***Facultatea de Electronica Telecomunicații si Tehnologia Informației***

***Programarea funcțională în JavaScript***

***Studenți: Profesor coordonator:***

***Dunărențu Radu Florin Dr. Ing. Ștefan Stăncescu***

***Dună Robert***

***441A***

***Cuprins:***

***1. Introducere*** *(Dunărențu Radu)*

* *Importanța programării funcționale*

***2. Limbaje de programare funcțională*** *(Dunărențu Radu)*

* *Avantaje*
* *Lucrul cu funcții*
* *Funcțiile IIFE și Closures*
* *Funcții de ordin superior*
* *Funcții pure*
* *Funcții anonime*
* *Înlănțuirea metodelor*
* *Recursivitate*
* *Evaluare leneșă*
* *Folosirea tehnicilor funcționale in JavaScript*
* *Exemple practice (Dună Robert****)***

***3. Concluzii*** *(Dună Robert)*

* *Limbaje funcționale vs orientate pe obiect*
* *Evoluția Limbajelor de Programare*

***4. Bibliografie***

Abstract

Cum software-ul devine din ce în ce mai complex, este cu atât mai important să fie organizat cât mai bine. Software-ul care este bine structurat este ușor de scris, înțeles și depanat, și oferă o colecție de module ce pot fi refolosite pentru a reduce timpul și costurile de producție viitoare. În această lucrare se dorește evidențiere diferenței față de celelalte stilului de programare și avantajele folosiri unui stil funcțional.

**1 Introducere**

Numele de programare funcțională provine din faptul că are ca operație fundamentală aplicarea de funcții pe argumente. Programul principal este scris ca o funcție care primește ca argumente parametrii de intrare ai programului și returnează rezultatul programului. De obicei funcția principală este definită ca alte funcții, care la rândul lor sunt definite de alte funcții, până când funcțiile de la nivelul de jos sunt primitive de limbaj. Toate aceste funcții de jos sunt asemănătoare funcțiilor matematice.

Caracteristicile și avantajele stilului funcțional sunt pe scurt următoarele. Programele funcționale nu conțin instrucțiuni de atribuire, o variabilă odată ce a primit o valoare nu mai poate fi modificată. Mai general, programele funcționale nu pot avea efecte secundare asupra altor programe. O apelare de funcție întotdeauna calculează numai rezultatul său. Aceasta abordare elimină o mare parte a bug-urilor și de asemenea face ordonarea funcțiilor irelevantă, aducând posibilitatea apelării oricând a unei funcții indiferent de stare, ceea ce duce la eliminarea flow controlului de către programator.

*Importanța programării funcționale*

Scopul principal al programării orientate pe obiecte este de a sparge problema în părți, exemplu figura 1.



Figură 1.1 Problemă spartă în componente obiect-orientate

De asemenea, aceste părți pot fi reunite pentru a forma părți mai mari după cum este ilustrat în figura 2.



Figură 2. Obiectele sunt compuse pentru a forma obiecte mai mari

 În funcție de legăturile și valoarea părților este descris un sistem în figura 3.



Figură 3. Diagrama de interacțiune a unui sistem obiect orientat

 Pentru comparație teoretică este ilustrat în figura 4 abordarea rezolvării unei probleme abordând stilul funcțional tot prin spargerea problemei în părți. Stilul OO sparge problema în grupuri de obiecte pe când abordarea funcțională sparge aceiași problemă în grupuri de funcții.

Într-un sistem OO, interacțiunea între obiecte cauzează schimbări interne asupra fiecărui obiect, ce duce la starea generală a unui sistem să fie compusă din mai multe schimbări minore de stări. Aceste schimbări minore si inter legate formează conceptual “o pânză de păianjen”. Cu creșterea complexității apare o confuzie legată de relațiile între obiecte și este necesară o cunoaștere a mai multor straturi pentru adăugarea a noi obiecte sau a unor proprietăți de sistem.



Figură 4. Problemă spartă în părți funcționale

 Pe de altă parte, un sistem construit folosind stilul funcțional minimizează modificările de stare. Deci, pentru a adăuga o nouă proprietate este nevoie de nevoie de a înțelege cum funcțiile pot să existe și funcționeze în contextul de date nedistructive si imutabile. Programarea funcțională nu este pentru a elimina schimbările de stare ci pentru a le reduce. Javascript suportă ambele stiluri de programare ceea ce înseamnă că un sistem poate fi și ar trebui creat folosind ambele stiluri.

