *defaultaction* method, which tells it to render the *templates/hello.tpl* template, which only contains the call to Quilt to print out the translation string “hello”:

```
{$_lc.hello}
```

that is retrieved from the *lang/en_US.UTF-8.php* translations file:

```
<?php return [‘hello’=>’Hello world!’];
```

The only thing a user has to do to activate the control is put it on the web page, through Quilt user interface.

3.6.3. Building a HelloWorld component using WordPress

WordPress components are called plugins. Each of the plugins resides inside its own folder, inside the WordPress' *wp-content/plugins* folder as a PHP file with functions that are called at certain events. WordPress defines events called hooks. The developer of the plugin should choose which events are important to the plugin and specify the plugin function that will be called when the event occurs.

In addition, a plugin may define one or more shortcodes. Each shortcode is a call to a plugin function that can be made through user-created content in WordPress. For example, a shortcode named "hello" will be invoked when the user publishes a page or post in WordPress and adds the following line to the content:

```
[hello]
```

A Hello world component in WordPress is built as a plugin with a *shortcode*, that renders the message "Hello world!" when invoked. This architecture is chosen because it's the simplest way to create a component that can render the message.

To create a plugin, a folder is first created inside wordpress/wp-content/plugins, called *hello*. Inside it, a file that holds the plugin code: hello.php is made, consisting of a custom function and a call to WordPress api function *add_shortcode* to register the plugin's custom function as a shortcode:

```
function hello_world($params) { return "Hello world!"; }
register_shortcode('hello', 'hello_world');
```

This makes the plugin usable by creating a page through WordPress back-end and entering [hello] as it's content, which will execute the plugin and write the "Hello world!" message when the user opens the page.

To make it complete, a localization file is created, using the *gettext* library, or sometimes called a *po*[10] file: *wp-content/languages/plugins/hello-en_US.mo* with a localization string "hello" which is used from the shortcode function:

```
function hello_world($params) { return __('hello'); }
```

[10].po files are a product of the *gettext* standard for creating localization files in Unix systems, https://en.wikipedia.org/wiki/Gettext

4. MEASURING SOFTWARE COMPONENTS' REUSABILITY

Reusability is a basic concept in component-based software, and broader in software engineering. There exists white/glass box reuse and black box reuse [64]. Black box components can be observed only by their interfaces, while the internal structure and working of the component is unknown. White/glass box components' can also be observed from the outside, by looking at their interfaces, but also their internal structure and functioning is observable - Illustration 8. Internal components can sometimes, depending on the component model, be replaced, and in some cases if a software component program code is available, that could also be modified.

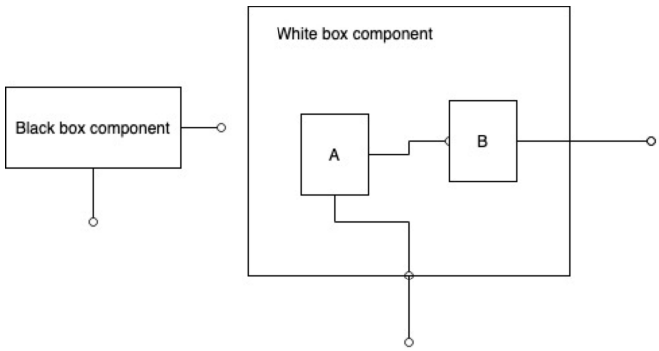

Illustration 8: Black vs White box components

There are various approaches to measuring component reusability. Reusability metrics are often evaluated together with component complexity and component coupling complexity. Usually, when component complexity metrics show that components exhibit less complex interactions, or when metrics like coupling are smaller, and cohesion higher, components' reusability is greater and easier to achieve because component functionality is well designed and separated [52].

Research on the topics of component complexity and, derived from that, reusability, design and development practices that produce reusable components, yielded common approaches and results through which researchers tried to define ways in which it is possible to compute complexity or predict reusability from the internal structure of components, structure of composite components or structure and functioning of component systems. This in turn

yielded methods and tools that can be used to describe components using metadata – both metadata programmatically extracted from analysis of program code and user-entered metadata, such as descriptions and annotations of components, composites or systems. Such metadata can be used to design component discovery and delivery systems, and as a consequence, component discovery and delivery can be automated. Based on all that research and produced knowledge, reusability frameworks have been developed [53] that made great advances in how software is reused in enterprise environments.

While reuse has been performed successfully and in a predictable systematic way, the industry is still striving with reusing components in simple scenarios. Rather than dealing with all the complexity of reuse, in many cases the same functionalities are repeatedly implemented whether because of heterogeneous component models or other reasons like uncertainty over component reliability, extra-functional properties etc. [54]. Component based software engineering for the web area of knowledge shows that many web application frameworks employ similar component models and component development approaches [11] but components that are produced are incompatible [14], [55]. This is improved by the FAC model. Software metrics provide the answer on whether reusability is augmented, and provide a reference on how much.

Most methods measure static complexity of the code. Static complexity entails non-functional properties of the code. Analysis of static code complexity can give answers to questions like how maintainable code is, how well does it comply with coding standards, how many classes and methods there are, and try to compute a numerical representation of reusability from those factors.

Another step of static analysis entails complexity of integration, which involves search for interfaces that are involved in integration of components. Number of components coupled and number of methods are counted because it is expected that it can show how hard it is to achieve said integration. On the most basic level, experiments have shown that the more lines of code a component consists of, the more complex the component is since it probably manipulates more complex data and does more with that date, and so the greater effort will be to integrate it into an application or another component [52].

Counting the lines isn’t enough, more static code properties need to be considered. Components provide different levels of functionality; components can be composite and any

part of the component or the whole is affected by a change made to any of the parts of the composite; components exhibit transitivity of dependencies; components can have different levels of cohesion and coupling and all these factors impact reusability and its cost. For example, a higher count of functionalities of a certain component might justify the cost of its integration into a system.

Many component models define components that are represented by a single class, or a class that ties all the sub-components and code constructs together into what is considered a component. The class itself is not the component, but there usually exists a central entity that connects all the parts of the component and exposes it towards the rest of the system. This is the reason why it is always important to evaluate components at a higher level, but also at the object-oriented level because the differences can be very abstract to notice or may change with perspective of evaluation. This property of a single class component is shared by many of the popular component models of web application frameworks that are direct competition to FAC, like Laravel, Symfony, .NET MVC [18], [19], [20], [21], [23], [25] etc.

We will therefore consider components as entities that result from evolution of code entities in object-oriented software development. Such components comprise of a single class or multiple composed classes. Accordingly, researchers have found that there are two kinds of software metrics that can be applied to components [56]:

- object-oriented software metrics,
- and as an extension to the previous, component-based software metrics.

As is previously noted, component reuse starts with extending or using a certain class that defines the component and serves as the main representation of a component, and as a continuation of that paradigm, component-based software metrics are a sort of extension of object-oriented software metrics. In fact, both can be used in parallel to further our understanding of properties of various types of software components.

4.1. Object-oriented software metrics

Object-oriented software metrics are used to expose extra-functional properties of programming code, such as internal structure of classes, their complexity, functional completeness, dependencies and their interactions – in a numerical way. Background of such metrics lies in measures of software complexity, which has never been easy because of a high

level of abstraction that software inherently represents. Some of the most commonly used software complexity metrics are **McCabe's cyclomatic complexity metric** [59] which is used to compute a number of independent paths through a source code can be visualized as a control flow graph. Nodes on the graph are forks of control flow like *if* or *switch* statements, while edges connecting the nodes are other functional code statements. For example, if a program had not if or similar conditions.

There also exists **Halstead's complexity measures** [58] which are a method to compute a quantitative measure of complexity from operators and operands in the source code of the program.

Finally, one of the best know metrics, both in impact since it was invented and in terms of applicability to our needs to measure component reusability, that is also basis for many future work and analysis of object-oriented and component-based software is a metric introduced in the paper "Metric suite for object-oriented design" by Chidamber and Kemerer [57].

Based on mentioned initial work on software metrics, additional methods and approaches to measure object-oriented and component-based metrics, combined with software complexity metrics have been researched. Before applying these metrics, it is also important to decide what is the end goal of metrics analysis. For instance, maintainability of a certain class is of interest, its complexity and sheer size in terms of lines of code will be of value. While it may be very easily reusable as a black box component, the cost of making a large and complex component itself work, in the long-term, might be too high. But, this is a separate concern, that applies to all software in general.

To be able to formally define classes, objects or their relationships, a notation is needed. Chidamber and Kemerer then define an object X using the following:

$$X = < x, p(x) > \quad (1)$$

Where x is the name of the class from which the object is instantiated, and p(x) are properties of the object – its methods and instance variables. This will be useful to define what metrics actually mean in object-oriented complexity ontology.

We will briefly introduce some metrics, including some of **Chidamber and Kemerer's metrics** that can be used to measure complexity of object-oriented software, interactions between classes and class (and transitively component) reusability.

Weighted method per class (WMC): is a metric used to measure the sum of all the class' methods' cyclomatic complexity, which can be used to measure how hard it is to develop and maintain the class. This will intuitively make sense since it basically shows what the number of statements, conditions etc has to be written and maintained inside the class, and the more methods there are, more will be translated to child classes that extend it, and more complexity will ensue. In general, classes with a greater number of methods are usually more likely to be application specific, and are less likely to be reused.

To formally define WMC, a class C with methods $M_1, \ldots M_n$, with complexities $c_1, \ldots c_n$ respectively is considered. Then weighted method per class is defined as:

$$WMC = \sum_{i=1}^{n} c_i \quad 2$$

WMC may definitely be useful for code analysis, but there are indications that cyclomatic complexity doesn't work well in practice [60].

Depth in tree (DIT): measures how for down in the tree of inheritance a class is. A class that doesn't have a parent has a depth of 1. Influence of depth in tree should be considered because if there is a property P of a class, then each of the classes extending that class will also have it. Since classes with more properties tend to be more complex, the deeper the class is in the inheritance tree, the more complex it will be.

Response for class (RFC): number of methods that may be invoked when a class method receives an outside message (public method invocation). This will make sense for a larger set of public methods. A tester would have to perform more tests on the class, and understand the class in more detail. For example, a method is invoked, but after that, itself it invokes a number of private methods. This also includes other classes' and parent classes' methods. RFC is defined as:

$$RFC = M \cup R \quad (3)$$

where M is a number of all methods of a class and R is the number of remote methods any of the methods $\{M_i\}$ may call.

Coupling between objects (CBO): measures how many different other classes a class invokes, either by invoking their methods or by using a property or a local variable as an instance of another class type. Objects are coupled if one acts upon the other in any possible

way – by containing another object, referencing it, invoking methods of another object or producing instances of another object.

For two objects, say X=<x, p(x)> and Y=<y, p(y)>, coupling exists if one acted upon the other in the past. Acting upon the other includes both any action performed by {Mx} upon {My} or {Iy}, and vice versa – any action performed by {My} upon {Mx} or {Ix}.

Lack of cohesion (LCOM): measures how much methods of a class are similar, or rather different when lack of cohesion is considered. Methods that perform completely different tasks or have no common properties that they work on, and thus lack cohesion. Methods that access the their instance object's variables are cohesive. The bigger the cross section of a set of variables used by two methods is, the more cohesive those methods are. So, if methods are defined as M1, M2… Mn, and I1, I2… as class properties, or instance variables.

The degree of cohesion is then defined as:

$$\sigma(M1, M2 \ldots Mn) = (I1) \cap (I2) \ldots (\mathrm{In}) \quad (4)$$

Where σ() is a function of the degree of cohesion of methods Mi, dependent on the cross section of used object variables (properties). The greater the cross section is, the more cohesion an object exhibits.

4.1.1. Cohesion and coupling

In the subsequent sections, metrics and literature is introduced. It is clear that cohesion and coupling are metrics that are used when reusability of object-oriented code, classes or components is analyzed [74], [66], [62]… Decoupling, or loosely coupling components makes them independent of one another, and cohesion makes component functionally complete and replaceable. For decoupling, one might look to analogies in other engineering areas. For example, a car tire can easily be replaced without any impact on wheel or engine, while its intended function is very clear inside the system.

4.1.1.1. Component coupling

By making components (or modules, classes, code constructs) loosely coupled, is is ensured that they can more easily be used in different software systems, sometimes that also means different platforms, operating systems… Decoupled systems can also be decoupled in time. That might mean that messages are not synchronized – that once a component sends out a

message, it may take time before it is processed by the receiving end. Architectural patterns to decouple components are many, and only the ones that were used to ensure FAC isn't a tightly coupled component are introduced.

Connections via mediator will lower coupling between objects in the system where objects have to communicate with each other. Instead of each object being aware that there are quite a few types of them, and knowing how to communicate, they all rely upon the mediator object, know how to communicate with it, and communicate over it, thus lowering coupling.

Asynchronous communication is a pattern that is sometimes included within the event-driven architecture pattern. Instead of passing messages directly between objects, a message queue component is used (events can be transferred in this way). It usually conveys messages made up of basic data types, and in contrast to event-driven architecture, replies may be sent.

Event-driven architecture is a pattern in which components don't notify all interested parties (objects, classes, services) directly, but rather "fire" an event which gets delivered to all subscribers by the event system, usually the broker. The component that is the source of the event doesn't need to know about components listening to its events, thus, lowering coupling.

Dynamic bindings of service and consumer components means a user of the software can at any time bind two components together. This means that components are sufficiently decoupled in their implementation and design that they can be coupled during runtime in a simple generic way, which means their reusability is well designed, thus, coupling is low.

There are also other methods to lower coupling, that can be applied to any software, if the approach is accepted into the architecture. For example, coupling can be lowered if the **data is transmitted in a standard format** and using **standard protocols**, for example JSON or XML over HTTP, instead of using custom or proprietary formats or protocols.

Class methods should be designed to **accept standard data types** (like integer, string ..). Passing custom data types, like instances of certain classes or complex structures of arrays will cause more coupling because the receiving class and method will have to know of the foreign data structure, or another class.

Passing only key data to a method or service will also reduce coupling. For example, a method that should send an email to the user is most reusable when only the key, an id of the user from the database is required, and the method itself finds the address to which to send the

message. This approach can lower coupling because methods won't have to be invoked in a specified order, for example first retrieve the address of the user, then invoke send an email to the address of the user.

Loose coupling of application integration in business process automation contexts can be increased by **following a presentation layer integration model** in which automation applications interact with underlying automated applications through the presentation layer or graphical user interface. This means that every part of the automation is prepared to be connected to a subscriber, or rather publish-subscribe pattern. Each publisher is completely unaware of subscribers, that it send the messages to.

4.1.1.2. Component cohesion

Cohesion is a measure of the degree to which elements inside a software component, module, or class, belong together. Cohesion is an ordinal scale. It is possible to deduce that a component exhibits low, medium or high cohesion, but it isn't possible to distinguish between two components with high cohesion [66]. Components with high cohesion are preferred, or better, to the ones with low cohesion because high cohesion correlates with desirable traits of software components, such as robustness, reliability, reusability and understandability.

A cohesive class, for example, contains instance variables that together make up a cohesive whole if they together build a description of an abstraction just in the right amount – not too much and enough to describe it. In addition to data, a class is cohesive if methods of that class also serve some unifying purpose of the class, and perform actions and calculations on cohesive instance variables.

High cohesion will be achieved when functionalities of the class can be accessed through its methods, that perform similar activities and avoid handling coarsely grained data, or accessing unrelated data.

There are a number of types of cohesion, that can give us an answer how good cohesion is. The following list enumerates these types, from worst to best:

- **Coincidental cohesion** will be detected when parts of the module are grouped arbitrarily, the only relationship between the parts is the fact that they are grouped together. This is not to be mixed with a facade or mediator patterns, which help reduce

complexity when connecting to certain components in case of facade, and avoid direct coupling between certain classes in case of the mediator.

- **Logical cohesion** may be grouping components that handle all input signals, although they might not be the same in nature. For example, grouping together code that handles mouse and keyboard input.
- **Temporal cohesion** means grouping together components that are run in a certain point in the lifecycle of software. For example, application is loading, an error occurred and the log is being written.
- **Procedural cohesion** is grouping certain modules together in a certain way, because they (for some reason) should be run in a specified way. For example, when changing user's picture, check if it already exists, delete the old one, resize new one if needed and then save it.
- **Communicational / informational cohesion** means grouping components or parts of a component by data type that they are meant to process – e.g. date functions.
- **Sequential cohesion** is grouping components because output of one is the input of another, like an assembly line.
- **Functional cohesion** is the best type of cohesion, and one engineers intuitively try to achieve when creating classes for software. This cohesion type is achieved when components are grouped, or a part of a composed component, based on whether they contribute to a single well defined task. For example, mathematical function visualization.
- **Perfect cohesion** is atomic – when a module is reduced to a minimum. For example implementation of function $r(x) = 2x + 1 + 3x + 2$ is said to have perfect cohesion when it cannot be reduced any more than that: $r(x) = 5x + 3$.

In real life scenarios it is often only possible to achieve communication / information cohesion [66]. The reason for this lies in software itself – cohesion depends on ensuring the class interface presents a consistent and well modeled abstraction and modeling good abstractions is hard.

4.2. Component-based software metrics and reusability

Component-based software metrics, or component metrics, are metrics that evaluate code of components, but should measure properties of components at a higher level than object-oriented metrics. Components are considered a higher level of abstraction than objects or classes, although in some cases they may be considered as classes, when all component interfaces are exposed through a single class, usually a facade class.

In many component models, a component may be evaluated through such a single class, because (like in FAC component model), there exists a class that is the central abstraction, that represents a component to the rest of the system. In fact, separate extensions are visible through a single class, the implementation of the abstract *Extension* class, to the rest of FAC, but to other components through the universal adapter. In cases when that is impossible, when a component that is observed, evaluated or reused is made up of multiple classes, whole applications or libraries, measuring component metrics approaches consider components as black boxes, and measure properties based on observable interfaces or properties [61].

Measuring quantitative component reusability is hard, because of the abstract nature of components and software in general, so many approaches are qualitative, as Bhattacharya and Perry have found [72], reusability is in practice evaluated mostly through costs, quality and usefulness. It doesn't help either that there are multiple, similar but different definitions of a component [11], [62], [6]. This makes it inherent to define component metrics in different ways, to assess different properties of aspects of components, and then try to apply them in different ways to assess component reusability. One categorization of components, that seems to include most important types applicable for our case is done by Narishiman and Hendradjaya [62]:

- An operation-component consists only of traditional procedures or operations,
- A class-component is a component made up from a class, or that consists of multiple classes,
- A module-component is a component consisting only of modules,
- A super-component is a combination of all the above.

All the above can then additionally be categorized as white or black box components, and my be assessed for component reusability.

Mijač and Stapić [71] have assembled and analyzed a comprehensive list of component reusability metrics, and categorized them by component type (white or black box components). All approaches rely on well-know metrics. Some approaches provide ways to quantize reusability, while other hint that results from well known metrics should be interpreted for reusability. Coupling and cohesion are the most popular metrics most of the approaches use for assessing reusability, followed by complexity, reuse frequency or volume etc. There are very few methods that also use metrics like regularity, customizability, maintainability etc., which seem to be of value when pondering whether to reuse a certain component from a practical, for example, should I install this WordPress plugin perspective, not from a technical difficulty of integration point of view.

4.2.1. Measuring component reusability

Kumar and others [56], measure coupling of components to compute their reusability. There are approaches to measuring reusability by measuring coupling and cohesion, by including strength and transitivity of inter-component dependencies [52]. Transitivity is a concept that includes classes and their methods that are used indirectly by a class. It might, in some cases, show a better answer to how reusable a component is. The greater transitive coupling, lower the reusability.

Sook Cho E. and Sun Kim, M. [70] propose that reusability of components can be computed from the number of interfaces providing functions in a domain, divided by total interface methods of the component. From that, they also define component reuse level to be a measure of how much of component functionality is needed by the application that is using it. **Component reusability is then defined as**:

$$CR = \frac{\sum_{i=1}^{n} (Count(CCMi))}{\sum_{j=1}^{m} (Count(CIMj))} \quad (5)$$

where Count(CCMi) is the count of each interface method for providing common functions for application domain, and Count(CIM) is the total count of methods provided by the interface of the component.

4.2.2. Measuring reusability from complexity

Narasimhan and Hendradjaya [62], and Kumar et al. have defined a set of metrics that can quantize complexity of integrated components, that implies higher coupling and lower reusability. It also implies higher complexity and harder testing and maintenance of built software [56]. Complexities that can be measures stem from the interaction density of a component and from packaging density of components. Additionally, when multiple components are coupled, their interaction density can be computed, and also quantized, which gives another metrics for complexity of those interactions, which also translates into implications for reusability.

Interaction density of a component (IDC) is the ratio between connected interfaces of a component, and the maximum interfaces a component offers to other components of the system.

$$IDC = \frac{\#I}{\#I_{max}} \quad (6)$$

Respectively, incoming and outgoing interaction densities can be defined in a similar way. Instead of considering all interfaces, for incoming interaction density, the number of used incoming (required) interfaces is used. For outgoing interaction density, only the number of used provided interfaces over the maximum provided interfaces of the component are used.

The **average interaction density** (AID), or in some literature CAID [56], is the sum of all interaction densities of all components in the system divided by the number of components.

$$\mathrm{AID} = \sum_{i=1}^{i=n} \frac{IDC_i}{n} \quad (7)$$

Component packaging density (CPD) is the ratio of constituents in components, where constituents can be lines of code, number of objects or classes, number of modules, operations, etc., and the total number of components in the system.

$$CPD_{constituent_{type}} = \frac{\#constituents}{\#components} \quad (8)$$

For black box components only component interfaces and their interactions are analyzed. This means that it is either possible to analyze components separately or analyze them as part of the system. **Coupling complexity of a black box component** can then be defined as

$CC = II_c + OI_c$ [56], where II_c is the number of incoming or required and OI_c number of outgoing or provided interfaces.

Average coupling complexity is then defined in the same way as AID:

$$ACC = \sum_{i=1}^{n} \left(\frac{CC_i}{n}\right) \quad (9)$$

Using these metrics, it is possible to quantize the complexity of a software system built by integration of the host application and FAC and its extensions.

When measuring AID and CPD for a component system, there are four cases that indicate its nature:

- Low CPD and low AID represents a very simple or small software system;
- Low CPD and high AID often indicates it is a real-time system, that has low data processing and high computation – fewer optimized components doing lot of processing;
- High CPD and low AID indicates a system that processes high volumes of data, using many components. Systems like these are usually enterprise or business applications, having many functionalities and modules that handle those functionalities completely;
- When both CPD and AID are high, the evaluated system will be big and complex because a lot of components do big volumes of data processing, and communicate in the process.

4.2.3. Criticality metrics and implications for reusability

It is possible to analyze components by considering how critical any of the components in a system is. The most intuitive way to do this is to visualize components' interactions as nodes on a graph Illustration 9. The two systems are integrated or connected using a bridge component B, while each of them has components that are connected more to the rest of the components in the system. For System 1, that is component C1, and in System 2, component C2.

