UDC: 004.42.004.738.12 Info M str. 35-41

КОМПАРАТИВНА АНАЛИЗА РАЗВОЈА ЈЕДНОСТРАНИЧНИХ АПЛИКАЦИЈА КОРИШЋЕЊЕМ REDUX ПАТЕРНА И КОНВЕНЦИОНАЛНОМ МЕТОДОМ РАЗВОЈА
COMPARATIVE ANALYSIS OF SINGLE PAGE APPLICATION DEVELOPMENT USING REDUX PATTERN AND CONVENTIONAL METHOD OF DEVELOPMENT

Бојана Гошић, Саша Д. Лазаревић
Факултет организационих наука, Универзитет у Београду

РЕЗИМЕ: Појавом великог броја оквира који се заснивају на једној страни, по први пут се упознајемо са појмом компонената, њиховог стања које је потребно пратити, и синхронизације са осталим компонентама. Рад се бави изучавањем стратегије управљањем стањем у једностраничним апликацијама, конкретно, коришћењем Redux патерна. У самом раду, креиране су две једностраничне апликације (коришћењем оквира Angular 8), где једна од њих не користи патерн за управљање стањем, док је друга добијена на основу прве апликације, са истим софтверским захтевима, али коришћењем Redux патерна. Обе апликације ће користити исти сервер, API израђен коришћењем .NET Core оквира. У раду ће се детаљно размотрити разлике у комуникацији између компонената, употребом једног и другог приступа, и колико је надградња и одржавање апликације олакшано када се користи нека од стратегија за управљање стањем. На основу пажљиво одабраних софтверских метрике, упоредићемо апликације и дати препоруке када је боље користити патерн при изради апликације, односно када је конвенционална метода развоја примеренија за коришћење.
КЉУЧНЕ РЕЧИ: управљање стањем, Redux, једностраничне апликације, компаративна анализа
ABSTRACT: Appearance of vast number of frameworks, based on single page, introduced term components to us, state of components that should be tracked and synchronization with other components. Paper will introduce strategies for state management in single page applications, specifically, using Redux pattern. In paper, we created two single page applications (using framework Angular 8), where one of them is not using state management pattern, and other is derived from first one, with same software requirements, but using Redux pattern. Both applications will use same server, API implemented using .NET Core framework. In paper, we will tackle, in details, difference in communication between components, both with and without state management pattern, and explain how adding new features to the application and maintenance is facilitated. Based on carefully chosen software metrics, we will compare applications, and give recommendations in which case it is better to use state management pattern, and when it is more appropriate to use conventional method of development.
KEY WORDS: state management, Redux, single page applications, comparative analysis

1. УВОД

Све већа популарност једностраничних апликација довела је до потребе за проналажењем што бољих решења за њихову имплементацију. Као један од главних проблема који су уочени при развоју једностраничних апликација јесте начин на које ће компоненте да комуницирају и да се преноси стање од једне до друге компоненте. Уколико је овај део око комуникације и преношења стања између компонената добро имплементиран, то касније утиче на разумљивост кода, комплексност апликације и њене перформансе.

Ови проблеми довели су до појаве првих патерна за управљање стањем. Једно од најпознатијих решења јесте Redux патерн, који води порекло од *Flux*-а. "*Flux* је патерн архитектуре. Развијен је као алтернатива тренутним *MV** патернима који се користе у модерним оквирима попут *Angular*-а, *Backbone* или *Ember*."[1] *Flux* не представља конкретну библиотеку или имплементацију која може да се користи. У студијском примеру коришћена је *NgRx* библиотека, која је заправо имплементација *Redux* патерна, прилагођена једностраничним апликацијама.

На основу бројних истраживања и примера који су доступни, циљ је да се покаже шта нам то доноси *Redux* у односу на имплементацију апликације без патерна? Да ли нам увођење патерна доводи до повећавања број линија кода које морамо да имплементирамо? Да ли се сложеност апликације повећава или смањује? Шта се дешава у случају када треба да додамо нову функционалност у апликацији, како се понаша апликација која користи патерн за управљање стањем, а како она која не користи патерн? Колико је олакшано долажење компоненте до потребних података када не мора да се обраћа родитељској компоненти, већ може подацима да приступи директно из складишта? Да ли се број захтева ка серверу смањује, када не морамо поново да учитава податке сваки пут када корисник напусти и врати се на неки од погледа?