Figură 5. Funcție compusă din mai multe funcții

**2 Limbaje de programare funcțională**

 Limbajele de programare funcțională sunt acele limbaje care facilitează paradigma de programare funcțională prin folosirea funcțiilor de ordin superior, funcțiilor pure și imutabilitatea datelor. În cazul în care un limbaj include caracteristici necesare pentru programarea funcțională, atunci este un limbaj funcțional. În cele mai multe cazuri, stilul de programare determină dacă un limbaj este funcțional sau nu.

 Programarea funcțională nu se poate realiza in C sau Java. Acestea și multe alte limbaje nu conțin structurile necesare pentru a suporta această paradigmă. Ele sunt orientate pur pe obiect și strict non-funcționale. În același timp, paradigma de programare orientată pe obiecte nu poate fi aplicată pe limbaje pur funcționale cum ar fi Scheme, Haskell si Lisp. Există limbaje ce suportă ambele paradigme. Python este cel mai cunoscut limbaj in acest caz, printre altele mai sunt și Ruby, Julia si Javascript. În cazul limbajului Javascript caracteristicile de programare funcțională sunt ascunse.

|  |  |  |
| --- | --- | --- |
| Caracteristici | Imperativ | Funcțional |
| Stil de Programare | Efectuează sarcini pas cu pas si gestionează schimbările de stare  | Definește problema si transformările necesare pentru găsirea soluției |
| Schimbări de Stare | Importantă | Inexistente |
| Ordinea Execuției | Importantă | Neimportantă |
| Gestionarea controlului  | Bucle, condiționale și apeluri de funcții | Apeluri de funcții și recursiune |
| Unitatea de manipulare primară | Structuri și clase de obiecte | Funcții ca obiecte și seturi de date |

Sintaxa limbajului trebuie să permită folosirea anumitor structuri de date cum ar fi funcții anonime și sisteme de tip dedus. Esențial pentru un limbaj funcțional este permiterea folosirii calcului lambda. De asemenea strategia de evaluarea a interpretorului trebuie să fie non-strictă și cu execuție întârziată care permite existența structurilor de date imutabile și evaluarea leneșă.

 Avantaje:

* Programele funcționale sunt mai curate, simple și mici. Aceasta duce la simplificarea depanării, testării și mentenanței. Spre exemplu, fie o funcție ce convertește un vector cu 2 dimensiuni într-un vector cu o dimensiune. Atât funcția scrisă imperativ, Code 1, cât și funcția scrisă funcțional, Code 2, primesc același argument de intrare și returnează același lucru. Se poate observa cum exemplul funcțional este mai concis și curat.

Code . Stil funcțional

varmerge2dArrayIntoOne2 **=** **function(**arrays**)** **{**

 **return** arrays**.**reduce**(** **function(**p**,**n**){**

 **return** p**.**concat**(**n**);**

 **});**

**};**

Code . Stil Imperativ

**function** merge2dArrayIntoOne**(**arrays**)** **{**

 **var** count **=** arrays**.**length**,**

 merged **=** **new** Array**(**count**),**

 c **=** 0**,**

 i **=** 0**,**

 j **=** 0**;**

 **for** **(**i**;** i **<** count**;** **++**i**)** **{**

 **for** **(**j**,** jlen **=** arrays**[**i**].**length**;** j **<** jlen**;** **++**j**)** **{**

 merged**[**c**++]** **=** arrays**[**i**][**j**];**

 **}**

 **}**

 **return** merged

**}**

* Programarea funcțională implică spargerea problemei în părți. Asta înseamnă că codul este modular. Programele modulare sunt clar specificate, ușor de depanat și întreținut. Facilitează testarea deoarece fiecare modul in parte poate fi testat independent.
* Programele funcționale împărtășesc o varietate de funcții ajutătoare datorită modularității programării funcționale. Spre exemplu, o funcție ce caută prin liniile de configurare ale unui fișier poate să fie folosită și pentru căutarea într-o structură de date de tip hash table.
* Reduce inter-dependențele. Deoarece un programator funcțional scrie funcții de tip prima clasă, funcții de ordin superior și funcții pure ce sunt complet independente și fără efecte secunde asupra variabilelor globale, interdependențele sunt reduse. Modificarea unei funcții nu va schimba alta funcție atât timp cât se păstrează input-urile si output-urile.

*Lucrul cu funcții*

Principiul programării funcționale este de a descompune o problemă într-un set de funcții. De multe ori funcțiile sunt legate împreună, imbricate reciproc și tratate ca cetățeni de primă clasă. Framework-urile jQuery și Node.js sunt printre cele mai cunoscute tehnologii ce folosesc aceste tehnici.