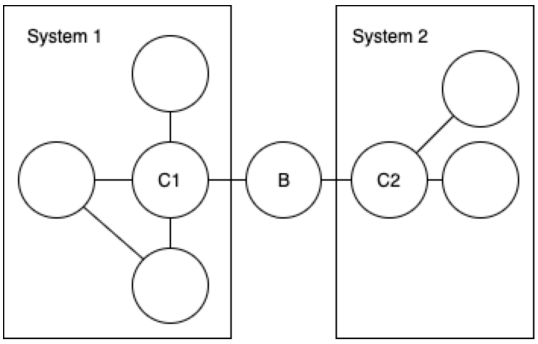

Illustration 9: Interactions of components

Properties that are visible in the graph can also be formally defined. Literature [56], [62], lists a number of different critical properties a component can exhibit in a system.

Link criticality is a measure of links a component has to other components. If a component has many links to other components, that is if many components use the provided interfaces of a single component, than that component's link criticality will be high.

Bridge criticality is a state of a component in which it serves as a bridge between two applications or components, or two sets of components. If the bridge fails, there will potentially be more problems.

Inheritance criticality means that a component is used as a blueprint or a base for other components, such as a base class from which all other components are built.

Size criticality is a case when a component is very large in size, which can be determined by other metrics, like lines of code. CPD for such components is very high and it might be hard to do maintenance, since larger size might also mean more bugs and it might be harder to find the source of the problem.

It is directly deductible that components with link, bridge and inheritance criticality are highly reusable, since many other components interface them for a common functionality (link), to establish a commonly reused functionality (inheritance) or connect to other components through them (bridge).

4.2.4. Reusability through properties of black-box components

When building software by reuse of black-box components, activities to choose and evaluate each component are performed:

- researching the functionality of the component;
- adapting the component to specific requirements;
- and porting the component to a new environment.

These are enabled by existence of meta information (description of behavior, customizability, non-functional properties and interfaces) and observability of the component (interfaces, inputs and outputs). The less external dependencies a component has, the greater portability (and reusability) will it have in other environments.

Washizaki et al. [73] introduced reusability metrics for Java Beans components, to evaluate those criteria. Their analysis is oriented towards Java Beans, but can be easily applied to black-box components in general.

Rate of component observability is the percent of properties that can be read in the component:

$$RCO(c) = \frac{P_r(c)}{A(c)}, for\ A(c) > 0 \quad (10)$$

Where Pr(c) is the number of readable properties in component c, and A(c) is the number of fields in the facade class of the component. If A(c) is = 0, then RCO(c) will also be 0.

Rate of component customizability is the percent of all the properties that are writable in the facade of class c.

$$RCC(c) = \frac{P_w(c)}{A(c)}, for\ A(c) > 0 \quad (11)$$

Where $P_w(c)$ is the number of writable properties of class c. If A(c) is = 0, then $P_w(c)$ will also be 0.

Self-completeness of component's return value is the percentage of all business methods of a class c, that have no return value. If there is no return value, it is expected that a component or class is more self-sustained, thus more easily portable or reusable.

Self-completeness of component's parameter is similar to the previous metric, the difference being that methods of the facade of class c without parameters contribute to reusability of the component or class. Percent of methods with self-complete parameters is:

$$SCC_p(c) = \frac{B_p(c)}{B(c)}, for\ B(c) > 0 \quad (12)$$

where $B_p(c)$ is the number of methods without parameters in c, and B(c) is the total number of methods.

From $\mathbf{SCC_r}$ and $\mathbf{SCC_p}$ the authors defined the **overall reusability metric (COR)**, which has been adapted for Java beans components, to discriminate for what they concluded were valid results, but which might be applied to other component models too:

$$COR(C) = 1{,}76\,\frac{V_{EMI}(C) + V_{RCC}(C) + V_{SCCr}(C)}{3} - 1{,}13 \quad (13)$$

where EMI(C) is 1 if metadata describing component C exists, 0 otherwise; while COR should be greater than 0 for reusable components.

5. FRAMEWORK AS A COMPONENT (FAC) COMPONENT MODEL

Component models have in more detail been introduced in section 3.1. A component model defines what a software component is, how it can be constructed, composed or assembled and finally deployed [9], [11], [14], [28].

Frameworks are traditionally abstract software constructs that provide facilities, rules and base building blocks to construct software by writing code. Formally, these rules, facilities and base building blocks implement and enforce the component model that defines how components are built and composed into composite components. A framework can thus be viewed as an implementation of a specific component model, or rather, the environment for building and running components that adhere to that specific component model. The user-written code then extends, reuses and builds upon generic framework functionalities. The resulting application and its components then adhere to the specified component model. This is elaborated in more detail in chapter 3.1.

The Framework as a Component component model (FAC) is in many ways the same set of rules and building blocks, though there are some notable differences opposed to other component models, that were the focus of this research. FAC requires that the framework is itself also a component that, at the same time, acts as a universal adapter between all components built using it, and all the third party components that will be integrated with it. The logic here is that by integrating a single component it is possible to implicitly integrate any number of components that were built adhering to the FAC component model. So, the FAC model defines a nested, recursive component architecture [12] and through that enforces that all the components have homogeneous, universal interfaces. This is just partly true for all component models where components must have uniform interfaces to connect to the underlying framework, and utility components a framework provides, to satisfy base classes' required interfaces for example. FAC, on the other hand, enforces that even provided interfaces of all components are uniform and are available through the framework itself, thus making them easier and straightforward to reuse.

In the following subsections, it is shown how properties of the component model have been achieved and how the model is defined:

- **What components are** and **how they are built** is described in section 5.2.5., which goes into detail what the architecture of each extension looks like, how parts of it communicate, what architectural patterns should be employed, how are multiple tiers of components described, what sub-components are and how can they be used. It also describes **how components are packaged** and **how they can be deployed**.
- Sections 5.2.2 and 5.2.3 go into detail **how components can be assembled** and deployed during implementation.
- In addition, FAC component model also defines **how FAC itself, and subsequently all FAC components, can be assembled** with heterogeneous applications or components, outside of FAC – through sections 5.2.2, 5.2.4 and 5.2.5. This is another important property of the FAC component model, one that makes it unique.

5.1. Domain and premise

The domain in which FAC is applicable is web applications. Other domains have not been included into consideration.

Web and mobile applications are currently most popular types of applications. Therefore, this research is targeting a large share of currently active applications, component models and frameworks. This popularity can be observed from the trend of the last few years in programming language popularity, from statistics that are published online, like the *RedMonk Programming Language Rankings* that use *GitHub* (the biggest online open source software repository) and *Stack Overflow* (a huge online community of developers helping each other with software or IT – related problems) to measure programming language traction [51]. GitHub is queried for languages by pull requests – the number of pull requests per repository for a certain language, with forked repositories excluded, and *Stack Overflow* is queried using their own Data Explorer Tool[11].

[11]Stack Overflow's Data Explorer Tool, can be used to retrieve statistics about questions on their site, https://data.stackexchange.com/stackoverflow/query/new

5.2. Architecture

Web applications are generally structured like a tree. Each of the nodes in the tree is a component. And each of those can be reduced to leaves of the tree until the view layer of each component, that in the end also makes up a tree of HTML elements. Each of the nodes (components) attaches itself to a parent that provides context. The root of the tree can thus be defined as the URL of the web application – from that, the application can deduce what other nodes (components) should be spawned, for the provided URL. Of course, context for those components will most often be more than just the URL, and parent components will in that case provide more context to child components. A tree-like structure shown by Illustration 10 should be considered.

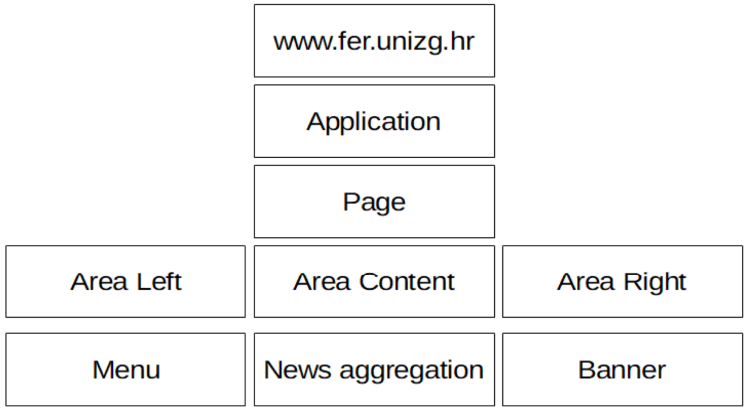

Illustration 10: Tree-like structure of a web application

A web application gets executed for a certain URL by the web server. This gives the application some context – the domain name, URL path and parameters. The next component that is executed is "Application". Application in turn activates many components directly or indirectly. In our example, Application uses a component called "Page" directly, which then in turn activates a number of components. In general, the Application component would usually be the application development framework – the one component that makes sure everything else runs, that provides all the necessary services and execution context for other components, and defines the component model for components that can be run in the first place. But for the sake of our example, all components are considered as simple as possible, without framework dependencies, only as a simple tree-like structure.

Thus, a Page component might need additional context for execution and the Application is supposed to provide it. Additional context might for example be the current user. It might also

need to translate the URL path to an id of the Page, so that the Page component can be initialized correctly, before the Page component takes over the main execution thread.

The Page component now initializes its child components in the same way the Application component did. Each of those components might need some additional context data and the Page is supposed to know how to provide that. Let's suppose that, for the sake of our example, the Page component should know which areas of the application should be rendered. Let's also suppose that each area is also a component called "Area". In our example, those areas are called "Content" and "Right". By pushing the execution context down the tree and initializing Area components for areas called Content and Right, both those Area components will now also know which child components they should initialize and execute.

Components Banner and News aggregation will be executed by Area "left" and a Menu component will be executed by Area "right".

For the sake of example, let's also define that each component is a three-tier architecture component, or a MVC architecture component. That means, each of those components renders a view for the user and manipulates data in some way.

This is a fairly simplified example of how a web application works, but it applies to many application architectures, component models [18], [19], [20], [21], [23], and how web application components in general receive their execution context: the common parts are provided by the application (development framework) and parts are provided by parent components. Just a side-note, this is completely natural to the way HTML is assembled.

This tree-like functioning of web applications and web application components is essential to understanding the architecture of the FAC component model.

In the following subsections, components of the FAC architecture and their properties are introduced and explained in more detail.

5.2.1. Architecture overview

In the following sections each element of architecture is explained in detail, but a high-level overview should be considered before details and specifics are explained. The high-level overview of FAC architecture is shown on Illustration 11.

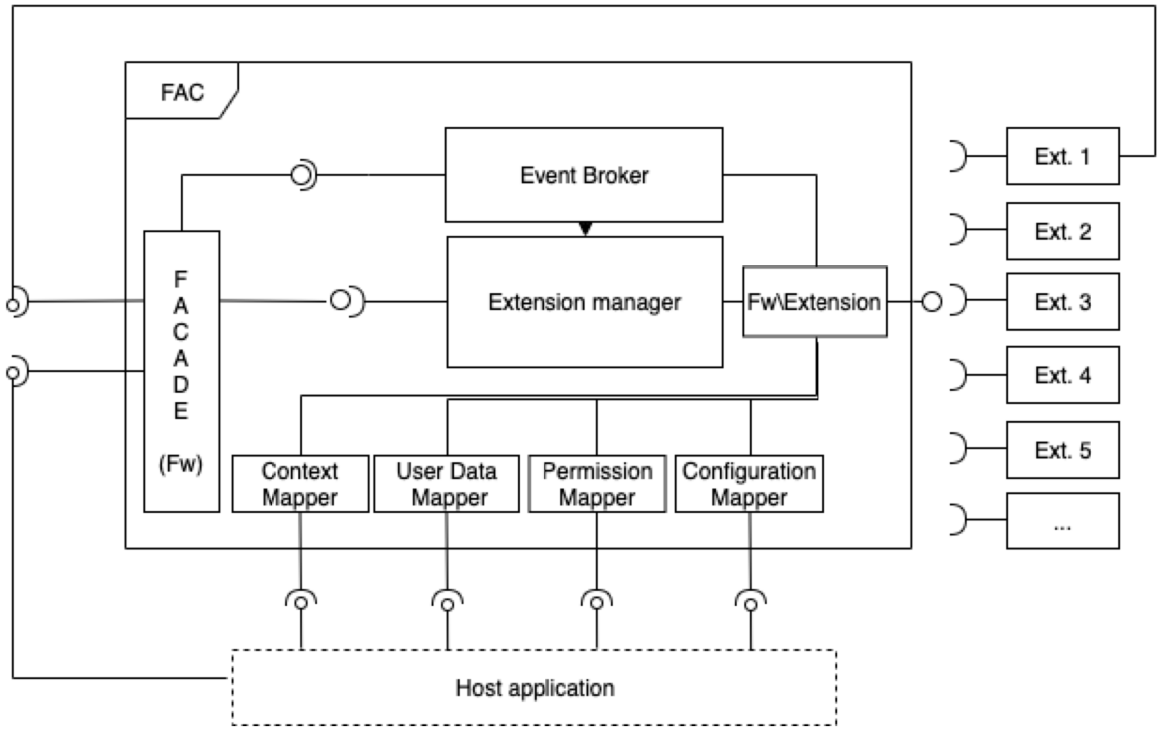

Illustration 11: FAC architecture and universal adapter visualization

The **host application** is visualized at the bottom, and connected to FAC through integration layer made up of **mappers** that translate host application execution context state and signals to FAC. Mappers are components that have to be implemented for every single host application integration, each being responsible for a part of context translation.

Facade is the main interface of the framework through which it gets bootstrapped, and through which the host application, or any components that wants to reuse a FAC component initiates that execution, or through which outside applications or mediators notify FAC of events that occurred. Actual handling of those events is then handed over to the Event broker.

Execution of components (extensions) through facade is handled by Extension manager. It instantiates actual extension objects (subclasses of *Fw\Extension*). The Extension base class computes input from the parent component, through Extension manager, from all the mappers and then the actual extensions are executed (to the right).

Each of those extensions, if it wants to use another extension, or another instance of the same extension, has to use the same Facade component, and use the FAC as an adapter to execute an actual extension. This has been called *the universal adapte*r, and is the main concept that drives design of FAC architecture.

5.2.2. Universal adapter

The tree-like structure of web applications and the way they are assembled using components, makes it possible to generalize the way in which components are interfaced, without loosing component functionality and purpose.

There are two sets of data that each component needs which will make sure it doesn't loose context and that it can execute, fulfill it's functionality. The first set of data is *common data* that all applications have and need; and the second is *data that the parent component provides*.

Consequently, FAC as a component model defines the development framework as a universal adapter and doesn't directly deal with creating common data. In fact, FAC is meant to be attached to other existing applications that have already solved all the common facilities like user authentication, URL handling, permissions and authorizations, etc., and deals with creating reusable components. The focus can then be pointed at extending existing applications, extending functionalities of existing components of other frameworks.

Common data is transmitted from the original application, component or framework, and translated into a form that FAC components understand. For this reason, components built using FAC are called *extensions*.

This implies that the first step to use FAC and any component built using the FAC component model is to integrate it with an arbitrary application/framework. Since the application which will host FAC is mandatory, it is called the *host application*. Only after this integration is achieved can extensions receive common context data, and be used to build or extend the host application or its components' functionality. How that happens and what it takes to create the context for extensions is described later, in section 5.2.4.

The two sets of data (common data and parent component data) have non-intersecting points that aren't common and are provided by parent components and points that are common (they do intersect) and are provided by the host application/framework. So, each FAC component receives context data from the host application and the parent component.

If required interfaces of the FAC are grouped (Illustration 12), that need to be connected to the parent component, or application, for an extension to run, a UML representation of FAC is simple.

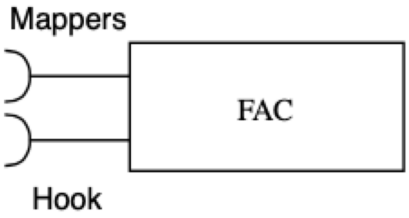

Illustration 12: FAC required interfaces grouped

Assembling components

Just as HTML tags are nested and used to build up the page using different tags, reusing components is possible in the same way, on the view layer. Each component that has a view layer displays a set of HTML tags, which actually is a natural way of composing web applications. To a simple way of running a component is required. A function or another similar construct is required the templating engine that is used to build the view layer of a web application component. This would fulfill all the previously stated data requirements of extensions.

The function and the data that is sent to the extension it activates is called a "hook". The function provides required data for the child component (extension) to hook onto the parent component – and render into a desired location in the view layer. The parent provides a simple way for the extension to hook onto it, while the FAC framework does the rest of execution context translation in the background.

The whole FAC framework is exposed towards reusing extensions and other components by using the facade pattern. The *ExtFw\Facade\Fw* implements all the public interfaces to the framework. These interfaces are later explained in more detail: framework initialization interfaces, extension execution interface, interfaces for extensions to retrieve context mappers instances, handling incoming events.

5.2.3. Hooks

A hook is a shorter name that was given to the universal adapter – the explicit part of it that is used by components to reuse other components.

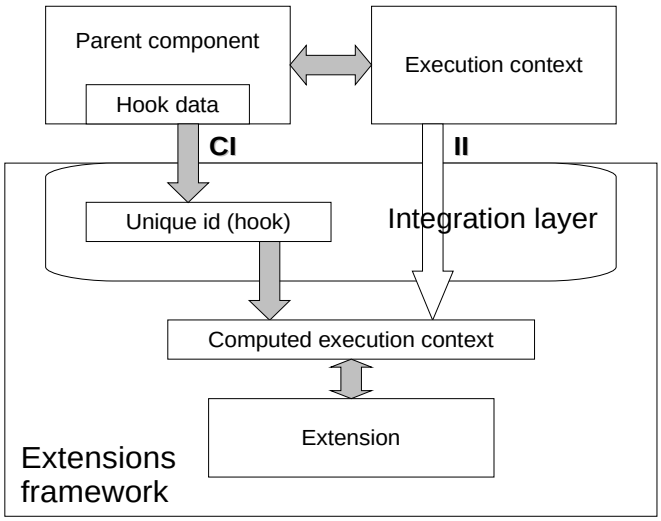

Illustration 13: Activating an extension through a hook

It is an interface through which FAC as a universal adapter is invoked and executes extensions, as shown in Illustration 13 [12].

Hooks are software constructs or functions, depending on the host application component model which defines how creation of view layer parts of components is supposed to be done. Hooks have to be prepared for each host application framework/component model. They are considered to be a part of FAC, not the host application. For example, *Laravel* applications utilize *Blade* templating engine, *WordPress* uses pure PHP. They both provide ways in which it is possible to activate FAC as a universal adapter through a hook, and execute extensions. So, a hook for *Laravel* is a *Laravel* component that works when the rest of FAC integration of *Laravel* is achieved.

The component that "provides" a hook doesn't do much work. The hook is invoked just as any other function or HTML tag would be invoked, on the view layer. There are only four arguments passed to the hook: name of the data, a unique representation of the data (preferably string), an id for the extension (just an id of a HTML element), and a Boolean argument whether the hook is being used inside of an HTML form. Optionally, a hook can invoke an extension directly by specifying its name, but if that information is omitted from the code, FAC provides ways for the user to attach extensions to "empty" hooks. FAC can also provide ways to add additional extensions to non-empty hooks. Elements of hook data described have already been hinted in the example at the beginning of this chapter, and the visualization of a component tree, as shown in Illustration 10.

An application written in Laravel is considered, with a *NewsAggregation* as an extension, not a native Laravel component. The invocation of the hook to execute *NewsAggregation* inside a Laravel view would be a Blade method call:

```
{{ \ExtFw\Fw::hook('Page', 423, 'NewsAggregation', 423.'aggregation-1') }}
```

This would simply invoke a FAC's facade class that resides in the ExtFw (*extensions framework*) namespace, and executes a hook method. The first argument is the name of the data type that the extension will hook onto, second argument is a unique identifier of that data, third argument is the name of the extension – *NewsAggregation* and the last argument is the id. The last parameter is concatenated from the actual id of the page and the string "aggregation-1" - so that it will always be unique. Giving extensions unique ids from parents is important as shown later in section 5.2.6, so that FAC can effectively parse and process submitted data.

A **generic hook** can also be defined, which doesn't require the name of the extension that should be used to be provided. Instead, it makes it possible to dynamically bind an extension, or multiple extensions to the provided generic hook. A generic hook should specify all the other arguments for the hook – the content name, id of the content and the unique hook identifier.

Unique ID objects

Behind the scenes, FAC calculates common context data and calculates a Unique ID object. Unique ID objects are actual data that each extension is provided, that it can hook on to. It contains information about what the name / type of data is, what is the unique identifier and the actual computed unique numerical representation of that object – the unique id.

FAC creates a database table to store these unique identifiers. Since extensions must never have direct database references to each other's db schema, and especially not to the host application's components' schema, this is a kind of a proxy that they can reference. Extensions can reference the unique id database table directly, but this shouldn't be done since it causes tight coupling. Instead, FAC's event subsystem should be notified on data changes, addition and deletion.

Using FAC event subsystem isn't ideal and in real-life scenarios because it entails firing events in the host application by directly invoking one of FAC components through the FAC

facade. A much better solution would be to use one of many available standard message queue components, like for example *Beanstalk* [12].

For larger applications, it is generally a good idea to fire events when models or data perform any CRUD operation. Most component models and frameworks already are event-driven, so forwarding events to a standard message queue handler shouldn't be a problem as it creates only one additional integration point with the FAC (described in more detail in chapter 5.2.4).

So, when an extension gets hooked to a parent, FAC creates an UniqueID object and then starts monitoring for relevant data events from the host application or parent extension, for data and type that that UniqueID object represents. Extensions that have used that particular unique id object as part of the hook are expected to remember it, so that they are able to consume and act upon events that FAC tells them happened, to that particular unique id object.

Enforcing loose coupling

To avoid specifying components that a hook should execute, FAC makes it possible to define which components should be executed by which hook id, for which content type. This enforces loose coupling between components, both extensions and the host application components and extensions.

5.2.4. Context mappers and integration components

Context mappers are main integration components that provide common data integration points. Technically, context mappers are abstract classes with abstract methods that have to be extended and implemented to achieve integration with a certain host application or framework. Each host framework or host application integration entails implementation of another set of mappers inside of FAC. Once mappers for a specific framework or application are implemented, FAC can easily be integrated with any instance of it. Only thing that has to be done to achieve that is provide basic configuration options.

5.2.4.1. Database access and persistent storage

Although not really a mapper but rather an abstraction, database access or persistent storage access is one of the most important functionalities for any application or framework. Thus,

[12] Benstalk work queue, https://beanstalkd.github.io

FAC provides database abstraction to its extensions. FAC can be configured to create database tables it needs inside an existing database, and prefix all the table names it uses, or use a separate schema. In terms of functionality it makes no difference, but is an important possibility since FAC is meant to be used as provider of components, extensions, rather than a complete application provider. The difference is in fact slight, but distinctive.

FAC or its components will never directly access host application's data, and an extension should never directly reference or access another extension's data. References should be done only through unique id objects (hooks) and integrations should be achieved through event and messaging systems. If that is true, than database access, in terms of mapper components, should be configured so that it never conflicts with host application's schema.

From the FAC perspective, the database access layer, when configured provides database access for extensions and other FAC components. It also provides object-relation mapping that extensions can use, if they want to use it.

Persistent file storage is also one of common functionalities in web applications. Users upload their files, or create files through the application, and those files have to be saved on the server. FAC also provides utilities necessary to save files and abstracts their real path on the disk for extensions. To be able to function, it needs to be configured so that it doesn't conflict with the host application.

