Razvoj okruženja za automatsko testiranje objektno orijentiranih programa

Dražić, Marino

SVEUČILIŠTE U SPLITU
PRIRODOSLOVNO MATEMATIČKI FAKULTET

DIPLOMSKI RAD

RAZVOJ OKRUŽENJA ZA AUTOMATSKO TESTIRANJE OBJEKTNO ORIJENTIRANIH PROGRAMA

Marino Dražić

Split, rujan 2022.

Temeljna dokumentacijska kartica

Diplomski rad

Sveučilište u Splitu
Prirodoslovno-matematički fakultet
Odjel za informatiku
Ruđera Boškovića 33, 21000 Split, Hrvatska

RAZVOJ OKRUŽENJA ZA AUTOMATSKO TESTIRANJE OBJEKTNO ORIJENTIRANIH PROGRAMA

Marino Dražić

SAŽETAK

U ovom radu biti će opisan razvoj okruženja ili aplikacije za automatsko testiranje programa. Specifični fokus okruženja je testiranje velikih količina objektno orijentiranih programa u kratko vrijeme. Također biti će opisane razne tehnologije korištene u razvoju sustava.

Ključne riječi: Imperativna paradigma. Objektno orijentirano programiranje, Okviri za testiranje, Nextjs

Rad je pohranjen u knjižnici Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu

Rad sadrži: 35 stranica, 9 grafičkih prikaza, 0 tablica i 16 literaturnih navoda. Izvornik je na hrvatskom jeziku.

Mentor: **doc. dr. sc. Divna Krpan,** *docent Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu*

Ocjenjivači: **doc. dr. sc. Divna Krpan,** *docent Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu*

izv. prof. dr. sc. Saša Mladenović, *izvanredni profesor Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu*

doc. dr. sc. Goran Zaharija, *docent Prirodoslovno-matematičkog fakulteta, Sveučilišta u Splitu*

Rad prihvaćen: rujan 2022.

Basic documentation card

Thesis

University of Split
Faculty of Science
Department of informatics
Ruđera Boškovića 33, 21000 Split, Croatia

DEVELOPMENT OF AN ENVIROMENT FOR AUTOMATIC TESTING OF OBJECT-ORIENTED PROGRAMS

Marino Dražić

ABSTRACT

This master's thesis will describe the development process of a integrated testing environment, primarily focused on testing object-oriented programs used in education. Along with all the various technologies that make a project like this possible.

Key words: Imperative paradigm, Object-oriented programming, Testing environments, Nextjs

Thesis deposited in library of Faculty of science, University of Split

Thesis consists of: 35 pages, 9 figures, 0 tables and 16 references

Original language: Croatian

Mentor: **Divna Krpan, Ph.D.** *Assistant Professor of Faculty of Science, University of Split*

Reviewers: **Divna Krpan, Ph.D.** *Assistant Professor of Faculty of Science, University of Split*

Saša Mladenović, Ph.D. *Associate Professor of Faculty of Science, University of Split*

Goran Zaharija, Ph.D. *Assistant Professor of Faculty of Science, University of Split*

Thesis accepted: September 2022.

Sadržaj

Uvod

Automatizacija je jedan od ključnih procesa u svakoj industriji. U posljednjih nekoliko godina informacijska industrija započinje proces automatizacije većeg dijela razvoja, testiranja i objavljivanja softwarea. U ovome radu se stavlja fokus na automatiziranju što većeg broja testova u svrhu oslobađanja vremena korisnika.

Objektno orijentirano programiranje je posljednjih godina postalo jedno od naj utjecajnih programskih paradigmi. Široko se koristi u obrazovanju i industriji, te gotovo svako sveučilište podučava objektnu orijentaciju barem u određenom dijelu svog kurikuluma. Veći dio šire IT zajednice se slaže da je podučavanje objektno orijentiranog programiranja korisno i ključno za budućnost. Elegantno podupire koncepte koji se pokušavaju podučavati dugi niz godina, poput dobro strukturiranog programiranja, modularizacije i oblikovanje programa. Također podržava tehnike za pristup problemima koji su nedavno su ušli u nastavni plan i program a to su programiranje u timovima, održavanje velikih sustava i ponovna uporaba softvera.

Alati za automatsko testiranje su značajno napredovali u svakom aspektu, uključujući brzinu, jednostavnost implementacije, načine testiranja i zadovoljstvu programera koji ga koriste. Nakon stabilizacije cijelog ukupnog područja testiranja u JavaScriptu proizašlo je nekoliko očitih pobjednika, kao i najbolje prakse i standardi unutar samog testiranja.

U ovome radu bit će opisane sve navedene mogućnosti alata za testiranje kao i njihova ugradnja u ostale napredne JavaScript biblioteke i okvire. Najveća pozornost je dana samom sustavu, njegovom načinu rada, mogućim načinima primjene i proširenja, kao i trenutnim manama i nedostatcima odabranih pristupa.

1. Programske paradigme

U znanosti i filozofiji paradigma je zaseban skup pojmova ili misaonih obrazaca, uključujući teorije, metode istraživanja, postulate i standarde za ono što predstavlja legitiman doprinos nekom području. (Waterlow, 1982)

Programske paradigme način su klasifikacije programskih jezika na temelju njihovih značajki. Jezici se mogu klasificirati u više paradigmi, te je rijetko da jezik pripada samo jednoj paradigmi.

Neke se paradigme uglavnom bave utjecajem na samo izvođenje programa, poput dopuštanja nuspojava ili je li slijed operacija definiran modelom izvođenja. Druge paradigme se uglavnom bave načinom na koji je kôd organiziran, kao što je grupiranje koda u dijelove zajedno njihovim modelom. Ostale se pak uglavnom bave stilom sintakse i gramatikom.

Glavne programske paradigme su:

- Imperativno programiranje
- Deklarativno programiranje

1.1. Imperativno programiranje

Prvi programski jezici dizajnirani su u skladu s fizičkom strukturom računala. Programski jezici koji se razvijaju s njima nazivaju se imperativnim jezicima i još uvijek su pod utjecajem arhitekture računala. Jezik imperativne paradigme odražava karakteristike Von Neumannova stroja koji pohranjuje podatke i programe u istoj memoriji. Ovi jezici su primjeri sastavljenih jezika i pružaju visoke performanse. Međutim, oni nude slabe apstrakcije i sigurnosne značajke. Prvi imperativni programski jezik je Fortran. Nadalje, Pascal i C odredili su programsku arhitekturu paradigme s funkcijama ili procedurama. (Bulent Tugrul, Gunduc, & Eryigit, 2017).

Pod imperativno programiranje spadaju sljedeće programske paradigme:

- Proceduralno programiranje
- Strukturirano programiranje

- Objektno orijentirano programiranje

1.1.1. Proceduralno programiranje

Proceduralno programiranje je programska paradigma koja je izvedena iz imperativnog programiranja, te se fokusira na konceptu pozivanja procedura koje jednostavno izvršavaju niz zadataka jedan za drugim

Vrh popularnosti proceduralnog programiranja je bio u ranim devedesetima s jezikom Pascal. Pascal je u tadašnje vrijeme bio primaran jezik za podučavanje programiranja, no već tada su se nazirali problemi s jezikom i proceduralnim programiranjem kao programskom paradigmom.

Pascalov manjak podatkovne apstrakcije te neuspjeh kao komercijalni jezik dovode do pritiska da se unutar edukacijskih sustava počne podučavati jezik koji omogućava lakše zapošljavanje na tržištu nakon studija.

Istraživanje provedeno na matematičkom odjelu fakulteta Virginia Commonwealth nastojalo je ustvrditi uzrok neuspjeha proceduralnog programiranja. Detaljan upitnik je poslan na 140 fakulteta od kojih je 45 odgovorilo. 66% ispitanika koji podučavaju proceduralnu paradigmu se žele prebaciti na drugu paradigmu, posebno ističu objektno orijentiranu paradigmu. Ispitanici ističu da je glavni razlog njihove želje za prijelazom činjenica da većina komercijalnog softwarea tada koristi objektno orijentiranu paradigmu. Glavni razlog istaknut kod ispitanika koji nastavljaju podučavaju proceduralnu paradigmu je pedagoška superiornost i povratna informacija od studenata koji tvrde da je proceduralna paradigma laka za naučiti za razliku od objektno orijentirane paradigme koja zahtijeva apstrakciju podataka. (Brilliant & Wiseman, 1996)

Veliki razlog istaknut za prebacivanje s proceduralnog programiranja je sami jezik Pascal gdje je 86% ispitanika istaknulo zelju za prebacivanjem na novi jezik. „Pascal je bio dobar jezik, no njegovo vrijeme je prošlo" (Brilliant & Wiseman, 1996)

1.1.2. Strukturirano programiranje

Strukturirano programiranje je podosta star koncept koji potječe iz samih početaka računalne znanosti, cilj strukturiranog programiranja je ukidanje GOTO naredbi te njihova zamjena s kontrolnim strukturama. Kontrolne strukture se sastoje od slijeda, selekcije ili grananja, rekurzije i iteracije ili ponavljanja. Svi programski jezici podržavaju

strukturirano programiranje. Strukturirano programiranje predstavlja same početke programskih paradigmi te samo po sebi nije dovoljno za moderne jezike. (BOHM & JACOPINI, 1966)

1.1.3. Objektno orijentirano programiranje

Objektno orijentirano programiranje je tehnika tj. paradigma za pisanje „dobrih“ programa za određeni skup problema. Pojam objektno orijentirani jezik znači jezik koji ima mehanizme koji dobro podržavaju objektno orijentirani stil programiranja. (Stroustrup, 1988)

1.1.3.1 Osnovni koncepti objektno orijentiranog programiranja

Termin objektno orijentirano programiranje može imati različita značenja, ali jednu stvar koju ovi jezici imaju zajedničko su objekti. Objekti su entiteti koji kombiniraju svojstva procedura i podataka budući da izvode izračune i spremanje lokalnog stanja. Jednoobrazna uporaba objekata je u suprotnosti s uporabom zasebnih procedura i podataka u konvencionalnom programiranju.

