Dekompozicija sistema za preračunavanje zavarovalno-tehničnih vrednosti

Martin Potrč, Dušan Bratuša, Vida Primožič, Nejc Maleš

Novum-RGI Germany GmbH Podružnica Maribor, Maribor, Slovenija
vida.primozic@novum-rgi.si, dusan.bratusa@novum-rgi.si,
martin.potrc@novum-rgi.si, nejc.males@novum-rgi.si

V našem podjetju smo se odločili za preobrazbo sistema za preračunavanje zavarovalno-tehničnih vrednosti, ki smo ga poskusili prestaviti iz aplikacije za informacijsko podporo življenjskih zavarovalniških poslovnih procesov (imenovano V'ger Life) v samostojno komponento. Za ta projekt smo izbirali med različnimi tehnologijami. Za implementacijo uporabniškega vmesnika smo uporabili ogrodje Quasar, ki temelji na Vue.JS, za zaledni sistem smo izbrali aplikacijski strežnik Quarkus, za implementacijo jedra sistema za preračunavanje pa smo izbrali Node.js. Komponento smo poimenovali Product Development Workbench –oziroma s kratico – PDW.

Ključne besede:
Quasar
Node.js
Product Development Workbench
PDW
Quarkus
Chrome
JavaScript
Aktuar

Univerzitetna založba Univerze v Mariboru
ISBN 978-961-286-772-0
DOI https://doi.org/10.18690/um.feri.8.2023.17

1 Uvod

Izračunava zavarovalno-tehničnih vrednosti iz vhodnih podatkov, pa naj si gre za izračun ponudbe zavarovanje, izračun vrednosti ob sklenitvi ali pa za izračune ob spremembah zavarovalne pogodbe, zna biti programerski izziv tako za načrtovalca aplikacije, programerja/testerja ter koordinatorja med aktuarji in konceptom rešitve. Vsi vpleteni želimo težiti k hitrejšemu lansiranju po eni strani novih zavarovalnih produktov na trg, kakor tudi implementacijo novih generacij obstoječih produktov ob rednem spreminjanju razmer na zavarovalniškem trgu.

V ta namen je potrebno izdelati sistem za preračunavanje zavarovalno-tehničnih vrednosti, ki omogoča izpolnitev omenjenih teženj.

Obstoječa rešitev, ki bo opisana v naslednjih poglavjih, se je sicer izkazala za uporabno, a smo z analizo ugotovili nekatere pomanjkljivosti ter poskušali najti rešitev, ki bi aktuarju na prijaznejši način omogočala razvoj in testiranje produktov še preden preidejo v produkcijske sisteme.

2 Koncept implementacije v obstoječem sistemu

Na tem mestu bomo na kratko opisali osnovne principe in algoritme, katere uporabljamo v sistemu za izračunavanje vrednosti ter prikazali primer definicije vzorčnega zavarovalnega produkta.

2.1. Osnovni principi izračunavanja

Za osnovo si predstavljajmo definicijo, ki na podlagi vhodne vrednosti izračuna dve drugi vrednosti. Definicija in primer objekta sta povezana preko unikatnega imena oziroma notacije.

Definicija	
UIme	"O1"
	Vrednost1 Izračun1 Izračun2

Objekt	
UIme	"O1"
	Vrednost1=10

Slika 1: Preprost primer definicije in konkretnega objekta.

Recimo, da želimo definirati izračun na sledeči način: *Izračun1 = Vrednost1 / 1.77* ter *Izračun2=zaokroži(Izračun1, 2 decimalni mesti)*. Na primeru iz slike 1 bi definiciji za oba izračuna izgledali takole:

```
var v1 = obj.get('Vrednost1');
var r = v1/1.77;
return r
```

```
var i1 = obj.get('Izračun1');
var r = round(i1,2);
return r
```

Slika 2: Izračun.

Za izgradnjo formule za izračun so na voljo sledeči konstrukti:

- Referenca na objekt – *obj*
- Pridobitev vrednosti v objektu – *obj.get(ime)*
- Klic vgrajene funkcije s parametri – *func(p1)*
- Referenca na globalno knjižnico funkcij (npr. GKF) ter klic funkcije na tej knjižnici – *GKF.func1(p1,p2,p3)*

Objekti so lahko povezani tudi v objektne drevesne strukture s povezavo *starš* (angl. parent) in *otrok* (angl. child). S tem lahko na eni strani pridobimo objekt, ki je trenutnemu objektu starš, s funkcijo *obj.getParent()* ter seznam otrok objekta s funkcijo *obj.getChildren(childRelationName)*. Ta funkcija nam omogoča definicijo agregatnih funkcij kot so vsota, povprečje in podobno.

Dodatna in tudi zelo uporabna funkcionalnost je izpeljevanje (angl. derivation). Definiciji lahko definiramo, da je izpeljana iz neke druge definicije in s tem pridobimo možnost, da neko vrednost definiramo na predlogi. Ta vrednost je nato dosegljiva s prej omenjeno funkcijo *obj.get(imeNaPredlogi)*.