5.2.4.2. Context mapper

Context mapper takes care of translating all the context information that isn't translated by one of the specialized mappers.

One such information is the current language of the resource being loaded/rendered to the user. Web applications can support multiple languages in parallel. Resources that are available to users can be in any of the different languages. Sometimes the URL of the resource will make it possible to explicitly determine the language, and sometimes that is done in a different way, so this mapper takes care of providing the information to FAC – what language should be applied for the current request.

Permission mapper and user data mappers provide information to FAC on what the current user permission level is and a reference to which to register data the user creates. They don't

provide information whether the current user is a logged-in user, or an anonymous user. So, this information is translated through the context mapper.

Sessions can be named and it is important for FAC to be aware of the name of the session, so that it can write to it without overwriting any of the host application data. Additionally, to be able to integrate logic that takes sessions in account within extensions, information on how long sessions last and when will they expire has to be available.

Many host applications are able to run and render for multiple DNS names, or domains. Providing this information to FAC is also the task of context mapper.

If the host application component is reusing an extension, or an extension is reusing another extension, and while doing so is rendering a form through which the user is editing data, and is in parallel requesting rendering of the extension it is using, FAC will make sure the reused extension also renders in “edit” mode – whether it is adding a new resource or editing an existing one.

One other final part of the context data that is given to an extension, that was found to be of use is whether the extension is being used recursively. For example, a *comments* extension is simpler to implement if it doesn’t explicitly take care of comment threads and replies. This would make are comments on comments are possible and further recursively in the same way. It is much simpler to implement an extension that makes it possible to write a stream of comments, and then reuse that same extension within itself to achieve threads and replies (comment on comments). FAC will detect this and provide information to the reused extension that it is being used from itself.

In conclusion, the *context mapper* is an integration component that translates and aggregates various variables that contribute to the way the application executes and what it shows to the user. Usually, most common data like that resides in the session so context mapper mostly translates data from the host application session, but it also has a wider reach. In the future, when more experimental features and extensions are developed the context mapper might grow or even be split-up into smaller specialized mappers.

5.2.4.3. User data mapper

Different applications will have an infinite number of user properties, but there are some that can be assumed. The task of user data mapper is to provide those properties to the FAC and its

extensions. This will make it possible for extensions to keep track of authors of content they themselves allow the user to edit, provide additional features to authors, but it also makes it possible to label the author of, for example, a blog post or a published photo, etc.

Common user data properties from various frameworks and applications that are provided by mapper methods have been found:

- User's first name and last name;
- Username – the username that the user uses to log into the host application;
- Password – since passwords should always be hashed, user data mapper doesn't provide a way for extensions to read the password, since that should be impossible. It rather provides a way for extensions to check whether a given password is valid – mapping the checkPassword method to host application's login/check password mechanism;
- Email address of the user – so FAC and extensions can send emails to the user;
- Photo URL, that provides an absolute path to the url of the user avatar, so that extensions can use that in their views;
- User display name is often the username, or user's first and last name, but this is a convenience mapper, so that extensions can provide the same feel as the rest of the host application.
- Title prefix provides a way to map if there's a title that should or in some cases, could be printed in front of the user's name, for example prof. Mario;
- Title suffix has the same purpose like the previous property, but that's the part of the title that goes behind the user's name;
- User's public url is mapped so that the extensions can link to a public url of the user, if such is provided by the host application;
- User's private (edit) url is mapped so that extensions can link to a route of the host application where the user can edit his or her user account settings and data.

5.2.4.4. Permission mapper

Web applications allow or restrict users to certain actions or content through roles or permissions assigned to them. Applications might only distinguish registered or logged-in users versus anonymous users. Additionally, they might provide additional configuration or management features to administrators or super users. Depending on application complexity, permissions may be more roughly or finely defined and distributed to users.

FAC should be able to easily interface with different host applications, similar permission levels have to be translated into FAC permissions. Through analysis of some of the most popular frameworks, web applications and a proprietary content management system, Quilt CMS[13], FAC permission levels have been defined: read, write, author, editor, admin and godlike – from least permissions to the highest (godlike) level [12], [18], [19], [20], [21], [25].

Permission levels are defined as:

- Read – users are allowed to see a certain resource, for example a page with blog posts, photo galleries or some other content module. Users are not allowed to contribute, comment or otherwise influence the content.
- Write – users can write a limited amount of data, that cannot directly modify the original content on the website, for example, users can star or "like" a blog post, write comments to a published blog post or at most might be able to suggest some content for publishing, similar to contributors in WordPress.
- Author – users can independently publish new content on a specific web page, but cannot choose what that content is. For example, if an administrator configured a certain page to serve as a blog roll, then the user can just add blog posts using that active module.
- Editor – users can perform the same set of actions as with Author level, but also edit other users' content.
- Admin – users can independently change the structure of a site, add pages, add modules to pages and publish content through those modules. Admin users can also

[13] Quilt CMS, a proprietary content management system for the web, developed at the Faculty of electrical engineering and computing, University of zagreb, https://www.fer.unizg.hr/quilt-cms/

specify which permission level is required to view a certain page and assign additional permissions to users or user groups for those pages.

- Godlike – administrator users that can access every setting for that instance of the application.

5.2.4.5. Configuration mapper

Configuration of the host application is not a concern to FAC. That is also true vice versa, since the two are meant to be as decoupled and independent. In spite of that, there are a couple of properties that were identified that extensions might need from the host application configuration: whether the application can be accessed publicly or from a certain IP range or list of addresses, whether debug and development modes are enabled, and what error function should be used. Applications are often optimized for production, with various debugging or other development options and logging levels disabled to improve performance. FAC is the same, and provides a way for the developer to switch to development and/or debug mode – through

Configuration mapper's purpose is to more tightly integrate FAC with the host application, but ensuring the components themselves are as loosely coupled as possible. By making it possible to use host application's configuration to restrict access to all FAC endpoints, including API, makes it safer and also easier to maintain and control.

By making FAC use the same error handler as the host application it makes it more intuitive and easier to develop both extensions and the host app using the given setup.

Configuration mapper makes it possible for extensions to access even more host application configuration properties by providing generic getters. This may be useful if a configuration naming scheme should be defined in the future, for common properties. As it is now, generic getters are implemented, but no extensions should actually try to use them, since in essence it means hardcoding reliance to a specific host application.

Applications are sometimes configured to be inaccessible for various reasons. Static resources, especially public ones, and AJAX or RPC endpoints of the FAC or extensions also need to be deactivated. Mapping such configuration preferences is also possible.

Host applications can often serve multiple DNS names, provide the same functionality, customized or completely different functionality depending on that. Often, there is one main DNS name and that might make all the difference. By providing this configuration information to extensions, it makes customizations depending on DNS possible.

5.2.4.6. Event broker

Most web applications and web application frameworks have a mechanism to process events that occur in the system. Depending on technology, they are processed right away on a different thread, or sequentially at a certain point in time of the request lifecycle of the application. The most important property of events is that they are not expected to process real-time, and are usually easy to intercept and expand the way in which they are processed and dispatched. Having a reliable event processing in FAC is crucial to long-term data integrity and cleanup. Since extensions "hook" their data onto data they are provided by their parent components, which can be host application's components, a mechanism to communicate consequent data changes must exist. An extension might want to reset its own data, or somehow notify authors of its own data, that parent component's data has changed. For example, it might make no sense to show the same comments to a post if the post has been changed. It might have a completely different meaning or show different facts, to which old comments may make no sense. Alternatively, the extension might want to clean up its own data if parent component's data has been deleted.

Event broker is a component that translates events such as these from the parent component, whatever that may be, into a generic data event in the FAC, which extensions understand and receive once they are hooked onto the data type for the first time. It is similar to a hook in that it has the same parameters with one additional that indicates which CRUD operation has been performed. That means it can easily be invoked from the code that manages event dispatch of the host application, an extension when it manages its own data, or from a third party massage or event dispatch component. Third party component being the preferred way, as described in section 5.2.4.

5.2.5. Extensions

Most important manifestation of the FAC component model are extensions. The purpose of FAC is to create an execution context for extensions. Extensions are reusable three-tier

components built using the model-view-controller architectural pattern. FAC is the execution environment – the catalyst that creates the execution context for extensions, and it is a universal adapter for extensions.

Each extension manifests when the base class, provided by FAC - *\ExtFw\Extension*, is extended and then implemented. The class, once implemented, becomes a controller of the reusable component. All this implies that the controller is not the only part of the said component. Extensions usually have a database schema, models representing the schema, persistent storage, and views. Extensions are built by writing code for each of the three tiers, that is packed inside a single folder. The architecture of an extension is described using an example folder tree of an example implementation of a *Photo gallery* extension. In literature, this property of the component model is often called packaging [11] – a way to store and distribute components.

- Folders in extension root:
    - css – Contains any additional styling for extensions views. Might contain a single or multiple css files and additional resources like pictures or fonts, optionally in subfolders.
    - js – JavaScript files, usually written as separated code that provides additional dynamic functionality for extension's views.
    - lang – Folder contains PHP files that are named after a certain locale, for example hr_HR.UTF-8.php. The file contains only a single PHP associative array – keys with values, that are automatically registered as variables in extension's views, this makes it possible to localize extensions.
    - sql – If the extension requires database tables into which it will write data, SQL schema definitions should be written in SQL files here. Files named init-n-xxxx.sql will be invoked on extension registration, where n is a number greater or equal to zero, and xxxx is a custom nave given by the developer – usually a description what the SQL file contains.
    - templates – user interface templates that define views that the extension will render and show to the user. Templates might also include JavaScript code or include use any of the files from *css* or *js* folders.

- classes – Folder contains other classes or code constructs that implement the business logic of the extension. These are often classes that represent models and code libraries that implement the business logic the extension implements.

- Files in extension root:
  - Pgallery.php – the main extension class file, containing the class that extends Extension base class. In the MVC architectural pattern, this class represents the controller part of the architecture.
  - The root folder of the extension might contain additional class files, the same as classes sub-folder. This is supported, FAC will find those classes if they are a part of the extension namespace, but usage of classes sub-folder is a better option because it ensures better code organization and the main extension class is clearly the only file in the root.

Aside from the architecture observable through code, each extension object contains sub-components that contribute to execution of the composed whole, shown in Illustration 14.

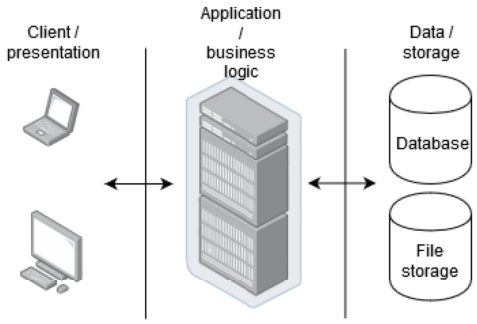

Illustration 14: Three-tier architecture

5.2.5.1. The ***presentation tier***

Each extension presentation tier, or sometimes called the client tier, is built through at least one templating object. It is instantiated and runs on the server, but once the template, or templates, specified to the object are rendered, the output will be sent to the client – the user's web browser, which will actually visually render it. Thus, the template object is the base component that builds up the view in the presentation tier, of the MVC architecture. Each template object, before being compiled and rendered server-side, gets pre-loaded by FAC with various contextual data by filling predefined variables, smarty plugins are executed and their

output is replaced in place of their invocation in the templates. This makes reuse of that data inside other components (as arguments), or showing that data to the user easy and uniform across extensions. It can also be the base upon which to build the extensions' JavaScript code, for example as an argument to initialize some JavaScript components that are part of the view layer.

Templating objects that are contained within extension objects are customized to work as a part of FAC. FAC instantiates those template objects and before the user-written code of the extension runs, pre-fills the template with common variables and data. FAC might also registers helper plugins, should the templating support that. Those will usually make it easier for extension developers to provide common extension components' functions, for example buttons that execute AJAX requests, automatically generate forms with CSRF (cross-site request forgery) tokens etc.

Code that is executed on the client can be found in the css, js and templates folders of the extension. Of those, CSS code and JavaScript are interpreted and executed on the client-side, while the templates are compiled and rendered on the server-side. In this case, rendering entails converting variables and plugins into final HTML, CSS and JavaScript, not actual visual rendering that will be done on the client-side by the user's web browser.

JavaScript that finally executes on the client's web browser can have in-built tokens or data, that make it possible to communicate with the application (server) tier. Integration of host application's JavaScript and extensions' JavaScript components is out of scope of FAC component model for now.

5.2.5.2. Application logic tier and data tier

The application logic, or business logic tier resides on the server-side. It connects the clients to the application server and data, compiles and prepares the views before they are sent to the client, runs all the other parts of the application and manipulates data that it stores or retrieves from the final, 3rd tier – the data tier.

Host application and FAC integration completely happens on this tier, so it is logical that the most important part of execution of extensions happens on this tier. Data that has been input from users through the client tier is processed, actions are performed to manipulate it and then it is finally stored on the data tier. The main extension class, \ExtFw\Extension, or rather the

Pgallery class that extends it in our gallery example, acts as the main controller for the component in the MVC pattern. It is also the main part of the component on the application tier. It invokes all the other components and libraries, and instantiates any models that might exist to abstract the input data. In our example, the model class for our example gallery extension could have a Gallery and Photo to abstract a group of photos and photos themselves.

The data tier is often taken care of by the framework or separate applications or components, instead of components implemented using a framework. FAC is the same, as it provides disk storage facilities to extensions, and provides database access. So, for our example, the component wouldn't have to directly choose where to write the photographs, but rather ask FAC for the path where to write them. This will also take care of restricting web access to those files. Saving information about files that have been saved and about the name or description of that particular gallery will be done in the database, that is the data tier. The database itself is often on a separate server reachable through the network, or is a separate service running on the same machine as the application.

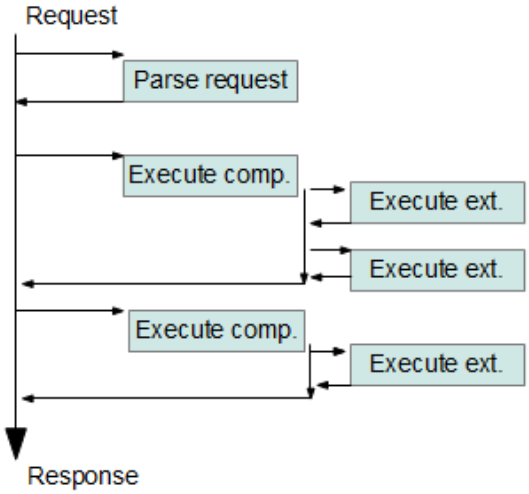

Illustration 15: Request processing flow

5.2.5.3. Extension lifecycle

Every extension lifecycle begins with a client request to the host application, which executes its components, of which some invoke the framework facade to execute one or more extensions. The sequence of a complete request processing of the host application, along with execution of extensions by host application components, is shown in Illustration 15.

This invocation of components to FAC facade to execute an extension transfers data to the framework which tells it what extension should be instantiated, what hook it is attached to, and then extension execution begins.

1. If the extension being executed is first one for that particular web request, FAC will compute the context through integration interfaces – mappers. This will prepare all the context data extensions might request and all the parameters that are required to instantiate and execute extensions.

2. FAC will search for an existing serialized object of the extension being executed in the session, because FAC saves extension server-side objects and their state between requests. If the session contains the extension object, it will be loaded.

3. FAC will determine if the extension is being run from within a form. For example, another extension or the host application component is in edit mode and it invoked the extension to also provide editing. For example, the programmer of a blog application wanted to provide gallery editing while editing a blog post in the host application, then the *defaultedit* method of the extension will be invoked.
   FAC will automatically detect that an extension is invoked from inside another extension's form, but host applications have to explicitly tell FAC that it is being invoked from a form.

4. If there was GET or POST data submitted for that extension, it is decrypted and used to invoke the right method of the extension object. It has to be decrypted because one has to use templating plugins (part of the presentation tier) to invoke actions on extensions, which is achieved through encrypting data being sent by the key specified in FAC configuration, usually through POST. If there is no data, then the *defaultaction* method is called without any parameters and the extension will render its default view. For our gallery example, that would be render photos in the gallery, or if there were multiple galleries created for that particular hook, show a list of all available galleries.
   It should be noted that there is almost always no GET or POST data for the extension if in step 3. FAC detected that the extension is running inside a form.

5. After step 4., control is given to the actual extension code that can then process input data, if there is need, persist the data in the data tier, prepare the view by specifying

which template should be rendered by the user interface templates objects, and providing it with data for template variables.

6. FAC compiles the extension's user interface and templates objects with both data provided by the extension and base variables and data that has been provided by FAC, renders the output and returns the output to the parent extension or the host application, which usually adds it to the output buffer and then when all processing is complete, sends the buffer to the client.
7. FAC saves the state of the extension object to the session.

This lifecycle is repeated for each extension, except for the first step which is performed only for the first extension invoked during a single request. So, the first step only runs once per request, other steps run for each extension instance, every request.

Once the request is processed and the host application sends the output buffer to the client, context mappers' data is discarded and rebuilt on the next request. Only the extensions objects are persisted through requests.

As far as session storage is concerned, it hasn't been optimized for the purposes of this dissertation. In the future, for any eventual production use, standardized components, such as Redis[14], could be used. This would ensure separation of host application and FAC session data, as well as ensure performance and scalability.

5.2.6. Extension manager

Another important part of FAC is the extension manager, or EM for short. EM is in charge of instantiating extension objects, persisting extension objects to session, retrieving them from session and executing required methods when clients send HTTP requests.

When facade receives the signal to execute an extension, the extension manager is invoked to actually create the execution context, instantiate an extension, parse parameters and execute it. Depending on the hook data a unique id, or instance id of the extension is computed and the extension object is instantiated. It is given the computed unique id, and it then gets registered by EM and saved to session. This makes it possible to persist objects and their state between

[14]Redis – in-memory data structure store, https://redis.io/

requests, and it makes it possible to identify which object corresponds to which hook between requests.

After the extension instance is created, the request is parsed by the EM that detects what data, if any, has been sent to the extension – by user input, by the host application component (or parent extension), or FAC. That data usually contains information which method (action) the extension should perform, and an array of arguments for that method.

There are a number of cases that can occur, which are of importance when executing an extension, and EM's job is to make sure each case is identified correctly. Correct functioning of extensions depends on this:

1. An extension is just invoked to render – display what it would by default, and no special arguments have been detected – no user input, no special settings, and the extension isn't being run inside a web form. In our example from section 5.2.5.3, this would be the case when a page with a gallery extension is rendered – and just a list of galleries or a single default gallery is shown.

2. An extension has submitted data from the client, whether as a single action, like a request to show a single gallery photo, or a complete form has been submitted, such as a gallery has been created and photos uploaded, or edited.
   In this case, the submitted data is encrypted inside a single POST, or sometimes GET request parameter. It is also possible that multiple extensions have submitted data at the same time, and that will be done in an associative array, or sometimes called arraylist. Each extension will have its data submitted inside a separate array, reachable through a key that will be set to the extension's identifier.
   EM will decrypt the submitted data, parse the array and detect which extensions have been sent some data by that sent identifier of the extension instance. Inside it, there will be a key "_name" which defines which action of the extension has to be executed, and the rest of the array will be handed to that particular extension method as arguments.

3. An extension has been invoked to render in *defaultedit* action for an existing hook. This would be a case when a blog component wants to show its form to edit an existing blog post and also show the extension's edit controls for the hooked gallery or galleries.

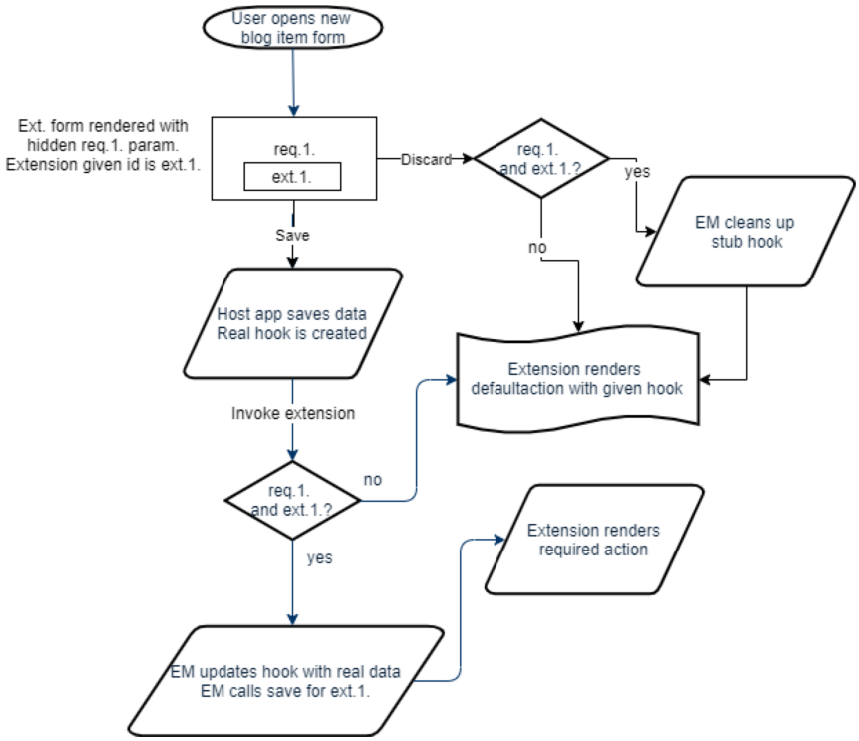

Illustration 16: Stub hook sequence

4. An extension has been invoked to render in "defaultedit" action for a non-existing hook. This is a case when a user is shown a form to create a new blog post. By that time there is no blog unique identifier to which to hook the gallery to, it is only known that the extension will be hooked to a blog post that will probably exist in the future. FAC handles this by recording request sequences and EM helps execute extensions by providing temporary (stub) hook data that doesn't exist yet (Illustration 16):

    1. Each request that is executed gets a sequential id. It is recorded in each extension object that is persisted in the session and it will also be rendered inside the extension web form that is shown to the user, as a hidden field.

    2. EM creates a unique hook by providing the request id and the extension id as the unique identifier for the data that the parent component specified.

    3. When a form from step 4.1. is submitted, the host app (or parent extension) will have to first take care of its own submitted data. In this case, the id of the extension that has been set by the parent will be of great importance. If the parent saves that data successfully, it will have to tell the extensions that it invoked that

"save" event has been fired. That event will automatically be tied to the previous request that resulted in form rendering and FAC will be able to tie it to the extension that was expecting its own data to be submitted as a part of the host application or parent extension form. This will be the job of EM. It will notify the extension that it will find by the id given by the parent, provide submitted data, tell it to perform the "save" action and then it will update the hook so that it references the actual parent's data. It will do this by concluding that the request was a save event, that the parent id for the extension is the same, that hook data changed – and that previously saved hook data was labeled as a stub.

4. If the form is submitted but the event is "discarded", the hook created by the previous request that created the form will be deleted. If the form is never submitted, FAC will clean up all orphaned stubs. It will find them because it records a stub hook with additional metadata: the session id in which it was created and the expire timestamp for that session. If the stub wasn't submitted and either saved or discarded by the end of session, it should probably be discarded.