Sve radnje u objektno orijentiranom programiranju dolaze od slanja poruka između objekata. Slanje poruka je oblik neizravnog poziva procedure. Umjesto imenovanja procedure za izvođenje operacije na objektu, objektu se šalje poruka. Selektor u poruci određuje vrstu operacije. Objekti odgovaraju na poruke koristeći svoje vlastite procedure (zvane "metode") za obavljanje operacija. Slanje poruka podržava važno načelo u programiranju: apstrakcija podataka. Načelo je da pozivni programi ne bi trebali stvarati pretpostavke o implementaciji i internim prikazima tipova podataka koje koriste.

Njegova je svrha omogućiti promjenu temeljne implementacije bez mijenjanja poziva programa. Tip podataka implementiran je odabirom prikaza vrijednosti i pisanjem procedure za svaku operaciju. Jezik podržava apstrakciju podataka kada ima mehanizam za spajanje svih postupaka za pojedini tip podataka. U objektno orijentiranom programiranju klasa predstavlja tip podatka, vrijednosti varijable su njegov primjerak, a operacije su metode na koje klasa odgovara. (Bobrow, 1985)

1.1.3.2 Značajke objektno orijentiranog programiranja

Značajke koje opisuju objektno orijentirano programiranje su (Bobrow, 1985):

1. **Višestruka reprezentacija** je najosnovnija karakteristika objektno orijentiranog stila u tome što kada se operacija poziva na objektu, sami objekt određuje koji se dio koda izvršava. Dva objekta koji odgovaraju istom skupu operacija tj. potječu iz istoga podtipa (Interface) mogu koristiti skroz različite operacije sve dok svaka instanca sa sobom nosi operacije koje rade za tu pojedinu reprezentaciju. Ove implementacije se zovu „objektove“ metode.
2. **Enkapsulacija** je interna reprezentacija objekta koja je općenito sakrivena od vanjskog pogleda na objekt tj. samo objektove unutarnje metode mogu izravno pristupiti i manipulirati njegovim podatcima. To znači da promjene na unutarnjoj reprezentaciji objekta mogu utjecati na mali dio programa. Ovaj pristup uvelike poboljšava čitljivost i lakoću snalaženja unutar koda, te poboljšava održivost programa. Apstraktni tipovi podataka pružaju sličan oblik enkapsulacije što osigurava da je konkretna reprezentacija njihovih vrijednosti vidljiva unutar samo jednoga modula te također osigurava kodu van objekta samo jedan način indirektne manipulacije podatcima kroz objektove metode.
3. **Podtipovi.** Tip objekta ili njegov „Interface“ je skup imena i oblika njegovih operacija. Objektova unutarnja reprezentacija se ne prikazuje unutar svog tipa, pošto ono ne utječe na skup stvari koje možemo izravno učiniti s objektom. Objektovi podtipovi prirodno zadovoljavaju njegove podtipne relacije. Ako objekt zadovoljava podtip A, onda također zadovoljava podtip B koji ima manje operacija od A, pošto u kontekstu koji očekuje da tip B objekt može pozvati samo operacije tipa B na sebi, prosljeđivanje objekta tipa A bi trebalo uvijek biti sigurno. Mogućnost ignoriranja dijelova objektova podtipa nam omogućuje da pišemo jedan dio koda koji može manipulirati objektima na različite načine u strukturiranom načinu.
4. **Nasljeđivanje.** Objekti koji dijele dijelove podtipa će često dijeliti i dijelove nekih ponašanja, te je cilj implementiranja nasljeđivanja iskorištavanje tih ponašanja vise puta. Većina objektno orijentiranih jezika postiže ovo upravo kroz upotrebu struktura zvanih klase. Klase su predlošci iz kojih izrađujemo objekte. Proces podklasiranja nam omogućava da izvedemo nove klase iz starih klasa, te

dodavanjem novih metoda i svojstava „nadjačamo“ te iste metode i svojstva iz starih klasa

5. **Otvorena rekurzija.** Još jedan koristan aspekt koji pruža većina objektno orijentiranih jezika je mogućnost da jedna metoda pozove drugu metodu unutar istog objekta koristeći posebnu varijablu zvanu „Self“ ili u nekim jezicima „This“. Ovo ponašanje omogućava pozivanje metoda iz različitih podklasa.

1.1.3.3 Ključni termini i koncepti objektno orijentiranog programiranja

Za uspješno korištenje objektno orijentiranog programiranja potrebno je znati ključne termine i koncepte.

1) **Klasa** - Klasa je opis jednog ili više sličnih objekata. Na primjer, klasa Jabuka je opis strukture i ponašanja jedinica, kao što su jabuka-1 i jabuka-2. Klase opisuju varijable instance, varijable klase i metode njihovih instanci kao i položaj klase u stablu nasljeđivanja.
2) **Klasno nasljeđivanje** - Kad se klasa smjesti u klasno stablo ona nasljeđuje varijable i metode iz svojih super klasa ili nadklasa. To znači da će se svaka varijabla koja je definirana više u rešetki klase također pojaviti u instancama ove klase. Ako je varijabla definirana na više od jednog mjesta, nadjačavajuču vrijednost određuje tok nasljeđivanja.
3) **Klasna varijabla** – Klasna varijabla je tip varijable spremljen unutar klase čija je vrijednost dijeljena sa svim jedinicama te klase. Razlikuje se od jedinice klase.
4) **Apstrakcija podataka** – Princip da programi ne bi smjeli donositi zaključke o implementaciji i unutarnjoj reprezentaciji podataka. Tip podataka je karakteriziran operacijama koje se izvršavaju nad njim. U objektno orijentiranom programiranju operacije su metode klase, a klasa predstavlja tip podataka i sve vrijednosti njihovih jedinica.
5) **Instanca (jedinica)** – Termin „instanca“ ili jedinica klase se koristi na dva načina. Termin „instanca“ opisuje odnos između objekta i njegove klase. Metode i strukture pojedine instance određuje klasa koju ta instanca predstavlja. Pojam „instanciranja“ se odnosi na stvaranje novog objekta prema nacrtu klase.
6) **Objekt** – Objekt je primitivni element u objektno orijentiranom programiranju, oni kombiniraju atribute, procedure i podatke. Objekti spremaju vrijednosti unutar

varijabli te odgovaraju na pojedinačne poruke tako što pozovu odgovarajuću metodu.

7) **Polimorfizam** – Polimorfizam je objektno orijentirani koncept programiranja koji se odnosi na sposobnost varijable, funkcije ili objekta da poprimi višestruke oblike. Jezik koji ima polimorfizam omogućava programerima da programiraju u općenitom, a ne u specifičnom..
8) **Podklasa** – Podklasa je klasa koja je niže na stablu nasljeđivanja od zadane klase
9) **Nadklasa** – Klasa koje je više na ljestvici nasljeđivanja od zadane klase

1.1.4. Podučavanje objektno orijentiranog programiranja

Objektno orijentirano programiranje je posljednjih godina postalo najutjecajnija programska paradigma. Široko se koristi u obrazovanju i industriji te gotovo svako sveučilište negdje u svom kurikulumu sadrži objektno orijentiranu paradigmu. Programerska zajednica se više-manje se slaže da podučavanje objektno orijentiranog programiranja je dobra stvar.

Međutim, podučavanje objektno orijentiranog programiranja ostaje teško. Mnoga izvješća o iskustvima onih koji pokušavaju poučavati objektno orijentirano programiranje uključuje dugačak popis problema s mnogo različitih zaključaka. Zašto je teško? Ili, točnije, zašto je podučavanje objektno orijentiranog programiranja teže od podučavanja strukturiranog programiranja?

Korišteni programski jezici su previše složeni kao i programska okruženja, te mogu biti previše zbunjujuća za studente. Neki sustavi koji se koriste za podučavanje su razvijeni za profesionalne softverske inženjere, te se studenti prve godine jako teško snalaze u njima. Dok drugi uopće nisu bili "razvijeni" nego su nastali iz povijesnih slučajnosti. Što dovodi do pitanja koji programski jezici su pogodni za podučavanje objektno orijentiranog programiranja? (Kölling, 1999)

1.1.4.1 Odabir programskog jezika za podučavanje objektno orijentiranog programiranja

Često se u literaturi mogu naći rasprave o zahtjevima koje bi objektno orijentirani jezici trebali ispunjavati. Izneseni su mnogi argumenti za i protiv određenih jezičnih konstrukata, te svaka moguća značajka ima svoje argumente za i protiv. Na primjer, pitanje je li višestruko nasljeđivanje potrebno, izazvalo je brojne rasprave, često vođene s vjerskim

žarom. Takve opće rasprave često su besmislene i neproduktivne. Jezici nisu dobri ili loši sami po sebi. Dobri su ili loši za određenu svrhu. Jezik koji je loš u jednom kontekstu može biti izvrstan u drugom (ili obrnuto). To je više pitanje pravog alata za pravi posao. Bespredmetno je raspravljati je li čekić bolji nego odvijač ako se ne svađate u konkretnom kontekstu onoga što želite čini. (Kölling, 1999)

Sljedeći kriteriji bi trebali biti ispunjeni kako bi jezik bio koristan za podučavanje objektno orijentiranog programiranja:

1. **Čisti i jednostavni koncepti –** Koncepti koje želimo podučavati moraju biti predstavljeni u jeziku na jednostavan, čist i konzistentan način. Također bi bilo poželjno da su ti koncepti reprezentirani na isti način na koji se govori o njima za vrijeme podučavanja objektno orijentiranog programiranja.
2. **Čista objektno-orijentiranost -** Termin čista objektna orijentiranost je izabrana na način da znači suprotno od hibridnih jezika koji također podržavaju objektno orijentirano programiranje. Opasnost korištenja hibridnih jezika je u činjenici da učenici s prethodnim iskustvom u proceduralnom programiranju nemaju nikakav poticaj kako bi promijenili svoj stil programiranja. Često se događa upravo suprotno, učenici jednostavno nastavljaju pisati u proceduralnom stilu, ignorirajući sve kljune aspekte objektno orijentiranog programiranja iako misle da pišu u stilu objektno orijentiranog programiranja.
3. **Sigurnost (jednostavnost pronalaženja grešaka) –** Princip sigurnosti se odnosi na greške koje kompajler ili sustav lako otkrivaju. Također te greške se trebaju pronaći rano i prikazati jasno uz poruku koja ukazuje na lokaciju i uzrok greške. Konstrukti koji su poznati po uzrokovanju problema bi se trebali izbjegavati gdje god je to moguće.
4. **Jezik više razine –** Programer se ne bi trebao zamarati sa trenutnim stanjem računala na kojem se njegov kod izvršava. Zadatci koje kompajler može jednostavno izvršiti nisu odgovornost programera.
5. **Jednostavan model izvršavanja –** Model izvršavanja treba biti jednostavan i lagan za razumjeti. Ovo uključuje modele objekata i modele memorije.
6. **Čitljiva sintaksa –** Sintaksa jezika treba biti jednostavna i lako čitljiva. Postoji mnogo razloga zašto bi sintaksa trebala biti jednostavna. Kao prvo, čitljiviji programi su vjerojatno i točniji, pretpostavka je da ćemo napisati točniji kod ako možemo lako razumjeti što naš kod sadrži i gdje se metode izvršavaju. Ključni

aspekt je preferiranje riječi umjesto simbola. U većini slučajeva riječi su mnogo intuitivnije korisnicima i početnicima nego simboli.