Ko nek vhodni objekt oziroma drevo objektov pošljemo skozi naš "računski stroj" nam sistem vrne novo izračunano drevo. Algoritem je zastavljen tako, da če se dostopa do neke vrednosti, ki še ni izračunana, s funkcijo *obj.get(ime)*, se ta vrednost izračuna samo enkrat, nato je dostopna vsem drugim funkcijam kot že izračunana vrednost.

Še ena od funkcionalnosti, ki je v tem članku ne bomo podrobneje opisali, je validacija vrednosti. Vsaka vrednost se lahko razširi s formulo za validacijo izračunane vrednosti. Rezultat validacije je po eni strani diskretna vrednost, ki prikazuje pravilnost, opozorilo ali napako, po drugi strani pa opis opozorila ali napake, ki se lahko doda v različnih jezikih. Najpreprostejši primer bi bil, da preverimo, če je vrednost v nekih osnovnih mejah (npr. $min < x < max$).

2.2. Primer produkta za izračun vrednosti življenjskega zavarovanja

Za primer in lažje razumevanje bomo predstavili formule za primer preprostega življenjskega zavarovanja z opcijo priključitve invalidnine za pokrivanje premije življenjskega zavarovanja v času invalidnosti.

Formule so definirane tako, da so razporejene v štiri-nivojsko drevesno strukturo:

- Skupina produktov (DL01EU2017)
    - Življenjski produkt (DL01EU2017)
        - Življenjski rizik (DL01EU2017)
            - Glavni del življenjskega rizika (DL01EU**M**2017)
            - Del živ. rizika za prilagoditev premije (DL01EU**A**2017)
            - Del živ. rizika po prenehanju plačevanja premije (DL01EU**P**2017)
            - Del živ. rizika za prilagoditev po prenehanju plačevanja premije (DL01EU**R**2017)
    - Rentni produkt za invalidnost (WL01EU2013)
        - Rizik rente za invalidnost (WL01EU2013)
            - Del rizika rente za invalidnost v dobi zavarovanja (WL01EU**R**2013)
            - Del rizika rente za invalidnost v dobi izplačevanja (WL01EU**L**2013)

Na vsakem izmed opisanih nivojev se nahajajo formule za vrednosti, kot so premije (riziko, stroški, dodatki ipd.), zavarovalne vsote, invalidnine, poročevalske vrednosti (rezervacije, premijski razrezi, amortizacije) ter še veliko drugih vrednosti, ki jih ne bomo naštevali, ker to ni relevantno za naš primer. Na vseh nivojih je možno uporabiti že izračunane vrednosti na drugih nivojih, kot na primer sešteti vse točno določene vrste premije. Primer: na najnižjem nivoju je definirana formula za riziko premijo, nakar je na višjih nivojih definirana formula za vsoto riziko premij v spodnjih nivojih, kar rezultira v riziko premiji na trenutnem nivoju, ter tako rekurzivno po nivojih navzgor.

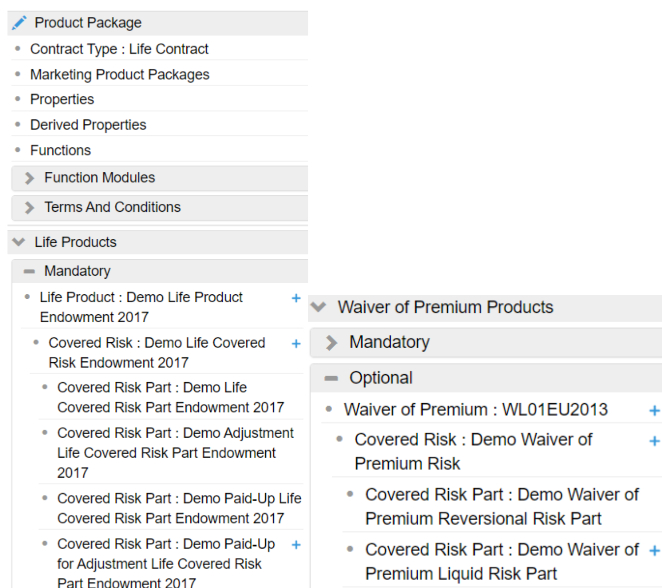
Za primere ponovne uporabe enakih formul obstajajo knjižnice funkcij, ki so v sistemu globalno dostopne po imenu in se lahko uporabijo v formulah štiri nivojskega primera s preprostim klicem na unikatno oznako knjižnice in definicijo metode za izračun (npr. *FN.round(x, factor, mode)*). Vse formule so definirane v skriptnem jeziku JavaScript.

Za primere podobnih produktov ali generacije produktov mora obstajati tudi koncept izpeljevanja. Koncept izpeljevanja spominja na razredno dedovanje, saj je možno definirati formule za vrednosti na nivoju predloge za določen nivo produkta, ki uporabljajo lastnosti konkretnega nivoja produkta. S tem se, kolikor se le da, izognemo podvajanju algoritmov.

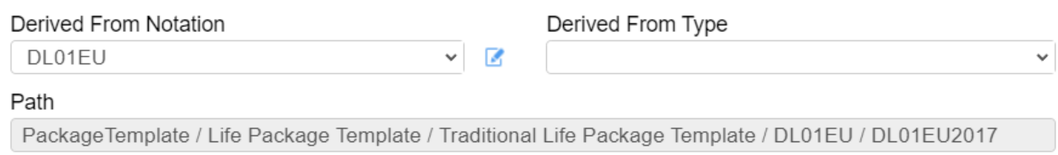
2.3. Primer definicije produkta v obstoječem integriranem sistemu

Na spodnjih slikah so prikazani deli uporabniškega vmesnika obstoječega integriranega sistema, kjer je mogoče pregledati ali spremeniti formule skupine produktov.