2. СОФТВЕРСКЕ МЕТРИКЕ

"Метрике представљају средства за праћење напретка, дају боље предикције циљева при изради софтвера, и доприносе развоју софтвера са минимумом грешака." [2]

2.1 *Број линија кода по компоненти*

"Број линија кода које садрже бар један карактер који није празан карактер, нити је део коментара."[9] Ову метрику можемо посматрати на нивоу пројекта, али и на нивоу компонената. На нивоу пројекта би представљала укупан број свих линија кода. Не можемо поставити конкретну границу, јер са новим софтверским захтевима од

klijenta, povećava se i broj linija koda koje moramo da implementiramo.

Kada govorimo o broju linija koda po komponenti *(CLOC – component line of code*) možemo je izračunati na sledeći način: [4]

$$CLOC = \frac{\sum_{i=0}^{n} Zi}{n} \quad Zi = \begin{cases} 1, \; ako \;\; je \;\; line_{c_i} \geq 150 \\ 0, \; ako \;\; je \;\; line_{c_i} < 150 \end{cases}$$

gde je

- n – broj kompонената,
- $line_{C_i}$ – broj linija koda u komponenti .

Broj linija koda nam svedoči o komplekscnosti komponente, i preporučeno je da ne prelazi 150 linija, po komponenti, kako bi se zadržala čitljivost komponente. Svaka vrednost preko toga, znači da je potrebno komponenту refaktorisati i po mogućnosti razdvojiti u više smislenih komponenti.

2.2. *Broj metoda*

"Broj metoda izmeren na nivou klase i predstavlja indikator složenosti klase." [4] Računa se prostim sabiranjem metoda koje se nalaze u klasi. Mi ćemo ih opet posmatrati na nivou komponenata, ali i na nivou projekta. Preporuka je da se broj metoda kreće od 2-7 kako bi klasa ostala čitljiva. Na nivou komponenata taj broj ćemo dobiti na sledeći način:

$$NOM = \frac{\sum_{i=0}^{n} M_i}{n}$$

gde je

- M_i – broj metoda u komponenti
- n – broj komponenata

2.3. *Ciklična komplekscnost*

"Ciklična komplekscnost predstavlja broj putevа kojima je moguće proći u jednom softverskom modulu. Povećanjem broja putevа kojim može da se prođe u jednom modulu, tok postaje složeniji, tezi za testiranje i nadgrađivanje."[4]

Ova metrika je zasnovana na teoriji grafova. "Računa se merenjem linearno nezavisnih putevа u grafu i predstavlja se brojem."[5] Cikličnu komplekscnost (*CC – cyclomatic complexity*) je prvi put predstavio McCabe i originalno se računa na sledeći način: [8]

$$CC = e \; - n + 2.$$

gde je

- CC – ciklična komplekscnost toka programa koji trenutno posmatramo
- E – broj grana u grafu
- N – broj čvorova u grafu

Kako ćemo cikličnu komplekscnost računati i na nivou komponenata, koristećemo navedenu formulu: [6]

$$ACCC = \frac{\left(\sum_{i=0}^{n} \sum_{k=0}^{l} CC_k\right)}{n}$$

gde je

- $ACCC$ – prosečna ciklična komplekscnost komponenata (eng. *average component cyclomatic complexity*)
- n – broj kompонената
- l – broj funkcija u komponenti
- CC_k– ciklična komplekscnost funkcije k

Duги niz godina ciklična komplekscnost je bila jedan od glavnih pokazatelja kvaliteta koda. Međutim, u novijim radovima je pokazano da poređ cikličnе komplekscnosti, treba razmatrati i kognitivnu komplekscnost.

2.4. *Kognitivna komplekscnost*

"Kognitivna komplekscnost se ne oslanja samo na matematički model izračunavanja komplekscnosti, već polazi od toga koliko je truda potrebno programeru da razume dati kôd." [7]

Pri izračunavanju kognitivne komplekscnosti uzimaju se u obzir sledeći parametri:

- Ugnježdenost – dodaju se poeni kada imamo ugnježdene strukture
- Uslovi – dodaju se poeni za svaki uslov koji imamo u nekom toku
- Ignorišu se strukture koje više izraza spajaju u jedan (npr. provera *null* vrednosti operatorom u jednoj liniji) [7]

Izračunavanje krajnjeg broja poena koji se dobijaju ovakvim pristupom, prepustićemo softveru *SonarQube*.

Kada govorimo o računanju komplekscnosti na nivou kompонената, koristićemo sledeću formulu:

$$CCC = \frac{\sum_{i=0}^{n} CC_i}{n}$$

gde je

- CC_i – kognitivna komplekscnost komponente
- n – broj kompонената

2.5. *Međuzavisnost kompонената*

Odnosi se na broj inpput parametara koje se prenose od jedne do druge komponente. U kontekstu jednostraničnih apликacija međuzavisnost kompонената (*CCF – component coupling factor*) se računa na sledeći način: [6]

$$CCF = (\sum_{i=0}^{n} C_i) / (n * (n-1))$$

где је

- n – број компонената,
- C_i – број зависности између компоненте i и других компонената.