Pentru a explica și demonstra avantajele programării funcționale vom folosi un exemplu practic. Dorim să alcătuim o listă de valori care sunt atribute generice ale unor obiecte. Aceste obiecte pot fi orice: date, obiecte HTML, etc.

Code . Exemplul 1

**var**

 obj1 **=** **{**value**:** 1**},**

 obj2 **=** **{**value**:** 2**},**

 obj3 **=** **{**value**:** 3**},**

 values **=** **[];**

**function** accumulate**(**obj**)** **{**

 values**.**push**(**obj**.**value**);**

**}**

accumulate**(**obj1**);**

accumulate**(**obj2**);**

console**.**log**(**values**);** // Output: [obj1.value, obj2.value]

 Codul de mai sus funcționează dar este volatil. Orice cod poate să modifice lista values fără să apeleze funcția acumulate(). Dacă nu este inițializat la început lista cu valoarea goală, [], atunci codul nu o să funcționeze. Dacă variabila este declarată înauntrul funcției atunci ea nu poate fi modificată de niciun alt cod. Pentru a rezolva aceasta problemă este necesară folosirea unei funcții de tip IIFE (funcție auto-apelată).

*Funcțiile IIFE și Closures*

Prin folosirea unei funcții IIFE putem să returnăm o funcție ce returnează la rândul ei valorile din lista values. Variabilele din funcție sunt disponibile pentru modificare numai în interiorul funcției.

Principul ține de domeniul variabilelor. Lista values este disponibilă în interiorul funcției accumulate() chiar și atunci cînd funcția este apelată din afara domeniului ei. Aceasta este un closure. Closures în javascript sunt funcții ce au acces la domeniul părintelui chiar și atunci cînd acesta este închis. Closures sunt o caracteristică a tuturor limbajelor funcționale. Limbajele imperative tradiționale nu permit existența lor din motive de securitate și gestiunea proastă a memoriei.

Code 4. IIFE & Closure

**var** ValueAccumulator **=** **function()** **{**

 **var** values **=** **[];**

 **var** accumulate **=** **function(**obj**)** **{**

 **if** **(**obj**)** **{**

 values**.**push**(**obj**.**value**);**

 **return** values**;**

 **}**

 **else** **{**

 **return** values**;**

 **}**

 **};**

 **return** accumulate**;**

**};**

**var** accumulator **=** ValueAccumulator**();**

accumulator**(**obj1**);**

accumulator**(**obj2**);**

console**.**log**(**accumulator**());**

// Output: [obj1.value, obj2.value]

*Funcții de ordin superior*

Funcțiile IIFE sunt de fapt o formă de funcții de ordin superior. Funcțiile de ordin superior sunt funcții care fie iau o alta funcție ca intrare sau returnează o funcție ca ieșire.

Funcțiile de ordin superior nu sunt des întîlnite în practică. În timp ce un programator imperativ ar putea folosi o bulcă pentru a repeta o matrice, un programator funcțional ar avea cu totul altă abordare. Prin utilizarea unei funcții de ordin superior , matricea poate fi prelucrată prin aplicarea unei funcții pe fiecare element al matricei pentru a crea o matrice noua.

Aceasta este ideea centrală a paradigmei de programare funcțională. Funcțiile de ordin superior permit programatorului sa transmită logica altor funcții, la fel ca la obiecte.

Pentru a observa aceasta facilitate, vom folosi funcția ValueAcumulator() din secțiunea anterioară.

Code 5. Funcție de ordin superior

// folosim forEach() pentru a itera lista si apelarea

// funcției de reapelare, accumulator, pentru fiecare element

**var** accumulator2 **=** ValueAccumulator**();**

**var** objects **=** **[**obj1**,** obj2**,** obj3**];**

objects**.**forEach**(**accumulator2**);**

console**.**log**(**accumulator2**());**

*Funcții pure*

Funcțiile pure sunt acele funcții ce returnează o valoare calculată numai din argumentele ei. Variabilele globale și cele din stări părinte nu pot fi utilizare, nu există efecte secundare. Cu alte cuvinte, argumentele nu pot fi alterate. Prin urmare, funcțiile pure sunt utilizare numai pentru valoarea lor de ieșire.

Un exemplu de funcție pură este o funcție matematică. Math.sqrt(4) o să returneze mereu valoarea 2, nu are setări sau stare proprie și nu o să aibă niciodată efecte secundare.

Funcțiile pure sunt adevărata interpretare a termenului matematic pentru **funcție**, o relație între intrări și ieșiri. Sunt simple, ușor de înțeles și sunt reutilizabile. Deoarece sunt independente, funcțiile pure se pot reutiliza frecvent.