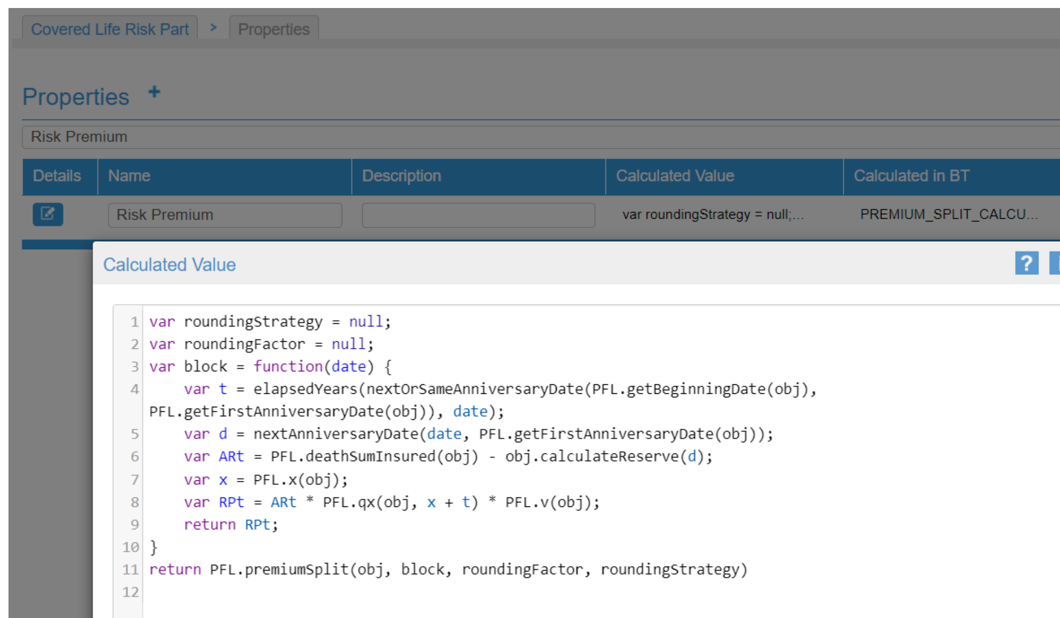
Prikaz štiri nivojske strukture v V'ger Life:



Slika 3: Prikaz strukture produkta v integriranem sistemu.



Slika 4: Prikaz izpeljevanja iz predlog v atributu '*Path*'.



Slika 5: Prikaz definiranja formule za riziko premijo

3 Analiza pomanjkljivosti

V nadaljevanju naštejmo pomanjkljivosti, ki so se pokazale pri uporabi samih definicij produktov, kot na primer iskanje napak, iskanje odvisnosti med formulami, iskanje razlogov za zmanjšanje zmogljivosti, kot so poraba pomnilnika in počasnost izvajanja. Hkrati smo želeli tudi iskati možnosti, kako bi aktuarji, tako v našem podjetju, kakor tudi pri poslovnih partnerjih, lahko na enostavnejši način in v čim krajšem času definirali nove generacije produktov iz obstoječih ter, kar je po navadi še pomembneje, testirali kalkulacije v sistemih izven produkcije ter jih po pozitivnih testnih rezultatih kar se da hitro vpeljali v produkcijo. Hkrati pa bi želeli ohraniti osnovne koncepte izračunavanja vrednosti, kot so izračuni po nivojih, uporaba lokalnih (znotraj gradnikov) ter globalnih (splošnih in neodvisnih od tipa produktov) funkcij ter koncept izpeljevanja konkretnih produktov iz predlog. Le-ti so se izkazali za največjo moč in uporabnost obstoječega sistema.

3.1. Ugotovljene pomanjkljivosti

Analiza je pokazala sledeče pomanjkljivosti oziroma izzive, s katerimi bi se želeli soočiti in jih premostiti v boljše rešitve:

- Priprava vhodnih podatkov za testiranje s strani razvijalcev zavarovalnih produktov je preveč kompleksna za nekoga, ki ni vešč programiranja v Javi – kot na primer aktuarji zaposleni pri zavarovalnicah
- Sledenje napakam s pomočjo razhroščevanja je bilo pri večini primerov možno le z uporabo tako imenovanih "štrcljev" – izpisov v konzole oziroma dnevnike
- Produkti razviti v paralelnem testnem okolju so težko prenosljivi v produkcijo
- Za razvoj in testiranje produktov je potrebna popolna paralelna testna aplikacija
- Preglednost formul in iskanje uporabljenih formul je v obstoječem sistemu praktično nemogoča oziroma zahteva preveč dodatnega programiranja
- Preglednost zgradbe štiri nivojske strukture vključno z izpeljavami iz predlog je zelo pomanjkljiva
- Sodelovanje med Java in JavaScript povzroča zmedo v razvoju in vzdrževanju algoritmov za izračunavanje vrednosti

- Obstoječa rešitev je temelji na JavaScript izvajalnem okolju implementiranem v Javi, kar je Java ponujala do verzije 15 [1][2]. Zaradi težav z vzdrževanjem in nadgradnjami JavaScript implementacije v Javi, ki so posledica hitrega razvoja JavaScript standardov in pomanjkanja vzdrževalcev implementacije, je bila podpora za to odstranjena iz JDK-ja. S to potezo se jasno kaže velika težava obstoječe rešitve, saj je to osnova za celotno njeno delovanje.