2.6. *Индекс преношења стања*

У једностраничним апликацијама, компоненте представљају градивне јединице сваке од апликација. Одлука о томе како ће се оне делити, комуницирати и добијати податке, представља једну од најважнијих одлука при развоју ових апликација.

Индекс преношења стања се рачуна на следећи начин: [6]

$$STR = \frac{\sum_{i=0}^{n} PS_i / SS_i}{n}$$

где је

- n – број компонената,
- PS_i – је број инпут параметара које компонента i добија од родитеља,
- SS_i – број инпут параметара који се преносе од компоненте i до деце компонената.

2.7. *Дубина графа зависних компонената*

Компоненте и њихове везе можемо представити у виду графа. Што је већа дубина графа, и компоненте више зависе од родитеља, то је теже препознати одакле стижу подаци. Дубину графа зависних компонената можемо израчунати на следећи начин: [6]

$$CDGD = \frac{\sum_{j=0}^{m} deap_{CDG_j}}{m}$$

где је

- m – број графова који су зависни од компоненте j
- $deap_{CDG_j}$ – дубина графа j

3. СТУДИЈСКИ ПРИМЕР

Креиран је софтверски систем који омогућава праћење и координацију активности запослених и студената на факултету. Приступ систему имају три врсте корисника: административни радници, наставно особље и студенти.

Како би студенти и наставно особље добили приступ систему, потребно је да од стране административних радника буду уписани у систем, након чега ће им бити додељена емаил адреса и насумична лозинка коју могу да промене након приступа систему. Административни радници, поред додавања друге две врсте корисника у систем, су задужени и за додавање предмета, за које означавају наставно особље које је задужено за њега. Они имају комплетан преглед свих информација везаних за студенте и наставно особље, као и могућност претраге и филтрирања њихових података по најразличитијим критеријумима.

Наставно особље је задужено за одржавање додељених предмета, попут додавања додатних информација, као што су наставне јединице које се обрађују на датом предмету.

Трећа врста корисника јесу студенти, који након додељивања и приступања свом налогу, могу да складно смеру коме припадају, изаберу предмете које желе да слушају, и у испитном року да пријаве изабране предмете за полагање.

3.1. *Имплементација – Конвенционална метода*

Систем је прво имплементиран без коришћења патерна, а затим је рефакторисан тако да користи патерн. Како је серверски део апликације исти у оба случаја, и није тема овог рада, неће бити приказан. Као пример ћемо узети пријављивање предмета за слушање од стране студента.

Пре него што кренемо на прављење компонената и било какве логике, дефинисали смо моделе за студента, предмет и слушање који одговарају моделима које добијамо са сервера. Како би студент пријавио предмет за слушање, потребно је да учитамо листу предмета које он може да пријави, тренутно пријављене предмете за слушање и да израчунамо колико ЕСПБ поена је до сада студент пријавио. У те сврхе смо направили сервис, који комуницира са *API*-јем и добија неопходне податке.

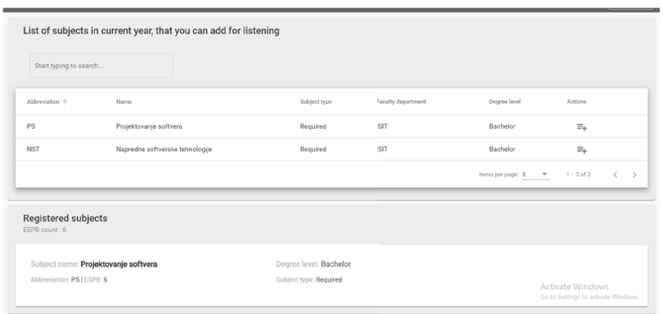
Пријављене предмете и оне које је могуће пријавити, сходно смеру студента, учитавамо на стандардан начин коришћењем *HttpClient*-а. Након што смо добили предмете израчунат је и број ЕСПБ поена за пријављене предмете. Направљена је одговарајућа компонента, која при иницијализовању учитава потребне податке на следећи начин, и иницијализује број ЕСПБ поена.

```
ngOnInit() {
    this.isLoading = true;

    forkJoin(this.subjectsService.getSubjectsInFacultyDepartment(this.currentUser.facultyDepartment),
        this.studentsService.getStudentsListenings(this.currentUser.id, Guid.createEmpty()))
            .subscribe(data => {
                this.subjectsInFacultyDepartment = data[0];
                this.studentsListenings = data[1].result;

                this.countESPB();
                this.isLoading = false;
            })
}
```

Слика 1. *Иницијализовање података у компоненти*

Како бисмо испратили принцип контејнер/презентационих компоненти, поред контејнер компоненте *StudentsListenings,* направљене су две презентационе компоненте. Једна презентациона компонента прима листу предмета за слушање, док друга добија листу пријављених предмета путем *@Input* параметара.