Pentru a arăta asta vom compara o funcție pură cu una non-pură.

În timp ce funcția non-pură se bazează pe starea obiectului fereastră pentru a calcula înălțimea și lățimea, funcția pură este autonomă și cere ca aceste valori să fie furnizate ca argumente. Acest fapt permite ca mesajul să fie tipărit oriunde, iar acest lucru face ca funcția să fie mult mai versatilă.

Code 6. Funcție non-pură vs funcție pură

// funcție ce afișează un mesaj pe centrul ecranului

**var** printCenter **=** **function(**str**)** **{**

 **var** elem **=** document**.**createElement**(**"div"**);**

 elem**.**textContent **=** str**;**

 elem**.**style**.**position **=** 'absolute'**;**

 elem**.**style**.**top **=** window**.**innerHeight**/**2**+**"px"**;**

 elem**.**style**.**left **=** window**.**innerWidth**/**2**+**"px"**;**

 document**.**body**.**appendChild**(**elem**);**

**};**

printCenter**(**'hello world'**);**

// funcție pură care face același lucru

**var** printSomewhere **=** **function(**str**,** height**,** width**)** **{**

 **var** elem **=** document**.**createElement**(**"div"**);**

 elem**.**textContent **=** str**;**

 elem**.**style**.**position **=** 'absolute'**;**

 elem**.**style**.**top **=** height**;**

 elem**.**style**.**left **=** width**;**

 **return** elem**;**

**};**

document**.**body**.**appendChild**(**

 printSomewhere**(**'hello world'**,**

 window**.**innerHeight**/**2**)+**10**+**"px"**,**

 window**.**innerWidth**/**2**)+**10**+**"px"**)**

**);**

În timp ce funcția non-pură poate părea opțiunea mai ușoară, pentru că efectuează adăugarea în sine în loc de a întoarce un element, funcția pură printSomewhere() și valoarea ei returnată se combină mai ușor cu alte tehnici funcționale de programare.

Code 7. Combinarea unei funcții pure cu alte tehnici funcționale

**var** messages **=** **[**'Hi'**,** 'Hello'**,** 'Sup'**,** 'Hey'**,** 'Hola'**];**

messages

 **.**map**(function(**s**,**i**){**

 **return** printSomewhere**(**s**,** 100**\***i**\***10**,** 100**\***i**\***10**);**

**})**

 **.**forEach**(function(**element**)** **{**

 document**.**body**.**appendChild**(**element**);**

**});**

În cazul în care funcțiile sunt pure și nu se bazează pe stare sau mediu, nu ne pasă dacă sunt sunt sau nu calculate. Aceasta face parte din principul de evaluare leneșă.

*Funcții anonime*

Un alt beneficiu de a trata funcțiile ca obiecte de primă clasă este apariția funcțiilor anonime. După cum implică numele lor, funcțiile anonime sunt funcții fără nume. Ele permit programatorului sa definească logica ad-hoc, după cum este necesar. Beneficiul lor îl reprezintă comoditatea, dacă o funcție este menționată o singură dată nu mai este nevoie pentru a folosi o variabilă.

Code 8. Exemplu de funcție anonimă

**function** powersOf**(**x**)** **{**

 **return** **function(**y**)** **{**

 // aceasta este o funcție anonimă!!

 **return** Math**.**pow**(**x**,**y**);**

 **};**

**}**

powerOfTwo **=** powersOf**(**2**);**

console**.**log**(**powerOfTwo**(**1**));** // 2

console**.**log**(**powerOfTwo**(**2**));** // 4

console**.**log**(**powerOfTwo**(**3**));** // 8

powerOfThree **=** powersOf**(**3**);**

console**.**log**(**powerOfThree**(**3**));** // 9

console**.**log**(**powerOfThree**(**10**));** // 59049

Funcția returnată nu trebuie numită, ea poate fi folosită oriunde înafara funcției powerOf(), deci este o funcție anonimă.

Un alt exemplu ce folosește funcția acumulator de mai devreme scrisă folosind o funcție pură, de ordin superior și anonimă. De asemenea este auto-apelată, indicat de structura (function() {…}) (), parantezele de după funcția anonimă face ca aceasta sa fie apelată imediat. În exemplul de mai jos, lista values este asignată ieșirii funcției anonime.

Un dezavantaj al funcțiilor anonime este dificultatea de a fi identificate în stiva ceea ce face depanarea mai dificilă.