3.2. Ideje za izboljšavo

Ugotovljene pomanjkljivosti so napeljale na nadaljnjo analizo možnosti izboljšav. Zastavili smo si cilje, s katerimi bi pomanjkljivosti odpravili ali pa vsaj omilili njihov vpliv na sistem, razvijalce in uporabnike.

- Ločitev razvojnega dela zavarovalnega produkta od same integrirane aplikacije
- Možnost uporabe ločenega sistema za izračunavanje preko REST vmesnikov iz integrirane aplikacije
- Možnost vpeljave v ločenem sistemu razvitega produkta v integrirano aplikacijo, če povezava preko vmesnikov ne bi bila zaželena
- Preprostejša priprava testnih vhodnih podatkov ter izvajanje testnih scenarijev – tudi za aktuarje brez večjih programerskih znanj in izkušenj
- Preprostejši način programiranja testih scenarijev za integrirano testiranje
- Možnost razhroščevanja in možnost vpogleda v vmesne vrednosti med razhroščevanjem
- Izboljšana zmogljivost – izboljšava hitrosti izračunov ter brez nepotrebnega tratenja pomnilniškega prostora ter
- Preprostejše iskanje funkcij ter programiranje z uporabo samodejnega izpolnjevanja klicev funkcij (angl. code completion)
- Prekinitev sodelovanja med Javo in JavaScript – amputacija v Javi razvitega algoritma izračunavanja ter premik tega v JavaScript

4 Prehod na samostojno aplikacijo za izračun vrednosti

Za namen prehoda smo preko pomanjkljivosti začeli s pregledom možnih orodij na trgu, s katerimi bi lahko pomanjkljivosti odpravili ter vzpostavili zadovoljivo arhitekturo, ki omogoča, da ohranimo vse bistvene funkcionalnosti, ki so glavne prednosti naših algoritmov.

4.1. Izbira tehnologij

Za ta projekt smo izbirali med različnimi tehnologijami. Za implementacijo uporabniškega vmesnika smo uporabili ogrodje Quasar [3], ki temelji na Vue.JS, za zaledni sistem smo izbrali aplikacijski strežnik Quarkus [5], za implementacijo jedra sistema za preračunavanje pa smo izbrali NodeJS [4].

Izbira teh orodij ni bila naključna. Po primerjavi orodij na trgu smo se odločili za Node.js, ker je ena izmed najbolj razširjenih rešitev za izvajanje JavaScript kode na strežniku. Quarkus smo izbrali zaradi njegove namembnosti, kajti omogoča integracijo v platformo za storitve v oblaku, kjer ponujamo tudi naš osrednji sistem. Ker je Quarkus primarno grajen za izvajanje v Kubernetes gruči, prav tako omogoča pakiranje v vsebnik (npr. Docker Image) in je primeren za hitro inštalacijo za potrebe tako testiranja kot tudi prezentacij. Quasar smo izbrali zaradi njegovih vnaprej pripravljenih gradnikov (Material Design), ki omogočajo enostavno, konsistentno in tudi hitrejšo izgradnjo uporabniških vmesnikov in spletnih aplikacij. Hkrati pa je sistem omogoča izgradnjo po uveljavljenih standardih in dobrih praksah dizajna uporabniških vmesnikov.

4.2. Arhitektura

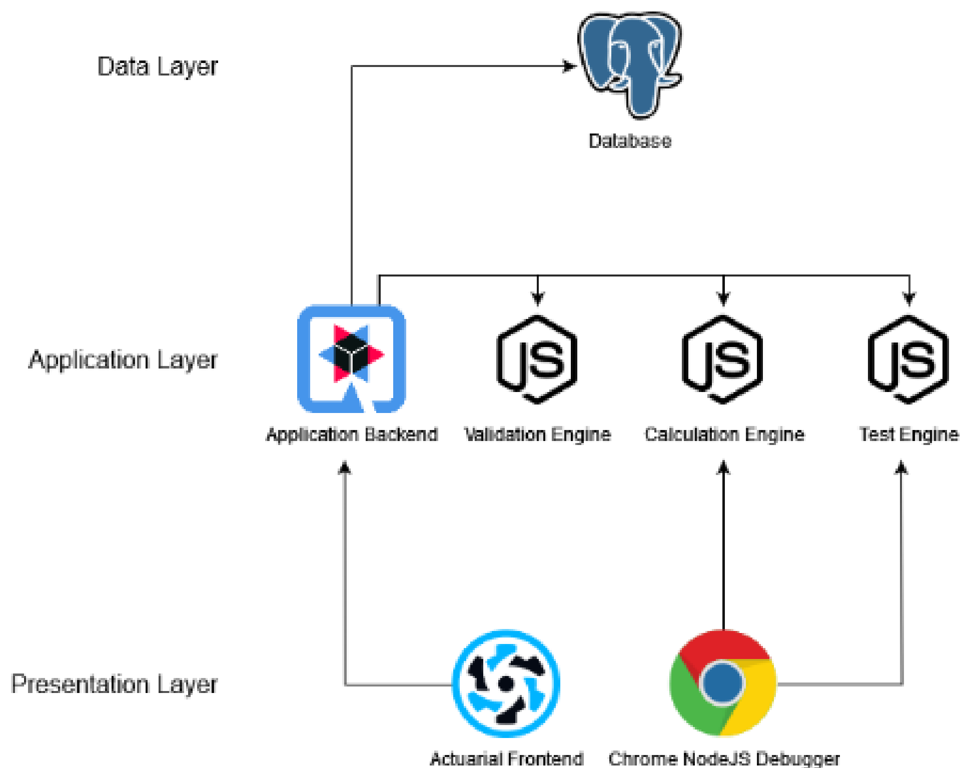
Sistem je tehnično zasnovan na klasični tri nivojski arhitekturi: podatkovni-, aplikacijski- in prezentacijski nivo.