7. **Bez redundantnosti –** Jezik bez redundantnosti znači da sve što želimo izvršiti u pojedinom jeziku mora imati jedan i samo jedan način za to postići.
8. **Maleni jezik -** Jezik bi trebao biti što je moguće manji, a da uključuje sve važne značajke o kojima se raspravlja za vrijeme podučavanja.
9. **Omogućava jednostavan prijelaz u druge jezike –** Jedan od ključnih aspekata bilo kojega jezika korištenog za podučavanje je prenošenje naučenog znanja u druge oblike i jezike. Studenti nakon završavanja studija moraju znati barem jedan jezik koji je široko korišten u industriji, ali to ne znači da oni moraju započeti učiti programiranje sa tim jezikom.
10. **Posjedovanje odgovarajućeg okruženja –** Odabrani jezik mora posjedovati dobro integrirano radno okruženje koje podržava razvoj aplikacija u odabranom jeziku.

Iako neki od zahtijeva poništavaju druge zahtjeve, kako bi smo odabrali pravi jezik za podučavanje potrebno je naći zlatnu sredinu između odabranih zahtijeva.

1.1.4.2 Odabir programskog okruženja za podučavanje objektno orijentiranog programiranja

Kako je rasla popularnost objektno orijentiranih jezika, razvoji u okuženjima za istraživanje i programskoj industriji su doveli do pojavljivanja mnoštva različitih programskih okruženja koja su pogodna za objektno orijentirano programiranje. U ispitivanju zahtjeva i prikladnosti okruženja javljaju se dva ključna aspekta koji su od posebnog značaja: podrška za objektnu orijentiranost i prikladnost za nastavu. Zahtjeve za integrirana okruženja možemo podijeliti u:

1. **Jednostavnost korištenja -** Jedan od najvažnijih čimbenika u odlučivanju o prikladnosti razvojnog okruženja za nastavu je jednostavnost korištenja. Okruženje mora biti dovoljno jednostavno kako bi ga početnici u programiranju mogli nakon vrlo kratkog vremena početi koristiti za rješavanje zadataka. To praktički implicira da okruženje mora imati grafičko korisničko sučelje. Lakoća korištenja također znači skrivanje nepotrebnih detalja. Aspekti operacijskog sustava, na primjer, kao što su pojedinosti o upravljanju datotekama, trebaju u potpunosti biti automatski, dopuštajući korisniku da radi na višoj razini apstrakcija. Na primjer, kada korisnik

kreira nove klase, njega ne zanima broj datoteka i u kojim se poddirektorijima nalaze informacije o tim klasama.

2. **Integrirani alati –** Zahtjevi integriranih alata su izravna posljedica zahtjeva zajednostavnosti korištenja. Kako bi okruženje bilo pogodno za poučavanje objektno orijentiranog programiranja poželjno je da ima sljedeće integrirane stavke:
    a. Konzistentno sučelje – Sučelje svih komponenti integriranog okruženja mora izgledati konzistentno i prikladno, te korištenje dodatnih komponenti sučelja mora biti jednostavno za naučiti.
    b. Maleno sučelje – Prenatrpanost različitim elementima može uzrokovati preopterećenje kod korisnika koji prvi put koriste alat.
    c. Povećanje produktivnosti – Dobra integracija često korištenih elemenata i prečaca može uvelike povećati produktivnost programera i brzinu učenja kod početnika. Cilj je ukinuti nepotrebne korake iz postupka izrade aplikacije.
3. **Podržanost objekata -** Mnoga postojeća okruženja za razvoj programa i aplikacija razvila su se tijekom vremena. Većina je izvorno razvijena za neobjektno orijentirane jezike i kasnije su prilagođeni za objektno orijentirane jezike. Ova prilagodba, međutim, obično ne uspijeva iskoristiti mogućnosti koji dolaze s objektnom orijentacijom – oni su po svom karakteru strukturirani te nisu objektno orijentirana okruženja. Iako postoje brojna okruženja za objektno orijentirane jezike, malo njih je objektno orijentirano okruženje.
4. **Podrška za recikliranje koda -** Olakšavanje ponovne uporabe postojećeg koda jedna je od glavnih motivacija za objektno orijentirano programiranje. U nastavi, moramo težiti da našim učenicima pružimo stvaran dojam rješavanja nekih od zadataka koje će susresti u stvarnom svijetu. Moramo pokušati uključiti stvarna iskustva programiranja u studentske vježbe što je više moguće, kako bi studenti mogli razumjeti probleme razvoja softvera te stekli dobre navike.
5. **Podrška za učenje -** Okruženje mora podržaviti neke od tehnika koje dokatano pomažu učenju koncepata programiranja. Nekih od tih tehnika su:
    a. Interakcija / eksperimentiranje - Pružanje praktičnog pristupa omogućavanjem interakcije s klasama i objektima može uvelike povećati razumijevanje koncepata objektno orijentiranog programiranja. Testiranje ideje da se, na primjer, mnogi objekti mogu stvoriti iz jedne klase i da se ti

objekti ponašaju slično, ali imaju zaseban identitet i različito stanje, može uvelike pomoći u razjašnjavanju odnosa između klasa i objekata.

b. Vizualizacija – Strukture o kojima govorimo za vrijeme poučavanja moraju biti jasno vidljive na zaslonu. Često iskustvo predavača objektno orijentiranog programiranja je da učenici u početku imaju poteškoća kod razmišljanja o klasama. Jedan od mogućih razloga je što učenici jedino na ekranu vide linije programskog koda , a prenošenje koda u vizualni oblik smatra se korakom kojeg mnogi početnici ne mogu jednostavno ostvariti.

6. **Grupna podrška –** Jedna od glavnih karakteristika programiranja u stvarnom svijetu je rad u grupama ili timovima. Jedan od ključnih razloga za uspjeh objektno orijentiranog programiranja je olakšavanje rada u timu.
7. **Dostupnost –** Jedan od temeljnih prepreka u korištenju integriranih alata je njihova cijena. Većina alata je dizajnirana za korištenje u velikim programerskim kompanijama kojima je cijena korištenja integriranog ograničenja poprilično beznačajna. Nažalost takve cijene su daleko izvan dostupnosti većini informatičkih odjeljaka fakulteta.

Nijedan od gore navedenih zahtjeva nije stvarno nov. Svaki od njih je već implementiran u neki postojeći sustav. Međutim, kombinacija ovih zahtjeva, je ono što je zaista važno u kontekstu podučavanja. Posebno kombinacija zahtjeva za jednostavnošću upotrebe i sofisticiranom tehničkom podrškom na prvi pogled može izgledati kontradiktorno. Tražimo snažan sustav koji korisniku izgleda jednostavno. Stupanj do kojeg je ova kombinacija postignuta bit će odlučujući faktor u procjeni prikladnosti sustava za podučavanje objektno orijentiranog programiranja. (Michael, 1999)

1.2. Deklarativno programiranje

Većina deklarativnih programskih jezika potječe iz rada na umjetnoj inteligenciji i automatizaciji potvrđivanja različitih tvrdnji. Ta područja zahtijevaju više razine apstrakcije i čisti semantički model. Osnovno svojstvo deklarativnih programskih jezika je to da je program teorija s nekom prikladnom logikom. Ovo svojstvo odmah daje precizno značenje programima napisanim u tim jezicima. S perspektive programera osnovno

svojstvo je podizanje apstrakcije programa. Na ovom višem nivou apstrakcije programer se može fokusirati na ono što želi da bude izvršeno a ne naočito na način kako to izvršiti. Programer žrtvuje određenu količinu kontrole nad programom ali ne i samu logiku programa. (Torgersson, 1996)

Ključne točke deklarativne paradigme:

- Ključna ideja deklarativnog programiranja je da je program zapravo teorija a implementacija je dedukcija te teorije
- Deklarativno programiranje treba uključivati logičko programiranje i funkcionalno programiranje, te se treba poprilično preklapati s ostalim područjima znanosti
- Deklarativno „debugiranje“ u svrhu uklanjanja pogrešaka je ključna tehnika potrebna za deklarativno programiranje
- Jednostavna semantika je ključna za primjenu mnogih tehnika, kao programska analiza, programska optimizacija, programska sinteza, verifikacija i sistemska konstrukcija
- Deklarativno programiranje ima ogroman doprinos u poboljšanju programerske produktivnosti, te čini samo programiranje dostupnije većem broju ljudi
- Deklarativni pristup meta-programiranja pomaže u ostvarivanju kompleksnih programskih alata kao kompajler-generatori.
- Sto više možemo učiniti jezik deklarativnim to će implicitni paralelizam koji možemo ostvariti biti veći
- Sto više možemo učiniti jezik deklarativnim to će paralelna implementacija biti lakša
- Polja funkcionalnog i logičkog programiranja treba spojiti

(Lloyd, 1994)

1.2.1. Funkcionalno programiranje

Osnovni princip funkcionalnog programiranja je realizacija programa koristeći funkcije. Riječ „funkcija“ ovdje ima matematičke korijene u smislu mapiranja ulaznih vrijednosti s izlaznim vrijednostima, gdje u programerskom smislu jezici pozivaju „funkcije“ kao pod rutine koje vraćaju vrijednost. Jedna važna razlika je ta što je funkcija u matematičkom smislu uvijek proizvodi isti izlaz kada je dan isti ulaz. Operacija poput

dobivanja sljedećeg retka iz datoteke nije funkcija jer svaki put kada se pozove ona proizvodi različitu povratnu vrijednost.