Code 9. Funcția acumulator scrisă ca funcție anonimă

**var**

 obj1 **=** **{**value**:** 1**},**

 obj2 **=** **{**value**:** 2**},**

 obj3 **=** **{**value**:** 3**};**

**var** values **=** **(function()** **{**

 // funcție anonimă

 **var** values **=** **[];**

 **return** **function(**obj**)** **{**

 // funcție anonimă

 **if** **(**obj**)** **{**

 values**.**push**(**obj**.**value**);**

 **return** values**;**

 **}**

 **else** **{**

 **return** values**;**

 **}**

 **}**

**})();** // auto apelată

console**.**log**(**values**(**obj1**));** // Returns: [obj.value]

console**.**log**(**values**(**obj2**));** // Returns: [obj.value, obj2.value]

*Înlănțuirea metodelor*

Înlănțuirea metodelor este des întîlnită în Javascript. Această tehnică este folosită pentru simplificare codului atunci cînd mai multe funcții sunt aplicate pe același obiect consecutiv.

Code 10. Înlănțuirea metodelor

// imperativ

arr **=** **[**1**,**2**,**3**,**4**];**

arr1 **=** arr**.**reverse**();**

arr2 **=** arr1**.**concat**([**5**,**6**]);**

arr3 **=** arr2**.**map**(**Math**.**sqrt**);**

// funcțional

console**.**log**(** **[**1**,**2**,**3**,**4**]**

 **.**reverse**()**

 **.**concat**([**5**,**6**])**

 **.**map**(**Math**.**sqrt**)**

**);**

arr**.**zip**([**11**,**12**,**13**,**14**).**map**(function(**n**){return** n**\***2**});**

// Output: 2, 22, 4, 24, 6, 26, 8, 28

*Recursivitate*

Recursivitatea este cea mai faimoasă tehnică funcțională. O funcție recursivă este aceea funcție ce se apelează reciproc. Când o funcție se reapelează reciproc ea se comportă ca o buclă, adică execută aceiași porțiune de cod de mai multe ori dar și ca o stivă de funcții. În lucrul cu funcții recursive trebuie evitată bucla infinită sau recursivitatea infinită, deci ca și în cazul buclelor se folosește o condiție de stop numită caz de bază.

Este posibilă convertirea oricărei bucle într-un algoritm recursiv și invers. În unele cazuri algoritmii recursivi sunt necesari pentru situații care diferă foarte mult de cele în care buclele sunt adecvate.

Un bun exemplu este parcurgerea unui arbore. Este ușor de realizat parcurgerea lui folosind un algoritm recursiv, pe când folosind o buclă gradul de complexitate crește.

Code 11. Parcurgerea unui arbore recursiv

**var** getLeafs **=** **function(**node**)** **{**

 **if** **(**node**.**childNodes**.**length **==** 0**)** **{**

 // caz de bază

 **return** node**.**innerText**;**

 **}**

 **else** **{**

 // caz recursiv

 **return** node**.**childNodes**.**map**(**getLeafs**);**

 **}**

**}**

*Evaluarea leneșă*

Evaluarea leneșă, numită și evaluare non-strictă, apelare cu necesitate sau execuție amânată, este o strategie de evaluare, care așteaptă până când este necesar pentru a calcula valoarea rezultată a funcției și este deosebit de utilă pentru programarea funcțională. Spre exemplu, este clar că linia de cod x = func() apelează pe x să fie atribuit valorii returnate de funcția func(). Cu ce este egal x nu contează până la necesitatea acestuia. Așteptând apelarea funcției func() până când este nevoie de x se numește evaluare leneșă.

Această strategie poate să crească performanțele, mai ales când sunt folosite metode înlănțuite și liste. Un alt beneficiu al evaluării leneșe îl reprezintă seriile infinite. Deoarece nimic nu este calculat până nu este nevoie de el este posibilă următoarea scriere.

Acest beneficiu deschide ușa unor noi posibilități cum ar fi: execuție asincronă, paralelizare și compoziție, printre multe altele.

Code 12. Serii infinite

**var** infinateNums **=** range**(**1 to infinity**);**

**var** tenPrimes **=** infinateNums**.**getPrimeNumbers**().**first**(**10**);**

*Folosirea tehnicilor funcționale in JavaScript*

O să demonstrăm facilitatea programării funcționale prin convertirea unui cod scris în stil imperativ în stil funcțional. O să fie examinată și o versiune orientată pe obiecte dar se poate observa similitudinea cu cel imperativ numai structurarea este diferă. Exemplul nostru va fi o aplicație de organizare a sarcinilor, preluarea datelor se face de pe un server. Pentru a realiza această aplicație este folosită librăria Ramda.js