When an extension is hooked to the parent, FAC automatically subscribes it to events for the hooked data. If the extension has never before been hooked onto the data type that the hook includes, Extension manager will create that subscription.

Extension manager is also in charge of **registering and unregistering extensions**. Once an extension package is installed – a folder created in the extensions subfolder of the framework, it needs to be registered with the framework before it can be used. This means saving the extension metadata into the database and subscribing the extension to "saved" and "discarded" events.

5.3. Limitations

Components can be monolithic or big and can have a significant number of functionalities. Components might not necessarily be big and monolithic in terms of lines of code, non-existent sub-modules, or because they are spanning multiple tiers of application architecture, but in terms of providing more functionality inside a single component (extension). Since every extension is potentially as big as a self-sustained application, building applications from application-sized components can be hard, and the size and a well-

rounded functionality of a single extension depends on the skill or prudence of the component designer and developer.

Visually, components built using FAC have a different look and feel than components of the host application. Conventions would be needed to ensure UI-level uniformity of design and integration. This has not yet been included in the component model, but another simpler approach might also be acceptable. Since user interface styling frameworks for the web are mainly dominated by Bootstrap[15] and Material design[16], a convention might be put in place, that requires developers to build main templates using both those two frameworks.

Retrieving aggregated data or reports can be slow without tight integration of components. Since every component defines its own data structure, database schema, it communicates data in a generic way through generic interfaces of the FAC.

FAC itself becomes a **critical bridge component** because it, as a universal adapter, connects every two components in the system.

Component nesting might be slower than actually handling a tree data structure directly from a component. Having the ability to nest and recursively use components might leave space for extension developers to overdo it and hinder performance.

Creating links to parent or child component views, dynamically and statically, is a challenge. Any form of actual coupling would involve more data traffic between components, than generic data hooks. This might be of interest if an extension functionality would require that it notifies the user of some event, and then provide the direct link to the web – the url where it could render the data. For example, a *Comments* extension might send an email to the user when her comment receives new replies. This would both mean child communicating with the parent and generating a URL that would be host application aware and signal all host application and FAC extensions correctly to show everything needed to load the right page, activate the right components in the right way, to activate the right *Comments* extension, so that it can show the comment in question. This is a research topic in itself, and is an area that should be addressed with future research.

[15] Bootstrap – world's most popular front-end component library and responsive design framework

[16] Material design – Google's front-end component library and responsive design framework

FAC component model is technology-dependent, if the framework was to be implemented in Java, it would be possible to develop Java applications and interface with Java host applications; the same is true for PHP, C#, … This shortcoming could be overcome by modifying FAC so that it can communicate with extensions and host application over web services. A single implementation could then work with any other host application, that doesn't even have to be on the physical or logical server. This is also a research topic for future work.

6. BUILDING SOFTWARE USING FAC FRAMEWORK

In this chapter a FAC framework prototype is introduced, and a method to build web application using it is explained. The process follows best practices identified for software engineering, introduced in chapter 2.1: software requirements definition, software design, software construction and software testing.

First the requirements are made. For requirements of FAC, component model defined in section 5 is used. After constructing and evaluating parts of the prototype, and identifying required shortcomings, changes were made both in the definition of the model and subsequently the implementation. This correlates to best practices identified in literature. For example, [3] states that fundamentals of software construction include, among other things, constructing for verification and expecting change. Even already in 1970, the Royce Waterfall Model acknowledged that each of the stages of software lifecycles had possible next steps in both directions – up and down the waterfall. Current agile methods all profess the same iterative repetitive processes of design, implementation and evaluation and testing [54], [69].

In the first stage of design, the choice of technology which will be used to build the prototype framework is done. PHP is chosen because it is a solid mature platform, and is the fifth most popular programming language according to GitHub's PYPL[17].

The choice of PHP also made it possible to integrate FAC with WordPress and Quilt CMS – WordPress being the most popular web application content management system currently in the world, while Quilt CMS is a custom made content management system, built at Faculty of Electrical Engineering and Computing at the University of Zagreb. If the most popular open source CMS and a custom web application can be integrated with FAC and software can be built, then FAC model works and it is possible to apply reusability metrics.

In the second stage, folder structure for FAC implementation was made. By creating the structure, **packaging** of FAC framework was also solved. The complete folder structure makes up the package to deploy.

[17] The PYPL PopularitY of Programming Language Index is created by analizing how often language tutorials are searched on Google, http://pypl.github.io/PYPL.html

In parallel to packaging, during the same design phase, **deployment** steps were specified. Provided integration mappers exist for the host application, deployment steps are:

- Copy the FAC framework to a folder on the web server;
- Create the database for FAC;
- Fill out the configuration Config.php file;
- Set-up the web server to serve the FAC public folder and give it access to the rest of FAC folders with PHP code;
- Install the host application component that provides FAC integration and reuse.

It is possible to use FAC through a WordPress *shortcode*, and this is the last and most important step. Shortcodes are parts of main WordPress components called *plugins,* defined by the WordPress component model. WordPress in itself was built as a blogging application, but through community-contributed plugins became the most popular content management system in the world, with a list of features (obtainable through plugins) that can hardly be matched. Integrating and building WordPress is described in more detail later, in section 6.3.1.

6.1. Method for building software using framework as component model

Software built in the previous two sections, using WordPress and Quilt CMS, have gone through the same standard design and construction phases. Design has been performed on all the participating software parties – the final required integrated applications, host applications, FAC implementation and extensions.

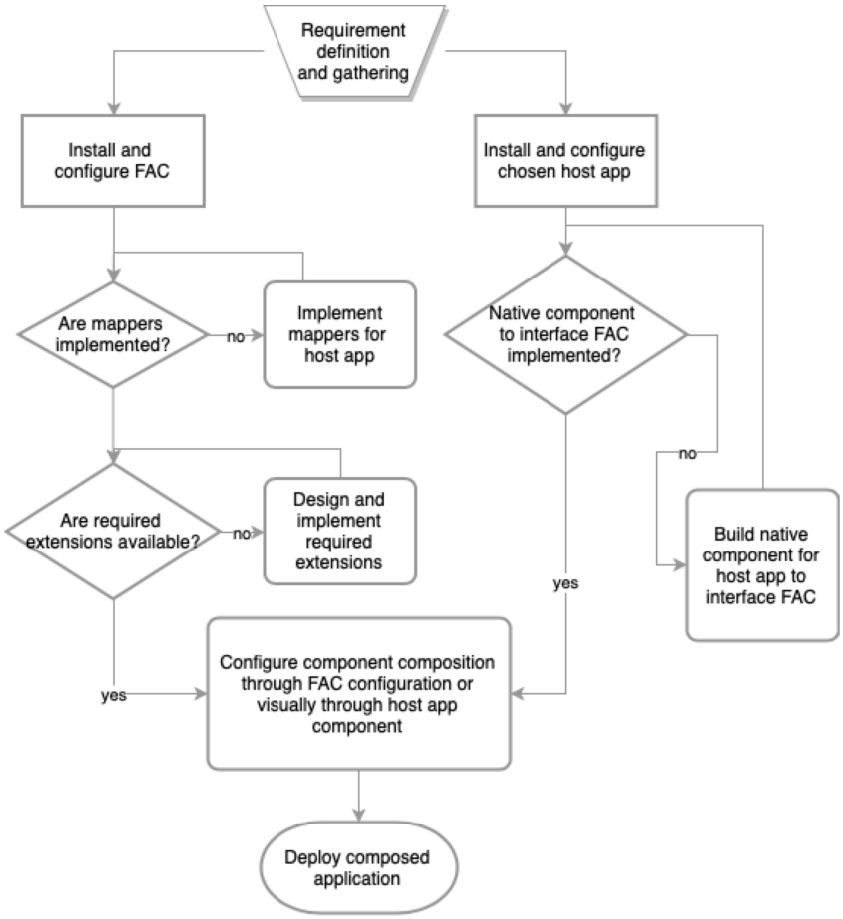

Illustration 17: Building software using FAC

The high-level overview of the complete process is given on Illustration 17. The process begins with requirements gathering and definition for the final software product (**step 1**).

Step 2: Through both functional and non-functional requirements, the design of software can be made. Components, functionality and data can be designed. During that phase, a host application should be chosen to provide part of the final software functionality, while the rest will be done through FAC components.

Step 3: The process goes on with the installation of FAC framework and the host application, which can be done in parallel.

Step 4: Next, both the native component for the host application, that is built using the host application component model, and the mappers for the host application in FAC framework have to be implemented. This step has to be done only once per host application, for example, once for building software with WordPress and FAC. Every consecutive implementation of

software with WordPress and FAC can skip this step. This step is described in more detail in section 6.1.2.

Step 5: When a working integration of the host application and FAC has been achieved, extensions that implement the required functionality have to be designed and built. It has to be noted that implementation of each extension is a project in itself, because it has to be built for reuse, described in more detail in section 6.1.3.

Step 6: Composition of components is the final step, and the main step in component – based software engineering is building software by reuse. Components may be composed statically through FAC configuration, or dynamically, by the end-user through the native host application integration component.

Step 7: The final step is not really related to building software, but it will usually follow software construction activities: deployment of the composed application into production.

6.1.1. Design principles application

Design principles should be applied when building software using FAC framework, but the method introduced in the previous section clearly shows that there are only a few steps when software design activities occur.

First, design activities will occur when the final software requirements are being collected and transformed into a specification, a plan how to achieve them – what host application should be used, what component should be made and how they should be composed.

Additionally, design activities will occur additionally as many times as there are specific extensions that have to be implemented to achieve the functionality that was planned during the first, high-level design of the software. For each extension, all the design principles will also be employed.

Abstraction is the first design activity. Each requirement was divided into components and a high-level overview was made of components that will be assembled to deliver the required functionalities. Data that each of those components will manipulate was also abstracted, like components, photos, galleries etc.

Abstraction, when performed for each of the extensions, will go into detail of how to build the extension from which sub-components, how to abstract the data etc.

Coupling and cohesion of components was considered next. To minimize coupling and keep the cohesion level as high as possible, FAC component model was chosen for non-native host application functionality. Cohesion of built extensions can be predicted to be very high since every extension is feature-complete in its domain.

Coupling and cohesion is evaluated for the internal structure of each of the extensions. How subcomponent interfaces are used, how subcomponents are cohesive etc.

Decomposition and modularization was done in parallel to coupling and cohesion planning – the whole of application was divided into submodules, and components, so that parts are replaceable and reusable as much as possible.

Each extension can surely reuse certain smaller components, for specific domains of use, for example, file manipulation, graph drawing etc.

Interface and implementation were clearly separated, for non host application components, simply as a result of the FAC component model. All the extensions have a uniform interface, through the universal adapter that a FAC framework is, which is a set of separate components (mappers), from the actual universal adapter that is being used.

This is the only design principle that doesn't have to be considered for each extension, since FAC enforces that extensions are used through uniform extension interfaces.

Sufficiency, completeness and primitiveness are enforced through the use of FAC framework. Components provide only basic data types for each other's interfaces, such as strings and integers.

Concerns were clearly separated in the final software implementation. The host application takes care of session handling, users, and basic web application functionality, with the addition of handling pages and articles, while separate components (extensions) in FAC provide all the other functionality, that can be reused even recursively – of which the components themselves are unaware.

Extension designers do have to take care of separation of concerns, for more complex extensions, that provide a higher number of functionality, additional subcomponents or external dedicated components should be used to lower and manage complexity.

6.1.2. Integration of FAC and host application

After performing design on the higher level, prerequisites have to be taken so that FAC extensions work, and so that it is possible to compose the final application. This means that FAC needs to be integrated with the host application. The only design activities that have to be performed are related to building the host application component.

To build this component, one has to consider the functional design of the host application, and the lifecycle of a request in the host application. Those are important because the integration component needs to receive host application's events and transfer them to the FAC framework.

In addition, the integration component needs to be invoked when host application components are rendering. It is possible to choose the way in which this can be done:

- Directly call exact FAC hooks from host application components for certain extensions (this will make integration tightly coupled and is the least desirable solution). This is possible only for white-box host application components;
- Create only named hooks without extension assignments, so that extensions can be added later statically or dynamically. This doesn't have to cause tight coupling of host application components and FAC, because the added hook may be added with a test if the FAC facade class is known during execution time;
- By the host application core (framework) when host application components have rendered, and additional processing can be performed upon the host application component output;

So the **first step** when integrating FAC with an arbitrary host application is to create a native host application component that (a) will interface with the FAC framework, the universal adapter, and invoke hook execution and add the output of extensions where appropriate inside the host application; and (b) react to host application events and forward them to the FAC framework.

In the **second step**, or rather even if there are two developers working on integration of FAC with a host application, in parallel to the first step, all the mapper components for the host application have to be built.

Since mappers are just classes/components that reside inside FAC framework, this is done by naming them in the required way and implementing them in the previously specified location (more on this in section 6.2.2). The classes need to implement interfaces prepared inside FAC framework, so the work is straightforward. Each of the methods in those interfaces retrieves a chunk of host application data. What each of the methods should retrieve from the host application is described in the code documentation in the interfaces' files and in section 6.2.2, which makes it a highly repeatable process for different host applications. Although repeatable, to be able to implement the mappers for a host application, one should be acquainted with its architecture and functional design.

Finally, the native host application component should be able to initiate the session, if the host application doesn't use it.

6.1.3. Building extensions

Building a single extension entails all the steps taken on the higher level of the application, while designing and afterwards, when building the software. A single extension can be just a small step in the implementation of the whole software, but from the point of view of the extension builder, it is a project in itself. In fact, as is mentioned in the introduction of this thesis, an extension can be considered a whole application in itself. Additionally, an extension is an application that provides an atomic and functionally complete functionality, unburdened by the rest of functionality provided by other components inside the application.

A software engineer, the builder of an extension, should keep in mind that each extension is being built for reuse. Since the component model enforces this, the engineer should just follow the component model to not tightly couple it with any other extension or host application functionality. It should be as self-sufficient as possible, with a clearly defined purpose. Concerns about the problem domain that the extension is solving should completely be encapsulated inside it, and the data it manipulates should be completely processed within the extension or one of its sub-components.

The lifecycle of an extension resembles a typical component lifecycle, described in section 3.2 and in Illustration 4 from that section.

During the **modeling phase**, a specification for the extension is built – description of how it should work, models can be made, and plans and decisions to reuse other extensions as sub-components can be made.

During **implementation / construction phase**, a software engineer has to keep in mind that an extension should be built for reuse. That means, all the usual methods introduced in section 2.1.3, for example most importantly:

- construct for variability through parametrization, but encapsulate that variability;
- write code comments and documentation;
- describe an extension in detail, provide enough metadata and extra-functional properties description;

Specific to extensions, FAC framework, extension developer should prepare empty named hooks in logical places in the presentation layer of the extension, so that it is easier to statically or dynamically configure child extensions in the future and use the extension to build by reuse in the future. These hooks will then be populated statically by *getStaticComponentBinding* configuration property of FAC, or visually by the user.

Once the extension is implemented, it is **installed** if basic extension metadata is written (registered) in the FAC database table *extensions*. **Installation** to another instance of FAC framework is possible by copying the folder and adding the metadata to the FAC database.

Error handling of extensions should be well thought-up. An extension should never stop the host application from running. FAC will try to make sure that this doesn't happen, but the extension developer should carefully consider possible error states, and apply most common fault-tolerance strategies like backing up and retrying, using auxiliary code, replacing an erroneous value with a phony value that will have a benign effect etc. [3].

Currently, **package of an extension** is the folder in which it has been developed. The package consists of all the possible subfolders and code files described in section 5.2.5. To install it into another instance of FAC framework, one would simply copy the folder into *framework/user* folder and write the extension metadata into the FAC database.

6.2. FAC framework implementation

To build FAC, standard software engineering process was followed. Requirements of the software were defined for the model. Then as a part of design process, component definition was made (chapter 5), alternating with software construction activities. In that way, some specifics of the model were also found and defined – by implementing and immediately testing it. By immediately constructing and testing validity of the model, wrong presumptions were detected early and changes to design could be made immediately, fixing the model and then immediately fixing the test implementation. This has proven to be the best approach, and only further confirms findings from previous research, that a framework that is used to build components is one of crucial parts of component model definition [6], [11].

During the design phase, **design patterns** and existing components that could be reused were identified and chosen for implementation. The framework main class, the one which is the actual universal adapter for any reusing component, is made using a **singleton pattern**. Access to it, and every other functionality of the framework, like the events interface, or log is accessible through the Facade class, which implies the **facade pattern** was also used. Events interface also hints at **event-based architecture**. A **factory pattern** was chosen for retrieving instances of mappers and the **mediator pattern** was employed to design interactions with actual extensions, through the ExtensionManager class, which is actually a mediator component for extensions. It was named differently because "managing extensions" sounds more intuitive.

Extensions must be implemented using the **model-view-controller** pattern, when the 3-tier architecture component is built, which doesn't limit the extension developer from implementing various architectural patterns and making different design choices for each of the layers in each of the tiers.

Components that were reused include **Smarty** templating library, while the main Smarty class was extended to provide functionality and data specific to FAC. The **whoops[18] error handler** was used to provide visual debugging aid. Doctrine database abstraction layer and Doctrine object-relational mapper are also included and can be used if extension developers are more

[18] Whoops – a popular error handler for PHP, that formats notices, warnings and errors in a developer-friendly way, http://filp.github.io/whoops/

fluent in a standard database abstraction library like that, then FAC's own SQLQuery class doesn't need to be used.

Composer package manager, a standard modern PHP package manager, is used to install those third-party components.

Depending on the configuration and requirements of extensions, FAC also requires some standard PHP libraries that usually need to be installed separately: **php-mcrypt** to handle encryption algorithms, required for cross-site forgery protection through request parameters encryption; **standard MySQL, PostgreSQL and sqlite libraries**, to connect to those database systems.

6.2.1. FAC framework root folder and components

The folder structure (defined in section 5.2) of the framework is designed so that only the *public* folder is served by the web server, as not to expose any data or code. Other parts of the framework implementation are separated into other folders and sub-folders. The separate *public* folder also makes it possible to easily serve static files, like *CSS and JavaScript*, while protecting other assets in inaccessible directories.

The **root folder of the framework**, contains both files and folders:

- framework – all the core framework code;
- opt folder – components that provide integration with host application, like WordPress plugins, that are built according to that host application component model, and should be integrated into it to provide FAC functionality;
- templates_c folder – Smarty's cache folder for compiled templates;
- user folder – folder intended for extensions and possibly, in the future, other FAC instance – related components;
- vendor folder – used by composer package manager, to install third-party components;
- readme.md – this file is a common practice, containing all the base important project metadata, like the author, repository url, description etc.
- index.html – an empty file, created to prevent folder listing by a poorly configured web server. Files like these are added inside almost all directories, for the same reason.

6.2.2. FAC framework folder and components

The *framework* folder contains all the FAC code, from extension base classes, and FAC components that execute them, to integration components like mappers, spread out in the following directories and files:

- lib folder – all the core code, organized into namespaces and classes, that implement FAC functionality, such as database abstraction, extension base class and extension handling, mappers for various host applications …
- public folder – folder that is served by the web server, e.g. Apache;
- Config.php class – framework configuration, such as specified special folders, database access configuration etc.;
- framework.php – FAC bootstrap file that has to be loaded by the host application;

Config.php is a configuration file of FAC. Currently, it is possible to configure the database access, to which FAC will provide read and write access to extensions. Sqlite, MySQL and PostgreSQL are supported, and the usual properties required to connect are configurable: the database, host, database name, tables' prefix and database password.

It is also possible to configure the encryption key, which is used to protect against cross-site request forgery (CSRF) – method of embedding encrypted data with every request which can be decrypted by the server using the secret key, when the form is submitted, thus making sure that the form that submitted the data was actually one generated by the server in the first place.

Setting a flag whether FAC should be working in development mode which triggers detailed errors and stack traces to be rendered in the browser, if an error occurs. It is also possible to specify 3 log levels – off, production and debug.

It is possible to specify where important folders are located. For example, the user folder doesn't need to be in the pre-defined location. Neither does the public storage folder, which contains files created by extensions, that should be publicly available, or the private storage folder, which contains files created by extensions, but need authorization checks by extensions before they are downloaded.

All of FAC and extensions' session data is saved inside a single session variable, which is named in the Config.php file.

Config.php also makes it possible to define default extension composition and default host application content type composition. For example, when building software with a host application and FAC, one might want to assemble everything in advance, so that certain extensions are always run for certain host application content types (usually that translates to host application components), and also that hooks inside extensions invoke other extensions by default.

Enforcing loose coupling for hooks, through *generic hooks*, as defined by the component model in section 5.2.3, is possible through configuration. In the future, FAC hooks might provide functionality for the user to choose which extensions a generic hook should run, but currently this can be done by the programmer while integrating FAC with the host application. The *getStaticComponentBinding* property defines an associative array that defines which extensions should always run for hooks that expose a certain data type. For example, for every hook that receives the data type *Post*, a *Comments* extension should run. Additionally, since hooks can have ids set, that make it possible for a single parent component to have multiple extensions of the same type that is hooked to different data, and shows different data (introduced in section 5.2.3), it is possible to specify those 'named' hooks in the configuration. This makes it easier to create a more detailed, and more finely controlled integration with the host application. The process of which is described later, along with examples of defined *getStaticComponentBinding* properties, in chapter 6.

Finally, the name of the host application can be set in configuration, and depending on the name, FAC will search for corresponding mappers to use when executing.

The **index.html** file inside the root folder is empty, added to prevent folder listing by a poorly configured web server.

Public folder is meant for public web server access. It may, for example, contain a subfolder named storage – which extensions are allowed to write to, if they need to store files that are publicly available. There is just one file of note inside it, called ext.php. This is a FAC XHR[19]

[19]XmlHTTPRequest – used to asynchronously transfer data between the web server and web browser, without reloading the whole page, https://en.wikipedia.org/wiki/XMLHttpRequest

endpoint, that responds to requests sent by extensions. All XHR requests also need to contain valid CSRF tokens, that is, *ext.php* is an API endpoint of FAC.

All the core functionality of the framework is in the **lib folder**, which in the future might get more children folder, but for now everything resides in the subfolder named **core**. It contains namespace folders which in turn contain classes or namespace folder hierarchies respectively.

6.2.2.1. The framework/lib/core folder

FAC's classloader, or sometimes in PHP world called *autoloader* resides inside the **autoload.function.php** of the core folder. It will know how to find a class and load it when some module tries to use it, in any of the FAC framework's namespaces, or if it is a part of any of the components that were installed through *Composer*. FAC classloader first tries to find the class inside FAC, and if that fails, searches for it using a composer-generated classloader. If that fails, an error will occur.

Classes subfolder

Main framework classes reside in the **classes subfolder** of core, and their namespace is implied to be *Fw*. Every other subfolder of core implies a sub-namespace of *Fw*. For example, folder ..core/*Mappers* will contain classes in the *Fw\Mappers* namespace.

Classes subfolder contains the singleton **Framework class** – the universal adapter that gets instantiated once per request. It is through that class that each extension execution is executed. Previously introduced framework.php file's mission is to actually load up everything and create the singleton instance of Framework. Once that is done, the framework is bootstrapped.

Both **Extension** and **ExtensionManager** classes, introduced in section 5.2.5., are implemented and stored in this folder. The custom ExtException exception class is also implemented and stored here, all easily reachable as main parts of FAC framework.