Podatkovna baza je v komunikaciji zgolj z zalednim sistemom za razvoj zavarovalno-tehničnih vrednosti in shranjuje poslovne definicije s tehničnimi formulami. Definicije, sestavljene iz formul v JavaScript jeziku (vnesenih preko spletnega vmesnika), se pretvorijo v skupek JavaScript datotek, katere so posredovane kalkulatorju (angl. Calculation Engine). Le-te je odgovoren za samo izvajanje vseh formul v pravilnem zaporedju, glede na odvisnosti vnosov in rezultatov.

Vsaka posamezno vnesena formula je najprej preverjena preko preverjevalnika (angl. Validation Engine), ki skrbi za skladnost z vnaprej definiranimi pravili skladnosti kodiranja. Rezultat preverjanja je več nivojski in vsebuje lažje in težje kršitve pravil, npr. lažja kršitev je uporaba ne-striktne enakovrednosti ("=="), katera na nivoju JavaScript jezika ne zagotavlja popolne enakovrednosti med dvema primerjanima objektoma, medtem ko je težja kršitev manjkanje rezultata formule. Vse kršitve pravil pisanja formul so sporočene uporabniku preko vmesnika, vendar v kolikor so v formuli zaznane težje kršitve, le-ta ni sprejeta v zbirko vseh definicij.

Uporabniku je na voljo tudi ogrodje za testiranje (angl. Test Engine), ki omogoča pisanje testov posameznih formul in z njimi povezanih definicij. Vse vnesene formule lahko uporabnik preveri s testiranjem pričakovanih rezultatov glede na podane vnosne parametre. Vsak posamezen test ali skupina testov se izvede z izračunom preko kalkulatorja in preverbo rezultatov. Sam rezultat je zatem prikazan preko vmesnika uporabniku, z namenom da se čimprej najde mesto kjer je test padel.

Poleg testov lahko uporabnik, ki je vešč v kodiranju v JavaScript, uporabi tudi razhroščevalnik izvajanja tako testov kot tudi samih formul preko razvojnih orodij ponujenih s strani Chrome (Chrome DevTools [6]) brskalnika namenjenih specifično za Node.js okolje. Ta arhitekturna zasnova je prikazana na sliki 6.



Slika 6: Arhitekturna zasnova.

Iz pogleda produkcijskega izvajanja izračunov prezentacijski nivo, razhroščevalnik, testirano ogrodje in preverjevalnik, niso relevantni. Zato je sistem zasnovan na način, da vsak posamezen del lahko komunicira z drugimi preko REST API vmesnika in tako ponudi možnost integracije individualnih delov v druge sisteme. Tako

se v produkcijskih okoljih, kjer razvijalec zavarovalnih vrednosti ni več prisoten, odstrani nepotrebne sisteme in integrira samo kalkulator. S tem je produkcijsko okolje optimizirano za izvajanje izračunu vrednosti.

5 Primeri uporabe sistema Product Development Workbench – PDW

V nadaljevanju bomo prikazali nekaj primerov uporabe samostojnega sistema, ki smo ga do neke zadovoljive mere implementirali. Z implementacijo smo uspeli priti tako daleč, da smo sposobni definirati zavarovalne produkte, ki jih je mogoče testirati.

5.1. Prikaz knjižnice funkcij ter definicij gradnikov v PDW

Tako v PDW, kot prej v V'ger Life, so formule definirane v jeziku JavaScript. V formulah se uporabljajo funkcije, ki so definirane v knjižnicah in splošne funkcije, ki so definirane v programu samem.
Na primer: knjižnica Product Function Library (PFL) ima definirano formulo *qx()*, ki vrne verjetnost umrljivosti iz tablice umrljivosti za glede na starost osebe. Knjižnica je prikazana na sliki 7.

Slika 7: Prikaz knjižnice funkcij vključno s funkcijo za umrljivost q(x).