Code 13. Datele de pe server

**var** data **=** **{**

 result**:** "SUCCESS"**,**

 interfaceVersion**:** "1.0.3"**,**

 requested**:** "10/17/2013 15:31:20"**.**

 lastUpdated**:** "10/16/2013 10:52:39"**,**

 tasks**:** **[**

 **{**id**:** 104**,** complete**:** **false,** priority**:** "high"**,**

 dueDate**:** "11/29/2013"**,** member**:** "Scott"**,**

 title**:** "Do something"**,** created**:** "9/22/2013"**},**

 **{**id**:** 105**,** complete**:** **false,** priority**:** "medium"**,**

 dueDate**:** "11/22/2013"**,** member**:** "Lena"**,**

 title**:** "Do something else"**,** created**:** "9/22/2013"**},**

 **{**id**:** 107**,** complete**:** **true,** priority**:** "high"**,**

 dueDate**:** "11/22/2013"**,** member**:** "Mike"**,**

 title**:** "Fix the foo"**,** created**:** "9/22/2013"**},**

 **{**id**:** 108**,** complete**:** **false,** priority**:** "low"**,**

 dueDate**:** "11/15/2013"**,** member**:** "Punam"**,**

 title**:** "Adjust the bar"**,** created**:** "9/25/2013"**},**

 **{**id**:** 110**,** complete**:** **false,** priority**:** "medium"**,**

 dueDate**:** "11/15/2013"**,** member**:** "Scott"**,**

 title**:** "Rename everything"**,** created**:** "10/2/2013"**},**

 **{**id**:** 112**,** complete**:** **true,** priority**:** "high"**,**

 dueDate**:** "11/27/2013"**,** member**:** "Lena"**,**

 title**:** "Alter all quuxes"**,** created**:** "10/5/2013"**}**

 // , ...

 **]**

**};**

Scopul va fi creare unei funcții care acceptă ca argument numele unei persoane(member), apoi preia datele de pe server și alege sarcinile care nu sunt finalizate de persoana respectivă. Returnează id-ul sarcinilor, prioritatea, titlul și data de finalizare, sarcinile sunt sortate după data de finalizare.

Deoarece preluarea datelor de pe server va fi asincronă, vom organiza codul cu promisiuni iar funcția va întoarce o promisiune care ar trebui să rezolve o listă cu obiecte ce conțin proprietățile necesare.

Pentru scopul demonstrativ, vor fi ignorate toate verificările de erori. Într-un caz real, de producție trebuie luate în considerare eșecurile de comunicare cu serverul și scenarii de date greșite.

Code 14. Rezultatul de ieșire al funcției

tasks**:** **[**

 **{**id**:** 104**,** complete**:** **false,** priority**:** "high"**,**

 dueDate**:** "11/29/2013"**,** member**:** "Scott"**,**

 title**:** "Do something"**,** created**:** "9/22/2013"**},**

 **{**id**:** 105**,** complete**:** **false,** priority**:** "medium"**,**

 dueDate**:** "11/22/2013"**,** member**:** "Lena"**,**

 title**:** "Do something else"**,** created**:** "9/22/2013"**},**

 // , ...