Након учитавања података, студент пред собом има следећи екран:



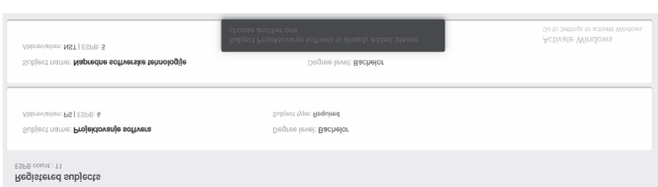
Слика 2. *Страница пријављивање предмета за слушање*

Студент кликом на дугме може да пријави нови предмет за слушање, при чему проверавамо да ли је предмет већ пријављен, и уколико није додајемо га у листу предмета за слушање.

```
registerSubject(subject: Subject) {
    const listening = new Listening();
    listening.studentId = this.currentUser.id;
    listening.subjectId = subject.id;

    this.studentsService.createListening(listening)
        .subscribe(response => {
            if (response.success) {
                this.studentsListenings.push(response.result);
                this.espbCount += response.result.subject.espb;
            } else
                this.errorService.showError(response.message);
        })
}
```

Слика 3. *Пријава предмета за слушање*

Уколико је предмет већ додат за слушање, студент добија оговарајућу поруку о томе, док у супротном смо додали предмет у листу и повећали број ЕСПБ поена.



Слика 4. *Пријављивање предмета за слушање*

Како бисмо било шта радили на страници, потребно је да учитамо предмете, и део кода који комуницира са сервером не бисмо могли да избегнемо. Поставља се питање шта се дешава када студент оде са стране за пријављивање предмета, и одлучи поново да се врати, у једном свом случају коришћења. Да ли је потребно поново учитати податке које смо једном добили, или можда постоји начин да се подаци сачувају код клијента?

Након што смо учитали податке, примећујемо да сваки пут морамо да израчунамо број поена које студент има, јер тај податак немамо у бази. Свако ново пријављивање предмета, резултује потребом да додамо предмет и поново израчунамо број поена. Постоји доста кода који се понавља, а могао би се избећи. Да ли постоји начин да се број поена повећа чим пријавимо нови предмет за слушање, без потребе да сваки пут размишљамо да ли смо га увећали? Поставља се питање колико се на овај начин отежава разумевање кода који је написан, као и како се повећава циклична комплексност метода у компоненти?

Већ на овом једноставном примеру смо приметили велики број проблема. Постоје делови кода које можемо у потпуности да избегнемо, да повећамо читљивост кода на тај начин и смањимо труд програмера који би у будућности додавали неке нове функционалности.

Запитајмо се шта би било у случају да желимо да омогућимо студенту одјаву предмета? Морали бисмо да прођемо кроз неколико корака. Одјава предмета, проверавање успешности одјаве, приказ поруке уколико није одјављен, и несмањивање броја пријављених поена. Уколико успешно одјави предмет, морамо да водимо рачуна да поред уклањања предмета из низа пријављених предмета, смањимо и број поена које је студент пријавио. Примећујемо да се неке провере понављају, као што је провера успешности одјаве, промена листе пријављених предмета и мењање броја ЕСПБ поена. Да ли можемо ова три корака само једном написати, без размишљања шта ће се десити ако желимо да одјавимо предмет?

Идентификовали смо неколико места на којима можемо да направимо побољшања тако да кôд буде читљивији, додавање нових функционалности лакше, да смањимо број линија кôда у компоненти, олакшамо цикличну и когнитивну комплексност, као и да смањимо број захтева ка серверу.

3.2. *Имплементација – Redux патерн*

Моделе, као и поглед код компонената, искористићемо од постојеће апликације, јер се софтверски захтеви нису променили. Сервиси који комуницирају са сервером такође остају непромењени. Оно што ће се променити код сервиса јесте место њиховог позива, као и елиминација потребе за поновним учитавањем података, након што их је корисник већ једном учитао, а затим напустио страницу. Самим тим кôд методе којим добијамо предмете који се налазе на смеру на коме студент студира, као и листа до сада пријављених предмета, не подлеже било каквим променама.