Na glavnem delu rizika DL01EUM je v formuli za riziko premijo uporabljen klic funkcije q(x). Prav tako je uporabljena splošna funkcija *elapsedYears(datefrom, dateUntil)*, ki vrne število let med dvema datumoma. Tretja možnost definiranja funkcij je definiranje le-teh na delu rizika (npr. *calculateReserve(date)* na "Traditional Life Covered Risk Part Template"), ki vrne vrednost rezervacije na določen datum. Na sliki 8 je vidno tudi opozorilo o slabi izbiri imena spremenljivke *Axn*.

Slika 8: Prikaz formule za izračun rezervacije.

Prikaz uporabe vseh treh tipov funkcij v formuli "Risk Premium" je na sliki 9.

Calculations

ADD PROPERTY | Risk Premium

Name	Property Type	Description	Code	Business Transactions	Validations	Action
CoveredRiskPartTemplate Risk Premium	DECIMAL		(see code below)		ADD VALIDATION	

```
1  let roundingStrategy = null;
2  let roundingFactor = null;
3  let block = function(date) {
4      let t = elapsedYears(nextOrSameAnniversaryDate(PFL.getBeginningDate(c
5      let d = nextAnniversaryDate(date, PFL.getFirstAnniversaryDate(obj));
6      let ARt = PFL.deathSumInsured(obj) - obj.calculateReserve(d);
7      let x = PFL.x(obj);
8      let RPt = ARt * PFL.qx(obj, x + t) * PFL.v(obj);
9      return RPt;
10 }
11 return PFL.premiumSplit(obj, block, roundingFactor, roundingStrategy)
```

MINIMIZE

Slika 9: Prikaz formule za riziko premijo.

5.2. Testiranje produktnih definicij v PDW

Prednost definiranja produktov v PDW je sprotno testiranje pravilnosti izračunov in sledenje morebitnim napakam. Primer testiranja skupine produktov DL01EU2017 z definiranjem štiri nivojske strukture:

RUN

Input parameters
Values given into calculation

```
1  {
2      "First Anniversary Date": "2022-06-01",
3      "Payment Frequency": 4,
4      "Notation": "DL01EU2017",
5      "Beginning Date": "2022-06-01",
6      "Change Date": "2022-06-01"
7  }
```

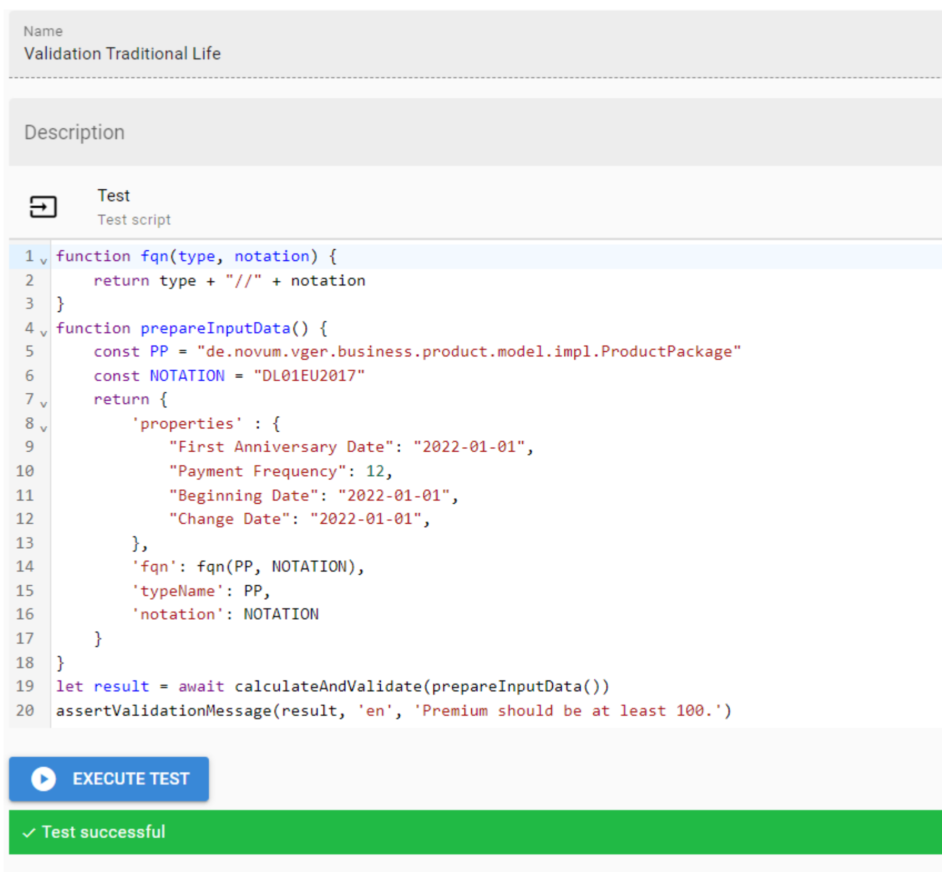
Input children
Children and parameters to be calculated

```
1  {
2      "Mandatory Life Product": [
3          {
4              "typeName": "de.novum.vger.business.product.model.impl.LifeProduct",
5              "notation": "DL01EU2017",
6              "properties": {
7                  "Insured Person Birth Date": "1976-06-01",
8                  "Insured Person Gender": "m"
9              },
10             "children": {
11                 "Covered Risk": [
12                     {
13                         "typeName": "de.novum.vger.business.product.model.impl.CoveredLifeRisk",
14                         "notation": "DL01EU2017",
15                         "properties": {
16                             "Beginning Date": "2022-06-01",
17                             "Insurance Period In Years": 20,
18                             "Sum Insured For Endowment": 250000
19                         },
20                         "children": {
21                             "Covered Risk Part": [
22                                 {
23                                     "typeName": "de.novum.vger.business.product.model.impl.CoveredLifeRiskPart",
```

Slika 10: Prikaz definiranja testa za skupino produktov.