**]**

*Exemplul imperativ de rezolvare a problemei*

Code 15. Abordarea imperativă a problemei

**var** getIncompleteTaskSummariesForMember\_imperative **=** **function(**memberName**)** **{**

 **return** fetchData**()**

 **.**then**(function(**data**)** **{**

 **return** data**.**tasks**;**

 **})**

 **.**then**(function(**tasks**)** **{**

 **var** results **=** **[];**

 **for** **(var** i **=** 0**,** len **=** tasks**.**length**;** i **<** len**;** i**++)** **{**

 **if** **(**tasks**[**i**].**member **==** memberName**)** **{**

 results**.**push**(**tasks**[**i**]);**

 **}**

 **}**

 **return** results**;**

 **})**

 **.**then**(function(**tasks**)** **{**

 **var** results **=** **[];**

 **for** **(var** i **=** 0**,** len **=** tasks**.**length**;** i **<** len**;** i**++)** **{**

 **if** **(!**tasks**[**i**].**complete**)** **{**

 results**.**push**(**tasks**[**i**]);**

 **}**

 **}**

 **return** results**;**

 **})**

 **.**then**(function(**tasks**)** **{**

 **var** results **=** **[],** task**;**

 **for** **(var** i **=** 0**,** len **=** tasks**.**length**;** i **<** len**;** i**++)** **{**

 task **=** tasks**[**i**];**

 results**.**push**({**

 id**:** task**.**id**,**

 dueDate**:** task**.**dueDate**,**

 title**:** task**.**title**,**

 priority**:** task**.**priority

 **})**

 **}**

 **return** results**;**

 **})**

 **.**then**(function(**tasks**)** **{**

 tasks**.**sort**(function(**first**,** second**)** **{**

 **return** first**.**dueDate **-** second**.**dueDate**;**

 **});**

 **return** tasks**;**

 **});**

**};**

*Exemplu OOP de rezolvare a problemei*

Code 16. Prima parte a exemplului OOP

// metoda principală

**var** getIncompleteTaskSummariesForMember\_objectOriented **=** **function(**memberName**)** **{**

 **return** fetchData**()**

 **.**then**(function(**data**)** **{**

 **var** taskList **=** **new** TaskList**(**data**.**tasks**);**

 taskList**.**chooseByMember**(**memberName**);**

 taskList**.**chooseByCompletion**(false);**

 **var** newTaskList **=** taskList**.**getSummaries**();**

 newTaskList**.**sort**(new** TaskListSorter**(**"dueDate"**));**

 **return** newTaskList**.**tasks**;**

 **});**

**};**

**var** TaskList **=** **(function()** **{**

 **var** TaskList **=** **function(**/\*Task[]\*/ tasks**)** **{**

 **this.**tasks **=** tasks**;**

 **};**

 TaskList**.prototype.**chooseByMember **=** **function(**memberName**)** **{**

 **var** results **=** **[];**

 **for** **(var** i **=** 0**,** len **=** **this.**tasks**.**length**;** i **<** len**;** i**++)** **{**

 **if** **(this.**tasks**[**i**].**member **===** memberName**)** **{**

 results**.**push**(this.**tasks**[**i**]);**

 **}**

 **}**

 **this.**tasks **=** results**;**

 **};**

 TaskList**.prototype.**chooseByCompletion **=** **function(**completion**)** **{**

 **var** results **=** **[];**

 **for** **(var** i **=** 0**,** len **=** **this.**tasks**.**length**;** i **<** len**;** i**++)** **{**

 **if** **(this.**tasks**[**i**].**complete **==** completion**)** **{**

 results**.**push**(this.**tasks**[**i**]);**

 **}**

 **}**

 **this.**tasks **=** results**;**

 **};**

Este important de menționat diferențele între stilul imperativ și orientat pe obiect în exemplu, înafară de câteva cuvinte “this“, diferența ține de organizare. Conținutul funcțiilor este asemănător, modul de organizare variază.

Code 17. Exemplu de rezolvare abordând stilul funcțional

**var** propEq **=** use**(**pipe**).**over**(**get**,** eq**);** // TODO: move to library?

**var** getIncompleteTaskSummariesForMemberFunctional **=** **function(**memberName**)** **{**

 **return** fetchData**()**

 **.**then**(**get**(**'tasks'**))**

 **.**then**(**filter**(**propEq**(**'member'**,** memberName**)))**

 **.**then**(**reject**(**propEq**(**'complete'**,** **true)))**

 **.**then**(**map**(**pick**([**'id'**,** 'dueDate'**,** 'title'**,** 'priority'**])))**

 **.**then**(**sortBy**(**get**(**'dueDate'**)));**

**};**

Code 18. A doua parte a exemplului OOP

TaskList**.prototype.**getSummaries **=** **function()** **{**

 **var** results **=** **[],** task**;**

 **for** **(var** i **=** 0**,** len **=** **this.**tasks**.**length**;** i **<** len**;** i**++)** **{**

 task **=** **this.**tasks**[**i**];**

 results**.**push**({**

 id**:** task**.**id**,**

 dueDate**:** task**.**dueDate**,**

 title**:** task**.**title**,**

 priority**:** task**.**priority

 **})**

 **}**

 **return** **new** TaskList**(**results**);**

 **};**

 TaskList**.prototype.**sort **=** **function(**/\*TaskListSorter\*/ sorter**)** **{**

 **this.**tasks**.**sort**(**sorter**.**getSortFunction**());**

 **};**

 **return** TaskList**;**

**}());**

**var** TaskListSorter **=** **(function()** **{**

 **var** TaskListSorter **=** **function(**propName**)** **{**

 **this.**propName **=** propName**;**

 **};**

 TaskListSorter**.prototype.**getSortFunction **=** **function()** **{**

 **var** propName **=** **this.**propName**;**

 **return** **function(**first**,** second**)** **{**

 **return** first**[**propName**]** **<** second**[**propName**]** **?** **-**1 **:**

 first**[**propName**]** **>** second**[**propName**]** **?** **+**1 **:** 0**;**

 **}**

 **};**

 **return** TaskListSorter**;**

**}());**

**3 Concluzii**

*Limbaje funcționale vs orientate pe obiect*

În această lucrarea am discutat despre diversele moduri în care programarea funcțională se leagă de principiile programării orientate pe obiecte. Programarea funcțională accentuează puterea reutilizării și compoziția de comportament prin funcții de ordin superior. Nu există nici o îndoială că structurile de date imutabile îmbunătățesc siguranța codului. De asemenea ceste caracteristici sunt disponibile și într-un context orientat pe obiect, tema comună o reprezintă beneficiile ce sunt atinse sunt universale la ambele abordări. Ca programatori suntem întotdeauna în căutarea de a scrie cod mai sigur și de construi un comportament mai flexibil.