Druga važna razlika je u tome što matematička funkcija ne "radi" ništa osim vraćanja vrijednosti. Ona ne bi trebala imati nuspojave - na primjer, ne bi trebala pisati u datoteku ili mijenjati varijablu u memoriji.

Ako se program sastoji od funkcija, a funkcije ne bi trebale mijenjati nikakve varijable, za što se onda koriste varijable? Tehnički za ništa, i zato pravilno implementirano funkcionalno programiranje nema varijable. Drugi nedostajući temeljni koncept su petlje. Koja je poanta petlje ako se ništa ne može promijeniti između ponavljanja jer nema varijabli? Petlje se u funkcionalnom programiranu zamjenjuju s rekurzijom.

Funkcionalne programske jezike možemo kategorizirati prema njihovom stavu prema neizbježnim nuspojavama. Čisti jezici (poput Haskella) to dopuštaju samo unutar posebnih jezičnih konstrukcija. Nečisti jezici (velika većina programskih jezika) u potpunosti ostavljaju odgovornost za korištenje jezika i nuspojava programeru. Kao što je to često slučaj, oba pristupa imaju svoje dobre i loše strane. (Hinsen, 2009)

2. Okviri za testiranje

Pri odabiru okvira za izradu projekta ključnu ulogu također igraju alati za testiranje. Alat Nextjs koristi JavaScript jezik kao svoju podlogu te nas to ograničava na alate za testiranje u JavaScriptu. To je do nedavno bio problem jer su alati za testiranje na webu bili teški za implementaciju, spori te nisu imali opširnu dokumentaciju kao današnji okviri. U zadnjih nekoliko godina okviri za testiranje na webu su se znatno stabilizirali što je potaklo testiranje na webu kao jedan od integralnih dijelova izrade web projekata. O zadovoljstvu sa novim okvirima za testiranje nam govori anketa „The state of JavaScript“ koja prikazuje da 96% korisnika Jest okvira je jako zadovoljno. (stateofjs, 2021)

2.1. Odabir alata

U svijetu JavaScripta izbora ima previše. Svima je poznato da novi JavaScript alat izlazi svaki dan te obećava biti bolji nego svi prijašnji. U ovome radu će se obraditi samo najpopularniji i najstabilniji okviri za testiranje.

Svaki okvir ima svoje ugrađene mogućnosti koje se mogu nadodati sa raznim proširenjima.

	Jest	Mocha	Jasmine	AVA	Tape
Assertions	✔	✘	✔	✔	✔
Spies	✔	✘	✔	✘	✘
Mocks	✔	✘	✔	✔	✘
Stubs	✔	✘	✔	✘	✘
Code Coverage	✔	✘	✘	✘	✘
Snapshot testing	✔	✘	✘	✘	✘
Globals	✔	✔	✔	✘	✘
Ready-To-Go	✔	✘	✔	✔	✔
Extensibility	✔	✔	✔	✔	✔

Slika 1 Značajke popularnih alata (Sattar, 2019)

2.1.1. Jest

Jest je okvir za testiranje koji je razvio Facebook. Nagli porast u popularnosti je doživio u 2017 te od tada drži prvo mjesto. U početku je bio temeljen na okviru Jasmine te je s vremenom Facebook implementirao svoje funkcionalnosti te dodao puno značajki povrh toga. Ključne značajke Jesta su:

- **Performanse** Jest se smatra bržim za velike projekte s mnogo testnih datoteka zbog svog mehanizma pametnog paralelnog testiranja.
- **Korisničko sučelje** – Jednostavno i korisno, sve što je potrebno
- **Odmah spremno** – Kao što je vidljivo iz gornje slike okvir jest dolazi sa svim značajkama koje su potrebne za većinu projekata. Naravno Jest je moguće vrlo lako proširiti sa dodatnim značajkama.
- **Globalni podaci** – Kao u Jasmineu, Jest stvara globalne podatke te nema potrebe za dodatnim potraživanjem. Ovaj postupak se može smatrati lošim jer ograničava fleksibilnost i ograničava kontrolu nad testovima ali u isto vrijeme značajno olakšava živote običnih programera.

2.1.2. Mocha

Mocha je najkorišteniji alat za testiranje. Za razliku od Jesta i Jasmine, Mocha sadrži samo okruženje za pokretanje testova te sve ostale značajke se moraju ugraditi koristeći dodatne ekstenzije. Najčešće ekstenzije su Sinon i Chai. Mocha je malo teža za postavljanje u početku, ali zato je nenadmašena u svojoj fleksibilnosti. Ključne značajke Moche su:

- Zajednica – Mocha ima mnogo ekstenzija i proširenja za testiranje jedinstvenih scenarija kao i vrlo učestalih scenarija
- Proširivost – Mocha je lako proširiva do te mjere da su proširenja i ekstenzije posebno dizajnirane kako bi radile samo na Mocha okviru.
- Globali – Kreiranje globalne strukture koje ne ograničava fleksibilnost, no nažalost pruža dosta manje funkcionalnosti od alata Jest

2.1.2.1 Chai

Chai je BDD („Behavioral-Driven Development“) / TDD („Test-Driven Development“) okvir koji sluzi za potvrđivanje vrijednosti pojedinih varijabli. Glavna primjena je u okruženjima za Node i Web preglednike, te se može koristiti sa svakim JavaScript okvirom za testiranje. Chai ima nekoliko sučelja koji omogućuju korisniku da odabere onaj s kojim se osjeća najudobnije.

Should

```
chai.should();

foo.should.be.a('string');
foo.should.equal('bar');
foo.should.have.lengthOf(3);
tea.should.have.property('flavors')
  .with.lengthOf(3);
```

Expect

```
var expect = chai.expect;

expect(foo).to.be.a('string');
expect(foo).to.equal('bar');
expect(foo).to.have.lengthOf(3);
expect(tea).to.have.property('flavors')
  .with.lengthOf(3);
```

Assert

```
var assert = chai.assert;

assert.typeOf(foo, 'string');
assert.equal(foo, 'bar');
assert.lengthOf(foo, 3)
assert.property(tea, 'flavors');
assert.lengthOf(tea.flavors, 3);
```

Slika 2 Različiti tipovi sintakse

Iako bi se moglo puno reći o tome koji je od ovih stilova poželjniji, trenutno vlada „Expect“ sintaksa. „Expect“ je funkcija koja uzima jedan argument, vrijednost koja se testira ili nadređenu vrijednost koja se testira, ovisno o testu.

Chai.js uključuje kozmetička svojstva koja nemaju utjecaja na ponašanje, ali umjesto toga dodaje prirodni jezik kako bi bio jasniji ljudima koji čitaju kod.

```
expect('foo').to.be.undefined; // ne prolazi
```

Figura 1 Kozmetička svojstva „to“, „be“

Proširujući prethodni primjer, dodavanje „to“ i „be“ ne mijenja ponašanje. Dodavanje kozmetičkih svojstava „to“ i „be“ jasno pokazuje što se očekuje od rijeci "foo".
Chai trenutno ima 15 kozmetičkih svojstava: to, be, been, is, that, which, and, has, have, with, at, of, same, but, does.
Chai također uključuje svojstva markiranja s takozvanim „zastavicama“ (eng. flags). Zastavica ne predstavlja tvrdnju. Umjesto toga ona postavlja zastavicu na lanac očekivanja koju druge tvrdnje u lancu mogu pročitati. Postojanje zastavice samo po sebi ne mijenja ništa. Na pojedinačnim tvrdnjama je da odluče kako će protumačiti zastavicu. Takva tumačenja mogu uključivati negacije ili više ili manje stroge tvrdnje.

```
expect('foo').to.not.be.undefined; // prolazi
```

Figura 2 Zastavica "not"

Chai trenutno ima 7 zastavica: not, deep, nested, own, ordered, any, all

2.1.3. Jasmine

Jasmine je testni okvir na kojem je zasnovan Jest. Jasmine ima ogromnu prednost u tome što je na tržištu već dugo vremena te možemo pronaći mnoštvo članaka, alata i odgovorenih pitanja na raznim forumima koje su kreirale zajednice korisnika okvira. Angular tim službeno koristi Jasmine kao svoj okvir za testiranje. Ključne značajke Jasmine okvira su:

- Odmah spremno - Jasmine dolazi sa svime što je potrebno za početak testiranja na webu
- Zajednica – Jasmine je na tržištu još od 2009 te je zajednica korisnika stvorila ogromne količine sadržaja. Nažalost u JavaScript svijetu stvari postanu beskorisne jako brze te većina starog sadržaja je u potpunosti neiskoristiva
- Angular – Podržava sve verzije alata Angular, te je službeno preporučen kao okvir za testiranje Angular aplikacija.

2.2. Odabir tipa testa

Načini testiranja se razlikuju od jezika do jezika i od okvira do okvira, no tipovi testova su isti za sva okruženja. Kako je projekt baziran na testiranju studentovog koda potrebno je odlučiti kakav tip testa želimo izvršiti na njegovom kodu.