5.3. Testiranje in validacija produktnih definicij v PDW

Robni pogoji produktnih definicij so definirani z pravili, ki preverjajo, ali so parametri posameznega produkta ustrezni. Primer takega pravila je, da preverimo, ali je premija zavarovanja nad določeno minimalno vrednostjo in to preverimo s testom. Na slikah 11 in 12 je prikaz uspešnega in neuspešnega testa validacije.



Slika 11: Prikaz uspešne validacije produkta.

Slika 12: Prikaz neuspešne validacije produkta.

V PDW lahko poleg osnovni testov testiramo tudi izračune produkta ob različnih vrednostih parametrov. Na primer izračun premije za različne starosti zavarovancev, različne dobe zavarovanja ali različne zavarovalne vsote. Ta način testiranja je namenjen za ugotavljanje obnašanja formul v robnih pogojih. S takimi testi zagotavljamo pravilnost kalkulacij pri mejnih vrednostih, preverjamo robustnost formul na neznane ali neobstoječe vrednosti parametrov, kakor tudi časovno in prostorsko zmogljivost neke definicije produkta.

```
const MONTHLY_PAYMENT = 12
const QUARTERLY_PAYMENT = 4
const YEARLY_PAYMENT = 1
const INSURANCE_PERIOD = 10
const BIG_INSURANCE_PERIOD = 20
const SMALL_SUM_INSURED = 10000
const BIG_SUM_INSURED = 25000
let result = await calculate(prepareInputData(MONTHLY_PAYMENT, INSURANCE_PERIOD, SMALL_SUM_INSURED))
expect(result.tree.properties['Annual Premium']).toBe(1549.61)
expect(result.tree.properties['Annual Premium With Additions']).toBe(1552.61)
expect(result.tree.properties['Premium Rate']).toBe(131.2)
```

Slika 13: Prikaz testiranja s pričakovanimi vrednostmi.

5.4. Razhroščevanje z razvojnimi orodji Chrome (DevTools)

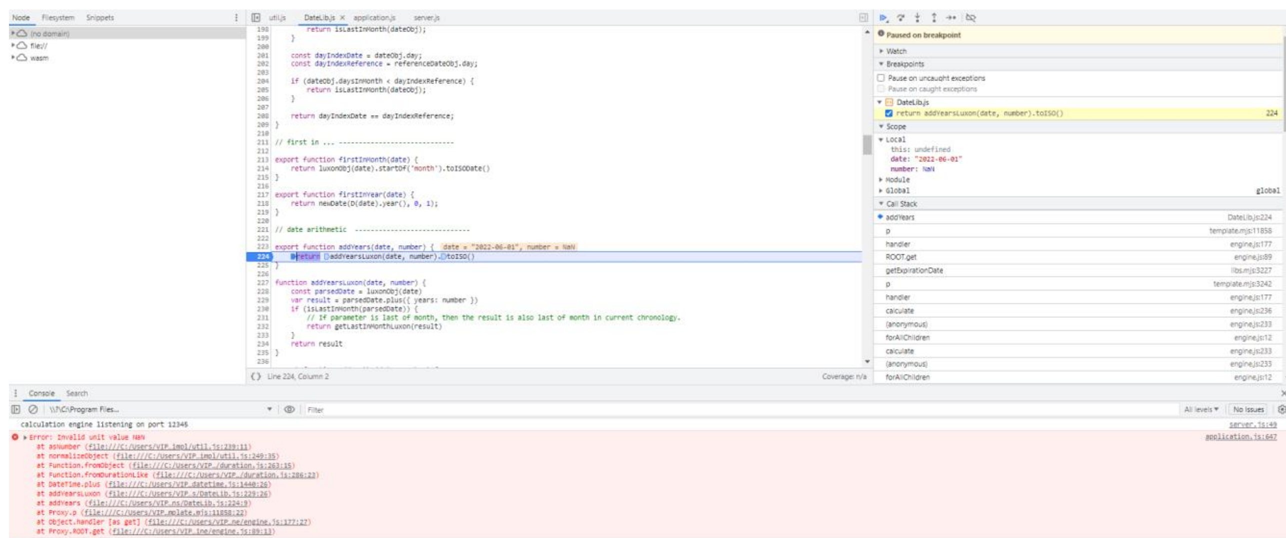
Dodatna pridobitev glede na integriran V'ger Life sistem je možnost razhroščevanja (angl. debugging) s pomočjo orodij, ki jih ponuja Chrome [6]. PDW omogoča povezavo na Chrome Node.js Debugger preko agenta, podobno, kot smo tega navajeni pri razvojnih okoljih za razvoj Java aplikacij.