Ideea programării funcționale este procesul de gândire. Nu este necesar un nou limbaj de programare pentru a programa într-un stil funcțional, deși unele caracteristici de limbaj ajută uneori. Introducerea expresiilor lambda in Java 8 reprezintă un pas înainte pentru folosirea stilului funcțional în Java. În timp ce programarea orientată pe obiecte a fost în mod tradițional despre încapsularea datelor și al comportamentului, acum se adaugă suport pentru comportament independent mulțumită ideilor din programarea funcțională.

Alte limbaje de programare cum ar fi Scala sau Haskell duc ideile funcționale la un alt nivel. Scala oferă un amestec de facilități funcționale și obiecte orientate, în timp ce Haskell se concentrează pe programarea pur funcțională.

Dezvoltarea software constă în producerea de software de o calitate superioară care satisface clienții și businessul. Un stil de programare funcțional poate atinge acest scop, dar nu este singurul.

*Evoluția Limbajelor de Programare*

Una dintre cele mai interesante tendințe în timp ale limbajelor de programare este trecere treptată la limbaje care sunt mai orientate pe obiecte și mai funcționale. În anii 1980 limbajele procedurale au fost eliminate și limbajele funcționale și orientate pe obiect au devenit populare.

Destul de interesant, la începutul creșterii în popularitate, limbajele combinau stilurile funcțional cu obiect orientat. La început limbajul ideal pentru un stil obiect-orientat era Smalltalk. Smalltalk 80 permitea expresii lambda și librăria de colecții era de natură funcțională, existau operații echivalente lui map, reduce și filter numai că sub alte nume.

Limbajul preferat pentru un stil funcțional era Common LISP. Acesta avea un sistem pentru orientarea pe obiecte numit CLOS (Common LISP Object System). Deci, în anii 1980, nu exista o cale pură în care să aplici aceste paradigme în producție.

În timpul anilor 1990, programarea s-a schimbat. Programarea orientată pe obiecte a devenit stilul dominant pentru a rezolva probleme business. Limbajele C++ și Java au crescut în popularitate. În anul 2001, conferința JavaOne a avut 28.000 de participanți.

În ultimii ani, trendul s-a schimbat din nou. Limbajele populare de programare ies din tiparele pure de funcționale sau obiect-orientate și migrează spre o abordare hibridă. Atît C++ cât și Java 8 au adăugat expresii lambda și facilitează folosirea elementelor funcționale alături de cele obiect-orientat.

Noile limbaje apărute suportă mai multe paradigme din start. F# este un limbaj puternic ce facilitează un stil funcțional și este la fel de bun ca C# în termeni de elemente obiect-orientate. Limbaje precum Ruby, Python și Groovy pot aborda stilul funcțional și obiect-orientat.

Viitorul programării este hibrid, constă în alegerea celor mai bune caracteristici și idei din ambele stiluri pentru a rezolva o problemă.

**4 Bibliografie**

1. [Functional programming in JavaScript – Dan Mantyla](http://shop.oreilly.com/product/0636920028857.do)
2. [Functional JavaScript – Michael Fogus](http://www.amazon.com/Functional-Programming-JavaScript-Dan-Mantyla/dp/1784398225)
3. [Why Functional Programming Matters – John Hughes](http://worrydream.com/refs/Hughes-WhyFunctionalProgrammingMatters.pdf)
4. [A practical introduction to functional programming – Mary Rose Cook](http://maryrosecook.com/blog/post/a-practical-introduction-to-functional-programming)
5. [Functional Programming – Scott Sauyet](http://scott.sauyet.com/Javascript/Talk/2014/01/FuncProgTalk/#slide-0)
6. [Functional Programming in Python – David Mertz](http://www.oreilly.com/programming/free/files/functional-programming-python.pdf)
7. [Object-Oriented vs. Functional Programming – Richard Warburton](http://www.oreilly.com/programming/free/files/object-oriented-vs-functional-programming.pdf)