Log is a simple class, implementing FAC framework logging functionalities, depending on configured log level, outputs log or debug messages into configured output.

Facade classes also reside here in the *Facade* subfolder, for now only the Fw class, that has multiple functionalities: initializing the main Framework singleton instance, executing extensions, retrieving correct mappers instances for the current host application, and receiving events emitted from host application or parent component / extension.

Db subfolder

This folder is intended to contain all the database abstraction classes for FAC framework. It currently provides a base abstract class called DBConnection. It serves as a template for each of the classes that implement support for each of the databases: *Sqlite*, *MySQL* and *PostgreSQL*, through **SqliteConnection**, **MySQLConnection** and **PgSQLConnection** classes. This is also the folder where one would, in the future, implement any additional integration with a database abstraction or provider libraries, and expose it to the rest of FAC and extensions, to use. Currently, only additional integration with Doctrine. Although simple and with limited support, it is there as a template, inside the **DoctrineDBALConnection** class.

Finally, this folder also contains the **SQLQuery** class, the simple class to execute queries and fetch results through any of the connection classes. SQLQuery will use a corresponding connection class, depending on which database is configured in Config.php file.

Events subfolder

Support for events in FAC is implemented through classes in this folder. Both extensions and host applications use these classes, through the Facade.

Base **Event** class resides in this folder. As is convention in event-based architectures, events should have names and are usually named through class names. This is also the case for FAC, and default events that happen on hooks are called through class names **ContentCreated**, **ContendDeleted** and **ContentChanged**, located in the *content* subfolder.

The event class is also a representation of the entity that FAC will write to the database before it starts dispatching it to all the listeners. Depending on the class name that extends the base Event class, name will be written.

Every extension should implement the **IEventListener interface**. FAC will expect that it exists so that it can uniformly notify extensions of events.

EventManager, is the FAC's event broker. It is the class that takes care of noting new event subscriptions to the database, handling fired events (writing them to the database, then marking them as processed when all listeners have consumed events), and also subscribing listeners to events. ExtensionManager helps in one part, when an extension gets connected to

a hook, it tells the EventManager to make sure that the extension will receive any future events that happen upon data of type specified in the hook.

Future work might include creating a new type of events that can be processed asynchronously.

Mappers subfolder

In this folder interfaces that a mapper for any host application must support are defined: **IConfigurationMapper**, **IContextMapper**, **IPermissionMapper** and **IUserDataMapper**.

Each of those interfaces is accompanied by a factory class inside the Factory folder: **ConfigurationMapperFactory**, **ContextMapperFactory**, **PermissionMapperFactory** and **UserDataMapperFactory**, each of them responsible to instantiate the correct mapper for the configured host application. This way, none of the components using the mapper don't need to know which is the actual type of the mapper they are using.

And finally, other subfolders in mappers should be named after each of the host applications that are supported, matching the mapper namespace name. That means that it is expected that in case of future use of FAC, the number of folders might grow. Currently, there are mappers implemented for WordPress and Quilt2.

Smarty subfolder

Custom FAC Smarty class is implemented here. It extends the default Smarty class, and by default, on instantiation, sets-up the object so that it looks for Smarty plugins in the **fw_plugins** folder, looks for templates in the extension's templates folder for which it was created, and sets-up some of the variables each extension template must have when it is processed. Examples of these variables are language resources, the title and data that the controller (extension object) assigned to the template.

Smarty plugins which make FAC tick, inside fw_plugins folder are:

- block.ext_form – renders form open and close tags, it expects arguments through which it tells FAC which controller action to call with submitted data. It also adds a CSRF token to be submitted with the form;
- function.ext_action – renders a link that, when clicked, runs a specific controller action of the controller;

- function.ext_actionpost – renders a link that does the same thing as the former, only through a hidden form, using POST method;
- function.ext_async_action – renders a link that submits a XHR on click, and invokes specified controller action, invokes a specified javascript function on XHR response;
- function.ext_form_async_submit – like the former, but it submits the whole form in which it is placed;
- function.ext – provides extensions with a simple way of invoking the plugin, in the background it creates a hook using provided arguments.

UniqueID subfolder

For each hook a UniqueID object is created and saved to the database. Those objects are represented by the UniqueID class, that makes sure that for each hook data a new object is created, or an existing one is retrieved from the database.

Each content type that a hook gets will also be enumerated into the database. This is the task of the **ContentManager** class. It also serves as a mediator between UniqueID objects and extension objects. Enumerated content types are also of use to the event manager, that

UniqueIDException and **ContentTypeUnqnownException** classes are also present in the folder, each representing an error state that may happen in the application.

Util subfolder

Purpose of the util subfolder is to provide common utilities and services to extensions, for example file uploads or image manipulation. Accordingly, currently only **Image** and **FileUploadHelper** classes exit.

6.3. Building software using FAC framework

Building software always starts with a need that is turned into a software requirements definition. In real-world scenarios requirements definition is done in detail, through definition of user stories and business analysis. A simple outline has been defined for the sake of this example:

1. Users need to be able to log-into the application;
2. Users need to be able to create web pages and publish text and multimedia content;

3. Users need to be able to publish news articles;
4. It should be possible, for every news article:
    4.1. To attach a photo gallery to it;
        4.1.1. It should be possible to comment to a gallery and / or photo;
        4.1.2. It should be possible to comment on comments;
        4.1.3. It should be possible to like a gallery or photo;
    4.2. It should be possible to like a news article;
5. Multiple permission levels for users should be available;
    5.1. Some users should be able to publish, edit and delete their own content;
    5.2. Some users should be able to publish, edit and delete their content and other users' content;
    5.3. Some users should be able to only suggest content for publishing, and wait until a user with greater permission publishes it.

As part of requirements definition and analysis, software that can be reused must be identified. From analysis of requirements 1 through 3, it is possible to conclude that WordPress is a good choice to base the solution on.

6.3.1. Building software using FAC framework and WordPress

WordPress may also solve 4.1, but not the rest of 4.1.x sub-items. Both 4.2 and 4.3 aren't possible with WordPress, but all the items under requirement 5. are. There are at least two options to solve perceived deficiencies, in terms of satisfying stated requirements.

The **first possible solution** might be to modify WordPress core, and add missing functionalities from 4.1.x, 4.2, although some part of it probably could be solved through plugins, so plugins could be developed or installed. By developing plugins, components that can only be used by WordPress are created. In addition, there is repeating functionality that should be applied to articles, comments, galleries and photos in galleries – commenting, assigning licenses and liking.

If similar requirements emerge in another system, components built will not be readily reusable, especially if it's not a WordPress based system. Or, if at some point a sort of product catalog is needed, that users should be able to comment and like products, it would have to couple comments and like components with the new cart and catalog. Comments and likes would be modified to interface with products, or a separate WordPress component would be built to mediate between plugins.

The **second solution,** is to use WordPress functionality and use FAC framework to create extensions to achieve the rest of stated requirements. After extensions are implemented, a generic add-on functionality will be available that can be used in other instances of WordPress, with other components, or with other host applications.

The like button extension should be attachable to the parent component, and it should enable the user to like the parent component's content. It should also count the number of users that have clicked the like button for that content, and show the counter. To do that, it should remember every single user that clicked it, because a user can only like content once, whether it is an article, a photo or a gallery. The user should always see if she clicked the like button, and the overall count of likes.

The comments extension should allow users to write comments to content the extension is hooked to. It should be possible to write an infinite number of comments and for each one of them, it should be possible to like it, and to write comments to the comment. This way comments would work in a hierarchical way, as threads with sub-threads. For each published comment, the author of the comment should be shown, at least using their name and a link to author URL.

The gallery extension should allow users to create a photo gallery with a title, an infinite number of photos, each with its own title. The gallery should be attachable to the parent component. It should be possible to like it and to comment it, as well as any of its photos.

6.3.1.1. FAC extension HelloWorld component structure

To show how extensions are built using FAC framework, a HelloWorld extension implementation is analyzed. The implementation of HelloWorld extension resembles to how the Quilt CMS HelloWorld portlet component is built, introduced in section 3.6.2.

As previously explained in section 5.2.5, a folder for the extension must be made, with the name of the extension namespace, inside a folder where it will be found by the framework. It consists of code files for the same purpose, and is similarly structured. The main difference is that FAC translation files are JSON, while Quilt uses PHP files to define arrays with translation strings.

The extension folder may contain model classes inside a *src* folder, user interface definitions inside a *templates* folder, language translations inside *lang* folder as well as JavaScript and CSS inside *js* and *css* folders.

Software engineer that is building an extension shouldn't have to care about routing, because it will be taken care of by the framework itself.

Routing is taken care of by FAC, thus a component builder isn't required to write route definitions.

In the extensions folder "*HelloWorld*", a *HelloWorld.php* file containing the main extension file with *HelloWorld* class that extends the base extension class *Extension* was made:

```
<?php namespace HelloWorld;
class HelloWorld extends Extension
{
 public function defaultAction( $input ) { $this→setTemplate("hello.tpl");
}}
```

When the extension is executed, FAC will invoke the *defautlAction* method, which will set-up the template *templates/hello.tpl* for rendering, containing only the invocation to render a Smarty variable *hello*:

```
{ $lc.hello }
```

This will in turn cause loading of a language string from the language JSON file *en_US.UTF-8.json*:

```
{  "hello": "Hello world!" }
```

Since FAC requires a host application, upon successful integration, for example with WordPress, to activate the Hello world extension, an engineer could write the static component composition as explained in section 6.3.1, or use the shortcode in a post content, or content of a page in WordPress:

```
[extension name="HelloWorld\\HelloWorld"]
```

6.3.1.2. WordPress theme

Many functionalities, when building custom software using WordPress, are implemented through the theme, and the *functions.php* file that is a part of it. A theme is not just the design theme, but rather an integration component of sorts. One could observe a WordPress theme as a mediator component. It defines how WordPress components will render, and basic additions to the definition of how they behave can be done too. This makes it possible to, for each WordPress component that renders, add some more processing, attach some custom components, or modify WordPress component's output before rendering it. For example, a theme will define how a WordPress Page renders, where the metadata about the page, comments and other default sub-components will render. For those reasons, custom WordPress applications almost exclusively entail custom themes. Or rather, most WordPress implementations start with theme implementation.

This is also our first possible choice to start implementation of FAC – WordPress application implementation. This would make it possible to specify exactly where hooks would be placed for WordPress pages and posts. In addition, hooks could be named through hook ids.

Each hook id will get prepended by the framework, by adding the name of the content that is given to the hook during run-time and the id of that content. In the end, if the component calling the hook specified content type as *post*, the id of the post *12* and the hook id *comments,* the hook's id will be computed and internally look like: *post12comments*.

This would make sure that for each post all the three hooks are unique, and it would be possible to attach a different gallery extension to each of those. This would also allow control where in the structure of a rendered post the extension appears, through placement in the theme templates. Having hook ids makes it easy to write the *getStaticComponentBinding* property for the FAC framework instance integrated with WordPress. For example, if array keys are defined as named hook ids:

```
'comments' => 'Phd\Comments',
'like' => 'Phd\Like',
'gallery' => 'Phd\Gallery'
```

This means, for each hook with given id of *comments*, hook up and execute the *Phd\Comments* extension, for hook with given id of *like*, hook up and execute *Phd\Like* etc.

Static component binding can be done in even more detailed way. We can specify the parent content type name, along with the extension, or extensions, which to hook up onto such a hook. For example:

```
'post?comments' => 'Phd\Comments'
```

would define that for each hook with given id of *comments*, that gets the content type of *post,* FAC should execute the *Phd\Comments* extension. The '*?*' character will be replaced by the id of the content given to the hook.

6.3.1.3. WordPress plugin

To extend WordPress functionality the ideal solution would be to allow the developer, or user, to invoke the FAC anywhere in the WordPress code, in theme template or by the end-user while writing content in the user interface. To achieve this, a WordPress modularity default entity, a plugin, has to be built. The plugin will then interface with FAC. A plugin can add functionality to both WordPress frontend and backend. Plugins are capable of extending functionality without any custom code being added to the WordPress theme, which makes coupling lower.

So, an ideal solution to build software with FAC and WordPress in terms of customizability where exactly extensions appear in the view, would be building both the theme and enabling the plugin. But a more generic and decoupled way can be achieved by using only the plugin, so this is the approach taken.

For production purposes, a comprehensive WordPress plugin that would allow the user to dynamically manage, in a visual way, what extensions are hooked onto content may be built. This functionality might extend the default article editor view in the WordPress backend. Since this isn't important to the definition of the method for building software using FAC, it will be left for future work.

The implementation of the FAC WordPress plugin is simple, and it invokes FAC automatically for each page or article (post), when the plugin is activated in WordPress. The plugin will deduce if a page or a blog post is being rendered, and assign appropriate hook data to a *facade* call. This is just an implicit version of writing the real facade function ourselves, somewhere in the theme template, for example for a HelloWorld extension, the plugin would execute:

```
ExtFw\Facade\Fw::executeExtension(
"HelloWorld\HelloWorld",
get_post_type(),
get_the_ID(),
array("param1"=>"This is an additional extension parameter!!")
);
```

The implemented plugin makes it possible for the user to add additional extensions to content, by invoking FAC when writing content in WordPress backend. For example when writing a blog post. This functionality is provided by plugin functionality called *shortcode*.

Shortcodes are functions that can be invoked from within the content by end-users. For the FAC plugin, the *extension* shortcode was named *extension*. It will be executed if the user writes "*[extension name="HelloWorld\\HelloWorld" type="post" content_id="1" id="?_custom_1"]*" anywhere in the body of a WordPress page or article. In the background, when processing the request, FAC WordPress plugin will invoke the facade method *ExtFw\Facade::executeExtension ($extensionName, $contentName, $contentUniqueIdentifier, $hookId)*, where for our example $extensionName is the class of the extension to use – HelloWorld\\HelloWorld; $contentName is post (invoking the extension from the post content); and $contentUniqueIdentifier is a number – 1. The id of the hook that will be computed is, as previously defined, computed from the given id: "?_custom_1"; and the content unique id, also provided: 1. So, the id of the hook would then be "1_custom_1".

The user can actually write whatever she wants. If arguments *type* and *id* are omitted, then FAC plugin will detect post or page, and assign the id, but this would equalize the hook created in the content with the one automatically run by the plugin when the post is rendered, but that is up to the user to decide. The hook id has to be specified, as our component model defines it, the hook id is the unique identifier of the component that will be executed.

All extensions can be reused in the exact same, uniform way – automatically for pages and articles, or they can be added through the content.

It is important to note that, when the user employs a shortcode to run an extension, it actually exposes and uses one of the most important FAC component models properties – runtime assembly. The two components will be assembled during runtime, into a composite component, while the system is rendering an article or page.

The plugin also knows how to interpret data from the FAC configuration, and depending on the value of the getStaticComponentBinding property, assemble default extensions for content types.

Event reactivity of FAC WordPress plugin

The most important FAC plugin functionality is reacting to events that happen in WordPress. Some of them will trigger execution of extensions, and some have to be transmitted to FAC for further processing. Every WordPress plugin can define which actions (WordPress equivalent events) are of interest, and specify which plugin function should be called when the event occurs.

To ensure correct event reactivity of FAC and extensions, FAC WordPress plugin will receive to the following actions:

- admin_notices – a plugin function will be called **extFwActivated** will greet the user when she visits WordPress back-end, at the top of the screen, where notices are expected to render;
- init – plugin function **extFwHookToPostOrPage** will be called, is used to process GET or POST parameters sent by extensions. This function will call the FAC *Facade* to process the request;
- wp_loaded – plugin function **extFwInit** will be called, to bootstrap FAC, after WordPress has been bootstrapped, and all data is ready for mappers to transmit to the framework, when needed;
- the_post – plugin function **extFwHookToPost** will be called when a post or page has been loaded. This makes it possible to execute extensions when a single post or a page is rendered;
- delete_post – plugin function **extDeleteEvents** will be called, when a post or page is deleted, so that deleted event can be transmitted to FAC and extensions;
- save_post – plugin function **extSaveAndUpdateEvents** will be called when a post or page form in backend has been submitted and saved, so that the event can be transmitted to FAC and extensions.

Actions either directly invoke processing of extensions, or transmit the events that occurred to the FAC framework Event broker, that forwards it down the component tree to interested extensions.

6.3.1.4. Implementing WordPress mappers

Extensions won't run before WordPress' execution context is integrated with FAC. All the FAC context mappers need to be built before extensions can work. Functionality introduced in section 5.2.4 on mappers and integration components define two additional integration components – database access and the event broker. Both concerns have been already mitigated, because database access is provided when FAC is configured correctly, and event mediation or brokerage has been implemented through the WordPress plugin.

PermissionMapper

WordPress distinguishes between the following permission levels [25]:

- registered / logged-in users called "subscribers";
- contributors, users that can write and manage their own posts or content, but cannot publish it, they need approval from authors;
- authors, that have all the permissions that contributors do, but can also independently publish blog posts or content;
- editors, have all permissions that authors do, but can additionally publish and edit posts of other users;
- administrators, who can access all the administrative features within a single site, like managing other users, apply site-wide settings etc.,
- and super administrators, that are able to manage the whole WordPress installation in a "site network" - a specially configured WordPress instance that can run multiple sites with individual users, domain, plugins etc.

A permission mapper for WordPress is created in the FAC folder structure where it searches for mappers: *framework/lib/core/Mappers/WordPress* folder. The file is called *PermissionMapper* and it implements the *IPermissionMapper interface.* Both FAC permission levels and the *PermissionMapper* have been introduced in section 5.2.4.4.

WordPress – FAC permission mapping that has been implemented is explained through data in table 1.

Table 1: WordPress to FAC permission mapping

WordPress permission level	**Mapped to FAC permission level**	**Explanation of mapping**
Subscriber	Read	Subscribers can only manage their profiles and view application pages, which corresponds to read permissions.
Contributor	Write	Contributors can write content in WordPress, but cannot publish it themselves, which corresponds to write permission level in FAC.
Author	Author	Authors in WordPress are allowed publishing their own content, the same is true for FAC.
Editor	Editor	Editors in WordPress are exactly what editors are in FAC, so this is an obvious mapping.
Admin	Admin	Administrators can change site-wide settings.
Super admin	Godlike	Super admins have all the privileges for the whole WordPress instance.

Implementation of WordPress **ContextMapper** is a relatively simple one. All the methods of the *IContextMapper* interface have been connected directly to WordPress helper function that provide the exact same data.

WordPress doesn't handle the session in a specific way, so ContextMapper's *sessionGet* and *sessionSet* methods perform data operations directly on PHP's $_SESSION object.

UserDataMapper

User data mapper has been implemented so that it takes advantage of the WP_user class. When an instance of *UserDataMapper* is created, it retrieves the instance of the current WordPress user by calling the *wp_get_current*_user WordPress function. That object is assigned an instance variable of the mapper, called *currentWordPressUser*, so that it can mediate all the calls made to the mapper, to the actual embedded instance of WP_user object. For example, mapper's *getUserPhotoEmail* returns the value of the email property of the WordPress user's object.

There are a few other mapper methods that forward the calls to other WordPress functions, such as *getProfileUrl*, or *getUserPhotoUrl*. A complete list of mapper methods and returned values from WordPress is listed in table 2.

Table 2: WordPress to FAC UserDataMapper – mapping list

Mapper method	**Retrieves WordPress data through**
getUserLoginName	$this → currentWordPressUser → login_name
getUserDisplayName	$this → currentWordPressUser → display_name
getUserFirstName	$this → currentWordPressUser → first_name
getUserLastName	$this → currentWordPressUser → last_name
getUserEmail	$this → currentWordPressUser → user_email
getUserPhotoUrl	get_avatar_url($this->currentWordPressUser → user_email)
getTitlePrefix	""
getTitleSuffix	""
getProfileUrl	get_author_posts_url($this->currentWordPressUser->ID);
getProfileEditUrl	get_edit_user_link()

Since WordPress by default doesn't include titles for users, empty strings are returned. Title fields could easily be added through plugins or a theme functions file, but that is not for FAC to handle, and is out of scope of this mapper.

ConfigurationMapper

Mapping configuration is relatively short work, since *ConfigurationMapper* doesn't have all that much work. Configuration keys shouldn't be referenced by name directly, as explained earlier in section 5.2.4.5, since that would cause tight coupling. Access to configuration keys by key name is made possible, saving those too, but it only reads and saves them to/from session, so those are temporary values.

Debug value is read from WordPress constant *WP_DEBUG*, and blog and site network home addresses are mapped through mapper's *getHomeAddress* and *getInstanceHomeAddress* methods.

6.3.1.5. Composing software

After the WordPress (the host application) plugin and mappers have been built, FAC and WordPress are integrated, extensions can be used to implement functionality that will be delivered to the user by WordPress.

Once the above steps are completed, it isn't necessary to repeat for any subsequent WordPress – FAC implementation. Instead, only the evaluation of existing extensions should be performed, whether extensions that provide required functionality exist. If not, design and construction activities should be performed to produce them.

In the case of example implementation described in previous sections, component composition and hierarchy should be specified before software is ready to run. For each post, the gallery, content liking and comments extensions must run. For each comment, content liking and comments should run, and for the gallery both comments and liking should run.

Compose are composed through FAC *Config.php* file. This will keep them loosely coupled, while providing required component composition. Since a custom theme wasn't built, and or custom hook ids defined, default extensions to specific WordPress data types will be assigned, and extensions will hook onto default hooks in used extensions.

Lastly, to make sure there aren't two different components providing commenting, built-in comments in WordPress should be disabled. This way it is possible to reuse the FAC comments extension and have all commenting in the resulting application appear uniform and

using the just one component. Composition of components, and in fact composing the application functionality then comes to writing a simple few lines of code:

```
'content'           => [
  'post'             => ['Phd\Like', 'Phd\Gallery', 'Phd\Comments'],
  'page'             => ['Phd\Like', 'Phd\Gallery', 'Phd\Comments']
],
'extensions'        => [
  'Phd\Comments'       => ['Phd\Like', 'Phd\Comments'],
  'Phd\Gallery.gallery'=> ['Phd\Like', 'Phd\Comments'],
  'Phd\Gallery.photo'  => ['Phd\Like', 'Phd\Comments']
],
```

On the left side the name of content type is specified, to which extensions in the array on the right side will be executed. This will attach extensions after the output of the post or page components, ant to the default generic hook inside the extensions.

The same works for extension class names on the left, then by specifying extensions on the right.

Another convention defined by FAC can be seen from the code – names of content are equalized to full extension namespace and class name, and for host applications, names of content are concatenated from a short abbreviation of the name of the host application and the content name, for example Wp.post represents a WordPress post content type.

An example of what the end result of execution of the WordPress as host application and FAC framework integration with extensions *Comments* and *Content like* is visible on illustration 18.