V definiciji formule za "Insurance Tax Rate" smo uporabili spremenljivko "n", ki je nismo definirali, kar lahko vidimo na sliki 14.

```
'Insurance Tax Rate': { enumerable: true, value: { p: function(obj) {
//let n = PFL.n(obj);
if(n < 10) {
    return 6.5
}
else return 0.0
}
, }},
```

Slika 14: Namerna povzročitev napake - neobstoj spremenljivke *n*.

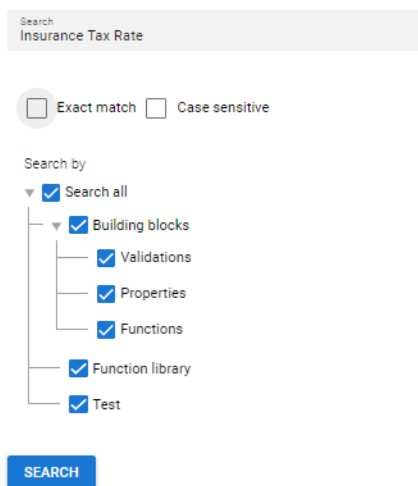
Pri zagonu testa lahko program na določenem mestu ustavimo in preverimo vrednosti spremenljivk sredi programa, kar je vidno na sliki 15.



Slika 15: Prikaz napake v chrome:inspect.

5.5. Preglednost formul in iskanje po formulah

Z razhroščevalnikom smo torej ugotovili, da imamo napako v formuli "Insurance Tax Rate". Z globalnim iskanjem lahko iščemo definicijo te formule po vseh produktih, knjižnicah in testih.



Slika 16: Prikaz globalnega iskanja.

Iz rezultata iskanja (slika 17) lahko vidimo, da se formula z napako nahaja na "Life Covered Risk Part Template"

Building Block Properties

Notation	Type ↑	Calculatable property name	Calculation
CoveredRiskPartTemplate	de.novum.vger.business.product.model.impl.CoveredLifeRiskPart	Insurance Tax Rate	1 `return obj.getParent().getPar`
Life Covered Risk Part Template	de.novum.vger.business.product.model.impl.CoveredLifeRiskPart	Insurance Tax Rate	1 `//let n = PFL.n(obj);` 2 `if(n < 10) {` 3 `return 6.5` 4 `}` 5 `else return 0.0`
CoveredRiskPartTemplate	de.novum.vger.business.product.model.impl.CoveredLifeRiskPart	Tax On Premium	1 `return PFL.roundPremium(obj,`

Slika 17: Prikaz rezultatov globalnega iskanja.

Kadar so formule definirane na različnih predlogah in se tako definicije "prepišejo", lahko v PDW to vidimo tako, da poiščemo definicijo formule na različnih predlogah. Primer: iščemo formule za "Premium Rate With Tax" v PDW. Na sliki 18 je "Premium Rate With Tax" definiran na predlogi DL01EU in ni prepisan na DL01EU2017, medtem ko sta "Minimal Premium Rate With Tax" in "Maximal Premium Rate With Tax" definirani tako na predlogi DL01EU kot tudi na DL01EU2017, kar lahko v PDW vidimo na ekranu hkrati.

Slika 18: Pregled formul.

6 Zaključek

Ugotavljamo, da smo nekatere cilje prehoda na samostojen sistem izračunavanja dosegli. Orodje smo tudi testirali tako, da smo vključili naše aktuarje v izgradnjo definicij naših zavarovalniških produktov, ki jih uporabljamo pri prezentacijah našega demo sistema. Izkazalo se je, da je mogoče iz stališča aktuarja s sistemom delati bolje kot je bilo to možno v integriranem sistemu.

Glede na to, da je PDW še vedno v fazi razvoja, pa je potrebno vložiti še nekaj truda v sam izgled orodja, kakor tudi dodati še nekatere funkcije, ki bi pripomogle k lažji in hitrejši implementaciji zavarovalniških produktov.

Manjkajoče funkcionalnosti, ki so že v fazi razvoja ali jih imamo namen v bližnji prihodnosti še razviti

- Samodejno izpolnjevanje programske kode (ang. code completion)
- Analiza pokritosti formul s testi (angl. code coverage)
- Moderna dokumentacija z vključenimi primeri uporabe ter dobre prakse
- Uvoz in izvoz definiranih gradnikov produktov ter knjižnic funkcij
- Možnost razširitve kalkulacijskega sistema z vtičniki (angl. plugins)
- Razvoj oziroma nadgradnja naših obstoječih rešitev z integracijo v PDW

Literatura

[1] https://openjdk.org/jeps/335: JEP 335: Deprecate the Nashorn JavaScript Engine, obiskano 6. 6. 2023

[2] https://openjdk.org/jeps/372: JEP 372: Remove the Nashorn JavaScript Engine, obiskano 6. 6. 2023

[3] https://quasar.dev Quasar: The enterprise-ready cross-platform VueJs framework, obiskano 8. 6. 2023

[4] https://nodejs.org/en/docs Node.js, obiskano 4. 6. 2023

[5] https://quarkus.io/ Quarkus, obiskano 1. 6. 2023

[6] https://developer.chrome.com/docs/devtools/ Chrome DevTools, obiskano 10. 6. 2023