Време је да наша компонента учита тражене податке. У претходном случају, компонента је директно комуницирала са сервисом ради учитавања података. Како *Redux* уводи нове елементе који су му потребни, а ми не желимо да компонента зна за све њих, искористићемо *Façade* сервис, како бисмо послали акцију за учитавање података. Акција се састоји од типа и параметара, па је кôд акције за учитавање предмета на смеру на коме се налази студент, дефинисан на следећи начин:

```
export class LoadSubjectsInFacultyDepartment implements Action {
    readonly type = SubjectActionTypes.LoadSubjectsInFacultyDepartment;

    constructor(public payload: number) { }
}
```

Слика 5 *.Акција "Учитај предмете"*

На исти начин смо дефинисали акцију за предмете које студент слуша.

Како бисмо позвали акције из компоненте, а не желимо да директно приступамо складишту и његовој методи

dispatch(), искористићемо *Façade* сервис. Овај сервис апстрахује све елементе Redux-а, и позива тражену акцију. Како бисмо избегли непотребне позиве ка серверу, проверавамо прво да ли су подаци већ учитани, и уколико нису учитавамо их. На овај начин решавамо проблем сувишних позива ка серверу када корисник напусти и поново се врати на страницу.

```
getSubjectsInFacultyDepartment(facultyDepartment: number) {
    this.store.pipe(select(fromStore.getSubjectsLoaded)).subscribe(subjectsLoaded => {
        if (!subjectsLoaded)
            return this.store.dispatch(new LoadSubjectsInFacultyDepartment(facultyDepartment));
    })
}
```

Слика 6. *Слање акције "Учитај предмете"*

Поред позива акције, компоненти су потребни и подаци. И ту нам помаже наш *Façade* сервис, који ће искористити селекторе и пружити компоненти неопходне податке, без да она има било какву потребу да директно комуницира са складиштем.

```
get subjects$(): Observable<Subject[]> {
    return this.store.pipe(select(fromStore.getSubjects));
}
```

Слика 7. *Селектор за узимање предмета из складишта*

Након што је акција позвана, и направљен селектор који ће да узима податке из складишта, потребан је неко ко ће податке вратити са сервера. Дефинишемо ефекат који ослушкује акције типа "Учитај предмете са смера", и када се подаци успешно врате, шаље нову акцију, "Предмети успешно учитани".

```
@Effect()
subjectsInFacultyDepartment$ = this.actions$.pipe(
    ofType(SubjectActionTypes.LoadSubjectsInFacultyDepartment),
    mergeMap((action: LoadSubjectsInFacultyDepartment) =>
        this.subjectsService.getSubjectsInFacultyDepartment(action.payload).pipe(
            map(response => new LoadSubjectsInFacultyDepartmentSuccess(response))
        ))
)
```

Слика 8. *Ефекат за учитавање предмета*

Reducer је ту да на основу дате акције, замени постојеће стање у складишту и спусти нове податке.

Компонента, не зна да се било шта од овога дешава у позадини, већ само позива сервис да изврши акцију и чека на податке. Када смо позвали акције за учитавање неопходних података, на нама је да у компоненти искористимо селекторе и сачекамо да дође до промене података. Приметићемо да више немамо методу која нам рачуна број ЕСПБ поена, њих сада рачуна селектор на основу пријављених предмета за слушање, који се налазе у складишту.

```
ngOnInit() {
    this.isLoading$ = this.studentsFacadeService.isLoadingListenings$;
    this.studentsListenings$ = this.studentsFacadeService.listenings$;
    this.subjectsInFacultyDepartment$ = this.subjectsFacadeService.subjects$;
    this.espbCount$ = this.studentsFacadeService.getESPBCount(this.currentUser.id);
    this.studentsFacadeService.getStudentsListenings(this.currentUser.id, Guid.createEmpty());
    this.subjectsFacadeService.getSubjectsInFacultyDepartment(this.currentUser.facultyDepartment);
}
```

Слика 9. *Иницијализовање **Redux** компоненте*

На овај начин свака промена у складишту, путем селектора, послаће обавештење компоненти и она ће освежити свој поглед, без потребе да било шта поново узимамо са сервера или рецимо рачунамо број ЕСПБ поена.

Следећи случај који смо имали, јесте пријављивање предмета за слушање од стране студента. Компонента више није задужена са поновно рачунање ЕСПБ поена, нити додавање пријављеног предмета у листу пријављених предмета, све што она сада зна јесте да пошаље акцију која треба да пријави предмет за слушање.

```
registerSubject(subject: Subject) {
    const listening = new Listening();
    listening.studentId = this.currentUser.id;
    listening.subjectId = subject.id;

    this.studentsFacadeService.createListening(listening);
}
```

Слика 10. *Регистровање предмета за слушање*

Тако смо избегли проблем понављања кода, који смо имали у другој клијентској апликацији. Такође, осигурали смо се да уколико дође до неког новог случаја коришћења, одјављивање предмета, компонента не треба да води рачуна о овим стварима, већ само да пошаље акцију која ће да одјави предмет.