Imamo četiri tipa testa:

- **Jedinični testovi** su testovi na vrlo niskoj razine i blizu samoj aplikaciji. Sastoje se od testiranja pojedinačnih metoda i funkcija klasa, komponenti ili modula koje koristi aplikacija. Jedinični testovi općenito su prilično jeftini za automatizaciju i mogu se vrlo brzo izvoditi pomoću kontinuiranog integracijskog poslužitelja.
- **Integracijski test** provjeravaju rade li različiti moduli ili usluge koje koristi aplikacija zajedno. Na primjer, to može biti testiranje interakcije s bazom podataka ili osiguravanje da mikro servisi rade zajedno kako se očekuje. Ove vrste testova su skuplje za izvođenje jer zahtijevaju da više dijelova aplikacije bude pokrenuto.
- **Funkcionalni testovi** – Funkcionalni testovi usmjereni su na poslovne zahtjeve aplikacije. Oni samo provjeravaju izlaz akcije i ne provjeravaju među stanja sustava prilikom izvođenja te radnje. Ponekad postoji zabuna između integracijskih testova i funkcionalnih testova jer oba zahtijevaju više komponenti za međusobnu interakciju. Razlika je u tome što integracijski test može jednostavno potvrditi da možete postaviti upit bazi podataka, dok bi funkcionalni test očekivao dobivanje određene vrijednosti iz baze podataka kako je definirano zahtjevima proizvoda.
- **„End-to-end“** testiranje replicira ponašanje korisnika sa softverom u kompletnom aplikacijskom okruženju. Provjerava rade li razni korisnički tokovi prema očekivanjima i mogu biti jednostavni poput učitavanja web stranice ili mnogo složeniji scenariji provjere obavijesti e-poštom, plaćanja na mreži itd. „End-to-end“ testovi su vrlo korisni, ali su skupi za izvođenje i može ih biti teško održavati kada su automatizirani. Preporuča se imati nekoliko ključnih end-to-end testova i više se oslanjati na vrste testiranja niže razine (jedinični i integracijski testovi).

Sam primjer koda kojeg će se testirati nam otkriva koji test ćemo koristiti.

```
class Postavke extends EventEmitter {
  constructor() {
    if (this instanceof Postavke) {
      throw "Statička klasa nema instance!";
    }
  }

  static coins = [];
  static spikes = [];

  static krajIgre = false;

  /** @type {Dinosaur} */
  static dinosaur = null;

  static cilj = 40; //4 novčića po 10

  static reset() {
    // važno! potrebno je resetirati postavke
    this.coins = [];
    this.spikes = [];
    this.dinosaur = null;
  }
}
```

Figura 3 Primjer koda za testiranje

Kao što vidimo kod je poprilično jednostavan te sadrži samo jednu klasu sa nekoliko svojstava. Za ovaj slučaj je najbolje koristiti jedinične testove kako bi smo mogli provjeriti zahtijevane elemente neovisno o sintaksnoj točnosti koda.

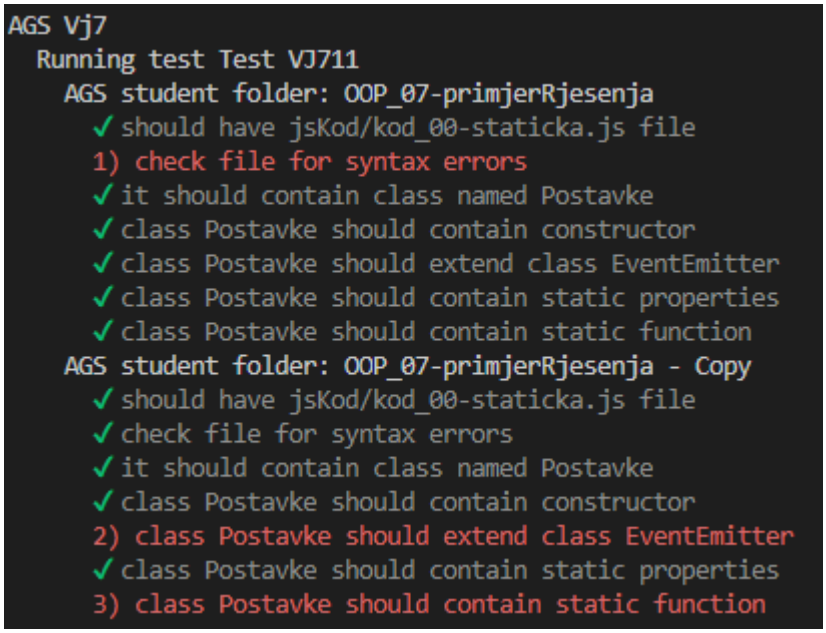
Iako je sam kod poprilično jednostavan, moguće je uočiti ograničenja JavaScripta kao objektno orijentiranog jezika. Ta ograničenja ne utječu samo na pokretanje i izvršavanje koda nego također utječu na pisanje i izvršavanje testova. Jedan od glavnih primjera je kreiranje konstruktora za statičku klasu. JavaScript je jako slobodan jezik, te protivno svim pravilima objektno orijentiranog programiranja on će dopustiti kreiranje objekta iz naše statičke klase te će za vrijeme stvaranja novog objekta doci do greške. Pravilan pristup bi bio onemogućenje stvaranja objekta iz statičke klase na prvom mjestu. Sa jasno definiranim pravilima i rezultatom testa, stvaranje objekta iz statičke klase je odličan primjer za jedinični test. Struktura jediničnog testa za utvrđivanje mogućnosti stvaranja objekta iz pojedine klase bi glasio:

- Učitaj klasu Postavke
- Pokušaj stvoriti objekt iz klase Postavke
- Očekuj pogrešku za vrijeme stvaranja objekta

Jako jednostavno, što je i najveća prednost jediničnih testova.
Također ostali zahtijevani testovi na ovom kodu mogu biti:

- Provjera sintakse
- Provjera postojanja konstruktora
- Provjera nasljeđivanja
- Zahtijevana svojstva
- Provjera postojanja funkcije
- Provjera nadopunjene funkcije

Svaki od ovih dijelova se može neovisno provjeriti te aplikacija vraća rezultat o točnosti svakog pojedinačnog dijela.



Slika 3 Zeljeni prikaz rezultata testa na ucenickom kodu

3. Inteligentni pomoćnici

Izrada AGS sustava je u velikom dijelu potpomognuta novim tehnologijama u umjetnoj inteligenciji. Istaknuti ću dva takva sustava koji stoje na raspolaganju svim programerima, te uvelike mogu promijeniti način izrade projekta

3.1. DALL-E 2

DALL-E 2 je novi system umjetne inteligencije baziran na kreiranju realističnih slika i umjetnosti iz zadanog teksta. DALL-E 2 kombinira koncepte, atribute i stilove različitih slika kako bi kreirao unikatna umjetničkih dijela. O uspjehu alata za kreiranje umjetnosti nam najviše govori nedavni uspjeh Midjourney sustava koji je osvojio nagrade na natjecanju u Coloradu. Iako DALL-E 2 nije osobito dobar u kreiranju teksta odabran je za izradu logotipa AGS sustava. Koristeći kratak opis projekta DALL-E 2 je uspješno generirao osam logo tipova za AGS sustav od kojih je odabran jedan.

Slika 4 Odabrani logo

3.2. Github Copilot

Github Copilot je jedan od najvećih iskoraka u programiranju ikada. Github Copilot je programerski pomoćnik koju pruža niz mogućnosti kao automatsko dopunjavanje koda, generiranje čitavih funkcija i transformiranje komentara u kod. Važno je naglasiti da trenutno Copilot ne razmišlja kao programer i ne programira kao programer. Njegova posebnost dolazi iz ogromnog seta podataka koji se sastoji od većine koda ikada napisanog

na Githubu. Sa tolikom bazom podataka Copilot je sposoban predlagati rješenja u svim programerskim jezicima i okvirima. Sa konstantnim korištenjem Copilot sve vise i vise razumije kod koji korisnik koristi te predlaže sve točnije i točnije rezultate. Prva verzija Copilota nije bila sposobna pogledati vise od trenutne datoteke te dati generalni prijedlog, dok trenutna verzija je sposobna pregledati čitav kod, te dati prijedloge u istom stilu kao i dosada napisani kod. Osim predlaganja koda Copilot također razumije kontekst podataka te uspješno razumije tip podatka te kakvu obradu je potrebno izvršiti kako bi se dobio željeni rezultat. Kao što je očito Copilot je jedan od najboljih alata za pisanje testove, zbog potrebe da se svaki test opise detaljno prije pisanja Copilot u potpunosti razumije ulazne podatke te rezultat koji treba postići. U ovome radu 100% testova je automatski predloženo od stanje Copilota. Čvrsto preporučam ovaj alat svima koji čitaju ovaj rad, te se nadam da će Microsoft nastaviti sa uspješnim razvijanjem ovoga alata.

4. Nextjs

Nextjs je JavaScript okvir inspiriran PHP-om te koristi JavaScript module, koji nam omogućavaju izvoz komponenti unutar aplikacije, što također omogućuje izvođenje pojedinačnih testova za svaku komponentu, kao i preuzimanje tisuća komponenti ili modula s NPM-a

Kada govorimo o aplikacijama u Next.js, moramo govoriti o CSS sustavu koji se zove „styled-jsx“ ili stilizirani JavaScript, ovaj sustav je posebno kreiran za rad s Next.js, te nam omogućuje da radimo sa svom snagom CSS-a izravno u JS datoteci.

Styled-jsx nam daje određene prednosti, na primjer, kada predstavljamo komponente samo generiramo CSS koji se koristi i, kada se komponenta više ne koristi, automatski uklanja CSS, što znači da nikada nećemo imati nepotreban CSS. Ovaj pristup smanjuje veličinu aplikacije te pruža brze vrijeme odaziva.

Fleksibilnost koju Nextjs nudi programerima i dizajnerima nema premca. Njegove značajke su brzina, responzivnost i prilagođenost SEO optimizira web stranicu za bilo koji uređaj ili razlučivost zaslona. Njegova analitika olakšava praćenje stvarnog učinka korisnika i automatski optimizira slike. Nextjs omogućava izradu potpuno prilagođenih stranica s jednostavnošću.

Next.JS je izvrstan izbor za projekte web razvoja, ali bitno je razumjeti osnove Reacta prije njegove upotrebe. Next.JS koristi usmjerivač koji se temelji na datotekama i podržava određeno dinamičko usmjeravanje. Osim ovih značajki, Nextjs također podržava TypeScript. Nextjs automatski konfigurira i kompilira TypeScript datoteke, osiguravajući sve prednosti TypeScripta bez problema sa postavljanjem i prevođenjem nazad u JavaScript.

Budući da je Nextjs okvir za izradu React aplikacija koje poslužujemo sa servera ključno je poznavanje Reacta

4.1. React

React.js je JavaScript okvir koji je brz, siguran i skalabilan. Pruža fantastično iskustvo za korisnike i programere. Štoviše, njegova popularnost raste jer ga podržavaju Facebook i šira zajednica programera. React.js je dominantna JavaScript tehnologija za izradu korisničkih sučelja i postaje sve popularnija.