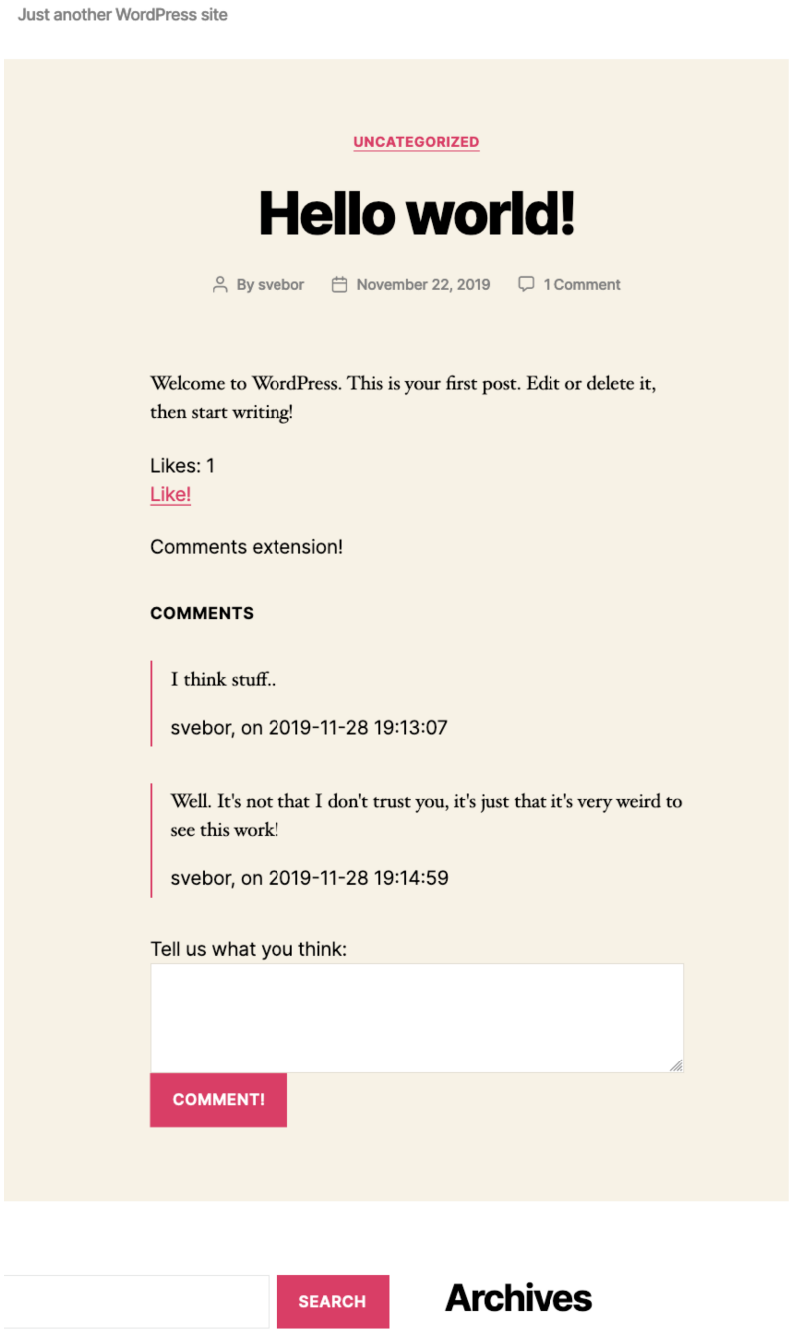

Illustration 18: WordPress and FAC framework running

A sample HelloWorld post that gets installed with every WordPress instance is shown, with two extensions attached to it: *Content like* and *Comments*. Some other user liked the post, so the current user has a “Like” button available.

Below that, a *Comments* extension is showing its title “Comments extension!”. There are two comments entered for the post, by the user Svebor, and there is a text area visible that is a part of the extension.

6.3.2. Building with Quilt CMS

Software is built to satisfy requirements defined at the beginning of section 6.3, but using Quilt CMS as the host application. The same extensions used in the previous section, to built software with WordPress are used.

Quilt is capable of publishing news articles and pages with static content through its *News* and *Content* portlets. Only integration of the FAC framework and Quilt should be done. After that, host application and extensions are composed in a similar way as with WordPress.

Portlets are main components that provide user-oriented functionality inside Quilt CMS. Portlets’ structure strongly resembles extensions’ structure. A class that is the center of a portlet, extends a base class and serves as a controller. The view is built using Smarty templates, and models and data access is provided by the separate layers in Quilt.

6.3.2.1. Implementing hooks

Quilt’s framework uses Smarty plugins to provide common functionalities for views just like FAC framework does.

One possible implementation of hooks towards FAC would be a Smarty plugin that any Quilt component can invoke anywhere in the view, that would automatically provide the name of the parent component (News or Content), but would require the developer to modify templates of those components to actually include the call to the hook Smarty plugin. This would work, but would require the actual reusing component’s view to be changed, which would tightly couple that component to FAC.

Quilt themes also make it possible, similar as WordPress themes do, to provide the system with a template that would be used to render certain portlets in place of default templates packaged with them. This would decouple FAC and components that are reusing extensions, but would require copying all the templates of the each of portlets’ views that should show extensions.

A more generic approach, similar to WordPress', based on events would be more suitable and would provide for a more decoupled integration. This has to be done because the current architecture of Quilt is lacking. Quilt core should be upgraded so that it fires events when a certain portlet has rendered, so that other components may act upon the rendered view, but that would also have to be analyzed in more detail, since there are no conventions for generic view names (defaultEdit, defaultAction...), such as FAC has, and it wouldn't be possible to decide if the view being shown is an editing view or a data display view. Since this isn't the focus of our implementation, and our exercise is to implement applications as-is, the former approach has been chosen, but foundations to mitigating this architectural flaw have been set, in the following section 6.3.2.2.

So, **the end result is a Smarty plugin** that has to be invoked with an id of the content that the portlet is showing, and the information whether the extension is to be rendered inside a form. An example call to the plugin, from the News portlet would then be:

{extfw name="Comments" id=$news.id content_name="news_article" form=false};

and from the Content portlet:

{extfw name="Comments" id=$content.id content_name="content" form=false}

For our Quilt implementation, configuration of statically binding components through configuration would look like:

```
'content'           => [
  'news_article'        => ['Phd\Like', 'Phd\Gallery', 'Phd\Comments'],
  'content'             => ['Phd\Like', 'Phd\Gallery', 'Phd\Comments']
],
'extensions'        => [
  'Phd\Comments'       => ['Phd\Like', 'Phd\Comments'],
  'Phd\Gallery.gallery'=> ['Phd\Like', 'Phd\Comments'],
  'Phd\Gallery.photo'  => ['Phd\Like', 'Phd\Comments']
],
```

6.3.2.2. Data change events

Data change events are another issue with Quilt integration, because just like for application events described in the previous section, there is no systemic event management for data events either. Since this is an issue that has to be solved, because the alternative is leaving

data that FAC saves to pile up, or loose integrity, required changes need to be built into portlets that participate in this integration.

The simplest solution is to modify each Quilt component, so that it notifies FAC directly when any CRUD operation is performed, but this would tightly couple components to FAC.

A separate component that periodically checks whether portlets' data has been changed in any way is thus built. This is done by attaching triggers in the database, directly to the tables of *Content* and *News* portlets. The trigger then writes the events that have occurred to a database table that is periodically checked. This component was called **EventBroker**, and it is an add-on utility class for Quilt.

Quilt's event broker uses Quilt scheduled jobs functionality called *QuiltCron*, that executes certain tasks in intervals, defined by cron[20]-like syntax. *EventBroker* only has one static method that checks the database table *event_broker_events* for events that the database triggers have written. It then checks the database table *event_broker_listeners*, that the architect of the system should fill, that contains a list of listeners that should be invoked for any type of event. Example entries in the *event_broker_events* are shown in table 3, while example listeners are shown in table 4.

Table 3: Events

Event	**Time of event**	**Event data**	**Dispatched**
News.edited	2019-11-07 18:54:50.246066+01	[the row data]	false
News.deleted	2019-11-07 18:54:50.246066+01	[the row data]	true

The trigger will fill the table with the name of the table in which the event occurred followed by a dot and the name of the event (deleted, updated, insert …). It will record the time, the actual row that was deleted, changed or inserted, set dispatched to false.

Table 4: Event listeners

Event	**Listener class**	**Listener method**

[20] Cron, software utility cron is a time-based job scheduler in Unix-like computer operating systems, https://en.wikipedia.org/wiki/Cron

	ExtFwNotifier	onEvent
News.deleted	Portlet_News	

In table 4, there are entries in the first column with the event name. If left empty, any event will be delivered to the listener. If the listener method is left empty, the event will be delivered to the default *onEvent* method. When an event is delivered to all the listeners, the row of the event in the *event_broker_events* table is set to true.

The *ExtFwNotifier* listener listed in table 4 is a class, built as a part of FAC integration layer inside Quilt. It's an add-on utility class. It's purpose is to transfer events that occur inside Quilt, to FAC's event manager. It is listening to all events. It translates the types of events to those FAC will understand, and transfers everything to FAC event manager. FAC will then make sense of what should be delivered, and if, to which components.

Two Quilt components were built to enable communication from the host application towards FAC: the hook Smarty plugin, and the event listener that will transfer events to FAC. This is very similar to the WordPress implementation, where the same concerns were mitigated by implementing multiple functions through a plugin. The difference here being, the Smarty plugin and the event listener being in two separate code files, or rather utility components.

6.3.2.3. Implementing Quilt mappers

Quilt execution environment and contextual execution data has to be translated to FAC, same as with context mappers for WordPress.

Permission mapper

There is one less permission level for users in Quilt, compared to FAC or WordPress:

- Read – users are allowed to see a certain resource, for example a page with blog posts, photo galleries or some other content module. Users are not allowed to contribute, comment or otherwise influence the content.
- Write – users can write a limited amount of data, that cannot directly modify the original content on the website, for example, users can star or "like" a blog post, write

comments to a published blog post or at most might be able to suggest some content for publishing, similar to contributors in WordPress.

- Author – users can independently publish new content on a specific web page, but cannot choose what that content is. For example, if an administrator configured a certain page to serve as a blog roll, then the user can just add blog posts using that active module.
- Admin – users can independently change the structure of a site, add pages, add modules to pages and publish content through those modules. Admin users can also specify which permission level is required to view a certain page and assign additional permissions to users or user groups for those pages.
- Godlike – administrator users that can access every setting for that instance of the application.

This means that one additional permission level from FAC be defined to a corresponding permission level in Quilt. It has been pragmatically assigned to the lower permission level, rather that the higher one. Mapping from Quilt to FAC, with description of Quilt permission levels is given in table 5:

Table 5: Quilt to FAC permission mapping

Host application permission level	**Mapped to FAC permission level**	**Explanation of mapping**
Read	Read	Read permissions in Quilt are corresponding directly to Read permissions in FAC – users are allowed to see content.
Write	Write	Write permissions in Quilt provide users with the ability to create content, but cannot publish it themselves. Or, users can contribute to published content indirectly – such as comment on it, like it

		etc. This corresponds to the Write permissions level in FAC.
Authors	Author	Authors in both FAC and Quilt are allowed to publish their own content.
Admin	Editor	Editors in FAC are what Admins are in Quilt – being able to edit other users' content.
Admin	Admin	Administrators in Quilt can change site-wide settings, and change structure of the site or page. This will also map to Admin permissions in FAC – which means a user can change settings or assign sub-components.
Super admin	Godlike	Super admins have all the privileges in both Quilt and FAC.

ContextMapper

All the methods of the IContextMaper interface redirect to directly calls of Quilt's functions or access Quilt's global data variables, containing the data in question.

There is no specific way to handle the session in Quilt, so ContextMapper's *sessionGet* and *sessionSet* methods perform data operations directly on PHP's $_SESSION object.

UserDataMapper

User data mapper has been implemented so that it retrieves the current user using the Quilt's *UserFactory* class. This will retrieve the object that will be of type of the current user, saved in a protected instance variable, called *currentQuiltUser*, and will provide all the required data that *IUserDataMapper* interface defines.

Each of the mapper methods forwards the call to a getter of the Quilt's user object, for example, *getUserFirstName* mapper method is forwarded to *getFirstName* getter of the object.

There are a mapper methods that may forward calls to other Quilt objects or ones that will perform more data processing before actually being able to retrieve the data, such as *getProfileUrl.* This method first tests that the Quilt's user object contains the method *getUserProfileUrl*. If it does, then the argument has to be provided – the id of the Quilt's language for which to retrieve the URL. This will first be done through the *ContextMapper*, and then handed off to *getUserProfileUrl*. If a corresponding language isn't found, then the default will be used (no arguments will be supplied). If the Quilt's method doesn't return any value, mapper will return null.

Lastly, *getTitlePrefix* and *getTitleSuffix* perform object retrieval by using Quilt's *Portlet_Portfolio* class, through its static method *getPerson*. Method will retrieve raw data about the user, and within it, title prefix and title suffix can be found as keys within an associative array.

ConfigurationMapper

Mapping configuration was easy for WordPress, and is also easy for Quilt. Configuration keys shouldn't be referenced by name directly, as explained earlier in section 5.2.4.5, since that would cause tight coupling. Access to configuration keys by key name is made possible, and saving by name is possible too, but the ConfigurationMapper will only read and save them to or from session – as temporary values.

The *debug* key, a boolean value, is read from the Quilt's global configuration variable - $_conf, which is an array. If the key "*developemnt*" is present and the value of if is true, then FAC will also run in debug mode.

The URL of Quilts's installation, or rather the URL of the main site, and the URL of the site that is rendering for the current request, are mapped through mapper's calls to Quilt's *Site* objects. The main site instance will be retrieved by invoking the

The getHomeAddress will thus retrieve the URL of the Quilt's site with the query to Quilt configuration key *domain_base*, while *getInstanceHomeAddress* will instantiate the current

site, for the request, also using Quilt's SiteFactory. If the site doesn't have a separate domain, then the *domain_base* will be used, along with the *url_prefix* given by the Quilt's site object.

7. EVALUATION OF APPLICABILITY

FAC applicability is evaluated through inspection of the component model and prototype implementation of FAC framework in the following ways:

1. **Component structure and framework utilities**: Component structure is compared, how components are built, composed and assembled using FAC framework and what framework utilities are available versus competing popular web application frameworks?
2. **Reusability**: Does the implemented FAC prototype enable reuse of extensions outside of their component model is investigated, and what is the measure of reusability of such components, is reusability predictable?
3. **Performance**: FAC performance is measured using standard benchmarking program, and whether performance is acceptable.

The first evaluation point analyses the **structure of components** built using FAC, and how similar they are to what competing component models provide. Software engineers must not be presented with a radically different approach. Components from both the host application and the FAC framework have to be composed, developed and maintained side by side. Thus, the approach of both frameworks should be familiar and intuitive. Common **framework utilities** that popular frameworks provide must also be available in FAC framework.

Second evaluation point shows how a real-life application composed of heterogeneous components is expected to perform. FAC or its extensions must not hinder performance of host applications it is integrated with.

Mappers have been implemented for WordPress and Quilt CMS, so **performance and reusability** are measured for those specific implementations, while component structure and framework utilities are additionally compared to the Laravel framework.

Acceptable **levels of measurable reusability** of extensions must be achieved, and the same must be achieved by the FAC framework model itself, since it is a component that must be integrated. If these are not achieved, use of extensions out of their component model is of little value.

7.1. Component structure and framework utilities

An analysis of how components are built using FAC is done, expecting that they should be structurally similar to what competing component models provide. Extensions should be structured in a similar way to how popular frameworks define component structure. Sub-components should also be used and composed in a similar. All these similarities are necessary so that FAC is usable and more easily maintainable alongside host applications.

7.1.1. HelloWorld component structure comparison

HelloWorld components built for WordPress, Laravel, Quilt CMS and FAC are compared, which have been introduced in that order in sections: 3.6.3, 3.6.1, 3.6.2 and 6.3.1.1.

A Hello world extension consists of the same number of constituents, just as a Quilt CMS component does. It resides inside its own folder, just as WordPress plugins and Quilt portlets do, and those constituents logically separate parts of the component in the same way as all the other frameworks do. Localization strings are saved inside JSON files in a separate folder and Smarty templates define views that are rendered to the user, also organized in a separate folder. The HelloWorld extension contains only one additional file, containing the main extension class, thus bringing the number of constituents to 3.

All evaluated frameworks make it possible to separate the view layer from the actual localization and the logic that decides what should be shown, except for WordPress, which lowers the number of constituents (code files). With WordPress, there is no explicit view layer and components are functions that render the HTML and are potentially both business logic and part of the view themselves. It is left up to the engineer to separate them, but no particular way is given or required.

Moreover, if any other available WordPress mechanism was used to render the plugin output, instead of the *shortcode*, the plugin developer decides at what times she wants the framework to call the plugin, by hooking onto “action”, instead of leaving it up to the framework to call the component at the right time. This is different to what Laravel, Quilt and FAC do, the standard way frameworks operate, called *The Hollywood principle* – “don’t call us, we’ll call you” [65]. WordPress plugin instead monitors the framework and decides itself when to render.

A controller is a mandatory part of a component in all the frameworks, except WordPress where plugin functions cover that functionality.

Routes for actions of components have to be defined explicitly only for Laravel, which causes Laravel to have the most constituents – the one additional code file.

The overview of component structure comparison of HelloWorld components is given in table 6.

Table 6: HelloWorld component structure comparison

	WordPress	**Laravel**	**Quilt**	**FAC**
Localization	gettext .po files	PHP array files	PHP array files	JSON files
View layer	-	Blade templates	Smarty templates	Smarty templates
Controller	No	Yes	Extended base class	Extended base class
Constituents	2	4	3	3
Route definition	No	Yes	No	No

7.2. Framework utilities

Web development frameworks, and frameworks in general, provide common utilities to make development of components and applications more simple and faster, as introduced in section 3.5.

Comparison between FAC, Laravel, WordPress and Quilt is given in an easy to review form, in table 7.

Storage providers are supported by all the frameworks, while Laravel has the most elaborate storage provider system of all, supporting both local, network and cloud providers. Other frameworks simply provide components with a path to which to save files.

Database accesses is the most common requirement of applications and components, and having a database access layer is mandatory for every framework. All the compared frameworks include this.

Table 7: Framework utilities comparison

	WordPress	Laravel	Quilt	FAC
Storage providers	Yes, simple *(wp-content)*	Yes	Yes, simple	Yes, simple
Database access	Yes	Eloquent	Yes	Doctrine
ORM	No	Eloquent	Yes, simple	Doctrine
Model definition and organization	No	Eloquent	Yes, simple	Doctrine
Composition	Runtime	Design-time	Runtime	Runtime and design-time
Packaging	Yes	No	Yes	Yes
Front-end development	Yes, simple	Yes	Yes	Yes
Localization	Yes	Yes	Yes	Yes
Authentication and authorization	Yes	Yes	Authorization, through mappers	Yes, inherited
Session handling	Yes	Yes	Yes	Yes, inherited

Object-relational mapping is a layer on top of the database access, that makes it convenient for components, or rather component builders, to not have to write SQL queries directly, but rather instantiate and use model objects, whose changes and actions are directly written to the

database. WordPress doesn't have an ORM layer, Laravel and FAC use standard libraries and Quilt provides a simple custom ORM.

Model definition and organization is provided by all component models, except WordPress. This is probably, at least partly, tied to the fact that there is no object-relational mapping in WordPress. To create models or any other business layer, one has complete freedom on how to organize or define it inside the plugin folder. Other component models point the user in the right direction, which is usually a better solution in the long-term.

Component Composition is possible using all the compared component models, FAC being the most flexible one. Quilt provides run-time composition options, but only in a limited way – the user can add a portlet to a certain web page, it isn't possible to nest or reuse portlets inside of portlets without tightly coupling them. The same is true for WordPress plugins.

Laravel has no run-time composition support, and reusing components within components is also not supported by the component model. Components would have to be tightly coupled if they were to be integrated. In fact, design-time composition is also not possible in a way that wouldn't tightly couple components. Laravel tutorials and how-tos will always point engineers towards coupling components, even at the database level.

FAC, because of its hooks mechanism, and because it is possible to define component composition at design-time using FAC configuration, and because it is possible to easily create run-time composition like in the case of a WordPress plugin, comes out as the most flexible framework between those compared.

Packaging of components is a fairly important aspect of a component model. How easily reusable a component is also depends on how portable it is, and that depends on how easy it is to create a component package. In general, a component will be easier to package if all the sub-components, or code constructs, are saved in the same place. In that case, an archive can be created and the component can be sent, downloaded over the internet etc.

All the compared frameworks group component's code constructs together, except for Laravel.

Similarly, all frameworks do provide conventions on how to separate different types of code files (localization files, templates, JavaScript, CSS...), except for WordPress, which leaves most of the flexibility to the user.

Front-end development provided by frameworks usually entails templating and ways in which templating can speed up repetitive tasks. For example, creating forms that will post data back to the component that generated it. All the frameworks provide this, except for WordPress. But, it does provide way to hook into actions that WordPress performs, and according to those actions perform plugin actions, so front-end development is supported by WordPress, just in a less comprehensive way.

Localization is supported in a very similar way throughout all the compared frameworks.

Authentication and authorization is provided by all the frameworks except authentication in FAC. FAC does support authorization, through mappers and integration with the host application.

Session handling is supported in all frameworks but WordPress, which expects components to write and read the session directly using standard PHP libraries.

7.3. Evaluation of reusability

It is important to note that measurement of reusability of frameworks, such as Laravel, Quilt or WordPress isn't possible. Frameworks that aren't made to be reused such as FAC, are never reused as components since the complexity of such an attempt would be very high. If only components built using said frameworks are evaluated and compared, there would be little direct comparative value to FAC components.

HelloWorld components compared in the previous sections for WordPress, Laravel and Quilt CMS are reusable inside other instances of their native frameworks. HelloWorld components are simple, but the same principles would apply for more complex components.

If a *Hello world* component built for Quilt was to be used inside Laravel, one would have to completely rewrite all the parts – the localization file, translate the template from *Smarty* to *Blade* template engine. Additionally, a route would have to be added for the component, and a part of the Quilt's portlet class, parts of methods could be copy-pasted into the Laravel controller.

The same is true for transferring any combination of source framework component to any target framework. The more complex the component is, the harder it gets to reuse any part of it to build the same component inside a different framework.

There are two main contributions of the FAC model: 1) FAC makes reusing components outside of their component model systemic, and repeatable for various host applications through mappers. 2) FAC enforces extensions reuse in a completely decoupled way – always reused through the universal adapter, without any tight coupling with the host application or other extensions, thus making reusability predictable.

To evaluate how reusable FAC component are, how FAC impacts component reuse in comparison to other component models and frameworks, reusability metrics are applied, introduced in chapter 4.

Reusability is measured through analysis of the example applications built in section 6. These applications use prototype FAC framework integrated with WordPress and Quilt CMS host applications.

7.3.1. FAC component reusability

The software system that has been modeled in section 6, in terms of components and visualization of component reuse, is shown in illustration 19.

It is visible that a component named *Host application* is using three different components. Components are assembled using a separate composition definition, the FAC configuration file. The only component the host application is really using is FAC framework component, through its native component (WordPress plugin or a Smarty plugin in Quilt) that connects to the FAC *facade* class.

Extensions (components on the right) also reuse other extensions. Gallery uses Comments and Like, while Comments use Like and recursively, Comments.

Since extensions are reusable only when FAC is reused, and FAC is reused so that extensions may be used, **reusability of FAC and single extensions is inseparable**.

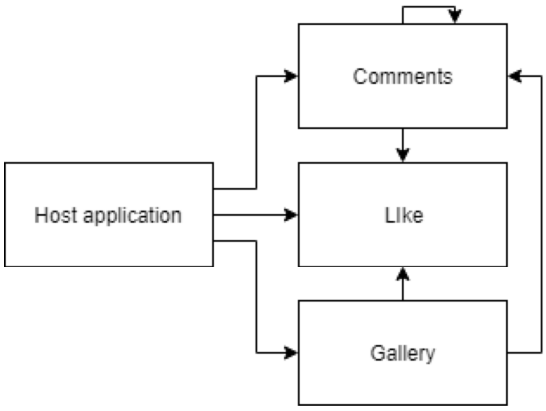

Illustration 19: Component usage

Another coupling that is not shown in illustration 19 is between the host application and the FAC framework itself through context mappers. This coupling is done in reverse, since mappers are actually a way for FAC and extensions to retrieve contextual data from the host application.

If FAC is illustrated as a white box component, containing all the extensions from our example, and include the native hook component in the host application, and the facade as the only exposed interface of FAC towards the host application hook, then only two interfaces that are used to compose the two components are left. The is because the interfaces between different extensions are actually interfaces through the facade – illustration 20.

For FAC prototype, all the mappers in total, invoke 10 host application methods in case of WordPress and 15 in case of Quilt. The difference being in the nature of applications, where a part of WordPress contextual data can be retrieved without using WordPress functions or components. For the sake of quantifying and measuring examples, an approximate number of interfaces that were used for both integrations is set to the value 15.

There are two additional methods that are called from the hook to FAC, through the Façade – 1) the invocation of the actual hook execution, and 2) sending the data events to FAC.

This makes 17 the total number of active interfaces between the host application and FAC.

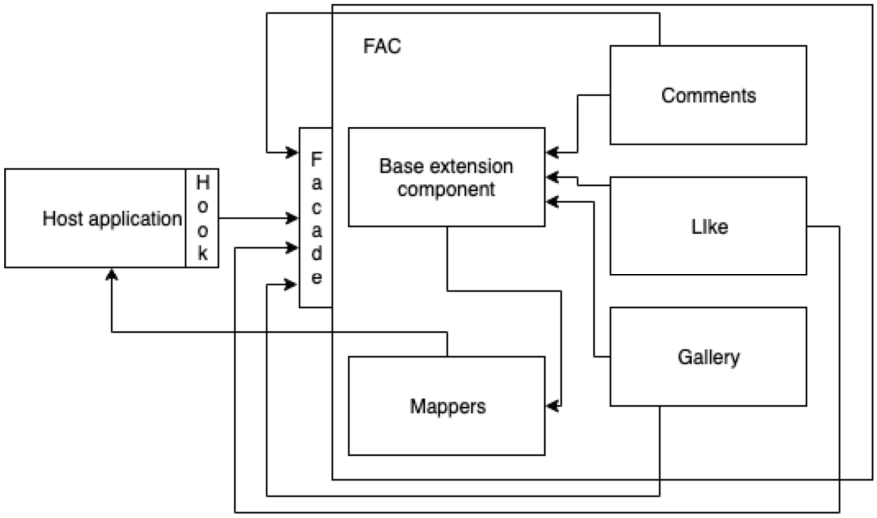

Illustration 20: Component usage with hooks and FAC as a white-box component

The number of interfaces between any two extensions is always two – the hook, and the event manager. Although extensions are effectively loosely coupled, since a mediator is used, and the composition is defined externally, not within components themselves.

7.3.2. FAC and extensions' component reusability

To use the definition of component reusability that was introduced in section 4.2.1, components that are measured have to be chosen. Reusability of FAC, and of each extension separately has been chosen.

In case of reusability of FAC, the total number of interface methods was found for FAC example implementations with WordPress and Quilt CMS. The value is, as shown in the previous section, 17. Two interface methods are added that provide common functions from the application (extension) domain (the hook and the events). Thus, equation 5 may be applied to compute *component reusability*:

$$CR = \frac{\sum_{i=1}^{n} (Count(CCMi))}{\sum_{j=1}^{m} (Count(CIMj))} = \frac{2}{17} = 0.11647 \quad (14)$$

This is a low score, since CR is expected in the interval [0, 1]. But this indicates only the FAC integration itself, without any extension reuse. Reusability of FAC may only be considered

transitively, and it depends on the number of extensions that are reused in the system. Each of those extensions rises the number of interface methods that provide functionality from the application domain. The equation must then be extended for FAC, so that it includes the number of extensions that can be reused:

$$CR = \frac{N * \sum_{i=1}^{n} (Count(CCMi))}{2*(N-1) + \sum_{j=1}^{m} (Count(CIMj))} \frac{2*N}{17+2*(N-1)} \quad (15)$$

N is the number of extensions in the system. If this equation is applied to compute reusability for 1, 2, 5, 15, 50, 150, 500, 2500 extensions, reusability will rise steeply to 0,869 for up to 50 extensions, and then asymptotically towards 1 (table 8).

Table 8: Component reusability of FAC, dependent on number of extensions

N	**CR**
1	0.117647058823529
2	0.210526315789474
5	0.4
15	0.666666666666667
50	0.869565217391304
150	0.952380952380952
500	0.985221674876847
2500	0.997008973080758

Visually, component reusability of FAC, dependent on the number of extensions in the system is shown on illustration 21.

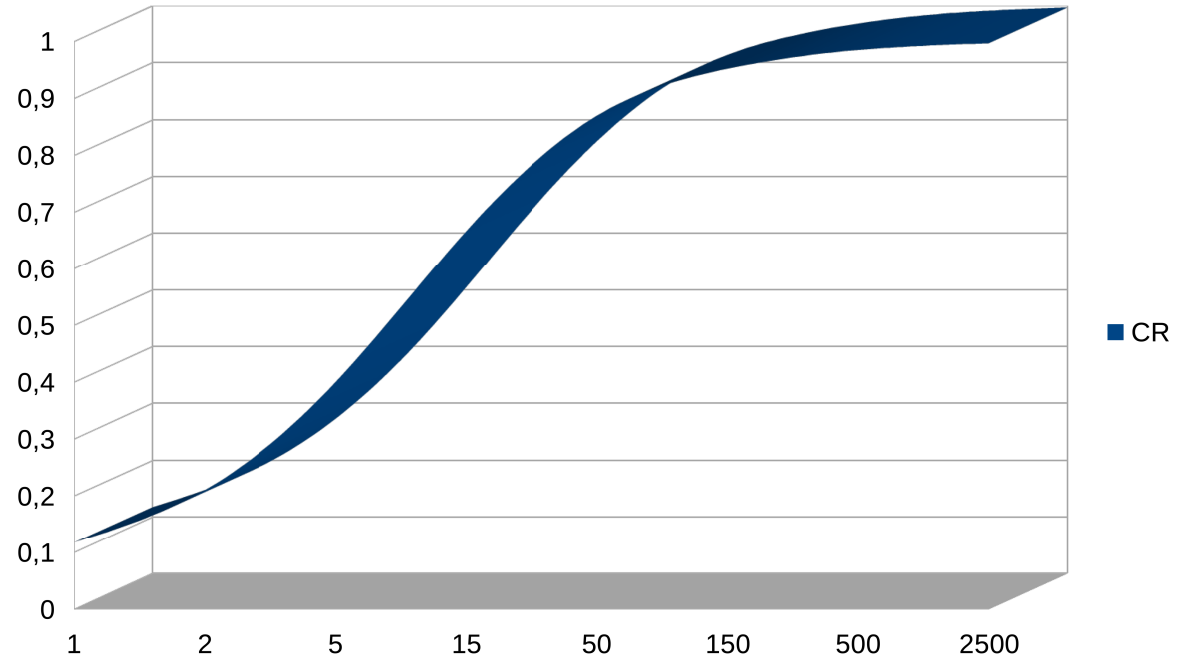

Illustration 21: Component reusability of FAC, dependent on number of extensions

Reusability of each extension, computed using equation 15, and when extensions' interface methods are considered to be only the ones exposed through FAC, towards the components that reuse them (other extensions, or host applications), is CR = 1.

The reason for this is that both the hook and events interfaces are domain specific to that particular extension.

This shows that FAC component model enforces creation of components that are perfectly reusable within the FAC component model, while reusability of each extension, towards other component models (host applications), or other extensions, depends on the number of extensions in the system and is defined by equation 15.

7.3.3. Measuring reusability from complexity

Metrics that measure component complexity and interaction density were introduced in section 4.2.2. The more complex the interactions of the component, or the component itself, the less reusable a component is.

Interaction density of FAC framework and extensions is computed for the FAC prototype implementation (section 6.3), using equation 6. Computed interaction densities for components in the system are given in table 9. Each extension has two interfaces – hook and

events interfaces, that will always be connected, so the interaction density of each extension is 1. The same stands for FAC framework – all of the 17 methods (mapper interfaces) will always be connected for FAC to work, so interaction density of FAC is also 1.

Table 9: Interaction densities for host application, extensions and FAC

IDC	Component
1	Like extension
1	Comments extension
1	Gallery extension
1	FAC framework

Average interaction density is then computed from these values, which is also 1. This is the highest interaction density a component-based system may have.

But extensions will never be used directly, they are accessed through the FAC framework, thus making the correct value of average interaction density of the system:

$$AID = \frac{ID_{FAC}}{n} = \frac{2}{4} = 0.5 \quad (16)$$

The host application should also be included into the formula, under the assumption that its only interfaces that can qualify for this component-based system are the ones that should always be connected to FAC. In that case its IDC is also 1. If they were to be included at some point, AID would be lower, thus making only lowering the final AID, making the system properties ever so slightly more positive. Thus, the final AID computation can be done using:

$$AID = \frac{ID_{host\ application} + ID_{FAC}}{n} = \frac{3}{5} = 0.6 \quad (17)$$

This value will fall with the number of extensions in the system, as shown in illustration 22.

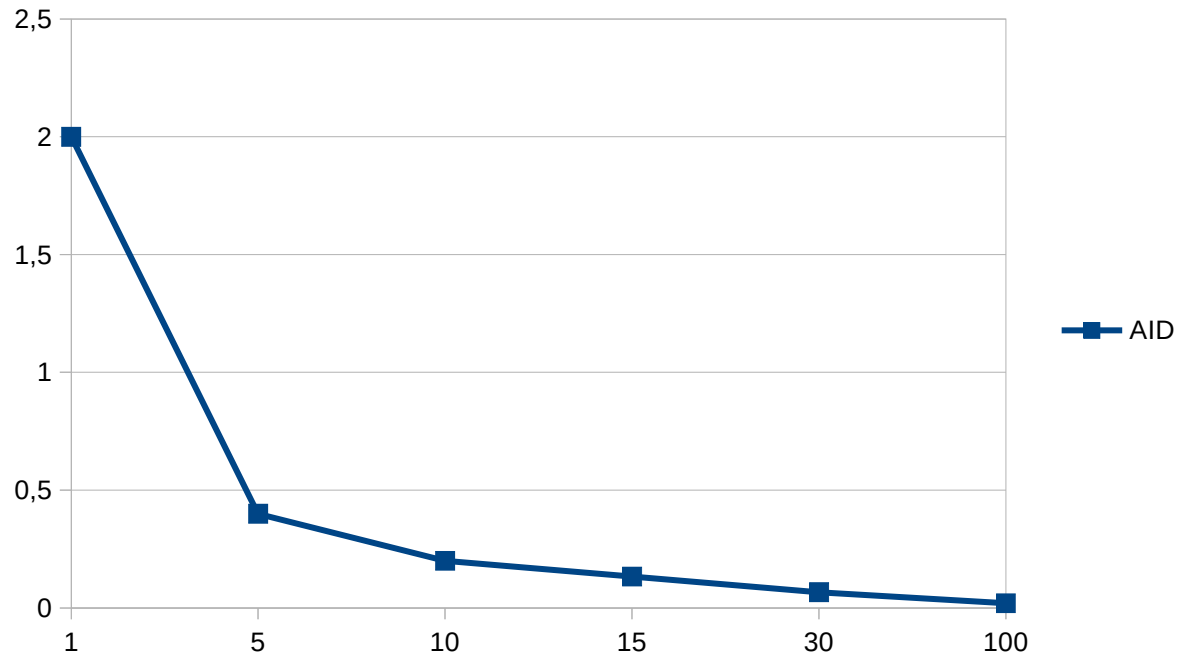

Illustration 22: Average component interaction density

The average interaction density value (y axis) quickly falls when as much as five extensions are in the system (x axis).

Component packaging density complements the meaning of component interaction density. CPD is measured to complement the value of AID = 1 of the system.

CPD is measured for extensions in the system, because host application numbers are variable and not important for reusability of FAC and extensions, and the integration with the host application (table 10).

Table 10: Components and constituents count

Component	LOC	Modules
Comments	260	4
Like	105	3
Gallery	262	6

$$CPD_{lines\ of\ code} = \frac{(260+105+262)}{3} = 209$$

$$CPD_{modules} = \frac{13}{3} = 4.33$$

If host application was included in computation of CPD, for example WordPress with more than 400.000 lines of code, values would rapidly rise. The same is true if there were more complex extensions used.

FAC, with its less than 4.000 lines of code in 54 files (modules) does raise packaging density slightly, but is a small and simple component in itself.

Based only on numbers of AID and CPD, it is possible to conclude that the composed component-based system is potentially **complex** – made up of many constituents and a small number of interactions. According to literature, it could be classified as a transaction processing system – with a high data volume being processed, with many components that exchange small amounts of said data.

This seems to be in line with expectations for FAC system – a composition of loosely coupled components that exchange only basic integration signals, each processing and completely encapsulating its own data.

In practice, complexity of the composed system will depend on the complexity of the host application, and complexity of extensions reused, but component reusability will always stay high because of the low value of AID.

AID doesn't correlate with a numeric reusability scale, but the low value of AID means higher reusability on an ordinal scale.

7.3.4. Coupling complexity

It is possible to compute the **average coupling complexity** of the resulting composed system using formula 9, and again discard coupling complexities of each of the extensions because they are coupled loosely and through interfaces that are achieved through integration of the host application and FAC, not their own interfaces. Coupling complexities computed for each of the components is given in table 11.

Table 11: Coupling complexities of components

Component	$II_c + OI_c$
Host application	0 + 17
FAC	17 + 0
Gallery	0
Like	0
Comments	0

Using formula 9 it is possible to compute average coupling complexity for our example system:

$$ACC = \frac{17+17+0+0+0}{5} = 6.8 \quad (18)$$

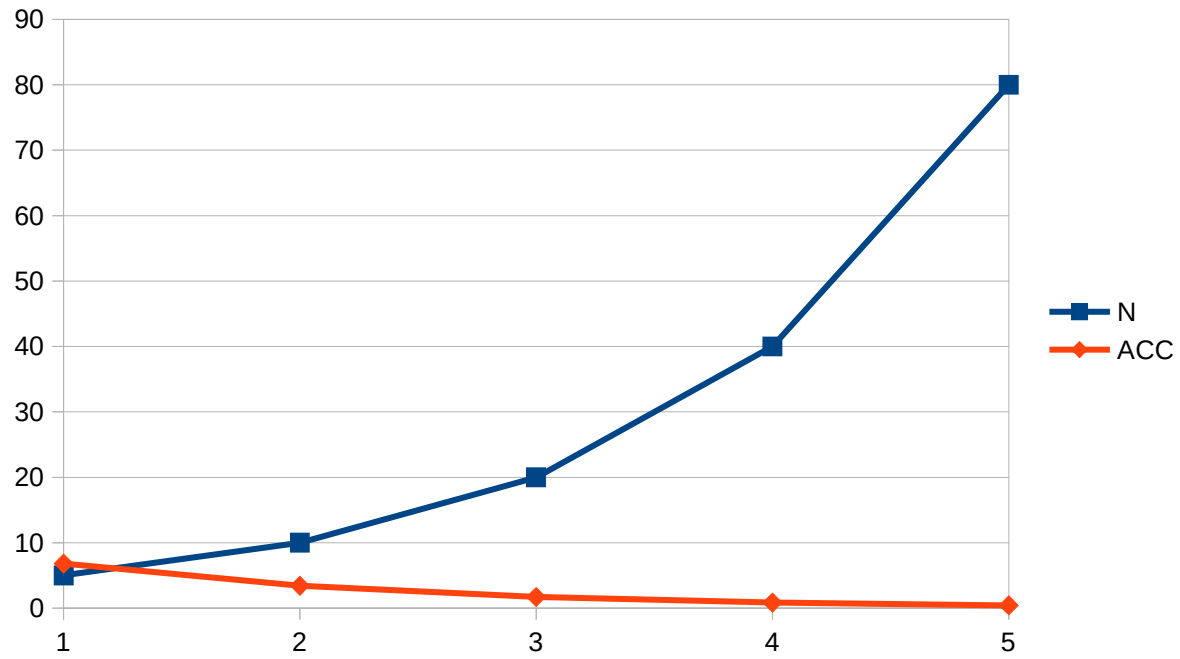

Illustration 23: Average coupling complexity, dependent on number of extensions in the system

Coupling complexity for FAC and host application is thus 17, and for our example system ACC = 6.8. Value of 17 is is not much if the achievement of integration between to component models is considered [56]. It is also very important to note how quickly it falls as more extensions are added to the system (illustration 23). ACC will fall to 3,4 for 10

extensions in the system, and to ACC of 1,7 for 20 extensions – an indication that coupling complexity of the FAC is very low, making it a low-coupled system.

7.3.5. Black-box reusability metrics

Black-box reusability metrics introduced in section 4.2.4 like ***rate of component observability*** *(RCO)* and **rate of component customizability** *(RCC)* are found to not be applicable to FAC framework, since extensions have no readable or writable properties, thus making formula 10 and 11 values equal 0.

These results are not valid for the rate of customizability, since customizability of every extension is left to the developer of that particular extension. Additionally, customizability should be possible through the user interface, or separate from the component interface (facade class), as RCC is defined. Customizability is a valuable property of each extension, but it doesn't seem to be quantifiable in such a way. It may be defined as a number of configuration options a single extension might have, but this cannot be systematically evaluated since it depends on the design and implementation of each extension.

Self-completeness of component's return value (SCC_r)

There are only two methods that can be invoked by components that reuse extensions – the hook and notify of events. Both methods are business methods of the extension, and only one of those could be considered self-complete, according to definition from section 4.2.3, thus making $SCC_r = 0,5$.

Self-completeness of component's parameter (SCC_p)

Both hook and the interface to notify of events receive parameters, so SCC_p will always be 1 for components of the FAC component model.

From **SCC_p** and **SCC_r,** the authors defined the **overall reusability metric (COR)**, which isn't applicable because their method discriminates components such as the ones FAC component model enforces as anomalies, and indeed the measurements come up negative.

7.3.6. Object-oriented software metrics

Object-oriented software metrics are evaluated on our prototype integration with WordPress. These metrics show how complex the integration link between WordPress and FAC is. Experiments have shown that one of the most reliable metrics for integration complexity are

lines of code written to integrate components [52]. In addition to lines of code (LOC), two metrics introduced in chapter 4.1 also may be used: RFC and CBO.

Mapper classes that provide integration of host application and FAC framework are analyzed. Native host application components that make it possible to reuse extensions are also analyzed. Through this analysis, it is possible to predict the complexity of components that have to be built so that a family of host applications, or a certain framework, is integrated with the FAC framework.

Response For Class for mapper classes will always be high. Mapper classes are supposed to access interfaces of the host application and translate host application data into data that FAC components (extensions) understand. This means that each invocation of a method of a mapper class will be forwarded to at least a couple host application classes and methods.

Our WordPress integration through mappers uses WordPress API functions to retrieve data, which is a large set of functions. Quilt integration uses global Quilt objects, Quilt factory classes and then object instances of various classes that will provide required data. For some other host applications, in addition to classes and methods, mappers might even need to invoke Web services.

RFC can be computed by applying the formula 3 from section 4.1 and values of WordPress mappers are: ContextMapper: 16; ConfigurationMapper: 5; PermissionMapper: 12; UserDataMapper: 18.

For Quilt, values are: ContextMapper: 9; ConfigurationMapper: 11; PermissionMapper: 15; UserDataMapper: 18.

CBO value for the prototype WordPress integration is low. User data mapping always integrates with a dedicated class or in case of WordPress, the *WP_user* class. The same is true for *PermissionMapper*. Each of the mappers interacts with only up to 3 different host application components.

There are four mappers, through which FAC and the host application are integrated, the value of CBO from FAC framework adapter point of view is 5 – there are four mappers, and there is a host application component that is reusing the FAC.

In literature, acceptable CBO values go up to 5, and acceptable RFC values for a class up to 100 [49], [50].

The FAC – host application integration provides the possibility of reuse of a significant number of components and CBO is still kept within acceptable bounds.

The same is true for RFC value, which is both for the Quilt and WordPress mappers much lower than 100.

Lines of code

In total, WordPress integration makes up around roughly 447 lines of code. For the sake of our example, this number is rounded to a much higher, but in total only 500. Integration of Quilt and FAC, also rounded to a higher value, makes up only 600 lines of code. Only a few hundred lines of code may be completed in only a few hours by an experienced engineer. The end result will thus depend on the level of knowledge an engineer has with a certain host application. Supposedly, FAC framework integration will be done by engineers that want to more easily reuse components (extensions). This means proficient engineers in that particular host application. For this example, even three days may be assumed. This time to integrate FAC with a host application is still acceptable, considering the criticality of components that are thus integrated, the potential number of extensions that will also be integrated through that work.

7.3.7. Criticality

Criticality is a known and expected side-effect of component reusability. The more component is reused, the more it becomes critical to the system or ecosystem in which it is used 4.2.4. FAC exposes two of three types of criticality, through different parts of the framework: bridge and inheritance criticality. In the previous section, we have shown through lines of code, that size of FAC isn't critical.

Bridge criticality of FAC framework is detectable as it acts as the bridge between various components, extensions, but also between the host application and FAC extensions. FAC framework is used as a bridge between any two components when an extension is used, even when an extension uses another extension, or itself recursively. Should any of the signal mediation capabilities of FAC fail, the probability of all extensions failing, or the whole system failing in which FAC framework is used, is very high.

From illustration 20, we can see that there are a number of smaller bridge-critical components in the system, all contributing to FAC achieving its functionality of a universal adapter.

The native host application component is the first one, exposing bridge criticality. In case of WordPress the plugin we wrote – because all communication and reuse that is initiated from WordPress side towards FAC goes through that component. The plugin then connects to the facade component, or rather, through the facade, to the *Framework* component.

Context mappers are also bridge-critical components, that support communication that is initiated by the FAC, towards the host application.

As a result, each FAC implementation includes 6 bridge-critical components: the native host application component, *Framework* component, and the four mappers.

The second form of criticality, **inheritance criticality** shows through the architecture of extensions, which all have their origin inside a single class – *Extension.* Should there exist a bug with that base class, or any part of design or architecture of the extensions, in terms of utilities of the FAC, then all extensions will have that same problem.

7.4. Performance impact of FAC

Performance is measured for the prototype FAC framework integration with Quilt CMS and WordPress.

WordPress and Quilt performance has been tested in two ways: 1) without FAC integrated and enabled; and 2) to measure the impact FAC makes, performance has been measured with FAC integrated and enabled, with extensions Like and Comments enabled.

Measurements have been done using ab[21], *Apache benchmarking tool*. It has been set-up to perform 1000 requests, while constantly performing 20 parallel requests, on a server with 4GB of RAM, two Intel Xeon E5-2620 processors at 2.1 GHz.

To test WordPress without FAC, a clean instance of WordPress has been installed. When WordPress is installed, it automatically creates a blog post called *Hello world*, that has a welcome message in it's content. We will perform WordPress benchmarking using that particular page.

To test Quilt without FAC, a copied instance of Quilt, running a copy of www.fer.hr website has been chosen. Benchmark will make requests to one of the published news articles.

[21] ab, Apache HTTP server benchmarking tool, https://httpd.apache.org/docs/2.4/programs/ab.html

It is important to note that in the WordPress test, both FAC and WordPress connected to the same MySQL database server.

For the Quilt test, FAC connected to its own MySQL database server, while Quilt used its PostgreSQL database. Connecting to two different databases is more overhead. Compared to the slowdown of smarty, this just shows how expensive performance-wise Smarty is.

Results for Quilt and WordPress, with and without FAC are listed in table 12:

Table 12: Performance comparison

	Quilt (no FAC)	**Quilt (FAC)**	**WordPress (no FAC)**	**WordPress (FAC)**
Time taken (seconds)	36.394	38.690	18.106	30.458
Mean time per request (ms)	727.875	773.793	362.127	609.169

It is observable that performance impact of having FAC enabled and executing two extensions in *WordPress causes 68.22% longer execution time.*

*P*erformance *impact on Quilt causes only an increase of execution time by 6.3087%.*

It is important to note that both Quilt and FAC use Smarty templates for the front-end layer. Once Smarty is loaded, both Quilt and FAC use it and the rest of the difference in performance is the actual impact that the rest of FAC overhead and the two extensions execution causes.

Then, 6% increase in execution time can be attributed for FAC with Quilt. Smarty impact performance already when initializing with Quilt, which can be seen from the overall longer execution time.

Wordpress with FAC enabled increase in execution time is 68% compared to without FAC. Smarty performance impact can be seen more explicitly, since WordPress doesn’t use any templating itself.

To test if Smarty really does induce such a performance impact, **additional testing has been performed. Smarty** was used inside a single PHP script with only two lines of code:

```
$msg = "This is a test!";
echo "This is a test message, with a template <br> $msg";
```

The script just renders a line of text from a variable. When instead Smarty is used:

```
$s = new Smarty();
$s->setTemplateDir('/tmp');
$s->assign("message", "This is a test!");
echo $s→fetch("test.tpl");
```

performance drops from 20.953 ms per request to 27.948 ms per request. This is a considerable 28.6% slowdown, that is visible already without any Smarty plugin use, multiple templating folders etc. It is a considerable increase especially because there are no contextual template variables being prepared by the system using Smarty.

Overall, the impact on a light-weight application like WordPress isn't negligible, but it also isn't unacceptable. Every page load under a second is acceptable [67]. It is still faster than Quilt's load time for the page without FAC enabled.

Another, less demanding test was performed using the same low-end server configuration, for a total of 40 requests with only 2 concurrent, performance is much better and impact is nearly the same.

Quilt with FAC enabled execution time is longer by *11.036%* compared to without FAC. Wordpress execution time with FAC enabled is longer by 101.73% compared to without FAC: table 13. Again, impact of Smarty is visible with Wordpress, whereas Quilt uses Smarty, and only 11.036% increase is due to FAC itself.

Both Quilt and WordPress with FAC are far below the one second threshold, meaning that performance is acceptable.

Table 13: Performance comparison for lower server load

	Quilt (no FAC)	**Quilt (FAC)**	**WordPress (no FAC)**	**WordPress (FAC)**
Time taken (seconds)	1.396	1.550	0.709	1.431
Mean time per request (ms)	69.781	77.482	35.473	71.559

8. CONCLUSION

Component-based software engineering is as pervasive as software engineering in general. Because of the very abstract nature of software, different component models exists for different application domains, different applications in the same domain, differentiated by different architecture styles, platforms and technologies. Component models are both implemented and defined by development frameworks. This thesis defines a framework as a component component model, and its prototype framework implementation to improve component portability between component models for three-tier web applications.

FAC model, as its name implies, defines the framework itself as a component, that serves as a universal adapter for all components (extensions) built using said framework. As a result, all extensions expose the same standardized interfaces through which they can be reused, improving their reusability, as shown in section 7.3.

The prototype implementation of the FAC framework in this thesis provides all the common utilities that popular open source frameworks provide. This causes methods and solutions to repeatable requirements and tasks that components need to satisfy similar to solutions of existing applications and frameworks. This similarity is important to engineers that would maintain and integrate the FAC framework and extensions with existing host applications. Using the most common architectural pattern (MVC) for extensions, makes FAC even more approachable to engineers maintaining or developing existing applications.

Integration of FAC with a host application is a well defined process through the use of mappers, that serve as blueprints for interfaces that have to be made.

Once this integration is achieved, FAC provides predictable and measurable benefits to component reuse and complexity of software being built, as shown in chapter 7. Numeric reusability of extensions built using FAC, in terms of complexity of integration and coupling, rises with both the number of extensions and components and host application that FAC is integrated with.

There are limitations to FAC framework prototype implementation and FAC component model that may need to be addressed in the future for certain use cases.

Loose coupling between components causes overhead and thus slower communication. So for real-time components or real-time systems, the component model would have to be improved.

Complete decoupling doesn’t solve requirements such as reporting. It is common for software to show reports using data from multiple components, and with FAC model no effort was made to support those requirements. Thus, separate custom components would have to be built.

The FAC framework prototype may be improved in the future through the addition of a web access layer. By implementing the hooks and mappers as web services, FAC framework would be independent from the underlying technology and platform. This would also make it possible to scale horizontally, further improve extension reusability across component models and platforms.

BIBLIOGRAPHY

[1] Software Engineering Body of Knowledge (SWEBOK Version 3), 2014, IEEE Computer Society, accessed 1. 8. 2017.

[2] Software Engineering Body of Knowledge (SWEBOK Version 3), 2014, IEEE Computer Society, Chapter Software design, accessed 1. 8. 2017.

[3] Software Engineering Body of Knowledge (SWEBOK Version 3), 2014, IEEE Computer Society, Chapter Software construction, accessed 1. 8. 2017.

[4] Software Engineering Body of Knowledge (SWEBOK Version 3), 2014, IEEE Computer Society, Chapter Software engineering models and methods, accessed 1. 8. 2017.

[5] J. Basha, S.A. Moiz, 2012, Component Based Software Development: A State of Art, Proceedings of the International Conference On Advances In Engineering, Science And Management, pp. 599-604.

[6] K. Lau, Z. Wang, 2007. Software component models, IEEE Transactions on software engineering, vol. 33, no. 10

[7] G. T. Heineman and W. T. Councill, Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley Longman Publishing Co., 2001.

[8] M. Broy et al, 1998. What characterizes a (software) component?, Software – Concepts & Tools 19, pp. 49-56, Springer – Verlag

[9] B. Wallace, 2010. There is no such thing as a Component, accessed 15. 9. 2014., <http://existentialprogramming.blogspot.com/2010/05/hole-for-every-component-and-every.html>

[10] Bishop, J., 2007. C# 3.0 Design Patterns, O'Reilly Media

[11] Crnković, I. et al., 2011. A Classification Framework for Software Component Models, IEEE Transactions on Software Engineering: Volume 37, Issue 5

[12] S. Prstačić, I. Voras, M. Žagar, 2011, Nested Componentization for Advanced Web Platform Solutions, Proceedings of the 33rd International Conference on Information Technology Interfaces, pp. 609-614.

[13] S. Prstačić, M. Žagar, K. Kroflin, 2012, Interfaces of nested Web application framework as reusable software component, MIPRO 2012 Jubilee 35th International Convention Proceedings, pp. 439-443.

[14] S. Prstačić, M. Žagar, 2013, Nested Web Application Components Framework A Comparison to Competing Software Component Models, ENASE 2013 8th International Conference on Evaluation of Novel Approaches to Software Engineering, pp. 141-148.

[15] S. Alpaev, "Applied MVC Patterns", VikingPLoP '2005

[16] MSDN, “Understanding Service-Oriented Architecture”, http://msdn.microsoft.com/en-us/library/aa480021.aspx, accessed 15. 9. 2014.

[17] T. Erl, 2005, Service-Oriented Architecture (SOA): Concepts, Technology, and Design; Prentice Hall

[18] Drupal documentation, "API reference", http://api.drupal.org/api/drupal, accessed 15. 9. 2014.

[19] Joomla documentation, "Developing a MVC component", http://docs.joomla.org/Component accessed 15. 9. 2014.

[20] Grails, 2012. Groovy on Grails documentation, accessed 5 May 2012., <http://grails.org/doc/latest/>

[21] Django, Django documentation, accessed 15 September 2019., <https://buildmedia.readthedocs.org/media/pdf/django/2.2.x/django.pdf>

[22] Altman, P., 2011. How I write Django Reusable Apps, accessed 5 May 2012., http://paltman.com/2011/12/31/how-i-write-django-reusable-apps

[23] Symfony, 2012. The book, accessed 5 May 2012., <http://symfony.com/doc/current/book/index.htm>

[24] MSDN, "Model – View – Controller", http://msdn.microsoft.com/en-us/library/ff649643.aspx 15. 9. 2014.

[25] WordPress, WordPress Developer Documentation, accessed 15 September 2019., <https://codex.wordpress.org/Developer_Documentation>

[26] CiviCRM, CiviCRM documentation, accessed 23rd October 2019, https://docs.civicrm.org/

[27] Portlet specification 2.0, http://www.jcp.org/en/jsr/detail?id=286 [4/4/2011]

[28] Alpaev, S., 2005. Applied MVC Patterns, VikingPLoP

[29] Walker, S., DotNetNuke 4.0 Module Developers Guide, accessed 20 April 2012 <http://www.dotnetnuke.com/Resources/BooksandDocumentation/ProjectandTechnicalDocumentation/tabid/478/Default.aspx>

[30] A. Bassam et al., 2010, Reusable Software Components Framework, ECS'10/ECCTD'10/ECCOM'10/ECCS'10 Proceedings of the European conference of systems, and European conference of circuits technology and devices, and European conference of communications, and European conference on Computer science, pp. 126-130.

[31] Zhang, L., Zhao, et. al., Design and Realization of Database Accession Middleware System, 2007, Hongqiang Engineering Institute of Engineering Corps, PLA University of Sci.& Tech., Nanjing 210007)

[32] T. Erl, 2005, Service-Oriented Architecture: Concepts, Technology, and Design, ISBN: 0-13-185858-0

[33] Crnković, I., Larsson, S., Chaudron, M., 2005, Component-based DevelopmentProcess and Component Lifecycle, Journal of Computing and Information Technology - CIT 13, 2005, 4, 321-327

[34] M. Bell, 2008, Service-Oriented Modeling (SOA): Service Analysis, Design, and Architecture, ISBN: 0470255706, 9780470255704

[35] R. E. Caballero and S. A. Demurjian, Towards the Formalization of a Reusability Framework for Refactoring, ICSR-7, pp. 293-308.

[36] J. Guo and Y. Liao. 2003, Integrating Software Components through Wrapper Technologies, Proceedings of IASTED International Conference, Software Engineering and Applications, pp. 441-446.

[37] Ching-Seh Wu and I. Khoury, 2013, Web Service Composition: From UML to Optimization, 2013 Fifth International Conference on Service Science and Innovation (ICSSI)

[38] J.Lee, J. Kim, and Gyu-Sang Shin, 2003, Facilitating Reuse of Software Components using Repository Technology, Proceedings of the Tenth Asia-Pacific Software Engineering Conference

[39] S. M. Filho, H. Mariano, U. Kulesza, T. Batista, 2010, Automating Software Product Line Development: A Repository-Based Approach, 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), pp. 141-144.

[40] Y. Lin, H. Ye, 2010, An Approach for Modelling Software Product Line Architecture, International Conference on Computational Intelligence and Software Engineering (CiSE)

[41] L. Tan, Y. Lin and Huilin Ye, 2012, Modeling Quality Attributes in Software Product Line Architecture, Spring Congress on Engineering and Technology (S-CET)

[42] Michael Mattsson, Jan Bosch, and Mohamed E. Fayad, 1999, FRAMEWORK INTEGRATION PROBLEMS, CAUSES, SOLUTIONS, Communications of the ACM Vol. 42, No. 10

[43] S. A. Zamudio, R. Santaolaya and O.G. Fragoso, 2012, Restructuring Object-Oriented Frameworks to Model-View-Adapter Architecture, IEEE LATIN AMERICA TRANSACTIONS, VOL. 10, NO. 4,

[44] F. Brooks, 1995, The Mythical Man-Month

[45] F. Brooks, 1986, No Silver Bullet – Essence and Accident in Software Engineering, University of North Carolina at Chapel Hill

[46] A. Burgess et al, 2004, Guide to the Software Engineering Body of Knowledge, IEEE Computer Society

[47] P. Clemens er al, 2010, Documenting Software Architectures: Views and Beyond, 2nd ed., Pearson Education

[48] P. Kruchten, 1995, Architectural Blueprints — The "4+1" View Model of Software Architecture, IEEE Software 12 (6), pp. 42-50.

[49] Gyimothy, T., Ferenc R., Siket I., 2005, 'Empiricalvalidationofobject- oriented metrics on open source software for fault prediction', IEEE Trans. Softw. Eng., 2005, 3, (10), pp. 897 – 910

[50] Rosenberg, L., Stapko, R., and Gallo, A., 1999, Object-oriented metrics for reliability. Presented at the IEEE International Symposium on Software Metrics

[51] S. O'Grady, The RedMonk Programming Language Rankings: June 2019, accessed July 2019 https://redmonk.com/sogrady/2019/07/18/language-rankings-6-19/

[52] Gui, G., Scott, P.D., 2009, Measuring Software Component Reusability by Coupling and Cohesion Metrics, Journal of Computers, vol. 4 no 9

[53] AL-Badareen, A.B., Selamat, M.H., Jabar, M.A., Din, J., Tuarev, S., 2010, Reusable Software Components Framework, Advances in Communications, Computers, Systems, Circuits and Devices

[54] Rai, M., Virk, K. S., 2017, History of Software Engineering: Status of Software Component, Reusability and Quality, CSI Communications

[55] Bose, D., 2011, Component Based Development – application in software engineering, Indian Statistical Institute

[56] Kumar, S., Tomar, P., Nagar, R., Yadav, S., 2014, Coupling Metric to Measure the Complexity of Component Based Software through Interfaces, International Journal of Advances Research in Computer Science and Software Engineering

[57] Chidamber R., Kemerer F., 1992, A metrics suite for object oriented design, Center for Information Systems Research Sloan School of Management Massachusetts Institute of Technology

[58] Halstead, Maurice H., 1977, Elements of Software Science. Amsterdam: Elsevier North-Holland, Inc.

[59] McCabe, T., 1976, A complexity measure, IEEE Transactions on Software Engineering, vol. SE-2, NO.4

[60] Hummle, B., 2014, McCabe's Cyclomatic Complexity and Why We Don't Use It, CQSE blog, accessed October 2019 https://www.cqse.eu/en/blog/mccabe-cyclomatic-complexity/

[61] Singh, S., Thapa, M., Singh, S., Singh, G., 2010, Software Engineering – Survey of Reusability Based on Software Component, International Journal of Computer Applications, Volume 8, No.12, October 2010

[62] Narasimhan, V. L., Hendradjaya, B., 2004, A new Suite of Metrics for the Integration of Software Components

[63] Goulão, M., Brito e Abreu, F., 2007, An overview of metrics-based approaches to support software components reusability assessment, Software Quality Measurement: Concepts and Approaches, ICFAI Books

[64] Sarbjeet, S., Thapa, M., Singh, S. and Singh, G., 2010, Software Engineering – Survey of Reusability Based on Software Component, International Journal of Computer Applications (0975 8887), vol. 8, No 12

[65] Matisson, M., Bosch, J., Fayad, M.E., 1999, "Framework integration problems, causes, solutions", Communications of the ACM, Vol. 42, October 1999

[66] McConnell, S., 2004, Code Complete: A Practical Handbook of Software Construction, Second Edition, Microsoft press, p168-171

[67] Fiona Fui-Hoon Nah (2004) A study on tolerable waiting time: how long are Web users willing to wait?, Behaviour & Information Technology, 23:3, 153-163, DOI: 10.1080/01449290410001669914

[68] Horowitz, E., et al, 2007, Computer Algorithms, 2nd edition, Silicon press

[69] Highsmith, J., & Cockburn, A., 2001, Agile software development: the business of innovation. Computer, 34(9), 120–127. doi:10.1109/2.947100

[70] E. S. Cho, M. S. Kim, and S. D. Kim, “Component metrics to measure component quality” in Eighth Asia-Pacific Software Engineering Conference, 2001. APSEC 2001., Macau, China, 2001, pp. 419–426.

[71] Mijač, M., Staić, Z., Reusability Metrics of Software Components: Survey, 2015, Central European Conference on Information and Intelligent Systems, Faculty of Organization and Informatics University of Zagreb

[72] S. Bhattacharya and D.A. Perry, “Contextual reusability metrics for event-based architectures”, 2005, International Symposium on Empirical Software Engineering, Australia, 2005.

[73] H. Washizaki, H. Yamamoto and Y. Fukazawa, A Metrics Suite for Measuring Reusability of Software Components, 2003, 9th International Software Metrics Symposium Proceedings, Sydney, Australia, pp. 211-223.

[74] J. Al Dallal, 2009, Software Similarity-based Functional Cohesion Metric, IET Softw., vol. 3, no. 1, pp. 46-57

BIOGRAPHY

Svebor Prstačić, m. sc. c. sc. graduated in 2007., at the University of Zagreb, Faculty of Electrical Engineering and Computing with the topic "Building Data-integrated Information Systems For Education". His professional interests include free and open source software, component-based software engineering for the web and integrated platforms.

In 2005 and 2006 he worked on building data integration mechanisms for the Faculty's web, and is awarded the dean's award for "..dedicated work on building FER's Quilt CMS Content Management System and FER's e-Campus". Today, Quilt CMS is a successful commercial solution, used by more than 10 institutions in Croatia and abroad.

Since 2012 he is the president of Croatian Association for Open Systems and Internet.

Since 2015., he is the Head of IT department at the University of Zagreb, Faculty of Electrical Engineering and Computing.

He is the co-founder and managing director of a company Ekorre Digital, which specializes in advanced web solutions, and offers commercial services to users of Quilt CMS.

List of published articles

Conference articles

1. **Prstačić, Svebor**; Voras, Ivan; Žagar, Mario

   Nested Componentization for Advanced Web Platform Solutions // Proceedings of the 33rd International Conference on Information Technology Interfaces (ITI) 2011.

   Zagreb: SRCE, 2011. str. 609-614 (presentation, international review, in extenso, scientific)

2. **Prstačić, Svebor**; Žagar, Mario; Kroflin, Krešimir

   Interfaces of nested Web application framework as reusable software component // MIPRO 2012 Jubilee 35th International Convention Proceedings

   Rijeka: Croatian Society for Information and Communication Technology, Electronics and Microelectronics - MIPRO, 2012. pp. 439-443 (presentation, international review, in extenso, scientific)

3. Kroflin, Krešimir; **Prstačić, Svebor**; Žagar, Mario

   Framework for Implementation of Complex Dynamic Web Forms // MIPRO 2012 Jubilee 35th International Convention Proceedings

   Rijeka: Croatian Society for Information and Communication Technology, Electronics and Microelectronics - MIPRO, 2012. pp. 436-438 (presentation, international review, in extenso, scientific)

4. **Prstačić, Svebor**; Žagar, Mario

   A model for Web application and Web service peer-to-peer hosting network architecture // Proceedings of the ITI 2013 35th International Conference on Information Technology Interfaces - ITI 2013 / Jare, Iva (ur.).

   Cavtat, Hrvatska, 2013. pp. 335-340 (presentation, international review, in extenso, scientific)

5. **Prstačić, Svebor**; Žagar, Mario

   Nested Web Application Components Framework A Comparison to Competing Software Component Models // ENASE 2013 8th International Conference on Evaluation of Novel Approaches to Software Engineering / Leszek, Maciaszek ; Joaquim, Filipe (ur.).

   Angers, France, 2013. pp. 141-148 (presentation, international review, in extenso, scientific)

ŽIVOTOPIS

Svebor Prstačić, dipl. ing., diplomirao je 2007. godine na Fakultetu elektrotehnike i računarstva Sveučilišta u Zagrebu, s temom „Uspostava podatkovno integriranih informacijsko-obrazovnih sustava“. Njegovi profesionalni interesi uključuju slobodne i otvorene tehnologije, programsko inženjerstvo orijentirano na komponente i integrirane platforme za web.

Godine 2005. i 2006. radio je na izradi mehanizama integracije podataka za sustav koji pokreće javni web i intranet Fakulteta, a za to mu je dodijeljena nagrada dekana za "za predani rad na izgradnji FER-ovog sustava za upravljanje sadržajima *Quilt CMS* i FER-ovog e-Campusa".

Nakon diplomiranja, zaposlio se na FER-u u web timu kao programer. Sustav Quilt CMS je potom vrlo brzo prodan i nekim drugim fakultetima u Hrvatskoj s kojima je Svebor Prstačić preuzeo komunikaciju, te koordinirao razvoj sustava Quilt CMS za potrebe drugih fakulteta. Quilt CMS je tako od 2017. uspješno komercijalno rješenje koje koristi više od 10 institucija u Hrvatskoj i inozemstvu, a Svebor Prstačić je jedan od dvoje osnivača *spin-off* tvrtke koja od 2018. godine između ostalog pruža i komercijalne usluge za sustav Quilt CMS.

Od 2012. godine, predsjednik je Hrvatske udruge za otvorene sustave i internet kroz koju je vodio i podržao mnoge projekte popularizacije otvorenih tehnologija, posebice za primjenu u javnom sektoru i obrazovanju. 2013. godine je bio član Radne skupine za primjenu otvorenog koda i otvorenih normi, Povjerenstva Vlade Republike Hrvatske za koordinaciju informatizacije javnog sektora.

Od 2015. godine voditelj je Centra informacijske potpore (CIP), Sveučilišta u Zagrebu, Fakulteta elektrotehnike i računarstva. CIP je služba koja broji 10 inženjera koji brinu za cjelokupnu informacijsku infrastrukturu, podršku korisnicima, informacijsku sigurnost, te razvoj softvera za podršku nastavnim i poslovnim procesima, njihovu integraciju s vanjskim uslugama i izvorima podataka, te održavanje razvijenog softvera. Kao voditelj, blisko surađuje s upravom i pomaže osigurati održiv i nesmetan rad FER-a.

Suosnivač je i izvršni direktor tvrtke Ekorre Digital, koja je specijalizirana za napredna web rješenja, kao i komercijalne usluge korisnicima FER-ovog sustava Quilt CMS.

Kroz tvrtku od samog osnivanja pruža usluge koordinacije, planiranja i vođenja projekata razvoja softvera za domaće i strane tvrtke, najčešće napredna web portalna rješenja ali i IOT sustave u biomedicinskoj domeni.