По угледу на горе описане кораке, и акција "Пријави предмет за слушање", ће имати своју акцију, ефекат, место у *reducer*-у, и на крају бити додата у складиште. Компонента већ ослушкује потребне селекторе, и чим дође до промене у складишту, она ће бити обавештена и освежиће поглед.

Већ у наведеном примеру видели смо бројна побољшања у односу на апликацију која не користи патерн за управљање стањем. Смањили смо понављање кода у самој компоненти и поједноставили будуће случајеве, као што је рецимо одјављивање предмета. Тако је смањена циклична и когнитивна комплексност функција у компоненти. Уколико корисник напусти страну, и поново се врати, не морамо поново да учитавамо податке, на тај начин смањујемо и број захтева ка серверу.

4. АНАЛИЗА ПРОБЛЕМА

Описане софтверске метрике, у зависности од тога да ли се односе на *TypeScript* кôд, или на преношење стања између компонената, поделићемо у две групе: *TypeScript* метрике и *Angular* стања. *TypeScript* метрике се односе на читљивост и лакше одржавање кода, док *Angular* метрике стања показују колико је тешко компонентама да међусобно комуницирају.

Метрике које се односе на *TypeScript* ће бити мерене на нивоу компонената и на нивоу пројекта. Оваква подела нам је битна, јер је компонента основна јединица грађе једностраничних апликација, и важно је да се читљивост и одржавање компонената олакша. Како *Redux* уводи нове концепте, као што су акције, reducer-и, ефекти и селектори, који неће бити у оквиру компоненте, приказане метрике смо мерили и на нивоу пројекта.

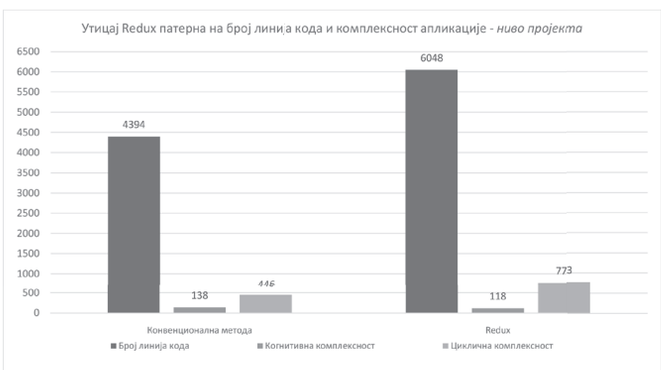
Друга група се односи на метрике које се тичу стања *Angular* компонената, као и односа између њих. Начин комуникације између компонената утиче на повезаност између класа као и кохезију метода и последично на квалитет самог кôда.

Измерене вредности метрика у студијском примеру су приказане у следећој табели.

	Метрика	Ниво	Конвенционална метода	Redux патерн	Препоручена вредност
TypeScript метрике	*Број линија кода*	компонената	0.2	0.03	0
		пројекта	4394	6048	/
	Број метода	компонената	6.65	5.24	2-7
		пројекта	/	/	/
	Циклична комплексност	компонената	9.86	7.48	10
		пројекта	446	773	/
	Когнитивна комплексност	компонената	4	1.55	5
		пројекта	138	118	/
Angular метрике стања	*Међузависност компонената*	компонената	1.3	1.06	1
	Индекс преношења стања	компонената	0.14	0.03	0
		SubjectDetails компонента	0.11	0	/
	Дубина графа зависних компонената	компонената	1.27	1	1

Слика 11. *Измерене вредности метрика*

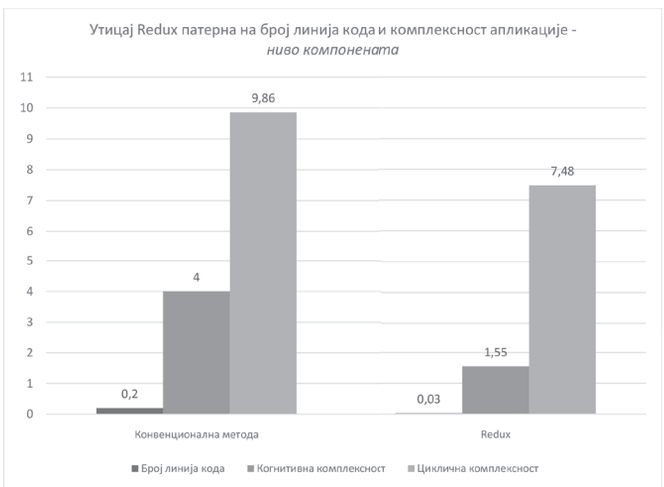
Прва група се тиче квалитета написаног *TypeScript* кôда. Као што смо рекли, *Redux* са собом уводи нове делове које је потребно имплементирати. Самим тиме број линија кода, као и број метода, које је потребно имплементирати на нивоу пројекта се повећао. Међутим поставља се питање како је то утицало на комплексност у нашој апликацији. Поред очекиваног повећања броја линија кода које је потребно имплементирати, видимо да апликацију заправо није теже разумети, није дошло до повећања когнитивне комплексности апликације, напротив, она је чак и незнатно смањена, са 138 на 118. Са друге стране, циклична комплексност апликације се повећала увођењем патерна са 446 на 773. Повећање цикличне комплексности, на нивоу пројекта, можемо оправдати повећањем неких услова, који се рецимо тичу постојања података у клијентској бази, како се не би поново захтевали са сервера. Иако је патерн захтевао доста више кôда да би се имплементирале функционалности, то није условило претеран раст комплексности пројекта.



Илустрација . *Утицај **Redux-a** на број линија кода и комплексност апликације – ниво пројекта*

Са друге стране, интересује нас какве је промене патерн условио на нивоу компонената, и да ли ту видимо нека побољшања. Када смо објашњавали начин имплементације патерна, видели смо да се доста логике помера са нивоа компонената, као што је пренос података другим компонентама, и селектовање података, јер сада имамо складишта и селекторе за такве ситуације. Сада је на компонентама да путем *Façade* сервиса пошаљу акцију када дође до промене, и да се претплате на селекторе, преко којих ће добијати податке из складишта. Захваљујући томе, и добијени подаци показују да је на нивоу компонената смањен број линија кôда које је потребно имплементирати, као и циклична и когнитивна комплексност компонената. Сада су компоненте много читљивије, лакше је додавати нове функционалности и разумети одакле долазе подаци које компонента користи.

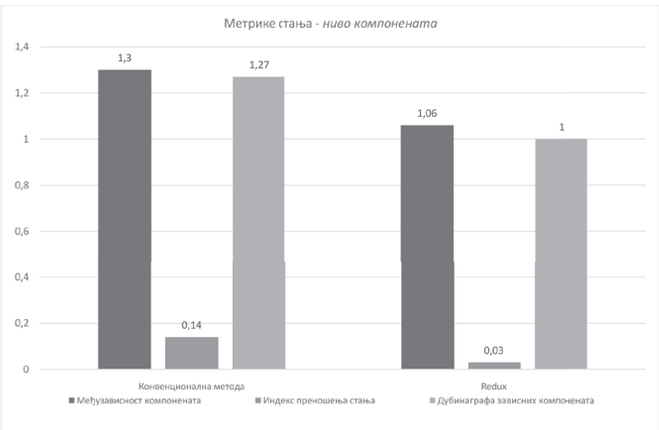
Однос броја имплементираних метода и комплексности апликације је сличан представљеном односу између броја линија кôда и комплексности.



Илустрација 2. *Утицај **Redux-a** на број линија кода и комплексност апликације – ниво компонената*

Уочили смо да је код већине метрика дошло до побољшања параметара на нивоу компонената, што нам је и био циљ, будући да је компонента једна од основних јединица грађе *Angular* апликација. Међутим, имплементирањем делова који чине *Redux*, имали смо нека погоршања параметара на нивоу пројекта. Увођење нових делова, како би се имплементирао *Redux,* резултовало је већим бројем линија кода и метода које је потребно имплементирати, што је допринело погоршању параметра цикличне комплексности на нивоу пројекта.

Са друге стране, уколико посматрамо метрике које се тичу преноса стања између компонената, видимо да је дошло до апсолутног побољшања у корист апликације која користи патерн. Побољшање је донекле и било очекивано, јер сада свака од компонената има могућност приступа подацима из складишта, и не мора да зависи од родитељске компоненте при добијању података. Видимо да се број *@ Input* параметара које компоненте примају смањио, као и број параметара које даље прослеђују деци компонентама, о чему нам говоре метрике са графикона. На овај начин је смањена и чврста повезаност између компонената, што нам омогућује да можемо поново да користимо компоненте.



Илустрација 3. *Метрике стања – ниво компонената*

Увођењем складишта смо дали могућност било којој од компонената да може да приступи директно подацима, и на тај начин смо осигурали један од главних бенефита увођења овог патерна, а то је смањење потребе за преношењем података великом броју нивоа у хијерархијском стаблу компонената. Апликација на тај начин постаје лакша за разумевање, што је и показано параметром когнитивне комплексности, како на нивоу компонената, тако и на нивоу целе апликације. Побољшања параметара *Angular* стања су наравно имала своју цену, а то је већи број линија кода и метода које смо морали да имплементирамо.

5. ЗАКЉУЧАК

Настанак једностраничних оквира, решио је проблем константног учитавања страница са сервера, који су имале вишестраничне апликације, али је донео нека нова питања и проблеме. Како се свака страница, засебно, није више учитавала са сервера, имали смо проблем да пренесемо стање од једне до друге компоненте. У том тренутку на сцену ступа *Redux* патерн, чијим смо увођењем осигурали централизовано место за чување стања и тако олакшали компонентама да приступе истом. Најједноставнији начин да се проблеми једностраничних апликација приближе читаоцу, било је креирање апликације која не користи патерн за управљање стањем, а затим рефакторисање клијентске апликације у ону која користи патерн. Све ово смо одрадили над истим сервером, *API*-јем израђеним коришћењем *.NET Core* оквира.

Разлике које су настале увођењем патерна посматрали смо са два аспекта. Са једне стране смо посматрали какве су промене настале са стране квалитета написаног *TypeScript* кода, *TypeScript* метрике, док смо са друге стране пратили шта се дешава са стањем, *Angular* метрике стања. На основу измерених *TypeScript* метрика, видели смо да је апликација са патерном захтевала више кôда да би се имплементирала, и повећала је комплексност целокупне апликације, док смо на нивоу компонената ипак остварили нека побољшања. Упркос повећању броја линија кôда и метода које је требало имплементирати, саму апликацију није било теже разумети, напротив, когнитивна комплексност се смањила. Када су у питању *Angular* метрике стања, приметили смо да је увођењем патерна дошло до апсолутног побољшања измерених вредности у корист апликације са патерном.

Током рада потрудили смо да дамо одговор на једно главно питање, да ли је увођење свих нових делова које са собом носи *Redux,* и времена да се патерн усвоји и имплементира, вредно труда? На дуже стазе, да. Након улагања иницијалног напора да се елементи *Redux*-а имплементирају и повежу, свака следећа одлука, која се тиче добијања података, одлучивања о томе шта урадити у случају успешне акције, шта у случају неуспешне, разумевања одакле долазе подаци и како освежити податке на једној компоненти уколико их друга промени, постало је знатно брже и интуитивније. Пун потенцијал патерна би се показао на "*enterprise*" апликацијама, где би константним доласком захтева од стране клијента, настала и већа потреба за проширењем апликације и већим угњеждавањем компонената. Повећањем дубине хијерархијског стабла компонената, где више програмера ради на једној функционалности, и где полако постаје немогуће одредити одакле долазе подаци, патерн би увео одлучујућу разлику, што смо и видели по резултатима метрика.

На крају, остаје питање, да ли ћемо на самом почетку утрошити више времена да имплементирамо чврсте темеље апликације, или ћемо сачекати да се прошити до мере у којој више не знамо шта је чија одговорност и ко са ким комуницира?

6. ЛИТЕРАТУРА

[1] **Marc Garreau, Will Faurot**, *Redux in action*, Manning Publications, June 2018
[2] **David Bellin, Manish Tyagi, Maurice Tyler,** *Object-Oriented Metrics: An Overview,* [приступано августа 2020]
[3] SonarQube, [приступано августа 2020]
[4] **Jorge Arturo Wong-Mozqueda, Robert Haines and Caroline Jay,** Is Code Quality Related to Test Coverage?, [приступано августа 2020]
[5] **Ayman Madi, Oussama Kassem Zein and Seifedine Kadry**, On the Improvement of Cyclomatic Complexity Metric, [приступано августа 2020]
[6] **Yiwei Lin , Min Li , Chen Yang , Changqing Yin,** A Code Quality Metrics Model for React-Based Web Applications, [приступано августа 2020]
[7] **G. Ann Campbell,** Cognitive Complexity, [приступано августа 2021]
[8] **Сања Костић, Саша Д. Лазаревић,** *Компаративна анализа квалитета програмског кода добијеног развојем вођеним тестом и конвенционалном методом развоја*, ИнфоМ, 2018
[9] **Kaushal Bhatt, Vinit Tarey, Pushpraj Patel,** *https://www.researchgate.net/publication/281840565_Analysis_Of_Source_Lines_Of_CodeSLOC_Metric*, [приступано фебруара 2021]

Бојан Гошић, P3 група
Контакт: gosic.bojana115@gmail.com
Област интересовања: .NET, TypeScript, Redux

проф. др Саша Д. Лазаревић,
Универзитет у Београду – Факултет организационих наука
Контакт: sasa.lazarevic@fon.bg.ac.rs
Област интересовања: софтверско инжењерство, информациони системи, базе података, системи за управљање документацијом, .НЕТ платформа