React je alat za izgradnju komponenti korisničkog sučelja i cijelih korisničkih sučelja – sve što se tiče sastavljanja vizualnih elemenata, povezivanja podataka s tim elementima i određivanja logike koja njime upravlja.

React.js se može koristiti za stvaranje korisničkih sučelja u JavaScriptu za različite platforme. ReactDOM se koristi za web aplikacije, React Native za razvoj mobilnih aplikacija (dijeleći većinu koda između Androida i iOS-a) i višeplatformske hibridne stolne aplikacije s Electronom. Nedavno je Microsoft također izdao React Native za Windows.

Dva su moguća pristupa pri korištenju modernih JavaScript okvira – prikazivanje na strani klijenta, gdje preglednik preuzima kod i prikazuje korisničko sučelje, ili prikazivanje na strani poslužitelja, pri čemu se korisničko sučelje prikazuje na pozadini.

Glavna značajka React.js-a koja ga razlikuje od ostalih popularnih JavaScript okvira je fleksibilnost. Možete zgrabiti biblioteku i koristiti je za prikaz jednostavne stranice ili pogleda, ali također možete kombinirati React.js s drugim alatima i koristiti ga kao okvir koji će postaviti temelje za složenu aplikaciju.

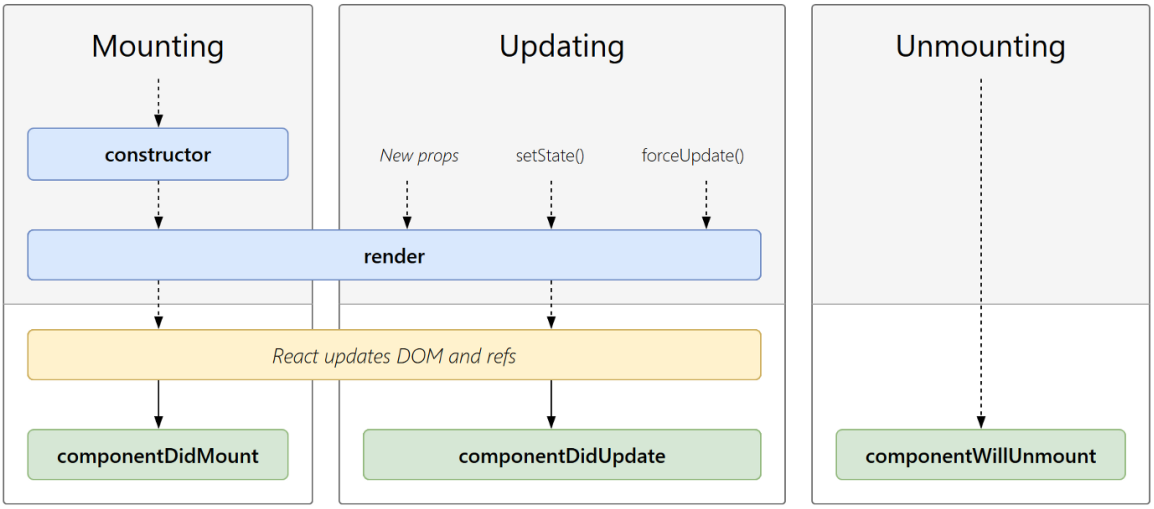
4.1.1. Komponente

Komponente su jedan od temeljnih blokova Reacta. Drugim riječima, možemo reći da će svaka aplikacija koju ćete razviti u Reactu biti sastavljena od dijelova koji se nazivaju komponente. Komponente uvelike olakšavaju zadatak izgradnje korisničkih sučelja, te inkapsuliraju pojedinačnu funkcionalnost. Možete vidjeti korisničko sučelje razbijeno na više pojedinačnih dijelova koji se nazivaju komponente i raditi na njima neovisno te ih sve spojiti u nadređenu komponentu koja će biti vaše konačno korisničko sučelje. One se mogu podijeliti na klasne i na funkcijske komponente.

4.1.2. Klasne komponente

Funkcionalne komponente lakše je konstruirati od React JS klasnih komponenti, koje su ES6 klase. Konstruktori, metode životnog ciklusa, funkcije renderiranja, kao i upravljanje stanjem ili podacima, uključeni su u svaku klasnu komponentu. Klasne komponente moraju proširiti React komponentu kako bi se mogle koristiti. Samo će React tada moći prepoznati da je ova specifična komponenta klasa i iscrtati ih ili vratiti potrebni React element.

Objekti klasne komponente imaju stanje, što znači da objekt može sadržavati informacije koje se mogu mijenjati tijekom životnog ciklusa objekta. Klasne komponente također mogu imati ulazna svojstva (poznata i kao „props“) koja im se prosljeđuju. Ulazna svojstva se prosljeđuju kao argumenti konstruktoru i trebaju se proslijediti klasi nadređene komponente pozivom „super(props)“. Podatci su dostupni tijekom cijelog vijeka trajanja objekta.



Slika 5 Životni ciklus React komponente, preuzeto iz (???, ???)

U Reactjs-u svaki proces stvaranja komponente uključuje različite metode životnog ciklusa. Ove metode nazivaju se životnim ciklusom komponente. Metode životnog ciklusa nisu jako kompliciranе i pozivaju se u različitim trenucima tijekom vijeka komponente. Životni ciklus komponente podijeljen je u tri faze. One su:

- Montiranje ili postavljanje (eng. *Mounting*)
- Ažuriranje (eng. *Updating*)

- Demontiranje (eng. *Unmounting*)

Montiranje je prva faza u životnom ciklusu komponente. To znači da je komponenta postavljena na DOM stablo. U ovome prvom koraku se definiraju njeni ulazni parametri, zatim nakon postavljanja prvog stanja komponente poziva se metoda „render()“ koja prikazuje ili iscrtava zadanu komponentu.

Nakon montiranja komponente ona sluša na bilo kakve promjene u njenom stanju i proslijeđenim parametrima. Prilikom promjene stanje pokreće se proces ažuriranja koji ponovo iscrtava komponentu sa novim stanjem.

Demontiranje je proces gdje se komponenta uništava svaki put kada napusti korisnikov prikaz tj. ekran.

4.1.3. Funkcijske komponente

Funkcijske komponente su samo normalne JavaScript funkcije. U današnje vrijeme se koriste isključivo funkcijske komponente, što znaci da se sve metode opisane u klasnim komponentama vise ne koriste. Kao zamjenu za te metode React je uveo kuke ili „hooks“ koje postižu isti ili cak bolji rezultat nego njegovi klasni predci. Kuke nam služe kako bi smo upravljali životnim ciklusom komponente u odnosu na promjene u stanju i ulaznim podatcima.

Funkcijske komponente su znatno poboljšanje u odnosu na klasne komponente te pokušavaju riješiti ceste probleme kod klasnih komponenti kao što su:

- Mijenjanje stanja komponenti kada je više ugniježđenih komponenti
- Kompleksne komponente postanu teške za čitati i razumjeti
- Klase su zbunjujuće u svojoj primjeni

Prednost funkcijskih komponenti je upravo to što je jednostavna JavaScript funkcija. Kao takva uvelike je laska za pročitati i razumjeti. Samim time što nisu klase nemaju potrebu za oslanjanje na kljucnu riječ „this“, što uvelike pomaže početnicima snalaženje unutar komponente. Također nemaju konstruktor što znaci da stanje se mora spremiti na dugi način, a taj način je korištenjem „useState“ kuke koja daje lako vidljivu vrijednost varijable i način za promijeniti vrijednost varijable na pravilan način koji će uzrokovati pravilno mijenjanje stanja i ponovo iscrtavanje komponente.

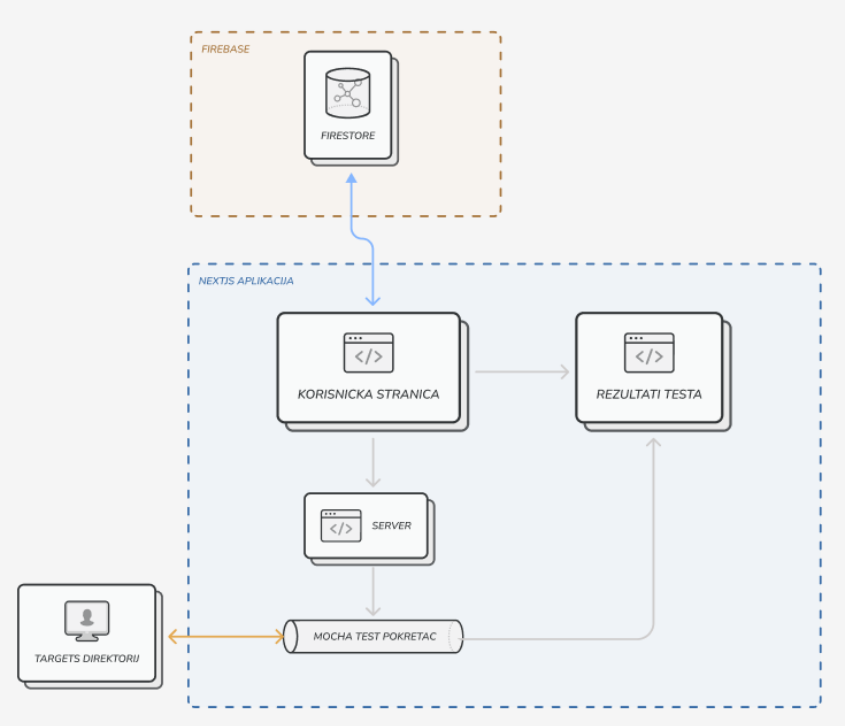
5. AGS

AGS (Automated grading system) ili system za automatsko ocjenjivanje je projekt sa ciljem ubrzavanja i automatizacije jednostavnih testova u području objektno orijentiranog programiranja. Sustav se sastoji od 4 dijela:

1. Stvaranje testova
2. Pokretanje testova
3. Izvršavanje testova
4. Pregled testova

5.1. Sadržaj AGS sustava

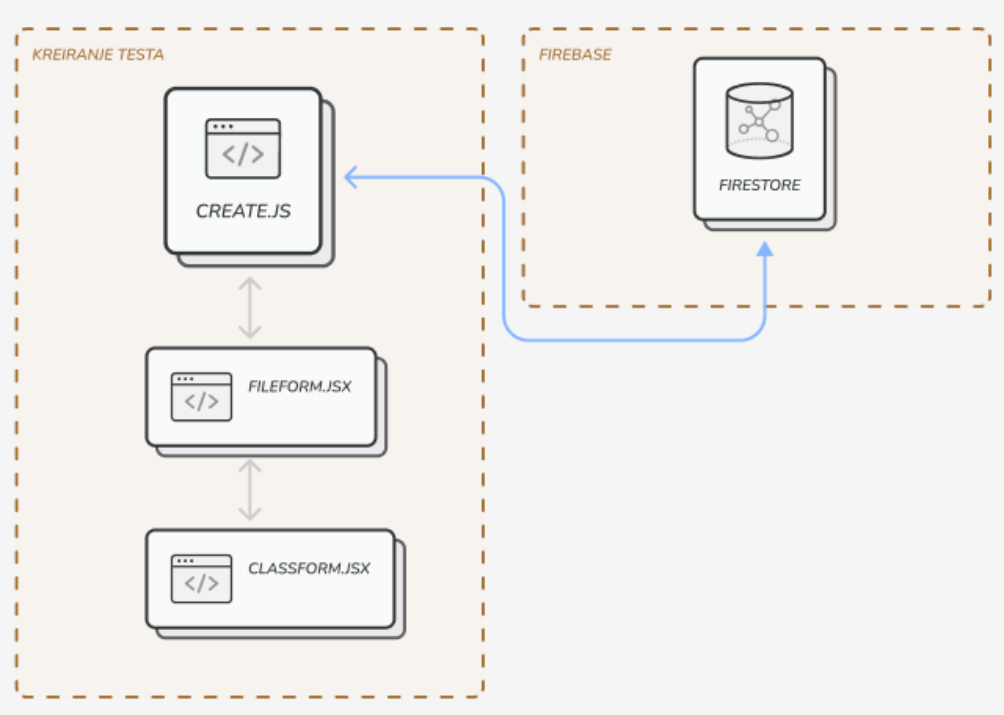
AGS sustav je od početka konstruiran da bude modularan i proširiv. Prednosti ovakvog pristupa omogućavaju proširivanje na druga područja kao programiranje mrežnih aplikacija ili čak i predmete koji ne koriste JavaScript kao ključni jezik. Sustav je razdvojen u vise dijelova koji koriste minimalnu interakciju između sebe.



Slika 6 Arhitektura AGS sustava

5.1.1. Stvaranje testova

Kako bi smo mogli pokrenuti test na učeničkome kodu prvo je potrebno definirati što točno naš test treba provjeriti. Iz tog razloga je izrađena stranica za stvaranje i izmjenjivanje testova. Dodavanje novih polja u test je lagano i intuitivno, no proširivanje same stranice kako bi se dodale dodatne mogućnosti ovisi o znanju Reacta i Nextjs okvira.



Slika 7 Arhitektura sustava za stvaranje testova

Arhitektura sustava za stvaranje testova se sastoji od 2 dijela, stranice Create i Firestore baze podataka. Stranica create sadrži pod komponente FileForm i ClassForm koji omogućavaju dodavanje neograničenog broja datoteka i klasa koje treba provjeriti za vrijeme pokretanja testa.

5.1.2. Pokretanje testova

Kako bi smo pokrenuli test prvo moramo odabrati koji test želimo pokrenuti. Jednom napisani testovi su dostupni svim korisnicima bez obzira na lokaciju. Ovaj pristup je odabran kako bi olakšao dijeljenje testova između korisnika te kako bi omogućio korištenje sustava sa vise lokacija bez poteškoća. Pokretanje testa se sastoji od četiri faze.

Prvi korak je dohvaćanje testova sa Firestore servisa. Svi testovi se moraju držati istog formata podataka kako ne bi došlo do kasnijih greška. Testovi su zatim spremljeni u JSON formatu te imaju sljedeću strukturu:


```
{
  "description": "Test vjezbe 7 OOP",
  "id": "bVgWfGvo39bUV8jCYHjP",
  "createdBy": "CuErhHs9mEPryHXcx2xf4bSo4sC3",
  "name": "Test VJ711",
  "testParams": [
    {
      "id": "bVgWfGvo39bUV8jCYHjP",
      "checkSyntax": true,
      "file": "jsKod/kod_00-staticka.js",
      "classes": [
        {
          "properties": ["coins", "spikes", "cilj"],
          "constructor": true,
          "className": "Postavke",
          "isStatic": true,
          "inherits": "EventEmitter",
          "functions": [{ "name": "reset", "overload": true }]
        }
      ]
    }
  ]
}
```


Figura 4 Struktura testa

Iz strukture testa važno je istaknuti nekoliko polja. Polje „testParams“ je ulazna točka Mocha test pokretača, te trenutno podržava samo jedan test koji se može pokrenuti. Ovaj tip podataka je niz što omogućava lagano proširenje u budućnosti. Polje „File“ sadrži lokaciju datoteke na kojoj je potrebno izvršiti test. Lokacija datoteke se nalazi u odnosu na svaku pojedinu datoteku unutar „Targets“ datoteke.

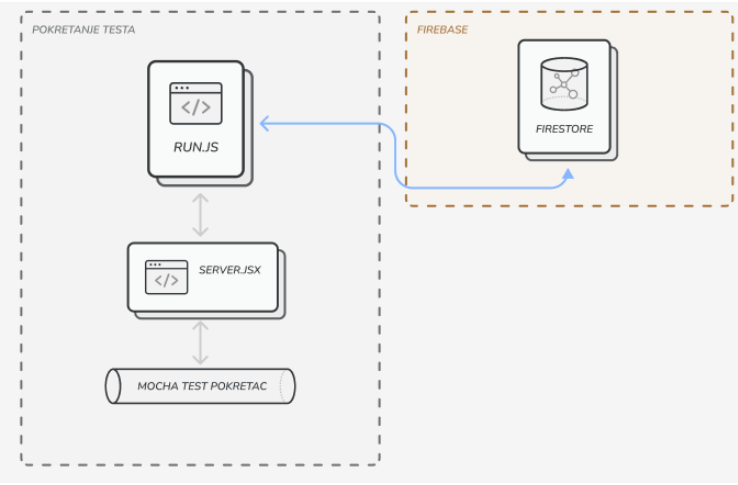
Drugi korak je odabir testa iz liste svih dohvaćenih testova. Nakon odabira testa on se prebacuje u drugi stupac koji ukazuje na trenutno odabrani test. Sa desne strane se također nalazi i treći stupac koji nam ispisuje sve pronađene direktorije u direktoriju „Targets“. Važno je naglasiti da sustav isključivo gleda direktorije te ostali tipovi nisu podržani.

Nakon odabira teste slijedi treći i zadnji korak koji se sastoji od pokretanja testa. Pritiskom na tipku Run šalje se POST zahtjev serveru na adresu "/api/runtest" sa svim podatcima trenutno odabranog testa. Nakon što zahtjev stigne na server izvršava se zapis testnih podataka u JSON datoteku pod nazivom „testconfig.json“, te se zatim pokreće Mocha testni pokretač.

```
const { exec } = require("node:child_process");
const fs = require("fs");

export default function handler(req, res) {
  fs.writeFileSync("./tests/testconfig.json", JSON.stringify(req.body));

  exec(
    "npx mocha tests/ags.spec.js --reporter mochawesome --reporter-options
reportDir=public/mochawesome-report"
  );
  res.status(200).json("test ran successfully");
}
```

Figura 5 Pokretač testa na serveru

Iz gornje figure možemo vidjeti da pokretanje testova je u potpunosti neovisno o poslanim podatcima i odgovoru servera. Što znaci da nismo u mogućnosti znati trenutni status teste te njegovu uspješnost izvršavanja. Povezivanje servera i Mocha testnog pokretača nije moguće jer testni pokretač koristi odvojenu paralelnu nit kako bi izvršio testove, te sami testovi se izvršavaju u paraleli. Zbog ovoga ograničenja rezultati testa će se uvijek prikazati pet sekundi nakon uspješnog odgovora sa servera. Spajanjem sva tri koraka dobivamo finalnu arhitekturu pokretanja testa.



Slika 8 Arhitektura za pokretanje testova

5.1.3. Izvršavanje testova

Nakon pokretanja testova koristeći dodijeljenu rutu na serveru, Mocha test pokretač je zaslužan za pronalazak svih datoteka unutar projekta koje se odnose na testiranje. Ove datoteke cesto imaju nastavak „.spec.js“. AGS projekt ima samo jednu datoteku za izvršavanje testova koja se zove „ags.spec.js“. Unutar te datoteke se nalazi sva potrebna logika za izvršavanje šest mogućih tipova testova.

Prije izvršavanja samog testa potrebno je pronaći sve datoteke unutar „Targets“ direktorija. Koristeći ugrađene mogućnosti Nextjs okvira možemo koristiti sve dostupne mogućnosti Node servera. Te mogućnosti uključuju skeniranje direktorija.

Nakon učitavanja svih direktorija vrijeme je za pokrenuti test na svim valjanim rezultatima.

Način izvršavanja testova je predstavljao jedan od temeljnih problema pri izradi projekta. Moguća su bila dva pristupa:

1. Prvi pristup se odnosio na učitavanje JavaScript koda unutar testnog okruženja te zatim izvršavanje zadanih testova. Ovaj pristup ima velike prednosti kao fleksibilnost testiranja svih mogućih zahtijeva i veću brzinu izvršavanja. Nažalost ovaj pristup također ima i veliku manu a to je nemogućnost testiranja sintaktički netočnog koda. Testno okruženje ne može prihvatiti i pokrenuti kod koji nije ispravan. Ako promotrimo ciljanu publiku na koja će pisati kod koji ćemo pokusati pokrenuti pronalazimo podosta sintaktičkih greška upravo mekog pristupa greškama unutar JavaScripta. Iz ovoga razloga način testiranja je morao biti prebačen na neki drugi način.
2. Drugi pristup se odnosi na učitavanje cijelog koda unutar JavaScript datoteke kao jedan dugi niz. Ovaj pristup uvelike ograničava fleksibilnost testiranja te stvari kao polimorfizam. Ovaj pristup je također bio preporučen od strane inteligentnog pomoćnika Copilot, te je odabran kao način izvršavanja testa

Nakon prihvaćanja drugog pristupa pri izvršavanju testova potrebno je napisati same testove. Ovdje također dolazi u pomoć pomoćnik Copilot koji je sam predložio i napisao sve testove unutar ovoga projekta.

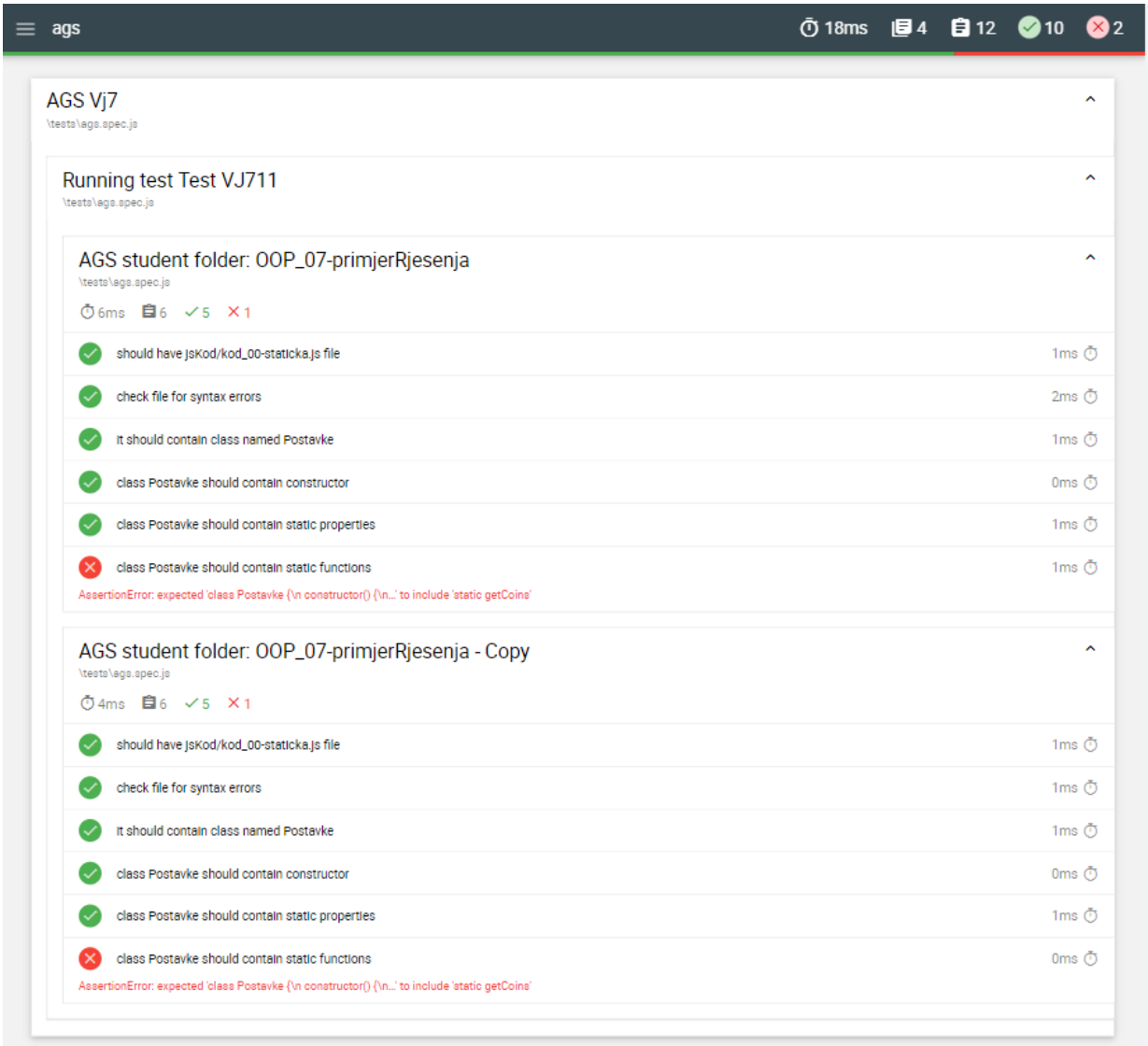
```
if (classData.inherits && classData.inherits.length > 0) {
  it(`class ${classData.className} should extend class ${classData.inherits} `,
function () {
    let kod = fs.readFileSync(workDirectory).toString();
    expect(kod).to.contain(
      `class ${classData.className} extends ${classData.inherits}`
    );
  });
}

if (testParam.checkSyntax) {
  it("check file for syntax errors", function () {
    let code = fs.readFileSync(workDirectory, "utf8");
    let script = new vm.Script(code);
    expect(script.runInNewContext()).to.not.throw;
  });
}
```

Figura 6 Primjer testova korištenih unutar aplikacije

U gornjem prikazu imamo dva tipa testova koji prikazuju zašto smo morali odabrati drugi pristup pri izvršavanju testova. Gornji test provjerava jeli zadana klasa nasljeduje drugu klasu. Ovaj test je relativno lagan za izvršiti s obzirom na zadana okruženja rada sa jednim jako dugim nizom. Donji test provjerava valjanost koda tj. provjerava sintaktičku točnost primljene datoteke. U koliko datoteka nije ispravno napisana to ne predstavlja nikakav problem u izvršavanju drugih testova i provjera valjanosti ostalih zahtijeva.

5.1.4. Prikaz rezultata testa

Nakon uspješnog pokretanja testa sa strane servera potrebno je prikazati rezultate testa. Mocha testni pokretač ima ugrađenu funkcionalnost izrade elegantne web stranice sa rezultatima testa. Za izradu rezultata se koristi proširenje zvano „Mochawesome“ koji generira „mochawesome-report“ unutar javnog direktorija. Generiranje rezultata unutar javnog direktorija omogućava jednostavan pristup rezultatima otvarajući novu stranicu u pregledniku sa lokacijom javnog direktorija.



Slika 9 Vizualni prikaz rezultata testa

Vizualni prikaz je podijeljen sa svaki direktorij na kojem je pokrenut test, te na same testove koji su odabrani da se izvrše na tom direktoriju. Alat Chai nam također omogućava pisanje detaljnijih poruka za ispis testova koji nisu zadovoljili uvijete.

Ovaj vizualni prikaz je također dostupan u JSON formatu sa mnoštvom dodatnih informacija, te ostavlja mogućnost proširenja u ostale sustave kao Moodle ili Elearning.

6. Zaključak

U ovome radu detaljno je opisana izrada aplikacije AGS-a za automatiziranje testova provedenih na objektno orijentiranom programima. Zbog modularnog pristupa izradi aplikacije ona je daljnje proširiva na ostale programske paradigme i tipove zadataka. Sama aplikacija je izrađena kao primjer na kojem se mogu prikazati neke od značajki i mogućnosti odabranih alata kao i njihova međusobna interakcija.

Vidljiv je napredak raznih tehnologija koje su omogućile bržu, jednostavniju i fleksibilniju izradu AGS sustava. Također je vidljiva evolucija JavaScript jezika koji omogućava primjenu širokog spektra alata i okvira u raznim situacijama.

Za razvoj klijentske strane aplikacije kao i serverske strane korišten je Nextjs i sve njegove mogućnosti. Za razvoj testova korišten je alat Mocha zbog svoje neusporedive fleksibilnosti i širokog spektra proširenja. Jedno od tih proširenja je biblioteka Chai koja omogućava brzo i jednostavno testiranje kao i pisanje testova.

Prikazan je razvoj okruženja za testiranje te bitne prepreke u testiranju koje postavljaju objektno orijentirani programi. JavaScript kao jezik nije baš poznat po svojoj primjeni u objektno orijentiranom programiranju te to stvara unikatne probleme u korištenju samog jezika kao i testiranju programa napisanih u njemu.

Literatura

Bobrow, M. S. (1985). *Object-Oriented Programming: Themes and Variations.* Intelligent Systems Laboratory, Xerox Palo Alto Research Center, 3333 Coyote Hill Road,: AI Magazine Volume 6 Number 4.

BOHM, C., & JACOPINI, G. (1966). *Flow Diagrams, Turing Machines aAd Languages With Only Two.* International Computation Centre and Istituto Nazionale.

Brilliant, S. S., & Wiseman, T. R. (1996). *The first programing paradigm and language dilema.* SIGCSE '96.

Bulent Tugrul, Gunduc, S., & Eryigit, R. (2017). *An Analysis of Imperative Programming Languages from Data and.* Ankara, Turkey: ICTACSE, 2017 .

Hinsen, K. (2009). *The Promises of Functional Programming.* S cie n t i f ic P rogr a m m i n g.

Kölling, M. (1999). *The Problem of Teaching Object-Oriented Programming, Part 1: Languages.* Journal of Object-Oriented Programming, 11.

Lloyd, J. (1994). *Practical Advantages of Declarative Programming.*

Michael, K. (1999). *The Problem of Teaching Object-Oriented Programming Part 2: Environments.* Journal of Object-Oriented Programming, 11.

PALUMBO, D. (2021). *The Flutter Framework: Analysis in a Mobile Enterprise Environment.*

Pierce, B. C. (2002). *Types and Programming Languages.* The MIT Press.

Sattar, A. (2019). *Comparison of different unit testing frameworks in JavaScript.* Medium.

stateofjs. (2021). *stateofjs.* stateofjs.

StMStNp, B. (1988). *What is Object-Oriented Programming* .

Stroustrup, B. (1988). *What is Object-Oriented Programming* .

Torgersson, O. (1996). *A Note on Declarative Programming Paradigms and the Future of Definitional Programming.* Chalmers University of Technology and Göteborg University.

Waterlow, S. (1982). *The Third Man's Contribution to Plato's Paradigmatism.*

POPIS SLIKA I TABLICA

Popis slika: