## Cuprins

# 1. Introducere

## 1.1. Argument : contextul general al programării aplicaţiilor în JavaScript

JavaScript este un limbaj controversat; în prima etapă a exploziei Internetului și-a atras disprețul specialiștilor prin utilizarea sa neadecvată și ca limbaj complementar : realizarea unor artificii inutile pentru interfața grafică – desenarea fulgilor de zăpadă, împodobirea cursorului sau a barei de status a navigatorului etc. –, modul de scriere a codului – adesea un șir lung de instrucțiuni, pe o singură linie în documentele HTML, inclus în proprietăți de tipul *onclick*, *onload*. Un alt motiv pentru antipatia față de JavaScript a fost lipsa standardizării. Ocupați cu implementarea constantă de noi funcționalități, creatorii limbajului (Microsoft și Netscape) au neglijat dezvoltarea unor unelte esențiale pentru programarea în JavaScript și au creat incompatibilități majore între versiunile oferite.

Astăzi, literatura de specialitate consideră că JavaScript este un limbaj matur, puternic, complex, flexibil și unic din anumite puncte de vedere, iar afirmația este susținută de dezvoltarea aplicațiilor web cu funționalitate vastă precum Gmail, Facebook, Yahoo!Mail, YouTube, hărți interactive (cum ar fi Google Maps sau Bing Maps) sau de aplicații pentru furnizarea datelor în timp real, dar și de posibilitățile tot mai variate de întrebuințare a tehnologiei în medii externe navigatorului (Rhino, Windows Scripting Host, Flash ActionScript și Flex – bazate și ele pe standardul ECMAScript sau extensii pentru navigator, dar și pentru programe consacrate cum ar fi Photoshop).

## 1.2. Contextul specific aplicaţiilor client în paradigma *Web*

> *JavaScript is the language of the Web.*
> [Stoyan Stefanov, *JavaScript Design Patterns*, p. 1]

Evoluția interpretoarelor de JS, efortul de standardizare prin ECMAScript apreciat deoportrivă de dezvoltatorii aplicațiilor Web, cât și de dezvoltatorii navigatoarelor și avantajul rulării fără nevoia de plug-inuri externe browserului, coroborat cu diversificarea recentă a terminalelor de acces pe Internet – apariția smartphoneurilor și a tabletelor – au dus la redescoperirea, sau, mai degrabă, utilizarea JavaScript într-un mod nou și productiv pentru a crea arhitecturi scalabile.

Paradigma Web actuală este formată din trei părți delimitate clar : conținut – în formă HTML – , prezentare – specificată prin CSS – și comportament. JavaScript oferă fundamentul pentru reprezentarea celui de-al treilea pilon al paradigmei : comportamentul. Este evident rangul primordial al JavaScript pentru dezvoltarea aplicațiilor și inovarea Webului. În acest context, calitatea codului JavaScript este esențială și trebuie tratată cu atenția și rigoarea cuvenite rolului pe care îl joacă acum acest limbaj.

Noul model de programare Web, bazat pe paradigma prezentată mai sus, a dus la apariția unui număr seminificativ de aplicații care utilizează intens JavaScript, oferind un set de funcționalități până nu demult specifice doar programelor desktop și adesea inimaginabile pentru mediul navigatorului. Din păcate, atunci când vorbim despre JavaScript, modelele de design sau

de arhitectură consacrate – prezente în programare dinaintea JS, aplicate cu succes în alte limbaje – și practicile recomandate sunt uitate și ignorate, rezultând, după cum precizează Alex MacCaw: *un amestec murdar într-o ciorbă de HTML și JavaScript* [Alex MacCaw, *JavaScript Web Applications*, p. 2].

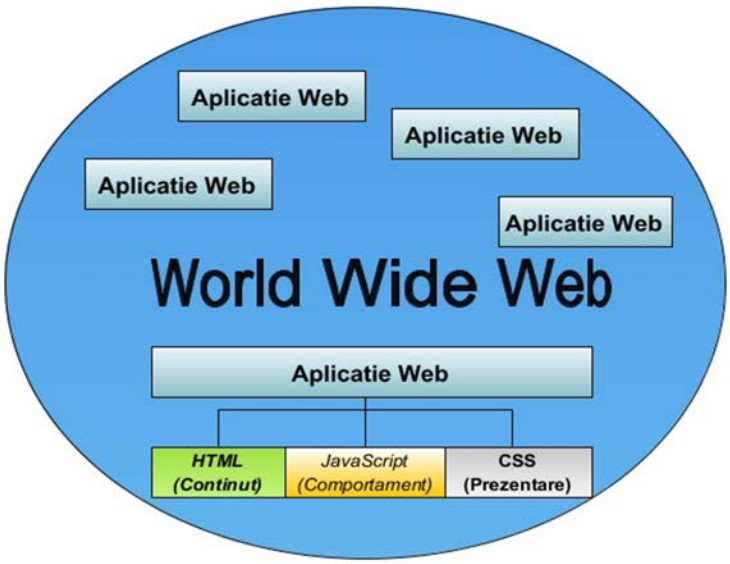



**Fig. 1.1 Paradigma Web**

Lucrarea de față susține tezele exprimate mai sus. În cele ce urmează, vom demonstra puterea, flexibilitatea, complexitatea, unicitatea acestui limbaj și vom prezenta metodele de aplicare a conceptelor clasice de programare orientată pe obiecte în JavaScript prin elaborarea unui framework aplicabil general în programarea aplicațiilor Web.

## 1.3. Domeniu de aplicabilitate

După cum am specificat mai sus, vom aborda în cele ce urmează subiectul şi problemele realizării aplicaţiilor *Web* client utilizând limbajul de programare JavaScript şi tehnologiile disponibile. Ne propunem tratarea cu o atenţie sporită a principiilor de programare obiectuală cunoscute, demonstrarea necesităţii de utilizare a acestora şi exemplificarea soluţiilor de implementare în mediul JavaScript. În acest sens, vom porni de la prezentarea sumară a limbajului cu accent pe elemente caracteristice. Pe baza conceptelor specifice JavaScript studiate vom aprofunda şi ilustra problemele puse de aplicarea conceptelor programării obiectuale şi de structure a codului conform unor modele arhitecturale.

Pornind de la premisa enunţată de E. Gamma că *ceea ce pentru anumite limbaje de programare reprezintă blocuri de bază, în altele poate fi un şablon de programare* corelată cu proprietăţile de bază ale limbajului JavaScript, suntem de părere că abordarea conceptelor fundamentale ale programării orientate pe obiecte – moştenire, încapsulare, polimorfism şi abstractizare – este esenţială în atingerea obiectivelor acestei lucrări.

Aspectele teoretice enunţate vor fi concretizate prin realizarea unui framework universal pentru realizarea aplicaţiilor client în paradigma *Web*. Productivitatea acestuia va fi demonstrată

printr-o aplicaţie practică ale cărei cerinţe funcţionale pot reprezenta un etalon raportat la alte frameworkuri asemănătoare.

## 1.4. Aspecte teoretice şi de implementare neabordate

Conform contextului general şi domeniului de aplicabilitate al lucrării prezentate anterior, ne propunem aprofundarea problemelor teoretice şi practice care ţin de aplicarea principiilor programării obiectuale în JavaScript.

Tema abordată în această lucrare este dezbătută amplu atât general, cât şi specific tehnologiilor în literatura de specialitate. În acest sens, menţionăm că, din punct de vedere teoretic, ne propunem abordarea unui set de subiecte pe care le considerăm fundamentale. În acest sens, având în vedere amploarea subiectului, vom trata şabloanele de design în mod selectiv în scopul demonstrării aplicabilităţii acestora în JavaScript. *Singleton* şi *Observer* fac parte din categoria modelelor de design fundamentale propuse iniţial de E. Gamma şi au un spectru larg de utilizare practică. Prin urmare, considerăm că limitarea prezentării bibliografiei consultate la cele două modele de design menţionate nu afectează scopul şi obiectivele lucrării.

Concretizarea practică a soluţiilor de programare obiectuală a căror aplicabilitate este discutată prin prisma JavaScript este de natura unei demonstraţii a conceptului şi trebuie tratată ca atare. În acest sens, implementarea nu va aborda teme adiacente subiectului principal precum intrumentele pentru concatenarea, minifierea şi optimizarea codului.

Una dintre diferenţele majore între implementarea propusă şi produsele existente asemănătoare, este tendinţa de oferi soluţii proprii pentru gama completă a funcţionalităţii oferite. Pentru moment, reţinem că nu vom aborda reimplementarea unor funcţionalităţi deja existente în comunitatea *open-source*. Notăm că decizia de integrare a unor soluţii existente va fi motivată şi dezbătută pe larg în capitolul următor.

În concluzie, suntem de părere că limitările enumerate şi impuse sunt în parametrii de nivel ai lucrării de faţă şi constituie un factor de echilibru pentru atingerea obiectivelor propuse.

## 1.5. Structura lucrării

În continuare propunem o trecere în revistă sumară a structurii acestei lucrări.

Primul capitol prezintă situaţia generală a programării utilizând limbajul JavaScript, a paradigmei *Web* evidenţiind probleme generale existente pe care le vom trata. Capitolul al doilea fixează obiectivele principale şi specifice ale acestui proiect. În capitolul al treilea – Studiu bibliografic – sunt prezentate elementele caracteristice importante ale limbajului, standardele actuale în vigoare şi sunt discutate problemele specifice domeniului în raport cu programarea obiectuală. Nu în ultimul rând, studiul bibliografic efectuat conţine o prezentare a lucrărilor de specialitate consultate. Capitolul patru tratează amplu conceptele teoretice fundamentale necesare programării orientate pe obiecte în JavaScript. În capitolul cinci sunt descrise detaliat cerinţele impuse frameworkului dezvoltat şi aplicaţiei demonstrative. Detaliile de implementare practică ale proiectului propus sunt înfăţişate în capitolul şase. Metodele de validare şi testare urmate în dezvoltarea practică sunt expuse în capitolul şapte. În capitolul opt se regăseşte setul complet de instrucţiuni cu privire la instalarea şi utilizarea acestui proiect. Capitolul nouă încheie lucrarea şi este dedicat concluziilor generale şi posibilităţilor ulterioare de dezvoltare.

## 2. Obiectivele lucrării

Creşterea rapidă a ponderii şi complexităţii aplicaţiilor Web exprimată prin tendinţa generală de reducere a responsabilităţii aplicaţiilor *server* în ceea ce priveşte procesarea la rolul de sursă de date şi securitate rezultă în nevoia imperativă de asigurare a calităţii codului pe partea de client. În teoria procesului de dezvoltare software, această stare de fapt se traduce prin împărţirea disproporţionată a nivelului de *bussiness* dintr-o aplicaţie între client şi server. Notăm că pe de o parte direcţia generală cu privire la acest subiect este realizarea operaţiilor de *bussiness* în totalitate sau în măsură cât mai mare posibil prin navigator. Pe de altă parte, majoritatea tehnologiilor de realizare a aplicaţiilor server includ instrumente sofisticate precum Hibernate în Java şi Entity Framework în C# de generare a codului din nivelele de prelucrare şi reprezentare a datelor, iar acestea nu îşi găsesc corespondent în tehnologiile disponibile clienţilor Web. Rezultatul discrepanţei dintre mediul server şi cel client este reducerea substanţială a eficienţei procesului de dezvoltare şi scăderea considerabilă a calităţii.

JavaScript este singurul limbaj de programare acceptat în mod general pe Web. Un aspect important pe care îl vom considera în stabilirea obiectivelor acestei lucrări este faptul că atât industria, cât şi cercetarea ignoră în general caracteristicile specifice ale limbajului asemănându-l eronat cu cele consacrate pe partea de server cum ar fi C# sau Java.

Menţionăm că obiectivele lucrării de faţă au fost stabilite conform contextului programării aplicaţiilor Web utilizând JavaScript, domeniului de aplicabilitate delimitat şi nu în ultimul rând în concordanţă cu aspectele neabordate. Este de reţinut că obiectivele fixate au fost identificate în acord cu criteriile SMART pentru a fi specifice, măsurabile, accesibile, relevante şi limitate temporal.

Din punct de vedere teoretic, considerăm esenţială cunoaşterea limbajului JavaScript şi înţelegerea trăsăturilor diferite ale acestuia. În acest sens, **aprofundarea cunştiinţelor teoretice de JavaScript** reprezintă un obiectiv al lucrării de faţă.

Corespunzător problemelor enunţate în introducere, **studiul şi exemplificarea metodelor de aplicare ale conceptelor clasice de programare obiectuală** constituie scopul principal al lucrării propuse.

Pentru realizarea practică a lucrării ne propunem aplicarea conceptelor teoretice expuse în **implementarea unui framework *open-source* JavaScript cu domeniu general de utilizare pentru dezvoltarea aplicaţiilor Web client integrate în arhitecturi de tip SOA**.

În vederea aprofundării cunoştiinţelor de JavaScript vom **prezenta sumar aspectele generale de limbaj precum şi standardele existente în vigoare**. Considerăm că, pentru îndeplinirea obiectivului de aprofundare a cunoştiinţelor teoretice, dezvoltarea subiectului **reprezentării datelor şi nuanţarea diferenţelor – cum ar fi închiderile sau paradigma prototipică – faţă de alte limbaje**, sunt esenţiale. Nu în ultimul rând, este importantă familiarizarea cu mediul gazdă al navigatorului.

Cu privire la aplicarea paradigmei obiectuale, ne propunem **expunerea teoretică şi practică a soluţiilor şi şabloanelor documentate în bibliografia consultată, disponibile în JavaScript**. În acest sens, vom aborda cu atenţie deosebită moştenirea, încapsularea, polimorfismul şi abstractizarea atât general, cât şi în contextul JavaScript. De asemenea, intenţionăm să **exemplificăm utilitatea şi modul de implementare al şabloanelor de design *Observer* şi *Singleton***.

După cum am precizat, partea practică a acestei lucrări vizează un framework JavaScript open-source care să faciliteze folosirea paradigmei obiectuală în dezvoltarea aplicaţiilor Web. Reţinem că multitudinea de soluţii şi abordări ale comunităţii open-source este în acelaşi timp

punctul forte, dar şi un dezavantaj major. Susţinem această afirmaţie prin faptul că multe dintre proiectele open-source sunt granulare şi rezolvă probleme specifice şi bine delimitate. Pe de altă parte, această caracteristică poate asigura performanţe superioare. În acest sens, **frameworkul propus în cele ce urmează va oferi o interfaţă de programare (API) transparentă în raport cu soluţiile utilizate**. Considerăm că **identificarea şi integrarea soluţiilor existente şi validate de către comunitate – cum ar fi jQuery sau Backbone – în frameworkul propus**, este un obiectiv de importanţă majoră pentru partea practică a acestei lucrări. Menţionăm că principiile şi funcţionalităţile care stau la baza frameworkului propus includ **manipularea arborelui DOM, implementarea unui nivel de reprezentare a datelor prin modele şi colecţii şi utilizarea şabloanelor arhitecturale şi de design**.

Utilitatea frameworkului propus va fi conturată în mare măsură de setul de componente abstracte destinate dezvoltării aplicaţiilor Web şi va fi demonstrată practic.

În concluzie, considerăm utilă însumarea sintetică a obiectivelor pe care şi le asumă această lucrare, împărţite după domeniul de aplicare : teoretic sau practic. Pentru ilustrarea conceptelor teoretice ne propunem următoarele :

1. Aprofundarea cunştinţelor de JavaScript prin prezentarea aspectelor generale de limbaj, a standardelor existente în vigoare
2. Prezentarea elementelor caracteristice de reprezentare a datelor în JavaScript
3. Studiul şi exemplificarea metodelor de aplicare ale conceptelor clasice de programare obiectuală

Din punctul de vedere al implementării practice, vom aplica rezultate teoretice după cum urmează :

1. Implementarea unui framework *open-source* JavaScript cu domeniu general de utilizare pentru dezvoltarea aplicaţiilor Web client integrate în arhitecturi de tip SOA
2. Realizarea unei interfeţe de programare transparentă în raport cu soluţiile utilizate la în nivelele superioare sau inferioare
3. Identificarea şi integrarea soluţiilor existente
4. Implementarea unui nivel de reprezentare a datelor prin modele şi colecţii
5. Utilizarea şi abstractizarea şabloanelor arhitecturale şi de design în implementarea frameworkului.
6. Implementarea unui set de componente vizuale reutilizabile, extensibile şi scalabile în dezvoltarea aplicaţiilor Web client.
7. Validarea frameworkului propus printr-o aplicaţie demonstrativă

# 3. Studiu bibliografic : Prezentare generală a limbajului JavaScript

JavaScript este un limbaj dinamic, slab tipizat, bazat pe prototipuri. Funcțiile sunt tratate ca obiecte de primă clasă. Paradigmele de programare suportate sunt multiple : orientată pe obiecte, imperativă și funcțională.

## 3.1. Elemente caracteristice

Între elementele care definesc unicitatea limbajului se numără cele enumerate mai sus, dar și flexibilitatea ieșită din comun. În funcție de cerințele aplicației și de cunoștiințele programatorului, JavaScript permite o abordare de o complexitate variabilă a problemei. Rob Harness și Dustin Diaz exemplifică această caracteristică considerând cazul programării unei animații după cum urmează :

| Abordare | Funcțională | Obiectuală (1) | Obiectuală (2) |
|---|---|---|---|
| Exemplu de cod | `function startAnimation() {...} function stopAnimation() {...}` | `var Anim = function() {...}; Anim.prototype.start = function() {...}; Anim.prototype.stop = function() {...}; /* Utilizare. */ var myAnim = new Anim(); myAnim.start(); ... myAnim.stop();` | `var Anim = function() {...}; Anim.prototype = { start: function() {...}, stop: function() {...} };` |
| + | ✓ Simplitate<br>✓ Complexitate scăzută<br>✓ Nu necesită cunoștiințe OO avansate | ✓ Abordare OO<br>✓ Permite reprezentarea stării obiectelor | ✓ Reluarea exemplului anterior cu sintaxa modificată rezultând într-un stil mai apropiat de cel al programării OO clasice<br>✓ Încapsularea codului într-o singură declarație |
| - | Lipsa capacității de reprezentare a stării obiectelor | | |

**Tabelul 3.1 Paradigme posibile în JavaScript**

| Abordare | **Obiectuală (3)** | **Obiectuală (4)** |
|---|---|---|
| Exemplu de cod | `function.prototype.method = function(name, fn) { this.prototype[name] = fn; }; var Anim = function() {...}; Anim.method('start', function() {...}); Anim.method('stop', function() {...});` | `function.prototype.method = function(name, fn) { this.prototype[name] = fn; return this; }; var Anim = function() {...}; Anim.method('start', function() {...}). method('stop', function() {...});` |
| + | | ✓ Permite înlănțuirea apelurilor |
| - | Nu permite înlănțuirea apelurilor | |

**Tabelul 3.2 Paradigme posibilie în JavaScript (continuare)**

După cum am precizat, JavaScript face parte din familia limbajelor slab tipizate. Considerăm că aceasta este o altă caracteristică definitorie a limbajului.

O altă proprietate importantă pentru a înțelege comportamentul JavaScript este tratarea funcțiilor ca obiecte de primă clasă. Practic, această abordare înseamnă că orice variabilă care nu reprezintă un tip de date primitiv, este un obiect – chiar și o funcție. Vom dezvolta acest subiect în secțiunea dedicată funcțiilor.

Nu în ultimul rând, trebuie precizată mutabilitatea obiectelor. În acest sens, o clasă sau un obiect pot fi schimbate după ce au fost definite sau instanțiate. În conjuncție cu tratarea funcțiilor ca obiecte de primă clasă, proprietatea de mutabilitate face posibilă utilizarea unor tehnici de programare unice după cum precizează Dustin Diaz și Rob Harness în *JavaScript Design Patterns*. Una dintre utilizările posibile este asignarea de atribute unor funcții, după urmează în exemplul de mai jos :

```
function globalCounter() {
    globalCounter.count += 1;
}
/* Utilizare */
globalCounter.count = 0;
globalCounter(); /* Apelul funcției rezultă incrementarea contorului
globalCounter.count */
```

Din punctul de vedere al execuției programelor, acestea vor rula întotdeauna într-un mediu gazdă numit motor JavaScript. Majoritatea motoarelor JavaScript sunt implementate în navigatoare web, însă a fost demonstrată posibilitatea utilizării lor și în alte medii. Atunci când vorbim despre mediul gazdă în care rulează programele JavaScript, trebuie precizat obiectul global. În cazul în care motorul JavaScript este găzduit de un navigator, obiectul global este însăși fereastra acestuia (obiectul *window*). În cazul *Global Host Object* este de remarcat că orice variabilă globală sau funcție definită în scop global (nivel bloc global) – vom defini mai jos conceptul de scop – devine atribut al acestui obiect.

### 3.1.1. Standardizare

Nucleul JavaScript – tipurile de date primite, obiectele native și funcțiile predefinite – se bazează pe standardul ECMA-262 și ISO/IEC 16262. Prima versiune a standardului a fost publicată în 1997. O versiune importantă a standardului este versiunea 3, publicată în 1999. Noutățile cele mai importante introduse de versiunea a treia au fost : suportul pentru expresiile regulate, manipularea îmbunătățită a șirurilor de caractere, instrucțiuni de control, tratarea blocurilor try/catch, definiția mai strictă a erorilor și formatarea ieșirilor numerice.

Conform http://en.wikipedia.org/wiki/ECMAScript versiunea actuală a standardului implementat în majoritatea navigatoarelor web importante este ECMA-262 ediția a 5a. Considerăm că cea mai importantă schimbare adusă de această variantă a standardului este modul strict – *acesta reprezintă un subset cu rolul de a asigura verificarea mai aprofundată a erorilor* .

[http://en.wikipedia.org/wiki/ECMAScript].

## 3.2. Reprezentarea datelor

### 3.2.1. Noţiunea de clasă în JavaScript

Având în vedere caracterul bazat pe prototipuri, rezumându-ne strict la modul de reprezentare a datelor, observăm că în JavaScript nu există noțiunea de clasă. De altfel, pentru crearea unui obiect, se poate utiliza literalul obiect sau o funcție constructor precedată de operatorul *new*. În cazul utilizării variantei a doua, este de reținut faptul că, în momentul definirii

funcţiei constructor, aceasta devine un obiect de tip *Function* referenţiat de o proprietate cu acelaşi nume a obiectului global, după cum vom arată în secţiunea dedicată acestui subiect. Afirmaţia este susţinută de către S. Stefanov[1] şi de definiţia limbajelor bazate pe prototipuri :
***Prototype-based programming** is a style of object-oriented programming in which **classes are not present**, and behavior reuse (known as inheritance in class-based languages) is performed via a process of cloning existing objects that serve as prototypes.*[2]

La nivel conceptual general, putem defini noţiunea de clasă astfel :
*O clasă defineşte structura şi comportamentul unui obiect, fiind un tipar al tuturor instanţelor de un anumit fel. O instanţă obiect este creată explicit pe baza unei clase şi este considerată un exemplu al acesteia.*[3]

Acceptând descrierea de mai sus ca fiind general valabilă şi având în vedere faptul că definirea unei funcţii în JavaScript poate îndeplini toate cerinţele necesare exprimate anterior – după cum vom arăta în cele ce urmează, în momentul interpretării declaraţiei se creează un spaţiu propriu de variabile; codul conţine specificarea comportamentului şi a structurii unui obiect; orice instanţă reprezintă un exemplu al abstractizării prin care a fost creată, aflată într-o anumită stare – considerăm că pentru lucrarea de faţă este util a face diferenţa între noţiunea de clasă şi cea de instanţă obiect. În continuare ne vom raporta la acest concept prin definiţia citată mai sus considerând că specificarea unei funcţii în scopul descrierii structurii şi comportamentului unui obiect reprezintă o clasă. Este de reţinut că, în aceste condiţii, specificarea unei clase reprezintă la rândul ei un obiect pentru motorul JavaScript, o abstractizare pe care programatorul o face pentru a oferi interpretorului informaţiile necesare despre structura şi comportamentul instanţelor într-o stare dată, aşa cum în alte limbaje de programare acest proces poate fi considerat o intrare pentru compilator.

### 3.2.2. Tipuri de date primitive

JavaScript defineşte următoarele tipuri de date primitive : *number, string, boolean, undefined* şi *null*. Observăm că, spre deosebire de alte limbaje de programare cu tipizare puternică, în JavaScript se utilizează un singur tip pentru reprezentarea datelor numerice (specific familiei limbajelor slab tipizate). Astfel, o variabilă de tip number poate să conţină atât valori întregi, cât şi în virgulă mobilă (float). Asemeni altor limbaje de programare, pentru tipurile *number, string, boolean* există obiecte wrapper.

Tipul *undefined*, de asemenea specific JavaScript, este atribuit unei variabile în cazul în care aceasta nu a fost definită, sau, spre deosebire de *null*, a fost definită, dar nu a fost iniţializată încă.

Orice variabilă care nu reprezintă unul dintre tipurile de date primitive este tratată în JavaScript ca obiect.

În ceea ce priveşte transmiterea parametrilor, tipurile de date primitive sunt transmise prin valoare, iar obiectele prin referinţă.

Având în vedere caracterul slab tipizat, trebuie menţionat faptul că o variabilă de tip primitiv (*number, string* sau *boolean*) îşi poate schimba tipul. Un exemplu ar fi conversia unei variabile *string* într-una *number* :

```
var stringToNumber = "100";
stringToNumber = stringToNumber * 1;
```

[1] (Stefanov, JavaScript Patterns 2010, 4)
[2] (Wikipedia)
[3] (Wikipedia)

### 3.2.3. Obiecte și tipuri de date predefinite

Tipurile de date predefinite în JavaScript sunt organizate ierarhic după modelul Java. Astfel, tipul de date rădăcină al limbajului este *Object*. În plus, limbajul conține următoarele obiecte derivate din tipul object : *Math, Date, RegExp – grupate ca obiecte utilitare -, Array, Function, Number, Boolean, String* – obiecte de tip wrapper de date. Tipul *Object* intră de asemenea în categoria obiectelor de tip wrapper de date.

Toate obiectele JavaScript conțin atributul *constructor*. În momentul instanțierii, atributul este inițializat cu o referință spre funcția constructor utilizată pentru crearea obiectului în cauză.

#### 3.2.3.1. *Prototipul unui obiect : atributul prototype*

Membrul *prototype* reprezintă un atribut special al tuturor obiectelor JavaScript. Acesta este prototipul de la care porneşte crearea tuturor obiectelor. Dacă nu este specificat altfel, prototipul este iniţializat cu o instanţă *Object*. În ceea ce priveşte funcţiile, acestea reprezintă în JavaScript la rândul lor un obiect. La crearea oricărei funcţii atributul prototype este iniţializat cu o referinţă spre un obiect similar cu cel care rezultă în urma apelului constructorului *Object()*. Diferența constă în faptul că atributul *constructor* al obiectului *prototype* conține o referință spre funcția creată şi nu spre constructorul obiectului predefinit de tip *Object*.

De menţionat este că utilizarea acestei proprietăţi este una dintre metodele prin care este implementat conceptul de moştenire în JavaScript – subiect pe care îl vom detalia mai jos.

În legătură cu **mutabilitatea obiectelor** în JavaScript, atributul are un rol esenţial deoarece orice modificare a atributului *prototype* al unui obiect de tip *Function* se va reflecta în toate instanţele clasei respective, indiferent de momentul creării lor (înainte sau după schimbare). Pentru a ilustra acest comportament, considerăm necesar următorul exemplu :

```
/*Definirea clasei Student*/
function Student() {
    this.nume = "Mihai";
}
/*Instantierea unui obiect de tip Student*/
var mihai = new Student();
/*Adaugam proprietatea "grupa" prototipului obiectelor de tip Student*/
Student.prototype.grupa = 1;
console.log(mihai.grupa);
=> 1
/* Observam ca instanta Student "mihai" a fost actualizata cu proprietatea
"grupa" si cu valoarea definita anterior */
/* Adaugam metodata "getNumeSiGrupa" */
Student.prototype.getNumeSiGrupa = function() {
    return this.nume + " din grupa " + this.grupa;
}
console.log(mihai.getNumeSiGrupa()); => "Mihai din grupa 1"
/* Adaugam metoda setNume */
Student.prototype.setNunme = function(nume) {
    this.nume = nume;
}
var ion = new Student();
/* Initial atributul nume are valoarea "Mihai" */
altStudent.setNume("Ion");
/* Dupa invocarea metodei setNume atributul nume al instantei ion isi
modifica valoarea */
console.log(altStudent.getNumeSiGrupa()); => "Ion din grupa 1"
```

```
/* Nu si atributul nume al instantei mihai */
console.log(unStudent.getNumeSiGrupa()); => "Mihai din grupa 1"
```

#### 3.2.3.2. *Literali obiect şi literali matrice*

Pentru instanţierea obiectelor de tip *Array* sau a celor definite de utilizator este suportată varianta întrebuinţată în majoritatea limbajelor de programare : new Array() sau new NumeClasă(). O altă metodă, specifică JavaScript este folosirea a ceea ce numim *literali obiect* pentru instanţierea obiectelor, respectiv *literali matrice* pentru instanţierea matricilor. A doua variantă este utilă mai ales pentru crearea unui obiect de tip *Array* gol : `var newArray = [];`. Analog, pentru instanţierea unui obiect, formatul este : `{[atribut : valoare[,atribut : valoare]]}.`

### 3.2.4. Funcții

Din definiţia oferită de Wikipedia reţinem că funcţiile unui limbaj de programare sunt considerate de primă clasă dacă ele reprezintă cetăţeni de primă clasă (*first-class citizens*). Astfel, limbajul suportă returnarea unei funcţii dintr-o funcţie, transmiterea acestora ca parametru unei alte funcţii, cât şi referenţierea unei funcţii printr-o variabială sau o structură de date. [http://en.wikipedia.org/wiki/First-class_function]

Unii teoreticieni consideră că pentru ca funcţiile unui limbaj să fie cetăţeni de primă clasă este necesar ca acesta să includă şi suportul pentru conceptul de funcţie anonimă.

Funcţiile JavaScript sunt obiecte de primă clasă. Tipul acestora este *Function*. De asemenea, limbajul suportă specificarea funcţiilor anonime. Acestea nu au nume, sunt create utilizând sintaxa `function() {...}` şi pot fi referenţiate printr-o variabilă. Utilitatea conceptului se evidenţiază în utilizarea tehnicii *callback* sau a funcţiilor imediate şi, după cum vom ilustra mai jos, în crearea închiderilor (*closures*).

Tehnica apelurilor *callback* nu este un concept nou în programare; în Java există o abordare similară sub forma claselor anonime. În JavaScript, această tehnică este posibilă datorită proprietăţilor pe care limbajul le conferă funcţiilor. Astfel, utilizarea apelurilor *callback* se realizează prin transmiterea unei funcţii anonime ca parametru unei alte funcţii.

Un comportament interesant al funcţiilor, pe care îl putem deduce din cele precizate mai sus, este abilitatea acestora de a se redefini sau de a fi redefinite, după cum se poate observa în exemplul următor :

```
function a() {
    console.log("Fac ceva!");
    a = function() {
        console.log("Fac altceva!");
    }
}
a(); => "Fac ceva!"
a(); => "Fac altceva!"
```

### 3.2.5. Scopul variabilelor

Conceptul de scop reprezintă mediul în care este executat un bloc de program (o funcţie în cazul JavaScript).

Spre deosebire de alte limbaje de programare, în JavaScript scopul este definit la nivelul unei funcţii. O variabilă definită în interiorul unei funcţii nu este vizibilă în afara acesteia, dar este vizibilă oricărei funcţii definite în interiorul celei curente. Acest comportament este numit lanţ de scop, se aplică recursiv şi poate fi exemplificat astfel :

```
function A() {
    var visibleToB = 1;
    function B() {
      /* variabila visibleToB este disponibila */
   }
}
function C() { /* variabila visibleToB nu este disponibila */ }
```

Aplicarea recursivă a lanţului de scop înseamnă că, în cazul în care în exemplul de mai sus, definim o altă funcţie D în interiorul lui B, D are acces la scopul funcţiei B, al funcţiei A, cât şi la scopul global. Lanţul poate continua la infinit în limita resurselor sistemului. Generalizând, o funcţie n are acces la scopul propriu, cât şi la scopurile tuturor părinţilor.

### 3.2.6. Închideri (closures) și spaţii de nume (*namespaces*)

În JavaScript există un scop global reprezentat adesea de obiectul global precizat mai sus. În *Object Orieted JavaScript*, Stoyan Stefanov ne îndeamnă să privim acest obiect global ca fiind „universul care le conţine pe toate”[1]. Având în vedere cele precizate despre funcţii şi scop în JS, să considerăm următoarea figură pentru a ilustra conceptul de închidere :

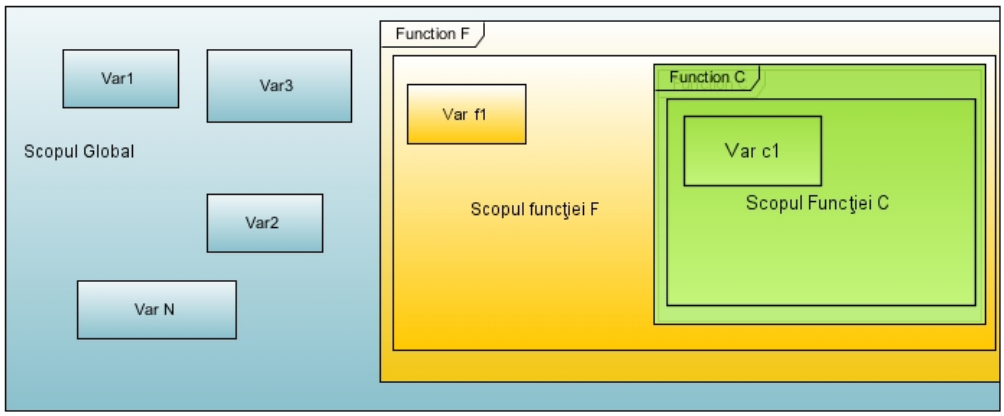



**Fig. 3.1 Ilustrarea conceptului de închidere**

După cum menţionează şi Stefanov[2], variabilele Var1, Var2 ... Var N şi funcţia Function F se află în scopul global. În acest sens, funcţia F are access la scopul global şi la scopul propriu (scopul funcţiei F în figură). Funcţia C se află în scopul lui F. Astfel, considerând caracteristicile funcţiilor JavaScript, deducem că aceasta are acces la variabilele din scopul global şi la scopul lui F, însă relaţia nu este valabilă şi invers. Conform lui Stefanov[3], dacă ne aflăm în punctul Var 1, Var f1 este invizibilă fiind în scopul F. În aceste condiţii, pentru a crea o închidere (*closure*) spaţiul (scopul) funcţiei C este externalizat, scos din F şi adăugat spaţiului global, după cum putem observa în figura următoare.

[1] (Stefanov, Object-Orieted JavaScript 2008, 82)
[2] Ibid. (83)
[3] (Stefanov, Object-Orieted JavaScript 2008, 83)

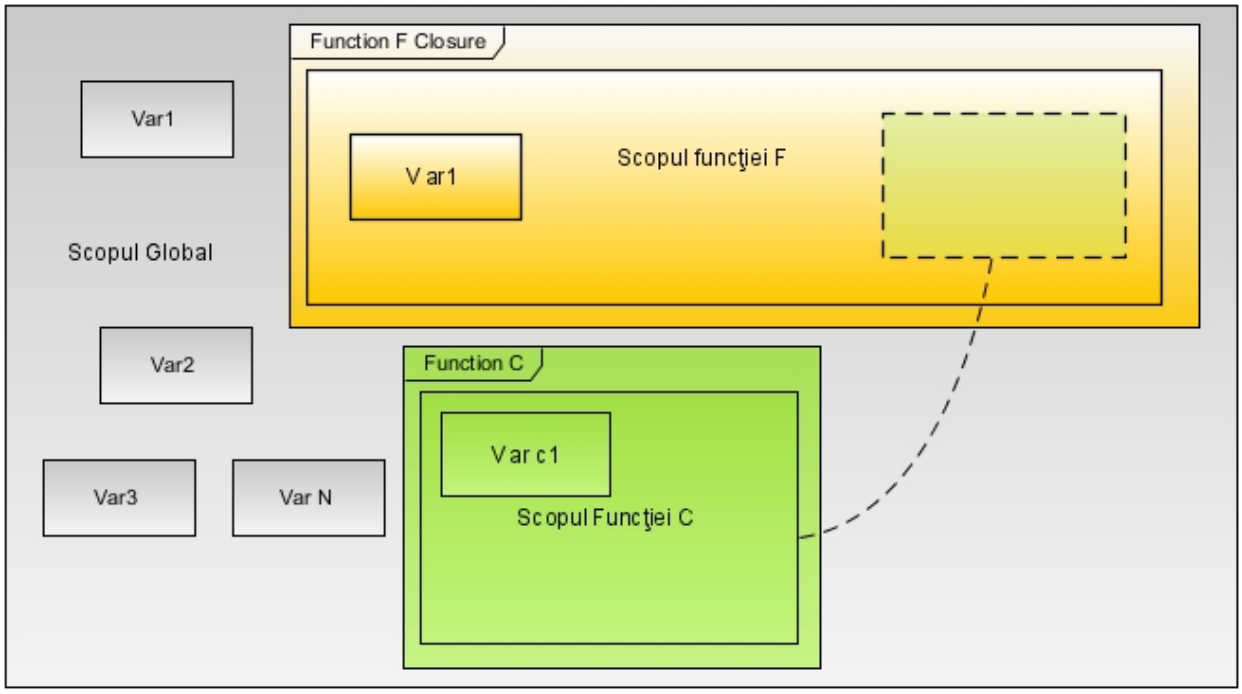



**Fig. 3.2 - Exemplu de închidere**

După cum putem observa în figura de mai sus, scopul funcţiei C şi oricare dintre Var1, Var2, Var3 sau VarN se află în acelaşi spaţiu şi anume cel global. Totuşi, VarF1 din spaţiul F este vizibilă doar funcţiei C, nu şi Var1, Var2, Var3, VarN care pot să reprezinte variabile, obiecte sau funcţii. Acest comportament se explică prin faptul că un obiect JavaScript de tip *Function* reţine mediul (scopul) în care a fost definit. Lanţul de scop descris mai sus este frânt în acest caz. Mai exact, acesta poate fi frânt prin globalizarea funcţiei C, adică expunerea acesteia în contextul global returnând-o din F, sau prin omiterea cuvântului cheie *var*. Exemplificăm acest comportament prin următoarele :

```
/* Exemplu de returnare a functiei C in spatiul global */
function F() {
    var Var1 = "Valoarea lui Var1";
    return function C() {
        console.log("Am acces la Var1 si valoarea acestuia este : ");
        return Var1;
    }
}
/* Exemplu de omitere a cuvantului cheie var */
function F1() {
    var Var1 = "Valoarea lui Var1";
    apelC1 = function() {
        console.log("Am acces la Var1 si valoarea este : ");
        return Var1;
    }
}
/* Exemplu de apel al functiei C din spatiul global.*/
var apelC = F();
apelC();
```

```
/* Exemplu de apel al functiei C1 din spatiul global. */
F1();
apelC1();
>>> Am acces la Var1 si valoarea acestuia este : "Valoarea lui Var1"
```

Conceptul de namespace (spaţiu de nume) este similar cu modul de organizare a codului în limbaje precum Java sau C#. În mod nativ, sintaxa JavaScript nu oferă o implementare specială pentru spaţii de nume. Totuşi, pentru organizarea codului conform conceptului, în JavaScript, putem utiliza un obiect global gol – şi, de preferinţă **unul singur**[1], după cum precizează S. Stefanov – iniţializat printr-un literal obiect şi care va conţine toată funcţionalitatea proiectului, adică funcţii, clase, interfeţe, eveniment, variabile statice etc.

Abordarea simplistă prezentată mai sus – crearea unui obiect global şi adăgarea manuală a funcţionalităţii în proprietăţi ale acestuia – nu elimină totuşi în totalitate riscul suprascrierii de funcţionalitate în cadrul proiectului. Erorile cauzate de înlocuirea sau schimbarea involuntară a unor variabile – acestea pot conţine informaţii de stare sau de comportament – sunt deosebit de dificil de detectat. De aceea este preferabilă utilizarea unei funcţii sau clase de creare a spaţiilor de nume. Funcţionalitatea acesteia trebuie să cuprindă verificarea şi evaluarea existenţei proprietăţii (denumirea spaţiului dorit) în obiectul global şi crearea acestuia în cazul în care numele este valid (nu ar suprascrie un atribut deja definit) sau semnalarea unei erori prin *aruncarea* unei excepţii în caz contrar. Această variantă asigură protecţia şi neutralizează posibilitatea suprascrierii spaţiilor de nume şi a funcţionalităţii incluse.

Avantajele utilizării spaţiilor de nume în JavaScript sunt multiple, tehnica fiind considerată adesea *un element important al programării responsabile*[2]. Dintre motivele care recomandă acest tip de abordare – mai ales în cazul aplicaţiilor sau al frameworkurilor de dimensiuni medii şi mari – le amintim pe următoarele : în primul rând, *reducerea poluării scopului global*[3], eliminarea în mare măsură a riscului de creare a situaţiilor conflictuale cu alte librării sau module în cazul suprapunerii numelor variabilelor, organizarea codului într-o manieră logică uşor de înţeles şi modificat. Considerăm că dezavantajele[4] pe care le menţionează bibliografia sunt nesemnificative raportat la cele enumerate mai sus şi având în vedere că există metode pentru a le depăşi. În concluzie, spaţiile de nume reprezintă un element esenţial în dezvoltarea oricărui proiect JavaScript de calitate.

### 3.2.7. Mediul gazdă specific navigatoarelor : B.O.M., D.O.M., evenimente

Aşa cum am precizat, codul JavaScript este rulat într-un mediu gazdă. Tehnologia JavaScript este întrebuinţată în special pentru dezvoltarea aplicaţiilor Web. Prin urmare, mediul gazdă cel mai frecvent este motorul JavaScript al unui navigator. Din punctul de vedere al standardizării, există două categorii de tipuri de date : native (cele cuprinse într-o versiune ECMAScript) şi specifice mediului gazdă. Majoritatea navigatoarelor definesc două astfel de structuri proprii – am putea să le considerăm un standard *de facto* – pe care le vom prezenta în cele ce urmează : obiectul model al navigatorului (B.O.M – *Browser Object Model*) şi obiectul model al documentului (D.O.M. – *Document Object Model*). De asemenea, navigatoarele oferă un set de evenimente JavaScript pentru fiecare tip de element HTML. Considerăm că a cunoaşte proprietăţile enumerate este esenţial întrucât acestea sunt fundamentele pe care se construieşte în mod direct sau interfaţat de un API / framework orice aplicaţie JavaScript.

[1] (Stefanov, JavaScript Patterns 2010, 87)
[2] (Harmes şi Diaz 2007, 66-67)
[3] (Stefanov, JavaScript Patterns 2010, 87)
[4] Vezi (Stefanov, JavaScript Patterns 2010, 88)

### 3.2.7.1. Obiectul model al documentului – DOM. Obiectul model al navigatorului – BOM

După cum sugerează numele, obiectul **model al documentului** este suma elementelor de conţinut ale unui fişier formatat după modelul unui limbaj de marcare. DOM este un standard dezvoltat de către World Wide Web Consortium[1] cu scopul de a reglementa independent de platformă sau de limbaj forma documentelor de tip XML sau descenţi ai acestuia. De asemenea, DOM nu este o caracteristică specifică doar JavaScript, ci o interfaţă de programare implementată în majoritatea limbajelor importante, specificând metode de acces şi de modificare a datelor. În concluzie, putem să definim obiectul model al documentului ca fiind *o modalitate de reprezentare a unui document XML sau HTML printr-o structură arborescentă*[2].

Revenind la problema standardizării, reţinem că versiunile DOM se numesc nivele. În prezent există 3 astfel de nivele, iar versiunea a patra se află în stadiul de *W3C Working Draft* după cum se arată în http://www.w3.org/DOM/DOMTR. Navigatoarele actuale implementează standardul DOM în măsuri diferite. După cum precizează Stefanov, în general nivelul DOM 1 este implementat complet în majoritatea navigatoarelor[3].

În ceea ce priveşte motoarele JavaScript, observăm că obiectul model al documentului este implementat separat şi cu mici diferenţe între navigatoare – cauzate în special de funcţionalitatea anterioară standardizării. Acest fapt îngreunează adesea programarea în JavaScript. În continuare, vom prezenta câteva proprietăţi DOM şi practicile recomandate pentru utilizarea sa.

Limitându-ne la aplicaţii Web, putem considera că obiectul model al documentului reprezintă interfaţa dintre motorul JavaScript şi conţinutul paginii curente.

În condiţiile implementării neuniforme, accesul şi manipularea DOM se pot număra printre factorii cu pondere majoră în scăderea performanţei unei aplicaţii necesitând reevaluarea structurii HTML şi redesenarea ecranului. În ceea ce priveşte accesarea DOM, pentru utilizarea optimă a resurselor sistemului client, Stefanov recomandă următoarele[4] : evitarea apelurilor de căutare a elementelor în bucle, referenţierea elementelor prin variabile locale, utilizarea *Selectors API*[5] dacă este posibil şi determinarea lungimii unei colecţii de elemente HTML în afara iteraţiei propriu zise

**Manipularea** este o formă specială a accesării informaţiilor obiectului model al documentului care implică modificarea acestora. Asemeni accesului generic, se recomandă reducerea apelurilor de modificare la nivelul minim posibil prin împachetare.

În cazul adăugării de informaţie, Stefanov sugerează în *JavaScript Patterns* utilizarea unui fragment de document (*document fragment*) pentru efectuarea modificărilor DOM. Acesta devine suportul de date intermediar, nefiind legat de arborele din memoria motorului JavaScript. Conţinutul de modificat este pregătit folosind fragmentul de document şi adăugat atunci când procesarea a fost încheiată în totalitate. Avantajul metodei constă în limitarea apelurilor de schimbare la unul singur, responsabil pentru toată structura de elemente pe care o conţine. Pe de altă parte, pentru operaţiile de actualizare a unor noduri, fragmentul de document poate fi înlocuit de o clonă a elementului asupra căruia dorim să efectuăm modificări. La încheierea procesării datelor, clona este inserată în DOM înlocuind elementul original.

Spre deosebire de DOM, obiectul model al navigatorului nu este standardizat deloc. BOM este compus dintr-o serie de obiecte implementate similar în navigatoare. BOM reprezintă

[1] (World Wide Web Consortium 2005)
[2] (Stefanov, Object-Orieted JavaScript 2008, 218)
[3] Ibid. (206)
[4] (Stefanov, JavaScript Patterns 2010, 183-184)
[5] Vezi (World Wide Web Consortium 2012)

legătura dintre motorul JavaScript şi funcţionalitatea specifică acestora pe de o parte – de exemplu accesul la date de tip *cookie*, ferestre de dialog, bara de adresă şi altele –. Pe de altă parte BOM este şi o interfaţă între JavaScript şi fereastra propriu-zisă a navigatorului oferind posibilitatea controlării unor ferestre noi, a redimensionării, repoziţionării sau închiderii acesteia Nu în ultimul rând, obiectul model al browserului oferă programatorului acces la evenimentele specifice ferestrei navigatorului cum ar fi părăsirea paginii sau deschiderea şi închiderea ferestrei. În mod similar, prin variabila globală *document* care conţine legătura cu reprezentarea DOM, pot fi detectate şi tratate evenimentele referitoare la conţinut.

### 3.2.8. Evenimente

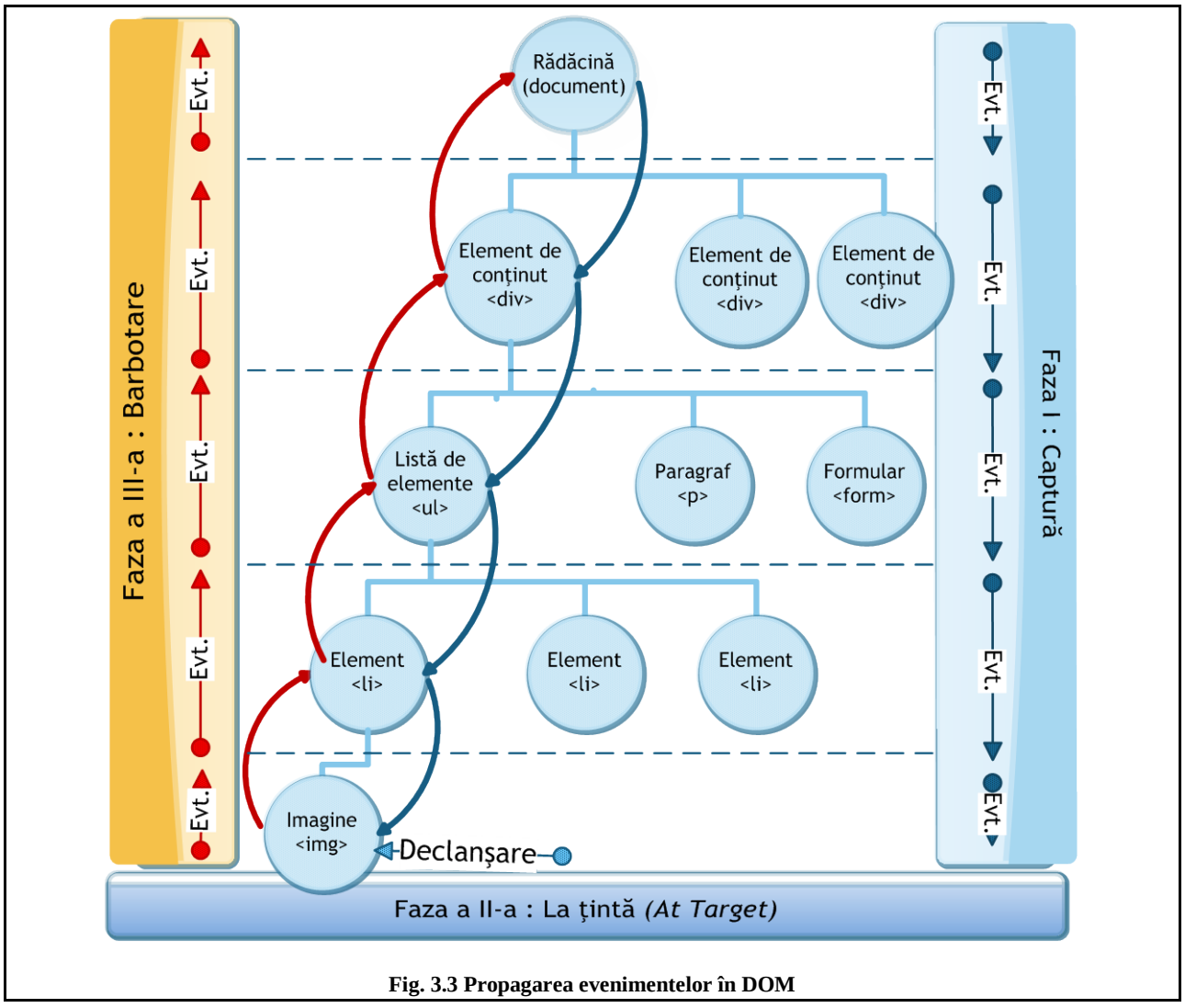



**Fig. 3.3 Propagarea evenimentelor în DOM**

Evenimentele constituie în majoritatea limbajelor de programare un factor primordial în interacţiunea dintre utilizator şi aplicaţie. Utilizarea acestora nu se rezumă însă la atât. Mai mult, programarea bazată pe evenimente este adesea soluţia care stă la baza unei arhitecturi software cu cuplare minimă – un exemplu ar fi modelul de design *publică – subscrie* (*Publish – Subscribe*).

Această tehnică de programare se poate utiliza şi în JavaScript cu diferenţe infime raportat la alte limbaje de programare precum Java sau C# în special datorită specificaţiei DOM 2 şi prin dezvoltarea unor frameworkuri şi API-uri care uniformizează comportamentul navigatoarelor. În acest sens, jQuery, frameworkul pe care îl vom utiliza pentru dezvoltarea practică a acestei lucrări, este una dintre librăriile care conţin o interfaţă uniformă pentru programarea bazată pe evenimente. Vom detalia funcţionalitatea asigurată de jQuery în secţiunea dedicată tehnologiilor utilizate.

Considerăm important a cunoaşte câteva detalii referitoare la implementarea şi comportamentul evenimentelor în JavaScript. Spre deosebire de alte limbaje de programare, evenimentele caracteristice elementelor de conţinut cum ar fi un bloc paragraf (<*p/*>) se propagă prin întregul arbore de noduri prin **barbotare** (***bubbling***) sau **captură** (***capture***) în funcţie de navigator. Istoric vorbind, cele două metode au fost introduse de viziunile opuse exprimate în navigatoarele de tip Microsoft Internet Explorer şi Netscape Navigator. În consecinţă, Microsft a implementat în Internet Explorer modelul bazat pe barbotare, în timp ce Netscape a preferat captura. Diferenţa între cele două modele constă în direcţia propagării evenimentelor prin arborele DOM. Barbotarea presupune parcurgerea arborelui de jos în sus, în timp ce captura propune traversarea de sus în jos după cum ilustrăm în figura următoare :
Problema a fost depăşită printr-un compromis făcut de World Wide Web Consortium în standardul DOM nivelul 2. Conceptual, DOM 2 specifică trei faze ale propagării unui eveniment[1] : captură, la ţintă (*At* Target) şi barbotare. Prima şi ultima fază corespund celor două viziuni prezentate anterior. Din punctul de vedere al implementării, această specificaţie a fost rezolvată prin adăugarea unui parametru în lista funcţiei *addEventListener*. Astfel, pe lângă apelul callback de executat la declanşarea evenimentului, *addEventListener* are şi un parametru pentru controlul modului de propagare. În mod implicit, parametrul de control este *true* iar propagarea se realizează prin barbotare. Este de reţinut şi faptul că propagarea evenimentelor poate fi oprită programatic prin apelarea funcţiei *stopPropagation*.

**Delegarea** este un alt subiect important atunci când discutăm despre evenimente în JavaScript. Scopul şi avantajul major al utilizării acestei tehnici este reducerea numărului de observatori de evenimente (event listeners). Implementarea delegării are la bază modul în care se efectuează propagarea prin arborele DOM şi atributul *currentTarget* al obiectului descriptor transmis ca parametru funcţiei de tratare a evenimentului. După cum arată MacCaw [2], considerând exemplul din figura anterioară, pentru a trata acţiunea de clic asupra tuturor elementelor de tip <*li*> dintr-o colecţie, putem defini un singur observator de evenimente la nivelul părintelui – <*ul*> în cazul de faţă – după cum urmează în fragmentul de cod de mai jos :

```
/* Crearea unei referinte locale spre colectia de elemente */
var ul = document.getElementById("#lista");
/* Adaugarea observatorului de evenimente */
ul.addEventListener("click", function(descriptorEveniment)) {
/*  Testarea nivelului de propagare prin utilizarea informatiilor
descriptorului de eveniment */
    if(descriptorEveniment.currentTarget.tagName = "li") {
        /* tratarea evenimentului */
        return false;
    }
}, false);
```

[1] (World Wide Web Consortium 2000)
[2] (MacCaw 2011, 24-25)

Un alt avantaj pe care îl precizează MacCaw este consistenţa observatorului de evenimente definit astfel cu posibilitatea adăugării unor elemente noi în colecţie la un moment posterior definirii sale.

Evenimentele personalizate sunt o altă temă semnificativă. Pentru moment, reţinem că, deşi standardul W3C DOM conţine specificaţii pentru evenimente definite de program, acestea au fost *ignorate de către producătorii navigatoarelor*[1]. Vom dezvolta subiectul personalizării evenimentelor şi al modului de utilizare a tehnicii delegării în secţiunea dedicată prezentării frameworkului jQuery.

## 3.3. Produse existente asemănătoare : Yahoo!UI şi ExtJS

Suntem de părere că evoluţia recentă a JavaScript spre a deveni componenta de ordin comportamental a aplicaţiilor Web este datorată în foarte mare măsură apariţiei librăriilor precum Yahoo!UI, jQuery, Dojo sau ExtJS. Am putea afirma că rolul librăriilor menţionate este comparabil cu cel al *stdlib* în limbajul C. În esenţă, prin simplificarea programării independente de navigator şi prin rezolvarea unor probleme comune – atât legate de interfaţă, cât şi de comportament –, Yahoo!UI, jQuery, Dojo, Prototype, ExtJS şi altele fac ca JavaScript să fie o opţiune viabilă în ceea ce priveşte programarea aplicaţiilor Web.

În continuare propunem o clasificare a librăriilor JavaScript în funcţie de mediul de utilizare – client-server sau navigator –, scop – general sau specific – şi licenţă – open source sau comercială.

| | **Open Source** | **Comercial** |
|---|---|---|
| **Client / Server** | AjaxCFC, Qcodo | Atlas, Backbase (Struts) |
| **Navigator** | Prototype, **jQuery**, Yahoo UI | Backbase, SmartClient |

| | **Navigator** | **Client-Server** |
|---|---|---|
| **Specifice** | Scriptaculous, moo.fx, Open Rico | AjaxCFC, Qcodo |
| **Generale** | Prototype, **jQuery**, Yahoo UI | Ruby on Rails, Cake PHP |

**Tabelul 3.3 Clasificarea librăriilor JavaScript**

Clasificarea de mai sus reprezintă un factor important în alegerea produselor existente asemănătoare pe care le vom trata cu o atenţie deosebită în continuare. Selecţia librăriilor asemănătoare este justificată, în egală măsură, prin implementarea asemănătoare a conceptelor de programare orientată pe obiecte şi prin gradul de utilizare[2].

În cele ce urmează vom prezenta librăriile Yahoo UI şi Sencha ExtJS din punctul de vedere al manipulării DOM şi CSS, tratării evenimentelor, componentelor oferite pentru realizarea interfeţei grafice, interfaţării AJAX şi, nu în ultimul rând, al arhitecturii şi al implementării conceptelor de programare orientată pe obiecte.

[1] (MacCaw 2011, 25)

[2] Conform http://w3techs.com/technologies/overview/javascript_library/all jQuery, baza de la care ponim în realizarea unui framework de uz general pentru aplicaţii web este utilizat în 51% din paginile web monitorizate, *YUI Library* – 1,6% şi ExtJS – 0,1%. Menţionăm că, în cazul YUI şi ExtJS factorul de similitudine primează în selecţionarea pentru prezentare în această lucrare.

### 3.3.1. Yahoo!UI

Yahoo UI (YUI) este unul dintre cele mai ample frameworkuri JavaScript *open-source* existente la ora actuală pentru dezvoltarea aplicaţiilor web interactive şi a contribuit decisiv prin inovaţiile aduse la adoptarea JS ca limbaj al web-ului. Notăm că interfaţa grafică a portalului Yahoo este construită începând din 2006 integral pe baza acestui framework. Din punctul de vedere al modelului de dezvoltare, o diferenţă importantă între YUI şi frameworkul propus în această lucrare, este faptul că primul oferă soluţii proprii pentru majoritatea subiectelor abordate, în timp ce, în cazul *jqCls* am optat pentru integrarea modulelor granulare existente confirmate de către comunitatea *open-source*.

Arhitectural, Yahoo!UI este construit modular conform *Module Pattern*. Funcţionalitatea principală este structurată în : ***Core***, ***Utilities***, ***Controls*** *şi* ***CSS Resources***.

Modulul **Core** oferă funcţionalitatea de bază pentru manipularea evenimentelor şi manipularea arborelui DOM. De asemenea, acesta defineşte obiectul global YUI – spaţiul de nume al frameworkului – în care sunt incluse utilităţi specifice limbajului şi de infrastructură precum şi un încărcător dinamic de scripturi.

Un aspect deosebit de important al modulului *Core* este infrastructura pentru facilitarea aplicării paradigmei obiectuale în JavaScript. În acest sens, YUI oferă o implementare a metodei moştenirii *clasice* prin funcţia *extend*. În acelaşi timp, frameworkul oferă şi posibilitatea utilizării modelului moştenirii prototipice prin funcţiile *augmentObject* şi *augmentProto*.

Modulul **Utilities** reprezintă nivelul de servicii al frameworkului. Dintre funcţionalităţile oferite la acest nivel suntem de părere că următoarele au o relevanţă majoră în înţelegerea YUI : **Connection Manager** (cereri prin *XMLHttpRequest*), **DataSource** (interfaţă generică pentru reprezentarea surselor de date), **Element** (reprezentarea obiectuală JS a elementelor HTML), **JSON**, **Selector** (căutarea elementelor HTML în arborele DOM pe baza sintaxei CSS3).

Modulul **Controls** poate fi asemănat cu pachetul **Components** din frameworkul propus. Astfel, acesta oferă implementări generale, abstracte ale unor componente complexe de bază pentru aplicaţii web interactive. Dintre acestea amintim : ***AutoComplete***, ***Calendar***, ***Charts***, ***LayoutManager***, ***Menu***, ***Rich Text Editor***, ***TreeView*** sau ***Uploader***. Considerăm că denumirile componentelor enunţate mai sus sunt autoexplicative.

În afară de funcţionalitatea JavaScript amintită mai sus, YUI oferă sprijin şi pentru stilizarea vederilor prin modulul de resurse CSS denumit ***CSS Resources***. Acesta este compus din patru submodule pe care le enumerăm în continuare : *Reset* – normalizează modul de interpretare şi redare al stilurilor, *Base* – înlocuieşte în mod consistent şi independent de navigator valorile CSS implicite, *Grids* – conţine şapte şabloane generale de pagină prin subsecţiunile cărora pot fi realizate peste 1000 de scheme diferite – şi, nu în ultimul rând *Fonts* – conţine definiţii de familii de fonturi (regulile CSS *font-family*, *font-face*), cât şi setări pentru dimensiune şi reprezentare.

De asemenea, frameworkul conţine instrumente pentru facilitarea procesului general de dezvolare şi testare grupate în modulul ***Developer Tools***. *YUI Test* este un framework independent de testare JavaScript cu unele caracteristici similare *JUnit* – cunoscut pentru Java – sau *nUnit*. Suntem de părere că elementele *Profiler* şi *ProfilerViewer* sunt în egală măsură deosebit de utile pentru monitorizarea performanţei unei aplicaţii.

În vederea susţinerii procesului de documentare, YUI este în strânsă legătură cu generatorul de documentaţie API ***YUIDoc***. Menţionăm că acesta a fost utilizat şi în realizarea documentaţiei API a *jqCls*.

**YUI Compressor** este o parte integrantă a frameworkului şi are o importanţă majoră în ceea ce priveşte susţinerea procesului de împachetare, optimizare şi minifiere a aplicaţiilor client.

Instrumentul este independent de framework şi poate fi utilizat la modul general, asemeni YUIDoc în ceea ce priveşte generarea documentaţiei API, pentru *compilarea* aplicaţiilor JavaScript.

### 3.3.2. Sencha ExtJS

ExtJS este prezentat de către compania producătoare Sencha ca fiind *lider de clasă al aplicaţiilor web enterprise oferind o suită completă de componente, documentaţie şi instrumente necesare pentru dezvoltarea soluţiilor software robuste*.

La origini, ExtJS a fost dezvoltat ca un *plug-in* pentru realizarea vederilor tabelare cu YUI. Notăm că, la început ExtJS s-a dozvoltat acoperind nevoia de control al fluxului de date în aplicaţiile construite pe baza YahooUI până la versiunea 1.1. Începând cu versiunea 2.0, – versiunea actuală, în momentul redactării acestei lucrări, este 4.2.1 – frameworkul s-a desprins total de YUI şi a devenit de sine stătător. În acest sens, observăm că, asemeni YUI şi spre deosebire de frameworkul propus – *jqCls* –, ExtJS oferă soluţii proprii pentru toată gamă de funcţionalităţi abordate.

Din punct de vedere arhitectural, ExtJS are o structură ierarhică de pachete şi clase. Începând cu versiunea 4.0, abordează o implementare OOP *clasică*. Menţionăm că sintaxa de specificare a entităţilor funcţionale din *jqCls* este similară cu aceea a ExtJS. În acelaşi timp este important faptul că implementarea sistemului de clase al *jqCls* este independent de cel al ExtJs.

Un alt aspect deosebit de important al ExtJS în relaţie cu paradigma obiectuală este implementarea moştenirii multiple prin ceea ce numim *mixin*[1]. În acelaşi sens al implementării principiilor programării cu obiecte, ExtJS conţine aspecte de încapsulare a funcţionalităţii şi injecţie avansată a dependinţelor. Concret, frameworkul generează şabloanele de acces la resurse şi suportă realizarea implementării de comportament specific aplicaţiei. Un exemplu ar fi sistemul de referenţiere al vederilor din *controller* prin selectori CSS. În acest caz, ExtJS generează automat metodele de acces de tip *getter* la obiectele de tip vedere. Clasa *Model* reprezintă un alt exemplu de implementare a principiului încapsulării. Un obiect de tip *Model* ExtJS are o structură complexă. Dintre proprietăţile funcţionale ale clasei *Model* considerăm că validarea datelor şi urmărirea schimbărilor sunt esenţiale. Accesul la datele concrete ale unui obiect *Model* ExtJS se face prin metodele *getter* şi *setter* generice *get* şi *set* care asigură completarea secvenţelor de validare şi notificare necesare.

După cum am precizat, ExtJS a apărut pe fondul nevoii unui nivel de control în aplicaţiile JavaScript cauzată de creşterea acestora în complexitate. Astfel, frameworkul oferă o interfaţă de programare completă pentru controlul fluxului de *bussiness* şi reprezentarea şi manipularea datelor în aplicaţii complexe. Clasele *Model* şi *Store* (reprezentarea colecţiilor de modele), precum şi implementarea după şablonul *Proxy* a nivelului de persistenţă asigură un punct de plecare solid în definirea domeniului de *bussiness* al aplicaţiilor web client. În acest sens, interfaţa colecţiilor ExtJS de tip *Store* cuprinde, pe lângă integrarea cu nivelul de persistenţă şi operaţii CRUD, funcţionalitate pentru sortare, filtrare şi căutare. Dintre obiectele de tip proxy care asigură persistenţa datelor implementate în ExtJs, considerăm că *LocalStorage*, *SessionStorage*, *Ajax, JSONP* şi *REST* sunt sunt esenţiale. Modul de interacţiune transparentă dintre obiectele de tip *Model* sau *Store* şi proxy-urile de persistenţă, precum şi utilizarea unui format acceptat (XML sau, preferabil JSON) de transport asigură independenţa totală a aplicaţiei

[1] Procedura numită ***mixin*** pentru realizarea moştenirii multiple în JavaScript a fost descrisă în subcapitolul 4.1.2.6 - Alte metode de reutilizare a codului. Clase prin părţi. Împrumutarea metodelor. Clase abstracte. Moştenire multiplă

în raport cu tehnologia utilizată pentru stocarea datelor (local sau *la distanţă*), cât şi în relaţie cu tehnologia server utilizată. În ceea ce priveşte modul de realizare a nivelului de persistenţă, menţionăm că în frameworkul propus, abordarea este una asemănătoare.

ExtJS satisface nevoia aplicaţiilor web JavaScript şi condiţia impusă în mod general în domeniul dezvoltării software de separare a responsabilităţii obiectelor în funcţie de categorie (*control, model* şi *reprezentare*) prin implementarea la nivel abstract a şablonului arhitectural MVC şi prin facilitatea modului de programare bazat pe evenimente. Din experienţa acumulată şi în acord cu părerile specialiştilor în domeniu, notăm că implementarea MVC ExtJS acordă o libertate foarte mare obiectelor de tip *Controller*.

Sistemul de componente pentru realizarea interfeţelor grafice propus de ExtJS este similar cu cel prezentat în cazul YUI. Implementarea nivelului de *bussiness* mult mai elaborată din ExtJS constituie diferenţa majoră în această privinţă. Abordarea modului de realizare a schemelor de pagină este o altă diferenţă remarcabilă. După cum am precizat, în YahooUI, acestea sunt realizate prin modulul CSS Resources, în timp ce în ExtJS există o soluţie programatică implementată similar cu varianta Java Swing în pachetul *Ext.layout*.

Din punctul de vedere al creării vederilor, notăm că ExtJS conţine un motor propriu, puternic pentru procesarea şabloanelor HTML. Acesta poate fi utilizat prin obiecte de tip *XTemplate*. În sensul personalizării elementelor grafice prin teme CSS, considerăm deosebit de utilă construirea acestor utilităţi pe baza funcţionalităţii oferite de translatorul **SASS** şi frameworkul **Compass**.

Având în vedere cele precizate mai sus, concluzionăm că ExtJS este o opţiune viabilă pentru aplicaţii web locale sau integrate în arhitecturi bazate pe servicii (SOA).

| Aspect | YUI | Sencha ExtJS | jqCls |
|---|---|---|---|
| **Arhitectură generală** | | | |
| Implementare modulară | Programarea modulelor conform unei interfeţe. | Da. | Programarea modulelor conform unei interfeţe. |
| Sistem de clase şi interfeţe | Implementare clasică sau prototipică | Implementare *clasică* | Implementare clasică cu elemente prototipice |
| | Implementarea interfeţelor fără verificare concretă a funcţionalităţii | | Verificarea strictă a funcţionalităţii în cazul specificării unei interfeţe |
| **Şabloane arhitecturale** | Nu | MVC | MVVM, MVC, MV-W |
| **Şabloane de design abstractizate** | | Observer, Singleton | Observer |
| **Componente vizuale** | Structură asemănătoare | | |
| **Mod de manipulare DOM** | Simplă | Complicat de realizat | Simplă prin jQuery |
| Utilizarea şabloanelor pentru vederi HTML | Nu | Limitată de componentă | Da, prin Underscore.js sau Knockout |
| **Nivel de reprezentare a datelor** | *Lightweight* prin interfaţa **DataSource** | Complet (Colecţii – **Store**, Modele) | Complet prin Backbone.js, Knockout.js şi Knockback.js |
| **Model de dezvoltare** | *Open-source, soluţii proprii şi integrare de plug-inuri* | | *Open-source, soluţii proprii şi integrate cu alte frameworkuri granulare* |

**Tabelul 3.4 - Prezentare comparativă a soluţiilor analizate**

# 4. Fundamentare teoretică : Programare orientată pe obiecte în JavaScript

În capitolul precedent, am prezentat câteva proprietăţi distinctive ale limbajului JavaScript a căror cunoaştere este esenţială pentru dezvoltarea aplicaţiilor Web. Pornind de la aceste trăsături ale limbajului, vom discuta în continuare modurile de implementare şi de utilizare ale conceptelor de programare obiectuală în JavaScript : noţiunea de clasă, încapsularea, moştenirea interfaţarea şi modelele de design.

## 4.1. Utilizarea conceptelor de programare obiectuală în JavaScript

### 4.1.1. Implementarea şi utilizarea noţiunii de clasă

După cum am precizat la începutul lucrării, JavaScript este un limbaj bazat pe prototipuri, adică nu implementează noţiunea de clasă. Subiectul este controversat. Pe de o parte, unele lucrări de specialitate recomandă evitarea acestei implementări, considerând că *adaugă sintaxă şi reguli noi de învăţat şi reţinut*[1] şi că *introduce noţiunea de clasă, derutantă în ansamblu şi inexistentă din punct de vedere tehnic în limbaj*[2]. Pe de altă parte, pornind de la criticile aduse acestui mod de programare în JavaScript, corelate cu neutralitatea multor librării faţă de metodologia organizării codului, MacCaw precizează că acestea pot determina ignorarea necesităţii structurării codului şi prezintă noţiunea de clasă ca fiind ***un instrument la fel de util în JavaScript ca şi în orice alt limbaj modern***[3]. La începutul lucrării am arătat că funcţiile constructor şi prototipul unui obiect pot satisface definiţia unei clase.

Există mai multe metode de implementare şi utilizare a conceptului de clasă în JavaScript. Cea mai simplă variantă este utilizarea a ceea ce numim clase statice. Acestea sunt de fapt obiecte create printr-un literal şi sunt inutile în cazul necesităţii creării unor instanţe multiple. O altă variantă este definirea funcţiilor constructor şi utilizarea operatorului *new* pentru crearea unei instanţe noi. Este de reţinut că operatorul *new* are un comportament similar în JavaScript cu alte limbaje de programare. Utilizarea acestuia schimbă contextul funcţiei. În consecinţă, cuvântul cheie *this* din interiorul funcţiei reprezintă o referinţă spre scopul local al instanţei curente şi nu spre cel global.

O variantă complexă de implementare a conceptului în JavaScript este utilizarea unei funcţii de definire a claselor denumită de obiecei ***klass***. Stefanov[4] şi MacCaw[5] se numără printre cei care au teoretizat subiectul. Această variantă derivă din cea prezentată mai sus, având la bază funcţiile constructor şi utilizarea operatorlui *new*. Notăm, de asemenea că, după cum am arătat în secţiunea dedicată funcţiilor JavaScript, acestea pot returna orice tip de date – chiar şi o altă funcţie (un obiect de tip Function). Astfel, rolul principal al unei funcţii de definire a claselor în JavaScript este crearea şi returnarea unei funcţii constructor. Stefanov evidenţiază trei similitudini între implementările metodei *klass* – numele este dat în general funcţiei de specificare a claselor – : convenţia pentru denumirea metodei cu rol de constructor, clasele definite astfel pot moşteni alte clase şi existenţa unei legături între clasa copil şi cea părinte. În ceea ce priveşte denumirea clasei de definit, prin convenţie, acesta trebuie să înceapă cu majusculă.

[1] (Stefanov, JavaScript Patterns 2010, 130)
[2] Ibid. (130)
[3] (MacCaw 2011, 6)
[4] Vezi (Stefanov, JavaScript Patterns 2010, 128-130)
[5] Vezi (MacCaw 2011, 6-9)

În ceea ce priveşte declararea metodelor şi atributelor unei clase, MacCaw arată că definiţia celor statice se face direct în corpul funcţiei constructor, în timp ce proprietăţile specifice instanţelor sunt specificate pe obiectul prototip (*prototype*) după cum urmează :

```
/* Definirea unei clase noi "ClasaMea" */
var ClasaMea = new Class;
/* Atribute si metode statice */
ClasaMea.atributStatic = valoare;
ClasaMea.metodaStatica = function([listaParametri]{...};
/* Atribute si metode ale instantelor*/
ClasaMea.prototype.atributInstanta = valoare;
ClasaMea.prototype.metodaInstanta = function([listaParametri]){...};
```

MacCaw consideră această sintaxă complicată, dificil de citit şi de ordonat şi propune o îmbunătăţire a funcţiei de specificare a claselor prin parametrizarea atributelor şi metodelor statice, respectiv instanţă. În consecinţă, atributele şi metodele vor fi transmise ca parametri reprezentaţi prin sintaxa de tip literal obiect funcţiei *klass* care va avea şi rolul de a le copia în corpul clasei. De asemenea, MacCaw recomandă includerea funcţiilor de copierea a proprietăţilor statice şi specifice instanţei în definiţia clasei create. Cu această variantă a implementării conceptului de clasă, în cazul exemplului de mai sus observăm structurarea într-o formă mult mai uşor de înţeles şi urmărit :

```
var ClasaMea = new Class({
    proprietăţiInstanţă : {
        atributInstanţă : valoare,
        metodăInstanţă : function() {...}
    },
    proprietăţiStatice : {
        atributStatic : valoare,
        metodăStatică : function() {...}
    }
});
```

În acest sens, MacCaw arată că unul dintre câştigurile oferite de această metodă este posibilitatea utilizării modulare a codului asemănător cu practica din Ruby. De altfel, această abordare este şi una dintre variantele de utilizare a moştenirii şi, implicit, a reutilizării codului.

Evitarea conflictelor de nume care pot cauza suprascrierea de funcţionalitate este o problemă importantă – cu consecinţe grave asupra codului, netratată corespuzător – în ceea ce priveşte implementarea claselor în JavaScript. Soluţia optimă în acest caz este utilizarea spaţiilor de nume prezentate mai sus. Notăm că diferenţa între un pachet Java şi un spaţiu de nume JavaScript constă în faptul că ultimul poate să conţină de fapt orice tip de date : de exemplu, în acest caz un obiect de tip *Function* reprezentând o clasă. Punând în relaţie funcţia *klass* cu cele precizate despre spaţiile de nume, precizăm că utilizarea tehnicilor de organizare a codului prezentate anterior este deosebit de importantă, iar includerea acestora indispensabilă.

Având în vedere scopul şi obiectivele fixate, vom utiliza şi exemplifica acest mod de abordare în dezvoltarea practică a lucrării.

## 4.1.2. Moștenirea

### *4.1.2.1. Principiu*

Paradigma programării orientate spre obiecte este cea mai apropiată de forma naturală a conceptelor şi proceselor din fragmentul de lume pe care îl modelează o aplicaţie. Din acest punct de vedere, putem asemăna moştenirea – adică organizarea ierarhică a codului în clase – cu ceea ce studiul limbii şi al comunicării numesc generalizare. În esenţă, la nivel logic, generalizarea înseamnă extragerea caracteristicilor comune ale unor obiecte şi gruparea acestora într-o noţiune abstractă.

În cursul de Programare orientată pe obiecte, Marius Joldoş oferă o definiţie teoretică a conceptului : ***moştenirea*** *este una dintre tehnicile principale ale programării orientate pe obiecte, modelează relaţii de tipul „este un" folosind* ***asemănările*** *şi* ***deosebirile*** *în reprezentarea ierarhică a unor grupuri de abstractizări înrudite şi presupunând, în primă fază, definirea unei forme generale de clasă completată ulterior prin specificarea unor versiuni specializate ale acesteia prin adăugarea de variabile instanţă şi de metode*[1]. Mai mult, Joldoş precizează că *moştenirea este un mod de organizare a informaţiei şi de creare a unei taxonomii de obiecte prin gruparea claselor şi modelarea asemănărilor acestora*[2]. Spre deosebire de alte limbaje, în JavaScript există modalităţi diferite de abordare a conceptului pe care bibliografia le împarte în două categorii : ***moştenirea de clasă*** (***clasică***[3]) şi ***moştenirea prototipică***.

Din punctul de vedere al complexităţii, metodele clasice sunt mai simple şi utilizate mai des, având în vedere cunoaşterea şi înţelegerea la scară mult mai largă a paradigmei orientate pe obiecte faţă de cea prototipică. Vom prezenta mai jos o comparaţie amplă a metodelor de moştenire clasică raportate la cele prototipice.

***Moştenirea prin copiere*** este o altă variantă prin care se poate realiza scopul de reutilizare a codului pe baza unei structuri de date ierarhice şi este discutată de Stefanov în *JavaScript Patterns*[4] şi în *Object Oriented JavaScript*[5]. Aceasta este o metodă simplistă de implementare a conceptului şi, din punctul de vedere al acestei lucrări, reprezintă doar o simulare a comportamentului dorit. După cum sugerează şi numele, metoda are la bază ideea copierii proprietăţilor unui obiect pentru a fi utilizate în altul. În comparaţie cu metodele prezentate mai jos, această abordare este mult mai simplă conceptual : nu presupune cunoştinţe legate de prototipul unui obiect şi nu implică crearea unui lanţ prototipic. În esenţă, membrii proprii ai unui obiect sunt copiaţi în altul. O problemă importantă de tratat şi în cazul acestei abordări este legată de obiectele create prin compoziţie. Vom dezvolta acest subiect mai jos în secţiunea dedicată *moştenirii prototipice*. În ceea ce priveşte copierea proprietăţilor, reţinem că există două posibilităţi : ***superficială*** şi ***adâncă***. Astfel, în prima variantă, conform celor expuse mai sus referitor la tipurile de date, proprietăţile primitive sunt copiate prin valoare în timp ce restul sunt copiate prin referinţă. Deosebirea majoră între posibilităţile de mai sus este că în cazul copiei adânci, parcurgerea proprietăţilor unui obiect trebuie să fie recursivă, asigurându-se astfel reinstanţierea membrilor descendenţi *Object*. De asemenea, Stefanov propune şi o combinare a

[1] (Joldos 2011, 39-40)
[2] Ibid. (41)
[3] Stefanov arată că termenul *clasic* nu are în acest caz sensul de model, scriere după canoanele obişnuite sau învechit, ci, se referă la o abordare a moştenirii care presupune utilizarea conceptlui de clasă ca formă de abstractizare şi este specifică limbajelor de programare orientate pe obiecte cu o implementare nativă a acestuia. (Stefanov, JavaScript Patterns 2010, 115-117)
[4] Vezi (Stefanov, JavaScript Patterns 2010, 133-135)
[5] Vezi (Stefanov, Object-Orieted JavaScript 2008, 179-182).

*copierii proprietăţilor* cu *metoda prototipică*[1]. Astfel, Stefanov arată că moştenirea prototipică se poate utiliza pentru clonarea unui obiect existent urmată de copierea proprietăţilor unui alt obiect.

Majoritatea modalităţilor de implementare a conceptului în JavaScript, atât cele bazate pe clase, cât şi cele prototipice, au în comun utilizarea prototipului de obiect (proprietatea *prototype* prezentată mai sus). Astfel, având în vedere cele precizate referitor la prototipuri şi după cum precizează Flanagan[2], putem considera că JavaScript are un *mecanism de moştenire dinamic : un obiect îşi moşteneşte prototipul chiar dacă acesta este modificat ulterior creării instanţei*.

### 4.1.2.2. Moştenirea de clasă (clasică)

În ceea ce priveşte moştenirea *clasică*, Stefanov prezintă cinci variante de implementare incrementale : ***modelul implicit***, ***împrumutarea funcţiei constructor***, ***împrumutarea şi stabilirea prototipului***, ***împărţirea prototipului*** şi ***utilizarea unei funcţii constructor temporare***[3].

Prima şi cea mai simplă variantă propusă de Stefanov, ***modelul implicit***, presupune crearea unei funcţii generice de moştenire care rezolvă instanţierea şi referenţierea câte unei instanţe de tip *Părinte* prin prototipul obiectului de clasă derivată. După cum am precizat mai sus, organizarea ierarhică a abstractizărilor se realizează în general în JavaScript prin legături între prototipurile obiectelor. Pentru a înţelege funcţionarea conceptului în JavaScript – asemănătoare de altfel mai ales în acest caz cu implementarea din alte limbaje de programare –, să urmărim lanţul de prototipuri creat, pornind de la exemplul lui Stefanov în care o clasă *Părinte* cu proprietatea *nume* şi metoda *vorbeşte* este extinsă de o clasă *Copil*.

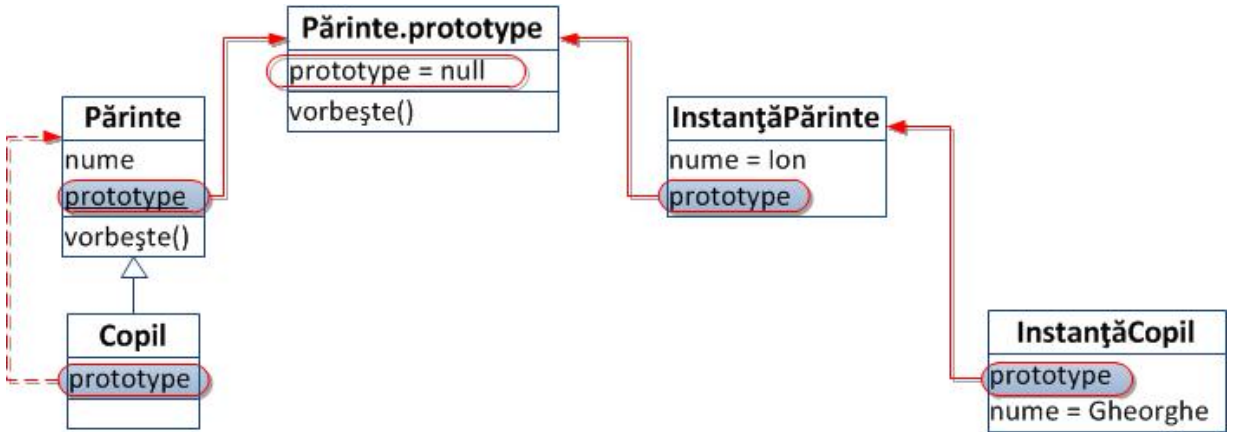



**Fig. 4.1 Lanţul de prototipuri după (Stefanov, JavaScript Patterns 2010, 118) cu adăugări**

Pornind de la instanţa clasei *Copil* remarcăm că, aşa cum arată de altfel şi Stefanov, prototipul acesteia (*prototype*) reprezintă o referinţă spre instanţa clasei părinte – aşa cum în Java super este o referinţă spre instanţa clasei părinte – iar atributul prototype al instanţei părinte este o referinţă spre prototipul clasei *Părinte*. În ceea ce priveşte apelul, accesarea şi modificarea membrilor instanţelor clasei *Copil*, reţinem că motorul JavaScript va parcurge obiectele din lanţul de prototipuri până la găsirea primului obiect care conţine proprietatea căutată sau, în

[1] Vezi Ibid. (187-188)
[2] (Flanagan 2011, 208)
[3] (Stefanov, JavaScript Patterns 2010, 117-128) – În continuarea secţiunii 3.1.2.2 se vor prezenta metodele descrise şi exemplificate în sursa citată.

cazul în care aceasta nu există, până când *prototype* are valoarea *null*. Astfel, considerând că în exemplul ilustrat mai sus atributul *nume* al obiectului de tip *Copil* nu este definit, – de exemplu imediat după instanţiere – în consecinţa invocării metodei *vorbeşte*() interpretorul va parcurge lanţul de prototipuri în ordinea *InstantăCopil* → *InstanţăPărinte* → *Părinte.prototype*. La întâlnirea prototipului clasei *Părinte* găsirea proprietăţii *vorbeşte* va determina oprirea căutării şi executarea funcţiei. Presupunând că implementarea funcţiei *vorbeşte* este de a tipări în consolă atributul *nume*, reţinem că în condiţiile menţionate, interpretorul va relua căutarea după cum am precizat mai sus, însă se va opri de această dată la instanţa *Părinte*. Pe de altă parte, în cazul în care *nume* este definit în cadrul instanţei *Copil*, parcurgerea lanţului prototipic se va opri la *InstantăCopil*, însă remarcăm că atributul instanţei *Părinte* îşi păstrează valoarea.

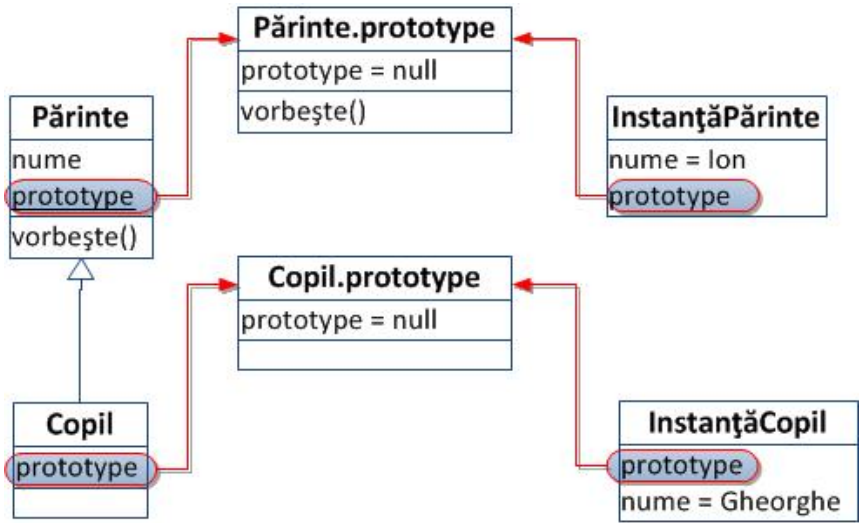



**Fig. 4.2 Lanţul prototipic în cazul împrumutării funcţiei constructor după (Stefanov, JavaScript Patterns 2010, 122) cu adăugări**

***Împrumutarea funcţiei constructor*** este o altă modalitate de implementare şi utilizare a moştenirii pe care o prezintă Stefanov. Ideea de bază a acestei tehnici este similară cu cea prezentată anterior. Diferenţa constă în modul de implementare. Spre deosebire de modelul implicit care presupunea definirea unei funcţii de extindere a claselor, în cazul de faţă, comportamentul specific conceptului de moştenire este asigurat prin apelarea metodei ***apply***[1] a constructorului părinte asupra instanţei *this* şi a listei de argumente (*arguments* [2] ) a constructorului subclasei. Stefanov arată că folosirea acestei metode rezolvă două probleme ale *modelului implicit* : posibilitatea transmiterii de argumente funcţiei constructor şi copierea membrilor moşteniţi spre deosebirea de referenţierea lor specifică primei variante. Dezavantajul major este limitarea moştenirii la membrii declaraţi în scopul *this* al constructorului împrumutat – în consecinţă, atributele şi metodele definite în cadrul prototipului clasei părinte nu sunt incluse, având în vedere întreruperea lanţului prototipic exemplificată în figura de mai jos. Considerând exemplul din figura următoare, notăm că invocarea metodei *vorbeşte* asupra instanţei *Copil* ar rezulta într-o eroare O altă caracteristică interesantă a acestei tehnici este

[1] Vezi (Mozilla Developer Network 2012) şi (Ecma International 2011, 119)
[2] Vezi (Ecma International 2011, 60)

posibilitatea implementării moştenirii multiple prin apelarea multiplă a metodei *apply* pentru fiecare superclasă. Notăm că rezolvarea oricărui conflict între denumirile proprietăţilor se rezolvă prin menţinerea ultimei valori.

***Împrumutarea constructorului şi stabilirea prototipului*** este o metodă care combină primele două tehnici prezentate. Stefanov este de părere că acest mod de abordare este, din punct de vedere comportamental, cel mai apropiat de limbajul Java. Astfel, utilizarea acestui model constă în apelul metodei *apply* a constructorului părinte şi referenţierea unei instanţe părinte prin prototipul subclasei. În acest caz se moştenesc atât membrii prototipului prin referenţiere, cât şi cei ai contexului *this* din funcţia constructor părinte prin copiere. În consecinţă, apelul metodei *vorbeşte* nu ar mai rezulta într-o eroare aşa cum am arătat în exemplul anterior. Unul dintre dezavantajele pe care le expune Stefanov este nevoia de instanţiere dublă a părintelui – în prima fază pentru *împrumutarea constructorului* şi, în a doua fază, pentru stabilirea prototipului. De asemenea, Stefanov menţionează că lanţul prototipic este acelaşi ca şi în cazul *metodei implicite*.

După cum sugerează şi numele, ideea de bază a celei de a patra metode este ***împărţirea prototipului***. În materie de implementare, această tehnică este similară primei, diferenţa constând în referenţierea protototipului clasei părinte prin atributul *prototype* al celei derivate după cum rezultă şi din figura următoare.

Avantajul acestei metode constă în efectivitatea şi viteza parcurgerii lanţului prototipic având în vedere existenţa unui singur element. În acelaşi timp însă, utilizarea unui prototip comun pentru *părinte* şi *copil* constituie şi un dezavantaj. Având în vedere cele precizate în capitolul anterior referitor la caracteristicile prototipului unei funcţii JavaScript, ca urmare a modificării prototipului la orice nivel al ierarhiei de derivare vor fi afectate toate obiectele existente, clasele specificate şi, implicit, înstanţele noi.

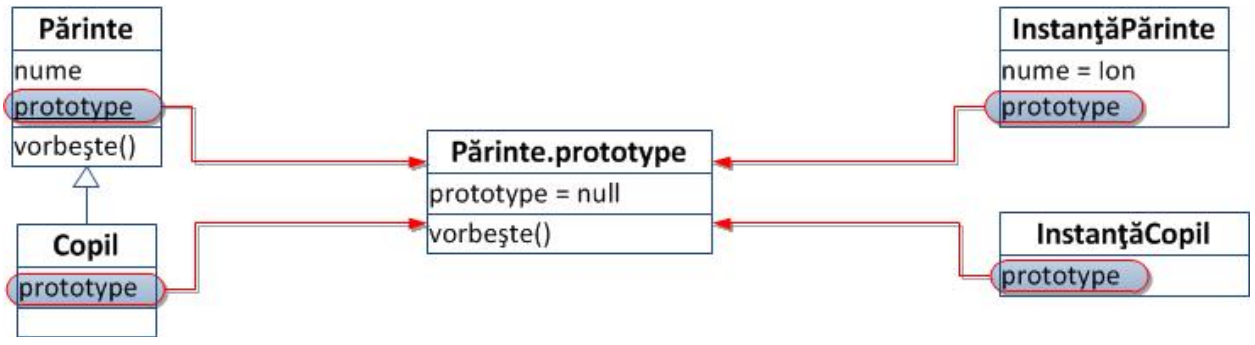



**Fig. 4.3 Prototip comun după (Stefanov, JavaScript Patterns 2010, 125) cu adăugări**

De reţinut este că, în acest caz, moştenirea se aplică doar în ceea ce priveşte prototipul – membrii contextului *this* al constructorului părinte sunt excluşi după cum rezultă şi din figura precedentă. Vom dezvolta această caracteristică în prezentarea următoarei tehnici.

Ultima metodă *clasică* pe care o prezintă Stefanov este cea a utilizării unui ***constructor temporar***. Am putea considera această tehnică o dezvoltare a precedentei prin aplicarea unui *proxy de protecţie*[1] al prototipului original. Acesta rezolvă problema modificării prototipului. De

[1] Vezi (Gamma, și alții 1995, 207-217, 208)

altfel, Stefanov precizează că tehnica este denumită adesea şi *funcţie proxy* sau *constructor proxy*.

Remarcăm că abordarea este similară primei, însă, în acest caz, legătura între prototipul instanţelor *Copil* şi cel al obiectelor *Părinte* este realizată printr-un *ProxyPrototip*. Astfel, prototipul obiectelor *ProxyPrototip* reprezintă o copie a proprietăţii clasei *Părinte* iar modificarea sa nu va avea efect asupra ierarhiei de derivare aşa cum se întâmplă în cazul precedent. Din punctul de vedere al comportamentului în cazul apelului metodei *vorbeşte*, menţionăm că acesta este similar cu primul model de utilizare a moştenirii clasice în JavaScript.

Implementarea acestei tehnici constă, după cum precizează Stefanov, în specificarea unei funcţii de *moştenire* cu parametri *clasăCopil* şi *clasăPărinte* care defineşte în scopul propriu o funcţie vidă reprezentând ***constructorul proxy*** – ***ProxyPrototip*** în figura de mai jos.

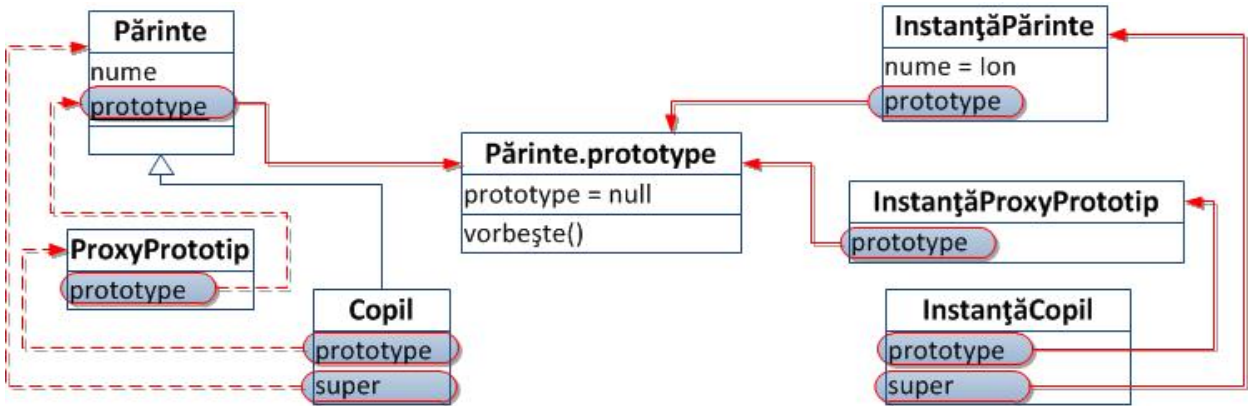



**Fig. 4.4 Constructor temporar, după (Stefanov, JavaScript Patterns 2010, 126) cu adăugări şi modificări**

Prototipul *constructorului proxy* este iniţializat cu cel al clasei părinte, iar cel al clasei *Copil* referenţiază o instanţă *ProxyPrototip*. O notă importantă legată de implementarea acestui model este reiniţializarea atributului *constructor* al prototipului subclasei cu aceasta însăşi. În javaScript atributul are un rol în principal informativ. Acest pas este util mai ales în vederea asigurării posibilităţii introspecţiei în timpul execuţiei.

Din punctul de vedere al legăturii cu o instanţă a părintelui, observăm că, după cum am arătat şi în figura de mai sus, aceasta este realizată prin adăugarea unui atribut referinţă în clasa derivată. Menţionăm că utilizarea denumirii din figură nu este posibilă în JavaScript întrucât *super* face parte din setul de cuvinte rezervate. Am ales această denumire pentru a păstra tonul specific altor limbaje orientate pe obiecte.

De asemenea, Stefanov prezintă o optimizare posibilă a implementării. Astfel, se creează o singură instanţă de tip *ProxyPrototip* păstrată într-o închidere de nivel global care returnează funcţia de moştenire ca *funcţie imediată*. Prin urmare, evităm crearea unei instanţe *ProxyPrototip* de fiecare dată când este apelată funcţia de moştenire, fiind suficientă schimbarea prototipului obiectului deja creat.

Revenind la discuţia despre moştenirea membrilor proprii superclasei – proprietăţile definite în contextul *this* al părintelui –, reţinem recomandarea lui Stefanov de a menţine componentele reutilizabile în prototip. După cum sugerează şi figura de mai sus, observăm că această tehnică nu presupune moştenirea membrilor proprii ai superclasei.

Aşa cum am precizat în secţiunea referitoare la metoda *klass*, implementarea conceptului de clasă în JavaScript poate să fie extinsă prin includerea uneia dintre tehnicile de moştenire clasică prezentate.

### 4.1.2.3. *Prezentare comparativă a tehnicilor de moştenire clasică în JavaScript*

Conform expunerii lui Stefanov [1] şi observaţiilor din secţiunea precedentă, propunem următoarea vedere de ansamblu a tehnicilor *clasice* de implementare şi utilizare a moştenirii în JavaScript :

| | (1) Modelul implicit | (2) Împrumutarea constructorului | (3) Împrumutarea şi stabilirea prototipului |
|---|---|---|---|
| Avantaje | ✓ Implementare simplă | ✓ Suportă parametrizarea | ✓ Suportă parametrizarea |
| | ✓ Mod de implementare similar comportamentului din alte limbaje OO | ✓ Suportă moştenirea multiplă | ✓ Moştenire prin copii ale membrilor proprii părintelui şi referinţe ale atributelor prototipului |
| | | ✓ Copierea proprietăţilor instanţei părinte elimină riscul suprascrierii acestora | |
| Dezavantaje | Moştenirea atributelor proprii, de obicei specifice clasei şi nefiind reutilizabile | Nu moşteneşte prototipul, ci doar atributele contextului *this* | Se poate dovedi ineficient din cauza nevoii de instanţiere dublă a părintelui |
| | Imposibilitatea parametrizării apelului de extindere a superclasei | | |
| | Lanţ prototipic ineficient | | |

**Tabelul 4.1 Prezentare comparativă a modurilor *clasice* de abordare a conceptului de moştenire în JavaScript**

| | (4) Împărţirea pro-totipului | (5) Constructor temporar |
|---|---|---|
| Avantaje | ✓ Nu necesită instanţierea părintelui | ✓ Permite modificarea prototipului |
| | ✓ Parcurgere rapidă a lanţului prototipic | ✓ Păstrează lanţul prototipic intact |
| | ✓ Sunt moşteniţi doar membrii prototipului | ✓ Permite legătura cu o instanţă a superclasei fără a moşteni atributele proprii ale acesteia |
| Dezavantaje | Imposibilitatea modificării prototipului subclaselor | |
| | Efecte asupra tuturor claselor din ierarhia de derivare în cazul modificării prototipului | |

**Tabelul 4.2 Prezentare comparativă a modurilor clasice de abordare a conceptului de moştenire în JavaScript (continuare)**

[1] (Stefanov, JavaScript Patterns 2010, 117-127)

#### 4.1.2.4. Moştenirea prototipică

Moştenirea prototipică este o altă variantă de implementare şi utilizare a conceptului în JavaScript. Bibliografia distinge în general clar această metodă de cele prezentate mai sus[1] : Harmes şi Diaz consideră că *moştenirea prototipică este o fiară cu totul diferită*[2]. Lucrarea de faţă se delimitează parţial de această abordare a subiectului, având în vedere că conceptul de *moştenire* în sine rămâne neschimbat, în materie de implementare există o serie de asemănări – în special cu metoda ***constructorului temporar*** **–**, iar trăsătura fundamentală de limbaj pe care această variantă o exploatează este, ca şi în cazurile precedente, aceeaşi, şi anume prototipul obiectelor JavaScript şi înlănţuirea acestora. Menţionăm că există totuşi anumite diferenţe care ţin în special de comportamentul motorului JavaScript şi de eficienţa utilizării resurselor. În secţiunea următoare vom prezenta o comparaţie amplă a celor două categorii de implementare a moştenirii.

Harmes şi Diaz sintetizează esenţa acestei abordări ca fiind ***utilizarea unui obiect JavaScript cu rol de prototip pentru toate instanţele sale viitoare***. În acelaşi sens, Stefanov arată că mecanismul constă în ***crearea de obiecte prin reutilizarea unuia deja existent a cărui funcţionalitate va fi moştenită de cele noi***. Pe de altă parte, Crockford[3] arată că, de fapt, avem de a face cu un model de ***moştenire diferenţială***[4]. Astfel, ***specializarea unui obiect este realizată prin personalizarea acestuia, arătând diferenţele faţă de baza din care a fost creat***[5].

Având în vedere că, utilizând acest model, un obiect poate să îşi moştenească funcţionalitatea de la un alt obiect, în această secţiune vom folosi termenul de *obiect prototip*[6] propus de Harmes şi Diaz pentru a ne referi la obiectele cu rol de *clasă*.

Din punctul de vedere al implementării, abordarea este similară cu metoda *constructorului temporar* prezentată mai sus. Astfel, implementarea *moştenirii prototipice* presupune definirea unei funcţii care primeşte obiectul de extins ca parametru. Corpul acesteia va cuprinde, similar cu metoda *proxy*, definirea unei funcţii vide, iniţializarea atributului *prototype* al acesteia cu prototipul părinte. În final se returnează o instanţă nouă a obiectului de tip *Function* cu rol de *proxy*. Pentru clarificare, propunem următorul fragment de cod bazat pe exemplele oferite de Stefanov[7] pe de o parte şi, pe de altă parte, de Harmes şi Diaz[8] cu menţiunea modificării denumirilor utilizate în scopul optimizării expresivităţii :

```
function mostenirePrototipica(obiectParinte) {
    function Proxy() {}
    Proxy.prototype = obiectParinte;
    return new Proxy();
}
```

Observăm asemănarea cu metoda utilizării unui *constructor temporar*, menţionând că diferenţa constă în definirea funcţiei *Proxy* la nivel global şi reducerea argumentului reprezentând clasa derivată. În continuare, este evidentă posibilitatea îmbunătăţirii implementării

[1] Vezi (Stefanov, JavaScript Patterns 2010, 130-133) şi (Harmes şi Diaz 2007, 45-49)

[2] (Harmes şi Diaz 2007, 45)

[3] Flanagan arată că Douglas Crockford este considerat în general a fi primul în a propune această abordare (JavaScript The Deifinitive Guide 2011, 119)

[4] (Crockford 2008, 51)

[5] Ibid (51)

[6] (Harmes şi Diaz 2007, 45) - Harmes şi Diaz arată că prin termenul ***obiect prototip*** nu ne referim la membrul *prototype* al unui obiect, ci la o entitate cu rol de şablon pentru cele pe care le vom crea pe baza acestuia.

[7] (Stefanov, JavaScript Patterns 2010, 131)

[8] (Harmes şi Diaz 2007, 46)

prin aplicarea unei închideri aşa cum s-a arătat în cazul metodei *constructorului temporar* pentru definirea unică a funcţiei *Proxy* :

```
var mostenirePrototipica2 = (function(){
    var Proxy = function(){},
        functieExtindere = function(obj){
            Proxy.prototype = obj;
            return new Proxy();
        };
    return functieExtindere;
})();
```

Pentru cunoaşterea proprietăţilor comportamentale de detaliu ale moştenirii prin prototipuri propunem analiza structurii de obiecte prezentate mai jos şi a rezultatelor obţinute în mod experimental într-o consolă JavaScript, menţionând că secvenţele de cod aferente se regăsesc în anexa 3 şi au la bază expunerea surselor bibliografice.

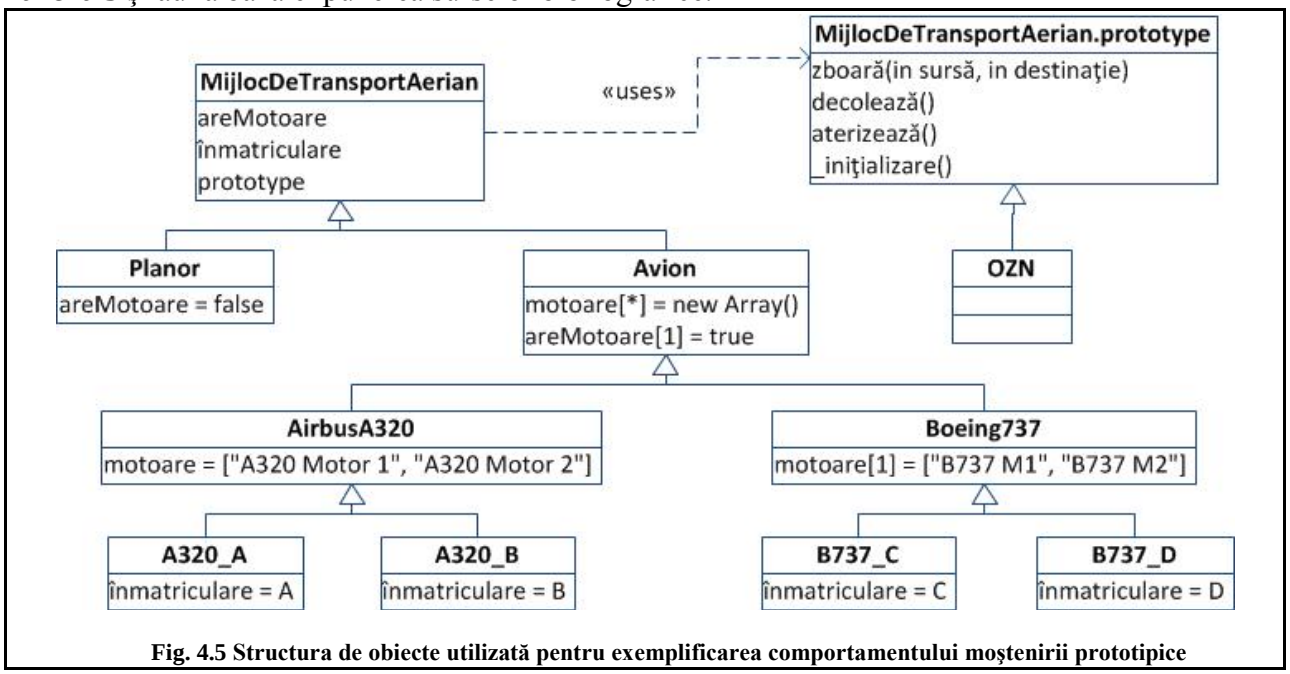



**Fig. 4.5 Structura de obiecte utilizată pentru exemplificarea comportamentului moştenirii prototipice**

Notăm că, în figura propusă, prezentăm ierarhia de obiecte care dorim să rezulte în urma apelurilor repetate ale funcţiei *moştenirePrototipică2*. Astfel, pornind de la un obiect *MijlocDeTransportAerian* al cărui prototip defineşte metodele *decolează()*, *zboară()* şi *aterizează()* specializăm entităţile *Planor*, *Avion* şi *OZN*. Tipul *Avion* este specializat la rândul său prin obiectele prototip *AirbusA320* şi *Boeing737*. În continuare, obiectele concrete *A320_A*, *A320_B*, *B737_C* şi *B737_D* sunt create pe baza prototipurilor de la nivelul superior. Observăm că, în ceea ce priveşte membrii obiectelor menţionate şi valoarea acestora, *Avion.areMotoare* este *true*, iar *Planor.areMotoare* este *false*. În acelaşi timp, *Avion* introduce câmpul *motoare* de tip *Array*. În ceea ce priveşte OZN, acesta nu moşteneşte decât metodele prototipului – vom reveni mai jos asupra acestui obiect. La următorul nivel, *AirbusA320* şi *Boeing737* iniţializează fiecare câmpul *motoare* cu valorile specifice prezentate în figură. Ultima treaptă a ierarhiei conţine obiectele concrete *A320_A*, *A320_B*, *B737_C* şi *B737_D* care iniţializează la rândul lor câmpul *înmatriculare* cu valorile din figură.

După cum vom observa mai jos, un prim subiect care trebuie tratat cu atenţie deosebită este legat de modul de transmitere a datelor în JavaScript. Amintim că tipurile primitive sunt transmise prin valoare, în timp ce tipurile care au la bază *Object* sunt transmise prin referinţă –

orice structură de date care nu este primitivă este descentă a *Object*. În continuare vom prezenta comportamentul moştenirii prototipice în cazul membrilor primitivi şi în cazul compoziţiei cu date de tip descendent al *Object* – un exemplu ar fi *Array*.

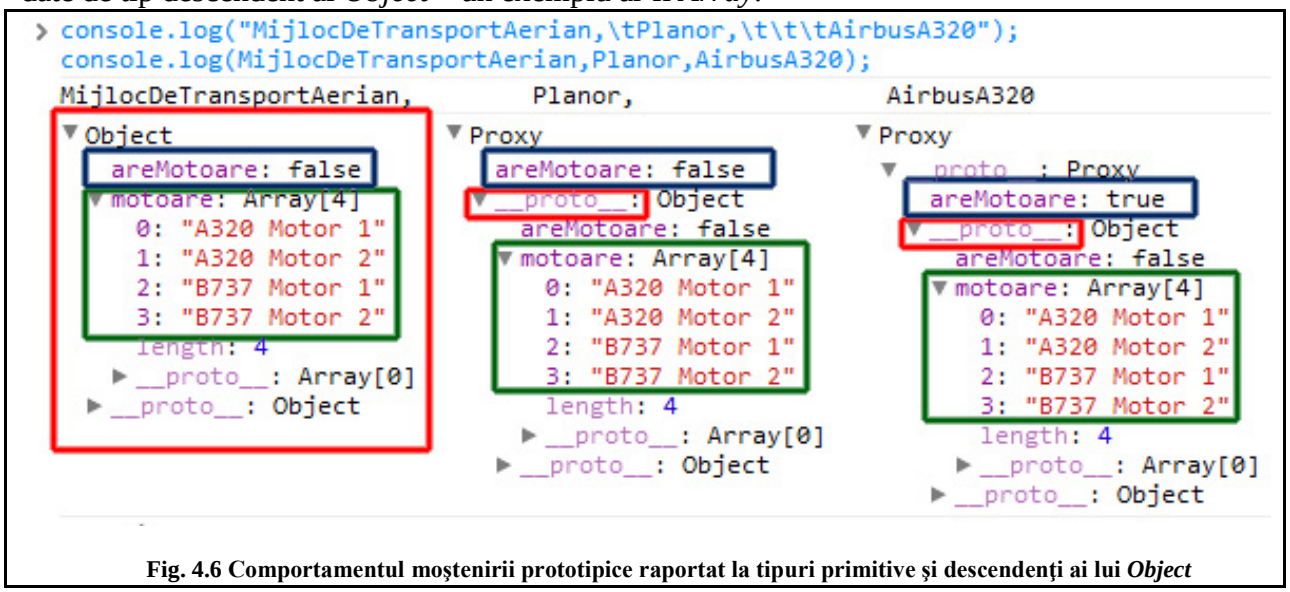



**Fig. 4.6 Comportamentul moştenirii prototipice raportat la tipuri primitive şi descendenţi ai lui *Object***

Figura de mai sus reprezintă rezultatul rulării primului caz de test din anexa 3. Remarcăm lanţul prototipic creat, evidenţiat cu chenare roşii. Astfel, membrii *__proto__*[1] ai obiectelor *Planor* respectiv *AirbusA320* puşi în evidenţă sunt referinţe spre *MijlocDeTransportAerian*. În consecinţă, făcând legătura cu cele precizate despre tipurile de date JavaScript, observăm că iniţializarea câmpului *areMotoare* – încercuit cu albastru – funcţionează corect şi conform diagramei prezentate în figura 3.5. Atributul *areMotoare* conţine date de tip *boolean*, deci primitive. Pe de altă parte, în cazul câmpului *motoare* care este de tip *Array* – descendent al *Object* – apelul metodei *push* pentru inserarea unui element nou are loc asupra instanţei definite în *MijlocDeTransportAerian*.

Diaz şi Harmes explică acest comportament prin ***citirea şi scrierea asimetrică a membrilor moşteniţi***[2]. Astfel, datorită modului de funcţionare a lanţului prototipic, membrii obiectelor *Avion*, ***Planor***, ***AirbusA320*** sau *B737_C* reprezintă iniţial legături spre cei ai prototipului (în cazul de faţă, *MijlocDeTransportAerian*). După cum putem observa şi în figura de mai sus, atributul *areMotoare* al obiectelor *Planor* şi *AirbusA320* este unul propriu definit prin iniţializarea sa în cazul de test. În consecinţă, atributul propriu obscurează valoarea proprietăţii prototipului. Mai mult, orice căutare în lanţul prototipic se va opri la întâlnirea atributului propriu. Pe de altă parte, atributul *motoare* nu a fost reiniţializat, ci doar modificat, ceea ce a dus la reprezentarea eronată a datelor. Am putea concluziona că problemele adresate de acest caz sunt similare cu cele ale clonării superficiale. Astfel, deducem că o soluţie imediată pentru moştenirea prototipică a obiectelor create prin compoziţie este clonarea adâncă. Implementarea acesteia este prezentată în cazul al doilea din anexa 3. Harmes şi Diaz recomandă iniţializarea membrilor care nu sunt de tip primitiv ai obiectelor create prin compoziţie utilizând o metodă definită în cadrul prototipului de extins. Avantajul abordării printr-o funcţie de iniţializare este decuplarea obiectului prototip de cel derivat.

[1] *__proto__* este denumirea dată atributului *prototype* de motoarele JavaScript ale unor navigatoare Web. Pentru testare s-a utilizat consola JavaScript a navigatorului Google Chrome versiunea 20.0.1132.57 care foloseşte denumirea *__proto__*.

[2] (Harmes şi Diaz 2007, 46-48)

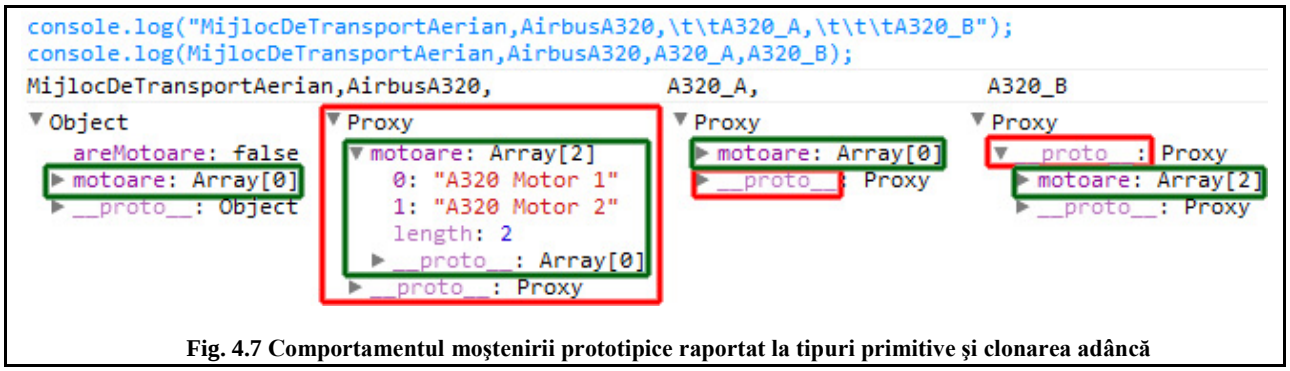



**Fig. 4.7 Comportamentul moştenirii prototipice raportat la tipuri primitive şi clonarea adâncă**

După cum putem observa în figura de mai sus, în acest caz reprezentarea este corectă : câmpul *motoare* al obiectului *MijlocDeTransportAerian* este o matrice fără elemente, în timp ce în derivatul *AirbusA320* valorile corespund diagramei prezentate la început. În ceea ce priveşte obiectele create pe baza prototipului *AirbusA320*, *A320_A* a fost creat cu iniţializarea câmpului *motoare* iar *A320_B* fără. În consecinţă, *A320_A* conţine un membru *motoare* propriu, în timp ce modificările afectuate asupra *A320_B.motoare* vor afecta câmpul definit în obiectul *AirbusA320*.

Revenind la subiectul *constructorului temporar*, amintim regula de bază exprimată în lucrarea lui Stefanov de a delimita elementele reutilizabile de cele specifice prin definirea celor generale în cadrul prototipului – ne referim aici la atributul *prototype* – şi a celorlalte ca atribute proprii.

În acest sens vom prezenta cazul obiectului *OZN* care este conform diagramei un derivat direct din prototipul lui *MijlocDeTransportAerian*. Varianta utilizării atributului *prototype* al unui obiect ca sursă de derivare este prezentată, pe bună dreptate, de către Stefanov ca fiind o metodă aparte de utilizare a *moştenirii prototipice*[1]. În figura de mai jos prezentăm starea obiectului *OZN* în ambele situaţii de test discutate până acum.

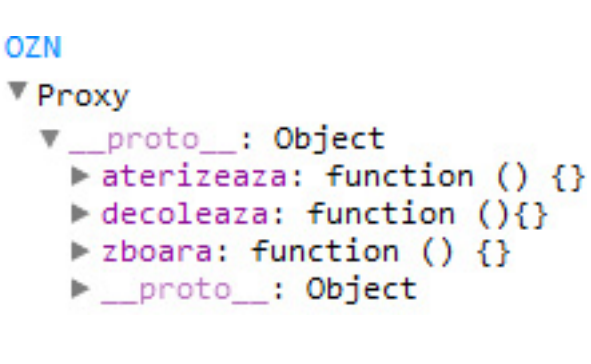



**Fig. 4.8 Starea obiectului OZN după executarea primului caz de test**

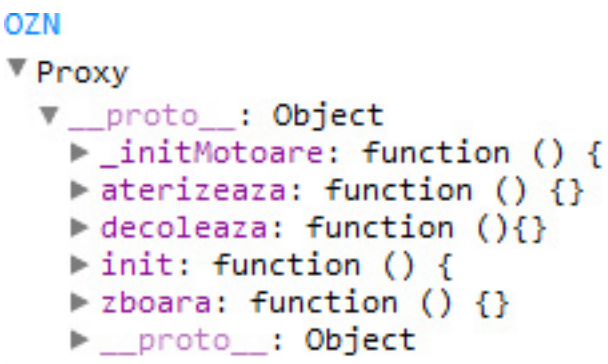



**Fig. 4.9 Starea obiectului OZN după executarea celui de-al doilea caz de test**

Astfel, observăm că în ambele cazuri, obiectul nu este afectat de alterarea câmpului *motoare* însă moşteneşte toţi membrii definiţi la nivelul *prototype* al *MijlocDeTransportAerian,* adică metodele *aterizeaza, decoleaza* şi *zboara*. Menţionăm că în ambele situaţii, câmpul *motoare* ar putea fi iniţializat ulterior, dacă este necesar, prin apelul metodei *_initMotoare* fără a avea efecte asupra restului ierarhiei de obiecte.

[1] Vezi (Stefanov, JavaScript Patterns 2010, 132)

O altă observaţie importantă este legată de obiectele definite în sintaxa literal. Notăm că omiterea iniţializării câmpului *prototype* cu un obiect gol duce la modificarea prototipului *Object*. Prin urmare, aceasta are efecte asupra întregii ierarhii de obiecte definite în motorul JavaScript la un moment dat. În ceea ce priveşte cazurile de test din anexa 3, menţionăm că s-a utilizat sintaxa *literal* pentru simplitate, însă este posibilă instanţierea obiectelor şi prin funcţii constructor.

Din punctul de vedere al standardizării este esenţial de reţinut că metoda *moştenirii prototipice* este parte a specificaţiei ECMA-262 începând cu varianta a cincea a acesteia fiind definită de metoda *Object.create*[1]. Consultând standardul, rezultă că implementarea prezentată corespunde cu specificaţia metodei *create*. O particularitate interesantă a metodei statice *create* este al doilea parametru pe care aceasta îl acceptă – primul fiind obiectul prototip de extins. Astfel, pe lângă baza care se doreşte a fi specializată, funcţia acceptă şi un obiect reprezentând o structură de date ale cărei elemente vor fi copiate ca *atribute proprii* în instanţa rezultat. Menţionăm că al doilea argument al metodei *create* este opţional şi, conform ECMA-262, dacă, acesta este prezent şi definit – diferit de *undefined* – efectul este cel al apelului *Object.defineProperties*. În concluzie, având în vedere cele precizate mai sus despre moştenirea prototipică şi utilizarea acesteia în combinaţie cu copierea proprietăţilor, observăm că acesta este şi comportamentul metodei statice *Object.create* în cazul specificării celui de al doilea argument.

În cele ce urmează, propunem o comparaţie obiectivă, neutră şi de sinteză a abordărilor conceptului de moştenire dezbătute până acum : *prototipică* şi *clasică*. Aceasta se bazează pe sursele bibliografice citate în dezvoltarea subiectului, prezentarea comparativă din lucrarea *JavaScript Design Patterns*[2] şi pe observaţii proprii.

| | **Moştenire prototipică** | **Moştenire *clasică*** |
|---|---|---|
| **Relaţii între** | Obiecte instanţă sau obiecte prototip | Clase (funcţii constructor) |
| **Implementare** | Similară, prin utilizarea părintelui ca prototip : iniţializarea atributului *prototype* al derivatului cu o referinţă spre entitatea generală şi crearea unui lanţ prototipic asemănător cel al constructorilor în alte limbaje de programare | |
| **Standardizare** | ECMA-262 versiunea 5 | Nespecificată |
| **Avantaje şi diferenţe** | ✓ Defineşte un sistem ierarhic de **obiecte** care pot fi instanţe sau tipuri de date | ✓ Defineşte un sistem ierarhic de **tipuri de date** |
| | ✓ Utilizarea mai eficientă a resurselor : iniţializează un câmp al instanţei derivate doar dacă este necesar şi există diferenţe faţă de părinte | ✓ Notorietate conceptuală mai mare prin implementarea în limbaje de programare precum Java sau C++ |
| | ✓ În acelaşi sens al utilizării mai bune a memoriei, păstrează o singură instanţă pentru metodele pe care le defineşte un obiect prototip, aceasta fiind utilizată de toate derivatele care nu o suprascriu sau supraîncarcă | ✓ Mai uşor de utilizat pentru specialişti cu o bază solidă în limbaje de programare cu o implementare nativă a conceptului de *clasă* şi cunoştiinţe minime de JavaScript |
| | ✓ Poate oferi suport pentru moştenirea stării unei instanţe | |
| | ✓ Concept de bază simplu | |

**Tabelul 4.3 Prezentare comparativă : Moştenirea prototipică - moştenirea clasică**

[1] Vezi (Ecma International 2011, 113) şi (Stefanov, JavaScript Patterns 2010, 132-133)
[2] (Harmes şi Diaz 2007, 49)

### 4.1.2.5. Încapsularea în contextul moştenirii : Moştenire funcţională (parazitică)

Douglas Crockford prezintă în *JavaScript : The Good Parts* un model de implementare pe care îl numeşte *funcţional* [1] (*Functional*). De asemenea, preluând modelul lui Crockford, Stefanov abordează subiectul în *Object-Oriented JavaScript* [2] într-o manieră pe care o considerăm similară în punctele esenţiale.

Conform lui Crockford, algoritmul acestui procedeu constă în patru paşi pe care îi vom discuta mai jos :

1. Crearea unui obiect
2. Specificarea opţională a variabilelor şi metodelor **private** ale instanţei
3. Augmentarea obiectului creat cu *metode privilegiate*[3]
4. Returnarea obiectului

Stefanov consideră că principiul fundamental al acestei tehnici este definirea unei funcţii de creare a obiectelor care, respectând paşii enunţaţi mai sus, preia funcţionalitatea unui obiect, îl extinde şi îl returnează.

Din analiza structurii procedeului deducem că modelul *parazitic* este un increment al celor prezentate mai sus. În acest sens, crearea obiectului de extins se poate realiza prin oricare dintre metodele de creare şi specializare prezentate mai sus – fie printr-o funcţie constructor, prin aplicarea unei metode *clasice*, *prototipice*, fie prin simpla copiere a proprietăţilor.

Considerăm că introducerea variabilelor şi a metodelor private – după cum rezultă din pasul doi – reprezintă cel mai mare câştig al acestui mod de abordare.

Pentru înţelegerea detaliilor de implementare a acestei tehnici considerăm util a urmări şi explica şablonul de pseudocod propus de Crockford pentru realizarea unui *constructor funcţional*.

| Pseudocod | Pas şi explicaţii |
|---|---|
| `var constructor = function(argInst, argPriv) { var obiectDeExtins /* alte variabile private */; argPriv = argPriv \|\| {};` | Declararea şi iniţializarea funcţiei astfel : ***argInst*** reprezintă valorile membrilor obiectului de extins – toate informaţiile necesare constructorului pentru crearea unui obiect după cum arată Crockford, ***argPriv*** conţine atributele private. După cum se poate observa, *argPriv* este opţional, fiind iniţializat cu un obiect vid în cazul în care nu este specificat. |
| Pasul 2 **: Declararea, iniţializarea variabilelor şi specificarea funcţiilor private. Astfel, variabilele şi metodele declarate în interiorul constructorului reprezintă membri privaţi.** | |
| `obiectDeExtins = /* creare */ ;` | **Pasul 1** : Crearea obiectului de extins |
| Pasul 3 **: Augmentarea obiectului de extins cu *metode privilegiate*** | |
| `return obiectDeExtins }` | **Pasul 4 :** Returnarea obiectului extins |

**Tabelul 4.4 Detalii de implementare a *moştenirii funcţionale***

[1] Vezi (Crockford 2008, 52-55).

Nu considerăm că numele este o trimitere directă spre paradigma programării funcţionale, ci descriptiv pentru modul de creare a obiectelor – prin definirea unei funcţii.

[2] (Stefanov, Object-Orieted JavaScript 2008, 190-191)

[3] Douglas Crockford defineşte *metodele privilegiate* ca fiind interfaţa unui obiect având acces la variabilele private ale acestuia.

În primă instanţă, observăm inversarea primilor doi paşi descrişi mai sus. Cauza este recomandarea generală de a defini toate variabilele din scopul unei funcţii la începutul acesteia.

Revenind la discuţia legată de încapsulare în contextul moştenirii, după cum am precizat mai sus, abordarea *parazitică* introduce conceptul membrilor privaţi. Astfel, toate variabilele şi funcţiile declarate în corpul funcţiei *constructor* – amintim că în JavaScript funcţiile reprezintă obiecte şi prin urmare pot fi referenţiate printr-o variabilă – se află în scopul *constructor*. De asemenea, superclasa (*obiectDeExtins*) se află tot în scopul funcţiei *constructor*. Făcând legătura cu cele precizate mai sus referitor la mediul specific unei funcţii (scopul), deducem că variabilele dintr-un anumit scop sunt vizibile doar din interiorul acestuia. Prin urmare, variabilele şi funcţiile din scopul *constructor* sunt vizibile doar în cadrul acestuia şi sunt ***private*.**

O notă importantă pe care o adaugă Crockford este definirea tuturor metodelor ca fiind private în primă instanţă şi referenţierea lor prin membrii privilegiaţi ai obiectului returnat. Conform lui Crockford, avantajul utilizării acestui stil este că, dacă metoda publică a obiectului este modificată, restul funcţiilor cu acces la definiţia privată nu vor fi afectate.

În concluzie, Crockford arată că puterea acestei abordări stă în flexibilitate, efortul de implementare scăzut şi în posibilitatea utilizării încapsulării şi a ascunderii informaţiilor. După cum precizează Crockford, *moştenirea funcţională* este o metodă de creare a obiectelor durabile care nu pot fi compromise şi sunt compuse dintr-un set de metode denumite *capabilităţi* restricţionând accesul la starea interioară peste nivelul de informaţii publice[1].

**Problema mutabilităţii obiectelor** este un alt subiect important, strâns legat de încapsulare în contextul moştenirii. Aşa cum am arătat la începutul lucrării, având în vedere caracterul dinamic al limbajului, toate obiectele JavaScript pot fi modificate după crearea lor : este posibilă adăugarea sau ştergerea unor proprietăţi şi, mai mult decât atât, alterarea metodelor.

Pentru tratarea acestei probleme, standardul ECMAScript 5 introduce o serie de funcţionalităţi deosebit de importante şi de utile pe care Flanagan le prezintă în *JavaScript : The Definitive Guide*[2].

În ceea ce priveşte definirea claselor imuabile, Flanagan prezintă utilizarea metodelor *Object.**defineProperty*** şi *Object.**defineProperties***.[3] După cum am precizat mai sus, acestea pot fi utilizate pentru definirea sau modificarea atributelor unui obiect, acceptând ca parametri instanţa asupra căreia se acţionează şi descriptorul de proprietate al membrilor de adăugat sau alterat. Un **descriptor de proprietate** (*property descriptor*) este un obiect în care se specifică în mod obligatoriu numele membrului şi, opţional politica de enumerabilitate, inscriptibilitate şi configurabilitate aplicată acestuia, cât şi *metodele accesoare* în cazul atributelor. Vom reveni asupra proprietăţilor speficificatorilor de proprietăţi în tratarea încapsulării. Pentru moment reţinem că definirea politicii de inscriptibilitate a membrilor este esenţială în ceea ce priveşte utilizarea moştenirii în JavaScript. În acest sens, observăm că restricţionarea inscriptibilităţii şi configurabilităţii unui membru – în special în ceea ce priveşte metodele – are în JavaScript un efect similar cu utilizarea modificatorului *final* în Java : se reduce riscul înlocuirii prin subclasare a comportamentului definit la nivelul părinte.

[1] (Crockford 2008, 55)

[2] Vezi (Flanagan 2011, 238-245).

Flanagan prezintă în subcapitolul 9.8 *Clase în ECMAScript 5* pe larg funcţionalitatea similară modificatorilor de acces din alte limbaje de programare orientate pe obiecte, introdusă în standard. Având în vedere subiectul discutat în această secţiune a lucrării, ne vom concentra asupra imuabilităţii şi prevenirii extinderii claselor, menţionând că vom prezenta celelalte detalii ale expunerii lui Flanagan în prezentarea utilizării încapsulării în JavaScript.

[3] Sepcificaţia metodelor se poate consulta în (Ecma International 2011, 113)

Flanagan consideră că, chiar dacă metodele prezentate mai sus sunt extrem de utile şi puternice, descriptorii de proprietăţi necesari acestora duc la reducerea lizibilităţii codului şi propune în consecinţă două funcţii utilitare exploatând reprezentarea de tip tabelă de dispersie (*hashmap*) a obiectelor JavaScript[1]. Astfel, prima funcţie, denumită de către Flanagan expresiv *îngheaţăProprietăţi* are rolul de a restricţiona inscriptibilitatea şi configurabilitatea membrilor unui obiect – notăm că aceasta acţionează doar asupra celor numiţi în apelul său sau în mod general pentru toate proprietăţile. În aceeaşi manieră, cea de a doua, utilitară denumită *ascundeProprietăţi*, restricţionează enumerabilitatea.

**Controlul extinderii unei clase** în ECMAScript 5 are la bază funcţionalitatea prezentată mai sus – în special descriptorii de proprietăţi. Remarcăm că Flanagan numeşte procedeul prevenirea existenderii de clasă, însă considerăm termenul control mai adecvat. Pentru a ilustra modul în care poate fi controlată extensibilitatea unei clase, Flanagan porneşte de la metoda *Object.preventExtensions*[2]. Apelul acesteia rezultă în restricţionarea adăugării de noi membri obiectului în cauză. În continuare, Flanagan arată că metodele[3] *Object.seal* şi *Object.freeze* reprezintă o alternativă standardizată pentru funcţiile utilitare prezentate mai sus. Astfel, *seal* poate fi considerată o extindere a metodei *preventExtensions* având aceleaşi efecte şi, mai mult, restricţionează posibilitatea de configurare a tuturor proprietăţilor obiectului în cauză. În consecinţă, membrii unui obiect *sigilat* – asupra căruia a fost apelată metoda *seal* – nu mai pot fi şterşi prin apeluri de tip *delete(obiect.proprietate)*. La rândul său, *freeze* este o extensie a funcţionalităţii lui *seal*. În consecinţă, pe lângă limitarea extinderii şi a configurării proprietăţilor, *freeze* restrânge şi inscriptibilitatea acestora.

Reţinem că, în cazul restrângerii inscriptibilităţii unui membru al unei clase ale cărei instanţe au rol de prototip pentru derivate, tentativa de definire a acesteia ca atribut propriu al subclasei va eşua. Aceasta este o caracteristică pe care Flanagan o consideră deosebit de importantă în ceea ce priveşte specificarea claselor JavaScript. Totuşi, pentru suprascrierea unui membru astfel definit, Flanagan aminteşte că este posibilă utilizarea metodelor de stabilire a descriptorilor de proprietate *defineProperty*, *defineProperties* sau *Object.create*.

În concluzie, menţionăm, după cum precizează şi Flanagan, că utilizarea descriptorilor de proprietăţi nu este o necesitate strictă, însă în anumite circumstanţe poate fi deosebit de utilă.

#### *4.1.2.6. Alte metode de reutilizare a codului. Clase prin părţi. Împrumutarea metodelor. Clase abstracte. Moştenire multiplă*

Bibliografia consideră că, spre deosebire de alte limbaje de programare, în JavaScript, moştenirea nu este unica modalitate de reutilizare a codului, arătând că adesea, datorită expresivităţii limbajului, există metode *mai elegante* şi mai eficiente pentru realizarea acestui scop. Harmes şi Diaz, Stefanov, Flanagan, dar mai ales Crockford prezintă alternative ale utilizării moştenirii stricte prin care, pentru specificarea claselor şi a obiectelor se practică combinarea diverselor componente. Dintre opţiunile de refolosire a codului, amintim conceptul de ***Mixin***[4] prezentat în *Object-Oriented JavaScript* şi în *JavaScript Patterns* de către Stefanov, dar şi în *Pro JavaScript Design Patterns*, ***împrumutarea metodelor***[5] abordată atât în lucrările menţionate, cât şi în *JavaScript : The Definitive Guide* de către Flanagan. În *JavaScript The*

[1] Vezi (Flanagan 2011, 240)
[2] Vezi şi specificaţia metodei în standardul ECMAScript 5 (Ecma International 2011, 114)
[3] Vezi specificaţia metodelor *Object.seal* şi *Object.freeze*, ibid.
[4] Vezi (Harmes şi Diaz 2007, 50-51) şi (Stefanov, JavaScript Patterns 2010, 135-136)
[5] Vezi (Flanagan 2011, 224-225) şi (Stefanov, JavaScript Patterns 2010, 136-139)

*Good Parts*, Crockford propune un model numit ***Parts***[1] pe care îl considerăm similar tehnicii de *împrumutare* pe care am amintit-o mai sus. De asemenea, pornind de la tehnicile enumerate mai sus, Flanagan propune un mod de implementare al conceptului de **clasă abstractă**[2].

Conceptul de ***Mixin*** nu este unul original JavaScript, însă în esenţă poate fi implementat în JavaScript ca o formă a moştenirii multiple. Stefanov arată că o abordare a conceptului poate fi dezvoltată pornind de la ideea de moştenire prin copierea proprietăţilor. Astfel, funcţia de extindere funcţionează, după cum arată Stefanov, la fel ca şi în cazul moştenirii prin copierea tuturor proprietăţilor unui obiect, dar acceptă un număr nelimitat de parametri reprezentând clase sau obiecte pe care le combină într-unul singur. Harmes şi Diaz propun o îmbunătăţire a acestei tehnici oferind posibilitatea specificării funcţionalităţii de copiat într-un nou parametru. Un exemplu de caz în care este recomandată abordarea prin *Mixin* ar fi, conform lui Harmes şi Diaz, o clasă generală pentru serializarea obiectelor. Metoda *serialize* poate fi implementată în JavaScript în mod general folosind reprezentarea de tip tabelă de dispersie a obiectelor prin parcurgerea proprietăţilor şi apelând metoda *toString* asupra acestora. Valabilitatea generală a acestei abordări de implementare a serializării face ca exemplul să fie ideal pentru utilizarea tehnicii *Mixin*. Harmes şi Diaz consideră că *aceasta este o modalitate uşoară de prevenire a duplicării codului, însă este utilă doar în cazul metodelor cu o implementare destul de generală pentru a fi utilizate în clase disparate*[3].

Tehnica ***împrumutării de metode*** este o altă posibilitate de organizare şi eficientizare a codului aferent unei aplicaţii. Flanagan consideră că *nu există nimic deosebit cu privire la metode în JavaScript, acestea fiind pur şi simplu funcţii atribuite proprietăţilor unui clase şi invocate „printr-o" sau „asupra" unei instanţe*[4]. Astfel, o funcţie poate reprezenta una sau mai multe metode şi, mai mult, poate fi utilizată ca metodă de clase diferite. De asemenea, Flanagan argumentează că *dintr-un punct de vedere prin prisma limbajelor de programare orietate pe obiecte, utilizarea metodelor unei clase ca metode ale alteia poate fi considerată a fi o formă de moştenire multiplă*[5].

Pe de altă parte, Stefanov arată că eficienţa împrumutării de metode devine evidentă în cazul nevoii de reutilizare a codului fără crearea unei relaţii de extindere de tip părinte – copil. Conform dispunerii sintetice pe care ne-o oferă Stefanov, reţinem că, în esenţă, *o instanţă a unei clase pretinde pentru moment că reprezintă o alta pentru a utiliza metodele celei din urmă*[6].

În ceea ce priveşte implementarea acestei idei la nivel de cod, Stefanov sugerează punerea în valoare a metodelor *call*[7] şi *apply*[8] specifice obiectelor de tip *Function*. O variantă mai simplă a aplicării practice a conceptului este prin asignarea unei funcţii de uz general unei proprietăţi aparţinând clasei specifice.

---

[1] Vezi (Crockford 2008, 55-57)

[2] Vezi (Flanagan 2011, 234-237)

Menţionăm că vom prezenta tehnicile de **moştenire multiplă** (***Mixin***), **împrumutare a metodelor**, utilizare şi implementare a modelului ***Parts*** şi a **claselor abstracte** utilizând ideile, conceptele şi conţinutul surselor bibliografice citate în notele 4, 5, 1 şi 2.

[3] (Harmes şi Diaz 2007, 51)

[4] (Flanagan 2011, 224)

[5] Ibid. (224)

[6] (Stefanov, JavaScript Patterns 2010, 136)

[7] Vezi (Ecma International 2011, 100,119) şi (Mozilla Developer Network Docs - call 2012). Conform documentaţiei, metoda apelează în contextul *this* primit ca prim parametru cu argumentele specificate individual.

[8] Vezi (Ecma International 2011, 119) şi (Mozilla Developer Network Docs - apply 2012). Metoda are un comportament identic cu cel al *call*, dar diferă prin specificarea argumentelor de transmis în formă de matrice.

Observăm că, la un nivel de complexitate ridicată, **stabilirea contextului de execuţie** al funcţiei împrumutate devine o problemă fundamentală. Un astfel de caz ar fi, după cum arată Stefanov[1], apelarea funcţiei *împrumutate* sub formă de *callback*. În această situaţie, conform lui Stefanov şi rezultatelor experimentale, contrar nevoilor obişnuite ale utilizării acestui procedeu – ne dorim în general ca scopul funcţiei să rămână instanţa prin care sau asupra căreia se realizează apelul –, obiectul global devine context de execuţie (*this*) al funcţiei împrumutate. Prin urmare, soluţia pe care o oferă Stefanov este utilizarea metodei *bind*[2]. Folosirea acesteia leagă contextul dorit într-un anumit caz de metoda apelată.

După cum am menţionat mai sus, putem considera că modelul *Parts* propus de Crockford este o variantă perfecţionată a aplicării tehnicii *împrumutării de metode*, având în vedere faptul că baza conceptuală a acestuia este tot ideea compunerii claselor şi a structurării funcţionalităţii prin ceea ce numim fragmente de cod cu aplicabilitate generală. Apreciem exemplul complex de ilustrare a aplicării modelului *Parts* ca fiind deosebit de practic pentru aprofundarea cunoştiinţelor şi înţelegerii problemelor în legătură cu reutilizarea funcţionalităţii prin *împrumutarea de metode*[3].

**Noţiunea claselor şi metodelor abstracte** reprezintă din punctul nostru de vedere unul dintre cele mai importante subiecte de aplicabilitate a reutilizării codului prin fragmente general valabile. Flanagan prezintă şi exemplifică pe larg tema implementării şi utilizării conceptului claselor abstracte în JavaScript, raportându-se la paradigma orientată pe obiecte[4]. Fără a aborda în detaliu ilustrarea complexă datorată lui Flanagan, găsim important a trata în cele ce urmează pe scurt o modalitate de introducere a noţiunii în JavaScript. Astfel, conform definiţiei teoretice a conceptului, o clasă abstractă nu poate fi instanţiată. În JavaScript această însuşire se traduce prin aruncarea unei erori în momentul unei încercări de instanţiere după cum arată şi Flanagan. O altă proprietate importantă este necesitatea existenţei uneia sau mai multor metode abstracte pe care clasele concrete derivate sunt obligate să le implementeze[5]. În consecinţă, Flanagan propune definirea unei funcţii globale cu rolul de a atenţiona asupra lipsei implementării metodei pe care o reprezintă – concret, *abstractMethod*[6] aruncă o eroare în cazul apelării. Punând în relaţie cele precizate până acum despre noţiunea de moştenire în JavaScript şi referitor la tehnica *împrumutării de metode*, concluzionăm că aceasta este o abordare viabilă a noţiunii de clasă abstractă aşa cum este definită în cadrul paradigmei orientate pe obiecte.

#### 4.1.2.7. Concluzii

Lipsa unei implementări native stricte a conceptului reutilizării şi organizării codului în structuri de clase ierarhice prin moştenire a permis apariţia şi dezvoltarea unor abordări variate a noţiunii. Având în vedere că bibliografia nu oferă în general o recomandare clară a modului de

[1] (Stefanov, JavaScript Patterns 2010, 138)

[2] Vezi (Ecma International 2011, 119-120) şi (Docs - bind : Mozilla Developer Network 2012).

Metoda *Function.prototype.bind* face parte din standardul JavaScript începând cu versiunea a cincea. Stefanov ilustrează în (JavaScript Patterns 2010, 138-139) o implementare cu acelaşi efect a acesteia pentru utilizarea în medii anterioare ECMAScript 5. Astfel, principiul de bază, de altfel similar celui specificat în standard, este returnarea unei funcţii – practic aceasta reprezintă o închidere – care apelează *apply* sau *call* utilizând scopul dorit – clasa sau obiectul – , specificat ca prim parametru şi argumentele de transmis funcţiei împrumutate, specificate în al doilea parametru.

[3] Vezi (Crockford 2008, 55-57)

[4] Vezi (Flanagan 2011, 235-237)

[5] Vom reveni asupra verificării implementării tuturor metodelor abstracte în secţiunea dedicată interfeţelor .

[6] (Flanagan 2011, 235)

tratare a moştenirii în JavaScript alegerea între o abordare *clasică* sau prototipică depinde în cea mai mare măsură de specificul aplicaţiei şi al funcţionalităţii cerute. Remarcăm totuşi tendinţa de standardizare a moştenirii prototipice prin includerea acesteia în specificaţia ECMAScript 5. După cum consideră şi Stefanov, studiul şi stăpânirea modelelor la care ne oprim – atât *clasice*, cât şi prototipice – contribuie la creşterea perfromanţei de întrebuinţare a limbajului, fiind un pas important în aprofundarea cunoştiinţelor de aplicare a noţiunilor de programare obiectuală în JavaScript.

În acelaşi timp, remarcăm că prin expresivitate şi, mai ales flexibilitatea dată de tipizarea slabă, obiectivul de bază – reutilizarea şi organizarea codului – poate fi satisfăcut prin alternative multiple a căror putere şi valoare este dată tocmai de simplitate.

### 4.1.3. Încapsularea şi ascunderea informaţiilor

#### *4.1.3.1. Principiu*

**Ascunderea informaţiei** este un principiu fundamental al paradigmei orientate pe obiecte şi se realizează prin **încapsulare**. Pentru a caracteriza esenţa acestuia, vom urmări exemplul propus de Joldoş în cursul de *Programare orientată pe obiecte*. Astfel, *un automobil încapsulează multă informaţie prin complexitatea construcţiei acestuia, dar nu este necesară cunoaşterea funcţionării pentru a-l conduce*[1]. Aplicând principiul ascunderii informaţiei asupra exemplului, observăm că *volanul şi schimbarea vitezelor constituie* ***interfaţa***[2] de utilizare în timp ce *motorul, transmisia sunt* ***implementarea (ascunsă)***[3]. Abstractizând exemplul de la care am pornit, înţelegem că în utilizarea unui obiect software este necesară cunoaşterea **capabilităţilor** acestuia însă nu şi a modului de implementare. În teorie, această relaţie se exprimă prin **vederile unei clase**. După cum precizează Joldoş[4], interfaţa reprezintă vederea **publică** *permiţând instanţelor să coopereze*, iar implementarea, pe cea **privată** formată din *proprietăţile care ajută capabilităţile să-şi îndeplinească sarcinile*. Prin urmare, principiul ascunderii informaţiei şi încapsularea garantează securitatea datelor şi contribuie tranşant la reducerea interdependenţei părţilor unui sistem prevăzând efectuarea comunicării prin ceea ce numim *canale bine definite*, adică interfeţele obiectelor[5].

Spre deosebire de alte limbaje de programare, în JavaScript nu există modificatori de acces – cum ar fi *private* în Java – prin care să poată fi pus în practică acest principiu. Având în vedere caracterul dinamic şi slab tipizat al limbajului, precum şi riscul real al reiniţializării eronate a variabilelor şi a membrilor unui obiect – atribute sau metode – considerăm imperativă utilizarea tehnicilor de încapsulare şi ignorarea acestora, lipsită de responsabilitate. În continuare, vom prezenta metodele prin care proprietăţile JavaScript pot fi utilizate în scopul îndeplinirii principiilor ilustrate.

#### *4.1.3.2. Încapsulare prin închideri*

Revenind la modelul moştenirii funcţionale propus de Crockford şi prezentat[6] mai sus, observăm că acesta oferă o soluţie generală de aplicare a principiului ascunderii informaţiei

[1] (Programare orientată pe obiecte : Note de curs, Cursul 1 2011, 48)
[2] Ibid. (48)
[3] Ibid. (48)
[4] Ibid. (49-52)
[5] (Harmes şi Diaz 2007, 26)
[6] Vezi secţiunea 4.1.2.5 - Încapsularea în contextul moştenirii : Moştenire funcţională (parazitică)

pentru programarea în JavaScript. Astfel, o funcţie constructor utilizată pentru specificarea unei clase reprezintă o închidere. Conform celor menţionate despre închideri, toate variabilele declarate în scopul unei funcţii sunt vizibile numai în interiorul acestuia şi sunt delimitate de mediul exterior aşa cum prevede şi principiul ascunderii informaţiei. Accesul la informaţia protejată a unei clase JavaScript este realizat prin ceea ce, în lipsa unei terminologii clare, Crockford numeşte **metode privilegiate** – acestea sunt publice şi au capacitatea expunerii membrilor privaţi în mod controlat.

În prezentarea problemelor pe care le pune încapsularea în JavaScript, Stefanov se opreşte în special asupra importanţei respectării ***principiului autorităţii minime*** *(Principle of Least Authority - POLA).* Acesta prevede reducerea informaţiilor din mesajele interschimbate de obiecte software la strictul necesar pentru îndeplinirea unui scop. De altfel, bibliografia de specialitate consideră satisfacerea *POLA* a fi o cerinţă elementară de securitate a programelor JavaScript. În consecinţă, Stefanov atrage atenţia asupra compoziţiei obiectelor şi recomandă[1] utilizarea metodelor accesoare pentru *structurile de date strict necesare unui consumator* în locul expunerii complete a unui membru prin interfaţa clasei. În acelaşi sens, amintim recomandarea implementării private (locale) şi expunerii prin metode publice a capabilităţilor unei clase, ilustrată în cadrul modelului moştenirii funcţionale[2].

Închiderile reprezintă baza principială şi pentru încapsularea obiectelor definite cu sintaxa literal. Pentru că în acest caz constructorul cu rolul de închidere de mai sus lipseşte, Stefanov arată că acelaşi comportament poate fi obţinut printr-o funcţie imediată. Utilitatea abordării prin funcţii imediate este relevată în situaţia nevoii de încapsulare a prototipului – acesta fiind adesea iniţializat printr-un obiect vid definit în sintaxă literal.

Harmes şi Diaz documentează[3] o abordare simplistă a subiectului bazată pe utilizarea unei **convenţii de denumire** a membrilor privaţi ai unei clase. Lucrarea de faţă se delimitează de pseudoîncapsularea prin convenţii de nume, având în vedere că aceasta nu are niciun efect practic asupra programelor. Mai mult, Crockford consideră că folosirea convenţiilor de nume este *un model de ascundere imaginară a informaţiilor adoptat în necunoştiinţă de cauză*[4]. Astfel, *în cazul necesităţii protejării unei proprietăţi prin acces privat, aceasta este denumită bizar* – de exemplu prin adăugarea unui caracter de subliniere la început – *în speranţa că alţi utilizatori ai codului se vor preface la rândul lor că nu o pot vedea*[5].

#### *4.1.3.3. Încapsularea în contextul ECMAScript 5*

Standardul ECMAScript 5 introduce o modalitate mai robustă de încapsulare a stării interne a unui obiect. Flanagan[6] arată că, în conformitate cu versiunea a cincea a specificaţiei, distingem între ***proprietăţi accesoare*** şi ***proprietăţi de date***. Cele două categorii se disting în definirea proprietăţilor de date prin nume şi valoare – ca şi în variantele anterioare ale standardului – în timp ce proprietăţile accesoare sunt create prin metode accesoare şi mutatoare (*getter* şi *setter*). Este important de precizat că, în consecinţă, interpretorul JavaScript va apela automat metodele de acces definite în cazul citirii sau asignării unei proprietăţi accesoare. Din punct de vedere sintactic, în cazul obiectelor literal, proprietăţile accesoare se declară după cum am precizat prin

[1] (Stefanov, JavaScript Patterns 2010, 93-94)
[2] Vezi secţiunea 4.1.2.5 - Încapsularea în contextul moştenirii : Moştenire funcţională (parazitică)
[3] Vezi (Harmes şi Diaz 2007, 30-32)
[4] (Crockford 2008, 52)
[5] Ibid. (52)
[6] (Flanagan 2011, 128-130)

metodele de acces înlocuind cuvântul cheie *function* cu una din variantele *get* sau *set* urmate de numele câmpului.

Inovaţia majoră pe care o introduce ECMAScript 5 în raport cu încapsularea este posibilitatea configurării atributelor unei proprietăţi. După cum precizează Flanagan, acesta este un avantaj important pentru dezvoltarea librăriilor JavaScript, permiţând specificarea de funcţii şi metode cu caracteristici similare celor native şi controlul comportamentului dinamic al obiectelor – de exemplu prin utilizarea metodelor *seal* sau *freeze* prezentate[1] anterior. Astfel, Flanagan[2] arată că putem defini un membru al unui obiect ca fiind format din denumire şi patru atribute care variază în funcţie de tip – accesor sau de date. Atributele de control ale politicii de enumerabilitate şi configurabilitate sunt comune proprietăţilor de date şi celor accesoare. Diferenţa între atributele proprietăţilor de date şi ale celor accesoare este similară cu cea constatată în cazul declarării. În consecinţă, celelalte două atribute ale unei proprietăţi de date sunt valoarea şi inscriptibilitatea, iar în cazul celor accesoare metodele de citire şi atribuire.

**Descriptorii de proprietate** sunt structura de date suport pentru manipularea atributelor unei proprietăţi şi conţin câmpurile aferente acestora. Astfel, *getOwnProprtyDescriptor* returnează obiectul descriptor de proprietate pentru un membru al obiectului asupra căreia este apelată în timp ce *defineProperty* sau *defineProperties* pot fi utilizate pentru crearea sau alterarea atributelor unui membru. Observăm că, în general *defineProperties* are un caracter restrictiv. În acest sens, metoda permite modificarea atributelor unui membru în sens restrictiv, dar nu şi opusul. În continuare propunem un tabel ilustrativ pentru cazurile în care *defineProperty* aruncă excepţie, menţionând că setul de reguli este conform celui prezentat de Flanagan[3].

| Observaţii | Enumerabil | Configurabil | Inscriptibil[4] | Extensibil[5] |
|---|---|---|---|---|
| Încercarea de declarare a unei proprietăţi rezultă în excepţie | **-** | | | **False** |
| Restricţionează orice modificare a atributelor.<br>Proprietatea nu poate fi ştearsă<br>Permite modificarea restrictivă a inscriptibilităţii | **-** | **False** | **-** | |
| Eroare la modificarea valorii | **-** | **False** | **False** | **-** |
| Permite modificarea valorii[6] | **-** | **True** | **False** | **-** |
| Proprietatea nu apare în cazul parcurgerii proprietăţilor unui obiect ca tabelă de dispersie | **False** | **-** | | |

**Tabelul 4.5 Utilizarea metodei defineProperties şi comportamentul acesteia în funcţie de parametri şi specificul obiectului**

[1] Vezi secţiunea 4.1.2.5 - Încapsularea în contextul moştenirii : Moştenire funcţională (parazitică)
[2] (Flanagan 2011, 131)
[3] (Flanagan 2011, 133)
[4] Corespunde specificării atributului *set* în cazul proprietăţilor accesoare
[5] Se aplică în cazul obiectelor după cum am arătat în secţiunea 4.1.2.5
[6] Situaţia este echivalentă cu inscriptibilitatea temporară : schimbarea atributului inscriptibil la valoarea *true*, scrierea noii valori şi revenirea la restricţionarea inscriptibilităţii.

### 4.1.4. Interfeţe

> *Program to an interface, not an implementation.*
> (Gamma, şi alţii 1995, 18)

#### *4.1.4.1. Principiu*

*Interfeţele sunt fundamentele în sistemele orientate pe obiecte*[1]. Gamma et. al. consideră[2] că interfaţa unui obiect este compusă din suma tuturor semnăturilor definite de operaţiile acestuia şi caracterizează setul complet de cereri care îi pot fi adresate. Numele unei interfeţe reprezintă conform Gamma et. al. un **tip**. Putem considera că un obiect are un anumit tip dacă este în măsură a răspunde tuturor cererilor de operaţii definite în interfaţa acestuia. Din punct de vedere relaţional, un obiect poate avea mai multe tipuri şi mai multe obiecte de clase diferite pot avea acelaşi tip prin partajarea totală sau parţială a interfeţei. Diferenţierea între **tip** şi **clasă** este un subiect important asupra căruia se opresc Gamma et.al. Astfel, *clasa unui obiect defineşte starea internă şi implementarea operaţiilor, în timp ce tipul se referă doar la setul de cereri care pot trata*[3]. Prin urmare, Gamma et.al. arată[4] că moştenirea de clasă presupune implementarea unui obiect în termenii altuia şi este un mecanism pentru partajarea codului şi a reprezentării, în timp ce moştenirea de interfaţă specifică posibilitatea interschimbării obiectelor.

Însumând cele precizate mai sus, considerăm că o interfaţă descrie comportamentul comun al unui gen de obiecte. Cunoaşterea şi utilizarea acestui concept în sistemele orientate pe obiecte este esenţială pentru că, după cum precizează[5] Gamma et.al., clienţii ignoră tipurile şi clasele specifice, cunoscând doar clasa abstractă care defineşte setul de capabilităţi cât timp obiectele pe care le utilizează aderă la interfeţele aşteptate. În consecinţă, reducerea semnificativă a dependenţelor de implementare ale subsistemelor duce la formularea principiului enunţat mai sus, *program to an interface, not an implementation*.

JavaScript nu oferă o implementare nativă a conceptului. Mai mult, utilitatea acestuia reprezintă un subiect de dezbatere în comunitatea JavaScript, argumentându-se că rolul interfeţelor este de a da o notă de flexibilitate limbajelor puternic tipizate şi de a specifica un contract pe care un obiect de un anumit tip trebuie să îl respecte – după părerea unora, inutil în limbajele dinamice. Lucrarea de faţă consideră aplicarea interfeţelor a fi substanţială în dezvoltarea unor arhitecturi scalabile, sigure, solide şi flexibile. În acest sens, amintim că, după cum rezultă şi din cele expuse de Gamma et.al., dinamismul conferit codului este un efect al aplicării abstractizării prin interfeţe şi nu un scop prin care s-a dezvoltat conceptul. Pentru ilustrarea avantajelor şi a dezavantajelor utilizării interfeţelor în JavaScript propunem următoarea prezentare sintetică pe baza opiniei formulate în *Pro JavaScript Design Patterns*[6].

[1] (Gamma, şi alţii 1995, 13)
[2] Ibid. (13)
[3] (Gamma, şi alţii 1995, 16)
[4] Ibid. (17)
[5] Ibid. (18)
[6] Ibid. (11-12)

| Avantaje | Dezavantaje |
|---|---|
| *Auto documentarea* | Lipsa implementării native |
| Încurajarea reutilizării codului | Scăderea performanţei[1] |
| Stabilizarea comunicării între clase | În JavaScript şi în alte limbaje dinamice pot fi considerate un mod de impunere a tipizării stricte, ceea ce reduce flexibilitatea nativă |
| Reducerea problemelor de interoperabilitate a obiectelor | |
| Aport semnificativ în testare şi depanare, în special în problemele legate de nepotrivirile de tip | |
| Stabilitatea codului prin asigurarea faptului că orice modificare a interfeţei unei clase trebuie să se reflecte şi în restul aplicaţiei | |

**Tabelul 4.6 Avantaje şi dezavantaje ale utilizării interfeţelor în JavaScript**

În cele ce urmează, vom prezenta o abordare posibilă a conceptului în JavaScript, pornind de la expunerea lui Harmes şi a lui Diaz[2] şi de la modul de implementare în alte limbaje de programare.

### *4.1.4.2. Emularea interfeţelor în JavaScript*

Harmes şi Diaz ilustrează în *Pro JavaScript Design Patterns* modul de utilizare al interfeţelor în Java, PHP şi C#. Pornind de la cele trei limbaje de programare menţionate, Harmes şi Diaz consideră că implementarea noţiunii într-o aplicaţie JavaScript trebuie să aibă în vedere trei aspecte importante : pe de o parte, **declararea şi verificarea existenţei interfeţelor implementate de o clasă**, pe de altă parte, **controlul definirii propriu-zise a metodelor abstracte specificate**.

Pentru **specificarea interfeţelor implementate**, Harmes şi Diaz recomandă[3] utilizarea comentariilor. Observăm că această practică nu are efecte concrete asupra aplicaţiei şi este utilă numai în scopul realizării documentaţiei.

**Verificarea atributelor** este o altă procedură utilă atât în declararea interfeţelor implementate, cât şi în verificarea existenţei acestora. Aceasta prevede precizarea interfeţelor implementate prin intermediul unei structuri de tip *Array* – ar putea să conţină numele în format *String* sau calea completă prin spaţiul de nume. În consecinţă, verificarea existenţei interfeţelor implementate este realizată prin parcurgerea elementelor menţionate în vectorul de interfeţe implementate. Observăm că, la nivelul codului, această abordare rezolvă problemele legate de scrierea, dactilografierea greşită şi, în acelaşi timp, este utilă pentru verificarea tipului unui obiect.

Pentru **a controla implementarea propriu-zisă** a metodelor abstracte definite de o interfaţă, Harmes şi Diaz pornesc de la ceea ce numim ***duck-typing*** după principiul lui James Whitcomb Riley : *When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck* [http://en.wikipedia.org/wiki/Duck_test].

[1] Observăm că scăderea performanţei este nesemnificativă. Mai mult, aceasta poate fi evitată complet.

[2] Vezi (Harmes şi Diaz 2007, 11-23). În capitolul 2, Harmes şi Diaz prezintă pe larg modul de utilizare a interfeţelor în Java, PHP şi C#. Considerăm acest noţiuni cunoscute, astfel încât ne vom concentra în continuare pe o abordare JavaScript. Având în vedere caracterul dinamic al limbajului, menţionăm că declararea explicită a tipului prin interfaţă nu este necesară pentru posibilitatea interschimbării obiectelor. Din punctul de vedere al acestei lucrări, principalul avantaj al utilizării interfeţelor în JavaScript este posibilitatea de verificare a implementării funcţionalităţii corespunzătoare. În consecinţă, interfeţele în JavaScript reprezintă o modalitate de prevenire timpurie a erorilor de programare şi contribuie seminificativ la îmbunătăţirea mentenabilităţii codului.

[3] (Harmes şi Diaz 2007, 14-15)

În programare, acest principiu poate fi interpretat ca o modalitate de determinare a tipului unui obiect în funcţie de capabilităţile sale. Criticii principiului completează judecata lui Riley considerând că, dacă un obiect are trăsăturile unei raţe, *ar putea fi un dragon cu înfăţişare de raţă* şi că *acesta este rareori deziderabil într-un lac pentru raţe, chiar dacă pretinde a se comporta asemănător*[1].

Harmes şi Diaz arată[2] că acest principiu poate fi utilizat şi în scopul asigurării implementării metodelor unei interfeţe. Soluţia propusă presupune crearea unei clase JavaScript – *Interface* – pentru reprezentarea interfeţelor cu o metodă statică – *ensureImplements* – de verificare a implementării metodelor definite. Astfel, pentru definirea unei interfeţe se creează un obiect de tip *Interface* specificând pe de o parte numele şi, pe de altă parte, într-un membru de tip *Array*, metodele de implementat. *ensureImplements* asigură implementarea unei intefeţe prin compararea matricii numelor de metode cu reprezentarea de tip tabelă de dispersie a fiecărei instanţe – numele metodelor declarate în interfaţă trebuie să se regăsească în lista de membri ai obiectului verificat. În acest sens, observăm că, în viziunea lui Harmes şi a lui Diaz, verificarea este efectuată la crearea fiecărei instanţe. Menţionăm că, în ceea ce priveşte acest subiect, dezvoltarea practică a lucrării de faţă porneşte de la soluţia ilustrată de Harmes şi Diaz, însă vom propune o abordare proprie îmbunătăţită.

## 4.2. Modele de design

> *The simple fact of the matter is that patterns have the potential to permanently alter the software engineering field, catapulting it into the realm of true elegant design.*
> (Steve Billow, *Journal of Object-Oriented Programming*)

### 4.2.1. Principiu

Erich Gamma, Richard Helm, Ralph Johnson şi John Vlissides sunt consideraţi a fi primii care au teoretizat modelele de design în programarea orientată pe obiecte. Întrebându-se ce este un model de design, Gamma et.al. răspund pornind de la raţionamentul lui Christopher Alexander care consideră că *fiecare model descrie o problemă frecventă în mediul nostru, oferind nucleul unei soluţii, astfel încât acesta poate fi reutilizat de nenumărate ori în mod unic*[3]. Aplicând principiul enunţat mai sus asupra paradigmei orientate pe obiecte Gamma et.al. arată că *descrierea de obiecte şi clase comunicante, personalizate pentru rezolvarea unei probleme software generale într-un context specific*[4] reprezintă esenţa unui model de design. Astfel, după cum precizează Gamma et.al. modelele de design au un aport semnificativ pentru rezolvarea problemelor software în ceea ce priveşte identificarea şi determinarea granularităţii obiectelor adecvate, cât şi în specificarea interfeţelor şi a implementării. De asemenea, modelele de design sunt importante în folosirea potrivită a mecanismelor de reutilizare şi în ceea ce numim *proiectare pentru schimbare*.

[1] Vezi (http://en.wikipedia.org/wiki/Duck_typing, accesat 7 august 2012)
[2] (Harmes şi Diaz 2007, 17-19)
[3] (Gamma, şi alţii 1995, 2)
[4] Ibid. (3)

Catalogul propus în *Design Patterns : elements of Reusable Object-Oriented Software* organizează modelele de proiectare software printre altele în trei categorii ample : **creaţionale**, **comportamentale** şi **structurale**.

Elaborarea practică a acestei lucrări îşi propune în mare măsură analizarea implementării modelelor de design – în scopul exemplificării am ales câte unul reprezentativ pentru fiecare categorie amintită mai sus – din perspectiva limbajelor de programare dinamice şi slab tipizate prin prisma JavaScript. În continuare vom prezenta o abordare posibilă a implementării modelelor *Singleton* (*creaţional*), *Observer* (*comportamental*) şi *Model View Controller* (*arhitectural*).

### 4.2.2. Singleton

#### *4.2.2.1. Principiu*

*Singleton* este un model de proiectare creaţional; adică este specializat în crearea de obiecte, după cum arată Gamma[1] et.al., dar şi Stefanov[2]. Gamma[3] et.al. caracterizează modelul *Singleton* arătând că scopul principal este garantarea unicităţii instanţei unei clase şi a unui punct global de acces al acesteia. De asemenea, Harmes şi Diaz[4] consideră că modelul reprezintă o modalitate de grupare a codului într-o unitate logică accesibilă printr-o singură variabilă. Din punctul de vedere al lui Gamma[5] et.al. necesitatea acestui model este dată de incapacitatea unei variabile globale de limitare a numărului de instanţe. Astfel, după cum precizează Gamma[6] et.al., prin *Singleton*, responsabilitatea urmăririi insntanţei unice revine clasei înseşi prin interceptarea cererilor de accesare a acesteia.

Considerăm structura descrisă de Gamma et.al. a acestui model de design relativ simplă. Clasa *Singleton* defineşte o operaţie statică *Instance* prin care sunt interceptate şi permise cererile de acces la instanţa unică.

Reţinem că, pe de o parte, în cazul general, prezentat de Gamma et.al.[7] utilizarea *Singleton* aduce următoarele câştiguri : accesul controlat la o resursă prin încapsularea acesteia, reducerea spaţiului de nume, rafinarea operaţiilor şi a reprezentării, flexibilitate sporită în raport cu operaţiile statice. Nu în ultimul rând, menţionăm că modelul permite un număr variabil de instanţe în funcţie de cerinţele sistemului. Pe de altă parte, în cazul specific al JavaScript, Harmes şi Diaz[8] sunt de părere că încapsularea diferenţelor între navigatoare prin ceea ce numim *branching* sau optmizarea codului în mod consistent se numără printre situaţiile însemnate de utilizare a *Singleton*. De asemenea, după cum am precizat şi în cazul general, *Singleton* are un rol esenţial în ceea ce priveşte tratarea spaţiului de nume global al JavaScript.

#### *4.2.2.2. Aspecte de implementare : generale şi în JavaScript*

Apreciem ca deosebit de importante aspectele de implementare legate de garantarea unicităţii instanţei şi de subclasare asupra cărora se opresc Gamma[9] et.al. Astfel, după cum am precizat

[1] Ibid (127-134)
[2] (Stefanov, Object-Orieted JavaScript 2008, 281)
[3] (Gamma, și alții 1995, 127)
[4] (Harmes și Diaz 2007, 65)
[5] (Gamma, și alții 1995, 127)
[6] Ibid. (127)
[7] Ibid. (128)
[8] (Harmes și Diaz 2007, 65)
[9] Vezi (Gamma, și alții 1995, 128-132)

mai sus, modelul maschează posibilitatea de creare a unui obiect pintr-o operaţie de clasă cu acces la instanţa *Singleton* şi care asigură iniţializarea acesteia înainte de returnare. Accesarea instanţei de către clienţi este posibilă exclusiv prin metoda statică menţionată mai sus, iar iniţializarea este târzie (*lazy initialisation*). Exclusivitatea accesului la instanţă prin metoda statică este garantată de declararea *protected* sau *private* a constructorului după cum precizează Gamma et.al.

În ceea ce priveşte derivarea, Gamma[1] et.al. consideră că *instalarea instanţei unice* reprezintă principala problemă şi exprimă esenţa acesteia ca fiind nevoia de iniţializare a obiectului *Singleton* cu o instanţă a subclasei. În consecinţă, abordarea pe care Gamma et.al. o găsesc a fi cea mai flexibilă este ceea ce numim un *registru de singletons*[2] (*registry of singletons*). Conform Gamma et.al., această abordare presupune înregistrarea obiectelor *Singleton* într-un registru bine cunoscut care are proprietatea de mapare între un nume specificat în format *String* şi instanţe.

De asemenea, reţinem că *Singleton* este utilizat şi în implementarea *Abstract Factory*, *Builder* şi *Prototype* ceea ce ne-a determinat în mare măsură să alegem modelul pentru ilustrarea în JavaScript.

Bibliografia propune o gamă variată de abordări ale problemei. Pentru lucrarea de faţă, având în vedere aspectele de implementare prezentate mai sus, le considerăm relevante doar pe cele numite de către Stefanov *cu sintaxă similară claselor*[3].

În ceea ce priveşte implementarea *Singleton* în JavaScript observăm că închiderile reprezintă un element cheie. Astfel, conform principiului, Stefanov[4] consideră că funcţia constructor a obiectelor *Singleton* are responsabilitatea de creare şi memorare a unei referinţe spre instanţa unică la primul apel şi de returnare a acesteia în cazul apelurilor succesive. Conform lui Stefanov şi celor menţionate de Gamma et.al, reţinem că există trei variante de asigurare a accesului la instanţa unică a unei clase *Singleton* după cum urmează : proprietate statică – aceasta este în general soluţia utilizată în alte limbaje de programare, însă în JavaScript are dezavantajul naturii publice, prin urmare nu garantează exclusivitatea formulării cererilor printr-o metodă controlată aşa cum prevede modelul –, variabilă globală – în general considerate a fi de evitat – şi **încadrarea într-o închidere**. Ultima variantă este preferabilă.

O primă implementare propusă de Stefanov[5] presupune redefinirea funcţiei constructor a clasei *Singleton* după instanţierea acesteia. Astfel, în prima fază, constructorul creează obiectul, menţine o referinţă într-un atribut privat definit în scop propriu şi se autoredefineşte pentru a returna instanţa unică la apelurile ulterioare. Problema principală a acestui mod de tratare este pierderea membrilor proprii definiţi între specificarea iniţială şi redefinirea funcţiei constructor. Cauza acestui comportament este lipsa membrilor proprii din prototipul clasei *Singleton*. De asemenea, în acest caz, Stefanov[6] atrage atenţia şi asupra inegalităţii membrului *constructor* al instanţei în raport cu clasa pe care o reprezintă explicând că proprietatea reprezintă în continuare o referinţă spre funcţia constructor iniţială. O soluţie posibilă pe care o ilustrează Stefanov este reiniţializarea prototipului clasei *Singleton* cu pseudovariabila *this* înainte de redefinire. Inconsistenţa câmpului *constructor* se rezolvă atribuind acestuia o referinţă spre clasa redefinită.

[1] Ibid. (130)
[2] Vezi Ibid. (130)
[3] (Stefanov, Object-Orieted JavaScript 2008, 281)
[4] (Stefanov, JavaScript Patterns 2010, 142)
[5] Vezi Ibid. (144-145)
[6] Vezi Ibid. (145)

O soluţie mai elegantă, documentată[1] atât de Stefanov, cât şi de Harmes şi de Diaz este utilizarea unei funcţii imediate. Această abordare păstrează ideea protejării instanţei unice prin încadrarea într-o închidere, dar tratează controlul instanţierii într-un mod mai apropiat de varianta clasică. În acest sens, funcţia constructor a clasei *Singleton* este responsabilă pentru iniţializarea şi verificare existenţei instanţei unice. Este important de reţinut că ambele variante prezentate permit *iniţializarea întârziată* a instanţei.

Precizăm că implementaea *Singleton* prin funcţii imediate este cunoscută în JavaScript şi sub numele de *Module Pattern*[2], fiind un procedeu de modularizare şi organizare a metodelor şi a atributelor care se pot asocia prin scop şi domeniu de abstractizare.

### 4.2.3. Observer

#### *4.2.3.1. Principiu*

*Observer* este un model de proiectare comportamental, adică reprezintă interacţiunea între obiecte şi, după cum arată Harmes şi Diaz[3], modelează relaţia între obiecte, acţiunile şi starea acestora. Gamma et.al arată că scopul acestui model este *definirea unei relaţii de dependenţă de tip unu-la-n între obiecte astfel încât dependenţii unui obiect să fie notificaţi şi actualizaţi automat la schimbarea stării subiectului*[4].

Justificarea pe care Gamma et.al[5] o dau acestui model este necesitatea unei abordări de cuplare slabă în stabilirea relaţiilor între obiecte care nu se *cunosc* unele pe altele. În acest sens, Gamma[6] et.al consideră că, în ceea ce priveşte *Observer*, obiectele cheie sunt *Subiect* şi *Observator*. Astfel, un subiect poate avea nenumăraţi observatori pe care se angajează să îi informeze cu privire la schimările sale de stare şi care, la rândul lor, pot să îi trimită interogări pentru a se sincroniza. Putem considera că astfel, modelul îndeplineşte scopul şi justificarea enunţate mai sus.

Abstractizând domeniile de utilizare ale modelului, Gamma[7] et.al recomandă ca utilă folosirea Observer atunci când două aspecte ale unei abstractizări sunt interdependente sau când schimbarea stării unui obiect afectează altele şi nu se doreşte o cuplare strânsă.

Având în vedere structura de clase mai amplă decât cea a *Singleton* cu care avem de a face în acest caz, apreciem esenţială cunoaşterea şi înţelegerea acesteia. În continuare, vom ilustra structura modelului prin diagrama de clase propusă şi explicată de Gamma et.al[8].

Reţinem că modelul *Observer* se realizează prin participanţii abstracţi *Subject* şi *Observer*, şi implementările *ConcreteSubject* şi *ConcreteObserver*. Astfel, subiectul (*Subject*) îşi cunoaşte observatorii şi reprezintă o interfaţă pentru ataşarea şi detaşarea unui număr nelimitat de obiecte de tip *Observer*. Prin *Observer* este specificată o interfaţă pentru actualizare. Subiectul şi observatorii concreţi (*ConcreteSubject* şi *ConcreteObserver*) sunt implementări ale *Subject* şi *Observer*. Un subiect concret reflectă starea de care sunt interesaţi observatorii şi este responsabil pentru trimiterea de notificări acestora. Un *ConcreteObserver* menţine o referinţă spre subiectul concret şi trebuie să aibă o stare consistentă cu cea a obiectului observat.

[1] Vezi Ibid. (146) şi (Harmes şi Diaz 2007, 70-75)
[2] Modelul este discutat pe larg de (Flanagan 2011, 246-250) şi de (Crockford 2008, 40-42)
[3] (Harmes şi Diaz 2007, 215)
[4] (Gamma, şi alţii 1995, 293)
[5] (Gamma, şi alţii 1995, 293-294)
[6] Ibid. (294)
[7] Ibid.
[8] Ibid. (294-295)

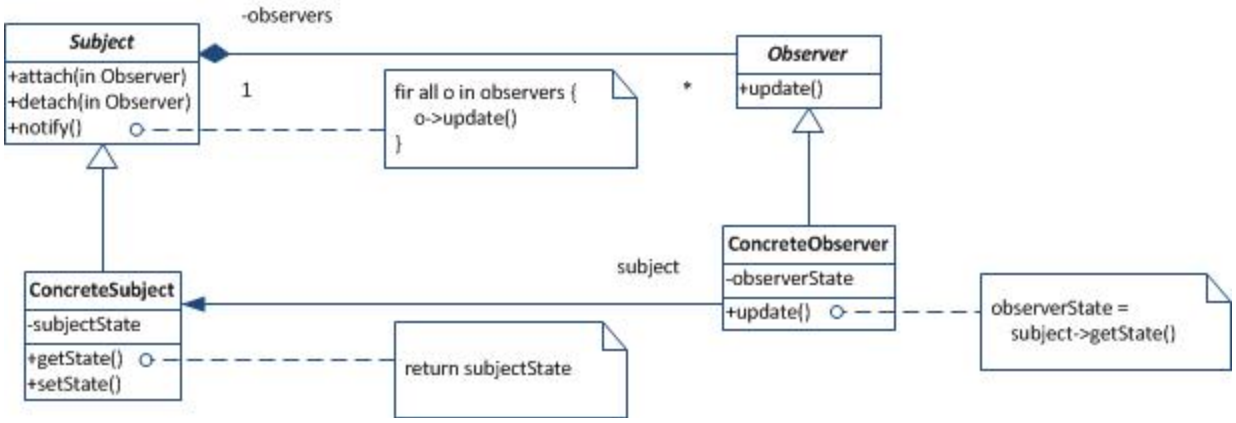



**Fig. 4.10 Diagramă de clase *Observer***

Conform Gamma[1] et.al consecinţele acestei abordări sunt cuplarea abstractă între subiect şi observator – un subiect cunoaşte doar o listă de obiecte de tip *Observer* – şi comunicaţia *broadcast* între subiect şi observatori. Un alt aspect de reţinut în utilizarea modelului este că, din cauza simplităţii protocolului, notificările trimise de un subiect nu conţin informaţii referitoare la modificări. Prin urmare, o operaţie aparent inofensivă asupra subiectului poate declanşa o cascadă de actualizări.

### *4.2.3.2. Aspecte de implementare : generale şi în JavaScript*

Gamma[2] et.al. descriu amplu subiectele de avut în vedere în implementarea modelului de proiectare *Observer*. Astfel, **maparea între subiecţi şi observatori** reprezintă o primă problemă de luat în calcul. Realizarea simplă a mapării este, din punctul de vedere al Gamma et.al, menţinerea unei referinţe către observatori. O abordare complexă şi care eficientizează utilizarea resurselor de memorie, însă presupune un compromis în ceea ce priveşte timpul de execuţie – creşte timpul de acces la observatori, este utilizarea unei tabele de dispersie pentru mapare. **Observarea subiecţilor multipli** este un alt aspect asupra căruia se opresc Gamma et.al. arătând că interfaţa de actualizare trebuie exstinsă astfel încât aceasta să conţină informaţiile cu privire la identitatea obiectului modificat. **Responsabilititatea declansării procedurii de actualizare** este o problemă fundamentală în implementarea *Observer*. Gamma et.al. arată că, pe de o parte, situaţia atribuirii acestei sarcini subiectului este avantajoasă prin lipsa nevoii de apelare explicită a metodei *notify* de către obiectele client cu care acesta interacţionează – subiectul trimite o notificare pentru orice modificare a stării sale –, însă este ineficientă având în vedere că nu permite secvenţe de schimbări. Pe de altă parte, atribuirea sarcinii clienţilor oferă posibilitatea declanşării procedeului de actualizare la sfârşitul unei secvenţe de modificări a stării subiectului, însă abordarea este predispusă erorilor prin omiterea apelului metodei *notify*. Preîntâmpinarea **referenţilor către obiecte *Subject* inexistente** este posibilă conform Gamma et.al. prin notificarea observatorilor de către subiect cu privire la distrugerea sau ştergerea sa înainte ca aceasta să aibă loc. De asemenea, Gamma et.al. recomandă **asigurarea autoconsistenţei** unui subiect prin metode şablon (*Template Method*[3]) înaintea declanşării procedurii de actualizare. O altă indicaţie preţioasă făcută de Gamma et.al. este **evitarea protocoalelor de actualizare**

[1] Ibid. (296)
[2] Vezi (Gamma, şi alţii 1995, 296-300)
[3] Vezi (Gamma, şi alţii 1995, 325) – Modelului de proiectare *Template Method*

**specifice push**[1] **şi pull**[2] care reprezintă extreme ale implementării *Observer*. **Specificarea explicită a domeniului de interes al observatorilor** reprezintă o eficientizare care poate fi adusă implementării. În acest sens, observatorilor le este permis să monitorizeze doar schimbarea ***aspectelor*** subiectului – declanşarea unui eveniment specializat – de care sunt interesaţi. În ceea ce priveşte JavaScript, evenimentele navigatorului pot fi considerate o astfel de implementare a modelului *Observer*. În situaţia unei relaţii complexe între subiect şi observatori, Gamma et.al. sugerează utilizarea unui administrator de modificări (*ChangeManager*) – soluţia este, de fapt o imlementare a modelului *Mediator*[3] – care reduce considerabil efortul de sincronizare.

Din punctul de vedere al punerii în practică a modelului în JavaScript observăm că, în acord cu bibliografia, implementarea este relativ simplă şi fără complicaţii, dar diferită faţă de cea a altor limbaje de programare. Soluţia întâlnită cel mai des în lucrările bibliografice consultate este prin utilizarea apelurilor *callback*. În *Object-Oriented JavaScript*[4], Stefanov propune specificarea unei clase *Observer* cu rol de subiect care poate fi extinsă de obiectele *observabile*. În abordarea lui Stefanov, înregistrarea observatorilor este realizată prin crearea unei liste de metode *callback* pe care subiectul le apelează în scopul notificării cu privire la modificarea stării sale.

### 4.2.4. Model View Controller

#### *4.2.4.1. Principiu*

*Model-View-Controller* face parte din categoria modelelor de proiectare arhitecturale şi a fost propus şi utilizat pentru prima dată de Trygve Reenskaug în platforma Smalltalk. Astăzi, *MVC* are un rol fundamental în dezvolatarea frameworkurilor pentru realizarea interfeţelor grafice ale aplicaţiilor interactive după cum precizează Fowler[5] şi Buschmann[6] et al.

MacCaw[7] arată că din perspectiva *Model-View-Controller*, o aplicaţie este împărţită pe trei nivele ierarhice : model – conţine structuri de reprezentare a domeniului aplicaţiei şi funcţionalitatea de bază –, vedere – prezentarea datelor – şi ceea ce numim *controller* – responsabil pentru gestionarea interacţiunii cu utilizatorul. Buschmann[8] et.al. şi MacCaw[9] detaliază scopul şi responsabilităţile categoriilor de obiecte menţionate mai sus.

Astfel, **modelul** menţine totalitatea datelor aplicaţiei, este decuplat de aceasta şi, în consecinţă, nu cunoaşte vederile sau obiectele de control. Totodată, modelelor le revine responsabilitatea de înregistrare a vizualizărilor şi a obiectelor de control dependente şi de notificare a acestora cu privire la schimbărea stării proprii.

**Vederile** sunt de asemenea independente de restul aplicaţiei şi pot conţine logică relevantă pentru prezentare prin ceea ce numim funcţii şi obiecte utilitare (*helper*) după cum arată MacCaw[10]. Buschmann[1] et.al. precizează că vederilor le revin următoarele sarcini : crearea şi

[1] În cazul implementării de tip *push*, subiectul trimite întreaga informaţie cu privire la schimbarea stării sale observatorilor.

[2] Tratarea modelului printr-o metodă de tip *pull* este extrema opusă protocolului *push* şi prevede trimiterea de notificări minime lăsând observatorul să interogheze subiectul cu privire la modificările de stare de care este interesat.

[3] Vezi (Gamma, şi alţii 1995, 273) – Modelul de proiectare *Mediator*

[4] Vezi (Stefanov, Object-Orieted JavaScript 2008, 287-290)

[5] (Fowler 2002, 300)

[6] (Buschmann, şi alţii 2001, 125)

[7] (MacCaw 2011, 3-5)

[8] (Buschmann, şi alţii 2001, 126-129)

[9] (MacCaw 2011, 2-5)

[10] (MacCaw 2011, 4)

iniţializarea obiectului de control asociat, observarea modelului implementând interfaţa *Observer* şi prezentarea informaţiei.

Din punctul de vedere al lui MacCaw[2], **obiectele de control** sunt legătura între modele şi vederi. Buschmann [3] et.al. apreciază că evenimentele generate de acţiunile utilizatorului reprezintă intrări pentru obiectele de control. Astfel, obiectele de control răspund de translatarea evenimentelor în ceea ce numim *cereri de servicii* (*service requests*). La rândul lor, obiectele de control pot observa starea unui model în cazul în care schimbarea stării acestuia rezultă în declanşarea unui eveniment relevant – un exemplu ar fi activarea şi dezactivarea unui buton.

Fowler[4] este de părere că, în esenţă, abstractizarea prin *MVC* presupune două delimitări fundamentale : în primul rând separarea prezentării de model şi, în al doilea rând detaşarea vederilor de obiectele de control.

În continuare, propunem o prezentare sintetică a consecinţelor utilizării *MVC* bazată pe observaţiile făcute de Buschmann[5] et.al.

| Avantaje | Dezavantaje |
|---|---|
| Reprezentarea aceluiaşi model prin vederi multiple | Creşterea complexităţii |
| Sincronizarea vederilor cu modelul printr-un mecanism de propagare a modificărilor | Poate duce la un număr excesiv de operaţii de actualizare după cum am arătat în prezentarea modelului *Observer* |
| Vederi şi obiecte de control *conectabile* modelului | Cuplarea între vedere şi obiectul de control |
| Potenţial de utilizare pentru dezvoltarea frameworkurilor | Ineficienţa accesării datelor - cereri multiple efectuate de vedere pentru obţinerea tuturor datelor necesare |

**Tabelul 4.7 Consecinţele utilizării MVC în dezvoltarea aplicaţiilor**

### 4.2.4.2. Aspecte de implementare

În *Pattern-Oriented Software Architecture – A System Of Patterns*, Buschmann et.al. ilustrează pe larg paşii de urmat în realizarea aplicaţiilor conform modelului *Model-View-Controller*.

Astfel, Buschmann[6] et.al. consideră că **separarea interacţiunii între utilizator şi sistem de funcţionalitatea de bază** prin analiza domeniului aplicaţiei reprezintă punctul de plecare în dezvoltarea conform modelului *MVC*. De asemenea, este esenţială implementarea mecanismului de propagare a modificărilor prin urmarea modelului *Observer* despre care am discutat mai sus. Următorii paşi pe care îi recomandă Buschmann[7] et.al. sunt **proiectarea şi implementarea vederilor şi a obiectelor de control**. Reţinem că reutilizabilitatea *controllerelor* poate fi obţinută prin aplicarea modelului *Command Processor*[8]. După realizarea paşilor menţionaţi este

[1] (Buschmann, şi alţii 2001, 128)
[2] (MacCaw 2011, 5)
[3] (Buschmann, şi alţii 2001, 128)
[4] (Fowler 2002, 300-301)
[5] (Buschmann, şi alţii 2001, 141-143)
[6] (Buschmann, şi alţii 2001, 132)
[7] Ibid. (134-136)
[8] Vezi Ibid. (277)

necesară **stabilirea relaţiei între vederi şi obiecte de control**. În acest sens, pentru cazul structurilor ierarhice de vederi, notăm că este recomandată utilizarea *Factory Method*. **Implementarea configurării *MVC*** este din punctul de vedere al Buschmann et.al ultima etapă esenţială de parcurs în dezvoltarea de aplicaţii pe baza acestui model presupunând iniţializarea modelului şi a vederilor în ordine, urmate de începerea procesării evenimentelor.

Pentru a obţine un grad de flexibilitate mai mare în aplicaţiile sau frameworkurile dezvoltate, Buschmann[1] et.al. consideră importantă implementarea unui *View Handler*[2] pentru administrarea vederilor în cazul creării dinamice a acestora. Nu în ultimul rând, Buschmann et.al. arată că libertatea de utilizare a codului poate fi crescută şi prin realizarea unei infrastructuri pentru vederi şi obiecte de control *conectabile* ierarhice.

## 4.3. Concluzii

În concluzie suntem de părere că, în acord cu principiile şi abordările practice prezentate mai sus, cunoaşterea şi aplicarea normelor de programare orientată pe obiecte în JavaScript are un impact însemnat în ceea ce priveşte eficienţa, viteza şi transparenţa aplicaţiilor. Nu în ultimul rând, un efect important al utilizării conceptelor discutate este gradul ridicat de compatibilitate cu alte programe şi librării. Prin natura sa dinamică, limbajul face implementarea modelelor de proiectare deosebit de uşoară, după cum am putut observa în cazul celor discutate mai sus. Menţionăm că alegerea modelelor expuse mai sus a avut la bază în primul rând criteriul aceptării acestora ca fiind fundamentale la scară largă aşa cum sunt *Singleton*, *Observer* şi *Model-View-Controller*.

Considerăm că modelele de design oferă şi în JavaScript aceeaşi modalitate deosebit de eficientă de identificare a abstracţiunilor necesare unei aplicaţii pe care o oferă în alte limbaje de programare consacrate.

Însumând, scopul de abordare a aplicaţiilor *Web* într-o manieră reutilizabilă şi modulară, propus la începutul lucrării, este o consecinţă a programării prin paradigma orientată pe obiecte în JavaScript.

[1] Ibid. (138-140)
[2] Vezi Ibid. (291)

# 5. Specificaţia sistemului

Având în vedere caracterul lucrării de faţă, propunem ilustrarea cerinţelor funcţionale, nefuncţionale şi de resurse ale sistemului atât la nivelul general şi abstract al frameworkului *jqCls*, cât şi concret prin exemplificarea în cazul unei aplicaţii specifice.

Considerăm că jocurile de strategie online multiplayer bazate pe navigator constituie o provocare majoră pentru funcţionalitatea interfeţelor grafice, fiind un domeniu în care aportul aplicaţiei client în atragerea utilizatorilor este seminificativ pentru succesul general. În consecinţă, în scopul demonstrării utilităţii şi a cerinţelor pe care le abordează frameworkul propus, vom realiza un simulator de gestiune a companiilor aeriene – *Airline Manager*.

În continuare vom prezenta capabilităţile cu caracter final avute în vedere pentru elaborarea, gestionarea şi evaluarea frameworkului *jqCls*. De asemenea, vom prezenta funcţionalitatea aplicaţiei demonstrative *Airline Manager* realizată prin *jqCls*.

## 5.1.1. Cerinţe funcţionale

### *Cerinţe funcţionale abstracte la nivelul jqCls*

#### Procesarea unui formular

Sistemul prezintă utilizatorului un formular HTML construit conform unui model de date. Utilizatorul completează câmpurile cu date adecvate şi declanşează evenimentul de procesare a formularului prin apăsarea butonului de trimitere de pe ecran, sau a tastei *ENTER*.

Sistemul începe procesarea datelor şi notifică utilizatorul prin afişarea unei notificări. În funcţie de caz, sistemul poate îngheţa starea formularului prin dezactivarea câmpurilor şi afişarea unei măşti de procesare.

După finalizarea procesării datelor (validare conform cerinţelor modelului de date, comunicare cu aplicaţia server în funcţie de caz), sistemul informează utilizatorul cu privire la rezultat şi afişează succesul operaţiei sau eventuale erori apărute. În cazul în care există comportament specific de executat la primirea răspunsului, sistemul continuă cu execuţia acestuia.

#### Operaţii CRUD

Sistemul afişează o componentă de entităţi tabelară, paginată în funcţie de cererea utilizatorului. Utilizatorul alege entităţile dorite şi poate solicita adăugarea, modificarea, ştergerea sau afişarea vederii de detaliu aferente. Sistemul intră în starea de procesare a cererii utilizatorului şi afişează răspunsul. În cazul ştergerii, sistemul va informa utilizatorul cu privire la rezultatul cererii : succes sau eroare.

În cazul adăugării, modificării sau cererii unei vederi detaliate, sistemul va aduce informaţiile necesare şi va afişa, după caz formularul corespunzător completat cu datele primite. Pentru adăugare sau modificare, procesul continuă conform specificaţiei pentru procesarea unui formular, cu menţiunea că, în cazul procesării cu succes a cererii, sistemul va actualiza vederea tabelară.

#### Căutare / Filtrare / Sortare

Sistemul afişează o componentă de entităţi tabelară, paginată în funcţie de cererea utilizatorului. Utilizatorul introduce termenul de căutare sau filtrare. Sistemul procesează datele introduse şi afişează rezultatele conforme cerinţelor de căutare sau filtrare introduse de utilizator.

Sistemul afişează o componentă de entităţi tabelară. Utilizatorul alege câmpul şi direcţia de sortare. Sistemul afişează datele sortate conform criteriilor utilizatorului.

#### Interfaţă MDI (Multiple Document Interface)

Sistemul oferă utilizatorilor o interfaţă grafică *Multiple Document Interface* în care blocurile funcţionale sunt prezentate în ferestre care pot coexista şi sunt administrate de o componentă de tip *desktop manager*. Utilizatorul poate deschide şi închide ferstrele interfeţei. De asemenea, utilizatorul poate modifica dimensiunile şi poziţia ferestrei pe ecran. Sistemul îi prezintă utilizatorului ferestrele deschise într-o componentă andocată pe una dintre extremităţile ecranului.

### *Cerinţe funcţionale concrete ale aplicaţiei demonstrative Airline Manager*

#### Înregistrare – exmplificarea procesării unui formular

Utilizatorii pot să se înregistreze fără constrângeri suplimentare în sistemul *Airline Manager*.

Sistemul va verifica unicitatea numelui de utilizator ales şi va securiza procesul împotriva înregistrărilor automatizate prin utilizarea CAPTCHA şi prin trimiterea unui email de confirmare a contului creat. Contul utilizatorului va fi activat doar după validarea acestuia prin codul de confirmare trimis în email. Utilizatorii care nu au fost validaţi în termen de 24 de ore de la crearea contului, vor fi şterşi automat.

#### Autentificare

Jucătorii şi administratorii sistemului trebuie să furnizeze date de autentificare valide pentru a accesa aplicaţia *Airline Manager*. Jucătorii îşi pot alege parola şi numele de utilizator la înregistrare. Administratorii primesc numele de utilizator şi parola în urma adăugării de către un alt administrator. Sistemul va permite oricărui utilizator să îşi modifice parola în panoul de administrare al contului.

#### Alegerea unui scenariu de joc

După autentificare, jucătorii pot alege scenariul de joc dintr-o vedere tabelară. Sistemul va reţine alegerea pe parcursul sesiunii unui utilizator şi va permite modificarea acesteia prin revenire la vederea tabelară iniţială. Un scenariu de joc fixează repere geografice, situaţia politică pentru perioada aleasă, precum şi durata simulării.

#### Deschiderea/închiderea unei companii aeriene

Fiecare scenariu va permite un număr limitat de companii aeriene în total şi per jucător. Orice jucător poate deschide o companie aeriană în limita restricţiilor scenariului. Pentru deschiderea unei companii aeriene sistemul va solicita numele acesteia şi sediul iniţial.

Sediul iniţial nu va putea fi modificat şi va fi utilizat în aplicarea restricţiilor politice conforme scenariului de joc. Sistemul va permite modificarea numelui, siglei şi a descrierii companiei. În orice unitate de timp, un jucător poate solicita închiderea unei companii aeriene proprii. Fiecărei companii aeriene îi este permisă deschiderea şi închiderea unui număr nelimitat de subsidiare ale căror sedii pot fi definite la înfiinţare.

#### Administrarea flotei – exemplificare pentru operaţii CRUD în vederi tabelare

Jucătorii pot cumpăra prin *cash* sau în sistem de leasing aeronave în funcţie de localizarea temporală a simulării. De asemenea, sistemul va permite utilizatorilor să caseze, închirieze şi să ofere spre închiriere sau vânzare aeronave. Sistemul nu va permite vânzarea unei aeronave, decât dacă pentru aceasta există un cumpărător.

#### Administrarea rutelor de zbor

O companie aeriană poate opera un număr nelimitat de rute cu aeronavele din flota proprie sau a subsidiarelor. Sistemul va aplica restricţiile politice impuse de scenariu pentru definirea rutelor de zbor şi va evalua următoarele criterii :

- Compania aeriană, alianţa sau subsidiarele deţin o poartă cu destule poziţii libere la aeroporturile selectate
- Aeronavele nu depăşesc orele de zbor permise într-o zi
- Pistele îndeplinesc cerinţele de aterizare şi decolare ale avioanelor selectate
- Aeronavele au autonomia necesară pentru completarea rutei sau a segmentelor din rută în cazul definirii escalelor.

Sistemul va calcula profitul sau pierderile în funcţie de costurile de operare, gradul de ocupare al frecvenţelor zilnice şi de preţul biletelor.

### Administrarea porţilor închiriate – exemplificare pentru operaţii CRUD în vederi tabelare

Fiecare companie aeriană poate închiria un număr nelimitat de porţi la un aeroport. Sistemul va restricţiona închirierea porţilor conform restricţiilor politice ale scenariului. De asemenea, fiecare aeroport are un număr limitat de porţi disponibile. În cazul epuizării porţilor disponibile pentru închiriere, o companie aeriană poate deschide un terminal propriu.

### Administrarea serviciilor la bord

Jucătorii pot să configureze serviciile multimedia şi de catering la bord în funcţie de distanţă şi numărul de locuri în aeronavă pentru clasele de transport *First*, *Bussiness* şi *Economy*. Sistemul va defini o capacitate specifică de transport pentru fiecare aeronavă în parte. Pe baza serviciilor de catering, a sistemelor de divertisment şi a configurării locurilor, sistemul va calcula scorul pentru popularitatea rutei. Nivelul de satisfacţie al pasagerilor va fi utilizat în determinarea împărţirii pasagerilor pentru rutele cu concurenţă ridicată.

### Administrarea terminalelor

O companie aeriană poate construi, vinde sau dezafecta terminale adiţionale în aeroporturi în limita resurselor financiare disponibile. Sistemul va limita această posibilitate aplicând restricţiile politice ale scenariului.

Raportul între costurile de operare a porţilor de îmbarcare într-un terminal propriu şi într-un terminal al aeroportului va fi de 1 la 10. Sistemul va genera pentru fiecare terminal propriu un număr de pasageri de până la 10% în plus pentru compania aeriană proprietară.

### Administrarea hub-urilor

O companie aeriană poate deschide sau închide *huburi*. Acestea reprezintă aeroporturi cu importanţă strategică deosebită şi cu un număr mare de conexiuni. Sistemul va restricţiona posibilitatea deschiderii unui *hub* în funcţie de numărul de zboruri al companiei din şi înspre aeroportul în cauză.

Sistemul va creşte costurile de închiriere a unei porţi în aeroport cu 100% la deschiderea unui hub. Pentru aeroporturile în care o companie aeriană înfiinţează un *hub* sistemul va procesa zboruri de legătură. Astfel, sistemul va face posibile rute de tip A – HUB – B.

### Vizualizarea situaţiei financiare

Sistemul va calcula situaţia financiară generală a fiecărei companii aeriene pe baza costurilor şi profiturilor zilnice. Jucătorii vor putea accesa datele relevante în panoul de prezentare a situaţiei financiare. În acelaşi timp, jucătorii vor putea să modifice parametri de salarizare ai personalului.

### Simularea timpului

Sistemul va simula trecerea timpului. O zi calendaristică (24 de ore) va fi reprezentată prin 24 de minute. Utilizatorul va fi informat cu privire la trecerea timpului simulat.

### 5.1.2. Cerinţe nonfuncţionale

Lucrarea de faţă îşi propune abordarea cerinţelor funcţionale şi a celor nefuncţionale conform modelului ***FURPS+*** [1] introdus de către Grady Boch. *FURPS+* clasifică cerinţele unui sistem în : ***f****uncţionale* – acestea reprezintă acţiunile sistemului într-un caz dat şi au fost exemplificate în capitolul anterior, de ***u****zabilitate* – descriu uşurinţa de utilizare, de soliditate (***r****eliability*) – măsură a fezabilităţii, de **p**erformanţă – cuantificată în capacitate de procesare, de mentenabilitate (**s**upportability) – reflectă uşurinţa cu care poate fi modificat şi de **s**ecuritate. În acelaşi timp, modelul consideră factori de implementare (limitări ale sistemului), interfaţă (constrângeri impuse de interfaţarea cu sisteme externe), operaţii definite de administrarea sistemului) şi, nu în ultimul rând: aspecte legale de licenţiere.

#### Cerinţe nonfuncţionale abstracte la nivelul *jqCls*

Din punctul de vedere al utilizabilităţii *jqCls* ca instrument de dezvoltare, considerăm că una dintre cerinţele nefuncţionale fundamentale este o curbă de învăţare scurtă. Prin aceasta, înţelegem că un programator cu cunoştinţe de JavaScript trebuie să poată utiliza frameworkul după 1-2 zile de studiu. Referitor la aplicaţiile client generate, *jqCls* îşi propune facilitarea realizării aplicaţiilor client cu interfaţă deosebit de intuitivă.

Suntem de părere că exactitatea şi robustitatea se numără, alături de performanţele ridicate printre cele mai de preţ atribute pe care le regăsim într-un framework. În acest sens, ne propunem eliminarea totală a disfuncţionalităţilor din nucleul *jqCls* prin testarea aprofundată. Cu privire la aspectul performanţei, ne propunem menţinerea resurselor necesare frameworkului la 2 MB şi încărcarea acestora prin 2 fişiere : unul pentru funcţionalitatea JavaScript şi unul dedicat stilizării componentelor prin CSS. Nu în ultimul rând, având în vedere că operaţiile cu arborele DOM se numără printre cele mai costisitoare, procesul de implementarea al *jqCls* va avea în vedere tehnicile eficiente recomandate ca standard în industrie.

Modificările şi personalizările efectuate de utilizatori la nivelul unui framework, fără ca acestea să fie propuse oficial spre adoptare nu sunt recomandate. *jqCls* aderă acestui principiu. Totuşi, suntem de părere că extensibilitatea şi uşurinţa de integrare cu alte produse sunt un factor cheie pentru adoptarea unui framework. În acest sens, ne propunem realizarea unei interfeţe de programare aplicabile alături de alte librării JavaScript modulare. Posibilitatea interschimbării modulelor de infrastructură *jqCls* este o altă caracteristică avută în vedere în scopul asigurării unui proces de mentenanţă şi dezvoltare facil.

Modelul actual al aplicaţiilor web – client lasă răspunderea securităţii aplicaţiei server. În acest sens, frameworkul propus nu va implementa module de securitate.

Din punctul de vedere al implementării, *jqCls* este un framework JavaScript dedicat realizării clienţilor în aplicaţii web. Acesta poate fi interfaţat cu orice tehnologie server capabilă de comunicare într-un format de transport al datelor acceptat (XML sau, preferabil JSON).

În ceea ce priveşte aspectele legale, *jqCls* se doreşte a fi un produs *open-source* protejat de licenţa GNU – GPL 3.0[2].

#### Cerinţe nonfuncţionale concrete ale aplicaţiei demonstrative *Airline Manager*

**Utilizabilitatea** *Airline Manager* constituie măsura obiectivă a gradului în care *jqCls* facilitează dezvoltarea aplicaţiilor web intuitive bogate în funcţionalitate. Având în vedere că aplicaţia se încadrează în categoria jocurilor, utilizabilitatea, uşurinţa de înţelegere a

[1] Vezi http://en.wikipedia.org/wiki/FURPS. Modelul a fost prezentat în cursul *Sisteme Informatice* al Prof. PhD. Ovidiu Pop.

[2] http://www.gnu.org/licenses/gpl.html

capabilităţilor aplicaţiei are un rol esenţial. Vom îndeplini cerinţele acestui punct prin crearea meniurilor intuitive şi prin utilizarea sistemului de interfaţă grafică *Multiple Document Interface*.

Succesul unui joc rezident în navigator depinde în egală măsură de capacitatea de a atrage utilizatori noi, de păstrarea celor existenţi şi de captarea atenţiei acestora pentru maximizarea timpului petrecut în aplicaţie. Suntem de părere că obiectivele enumerate cu privire la experienţa utilizatorului pot fi îndeplinite dezvoltând o aplicaţie **robustă** şi **performantă**. Prin urmare, ne propunem să oferim funcţionalităţile *Airline Manager* atât online, cât şi offline în măsura în care aceasta nu depinde de comunicarea la distanţă, utilizând capabilităţile *jqCls*. Nu în ultimul rând, serverele de aplicaţie alese trebuie să ofere un *uptime* garantat de 99%.

Studii recente referitoare la atenţia şi interesul utilizatorilor pentru aplicaţii web, arată că, pentru a obţine un grad de satisfacţie ridicat, o aplicaţie trebuie să reacţioneze la cereri într-un timp de până la 0.2 secunde. În acest sens, *Airline Manager* poate fi considerat un punct de reper pentru evaluarea **performanţelor** frameworkului propus.

Dat fiind domeniul *Airline Manager*, nu considerăm că datele stocate sunt sensibile. Consecutiv, modelul de **securitate** ales va fi unul standard.

Cu privire la aspectele de **implementare** şi **interfaţare**, *Airline Manager* abordează arhitectura bazată pe servicii. Aplicaţiile client şi server vor comunica prin formatul de transport JSON (JavaScript Object Notation). Această alegere arhitecturală asigură independenţa între client şi server şi va demonstra interoperabilitatea frameworkului propus.

### 5.1.3. Cerinţe de resurse

#### Cerinţe de resurse la nivelul *jqcls*

*jqCls* este un framework JavaScript. Ca atare, se aplică cerinţele de resurse general valabile în cazul produselor software din această categorie. Concret, pentru utilizarea *jqCls* este necesar un mediu cu un motor JavaScript integrat. Mediul de întrebuinţare vizat este un navigator, însă nu excludem alte posibilităţi.

Cu privire la alinierea la standardele existente, *jqCls* utilizează setul de funcţionalităţi specificate în ECMAScript versiunea 5.1.

#### Cerinţe de resurse la nivelul aplicaţiei demonstrative

Resursele necesare pentru rularea aplicaţiei *Airline Manager* le includ pe cele impuse de frameworkul propus cu privire la client, dar sunt independente cu privire la server.

Sistemul demonstrativ este o aplicaţie web client-server. Pe sistemul server se impune instalarea JRE 1.6 sau versiuni ulterioare, precum şi Tomcat 7+. Pentru stocarea datelor sistemul utilizează baze de date MySQL compatibile cu versiunea 5.6+.

## 5.2. Scenarii de aplicabilitate

Propunem descrierea scenariilor de aplicabilitate ale frameworkului *jqCls* metoda descrisă, motivată şi aplicată în subcapitolul anterior. Menţionăm că scenariile de aplicabilitate vor fi dezvoltate pe baza capabilităţilor menţionate.

### 5.2.1. Scenarii de aplicabilitate abstracte *jqCls*

În cele ce urmează propunem ilustrarea unei vederi de ansamblu a cazurilor de utilizare abstracte identificate pentru funcţionarea *jqCls*.

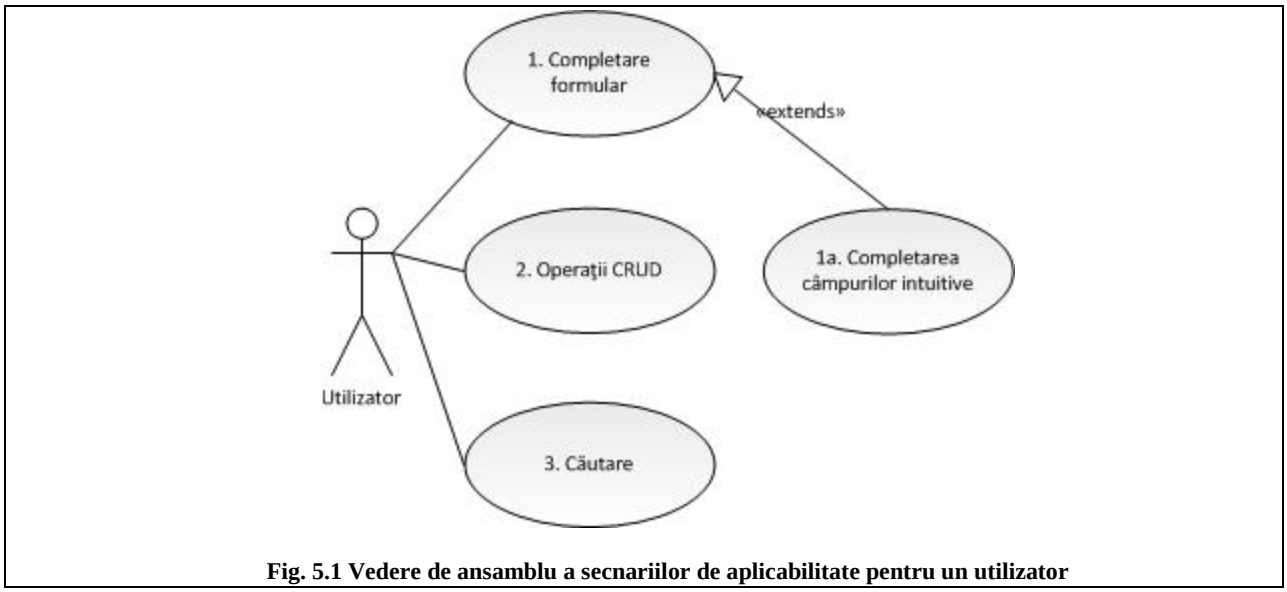



**Fig. 5.1 Vedere de ansamblu a secnariilor de aplicabilitate pentru un utilizator**

După cum reiese din figura 5.1, nivelul de abstractizare al frameworkului *jqCls* nu distinge între categorii de utilizatori. Operaţiile înfăţişate în diagramă reprezintă din punctul de vedere al acestei lucrări, entităţile atomice de funcţionalitate general valabile în aplicaţii. Astfel, frameworkul facilitează dezvoltarea interacţiunii pentru completarea formularelor (UC 1), efectuarea operaţiilor de creare, citire, editare şi ştergere a datelor (**C**reate, **R**ead, **U**pdate, **D**elete) prin UC 2 şi de căutare (UC 3). Câmpurile intuitive (autocomplete) au devenit un standard de facto în aplicaţiile moderne. Abstractizarea fluxului de informaţii pentru astfel de câmpuri este prezentată în extensia UC 1.a.

### 1. *Denumire :* ***Procesarea unui formular***

Actori principali : **Utilizatorul**

**Interesele părţilor**

1) Utilizatorul : introducerea datelor pentru îndeplinirea unui obiectiv – de exemplu creare unui cont sau autentificarea

2) Sistemul : colectarea datelor cu integritate completă

**Precondiţii**

Utilizatorul cere deschiderea unei vederi de introducere a datelor.

**Postcondiţii**

Sistemul salvează datele procesate .

**Scenariul principal de succes (Fluxul de bază)**

1) Sistemul afişează un formular bazat pe o entitate de date.

2) Utilizatorul completează formularul conform cerinţelor.

3) Utilizatorul notifică sistemul cu privire la finalizarea completării datelor prin apăsarea butonului de trimitere sau a tastei *ENTER*.

4) Sistemul colectează datele şi notifică utilizatorul cu privire la intrare în starea de procesare prin afişarea unei măşti de încărcare.

5) Sistemul validează datele conform configurărilor entităţii de date.

6) După finalizarea procesării datelor, sistemul notifică utilizatorul prin afişarea vederii încheierii operaţiunii cu succes.

**Extensii (Fluxuri alternative)**

**1.a.** Denumire : **Completarea câmpurilor intuitive**

Formularele pot conţine câmpuri intuitive (de tip *autocomplete*) în scopul simplificării şi normalizării introducerii datelor.

Utilizatorul poate modifica parametrii de căutare introduşi în orice unitate de timp.

1) Utilizatorul introduce minim trei caractere în câmpul intuitiv.
2) Sistemul caută în baza de date intrări care conţin şirul de caractere introdus şi notifică utilizatorul cu privire la schimbarea stării.
3) Sistemul afişează rezultatele căutării.
4) Utilizatorul selectează un element sau poate modifica parametrii introduşi.

**5-6.a.** Denumire : **Tratarea erorilor**

Sistemul va monitoriza încheierea cu succes a unei cereri sau va notifica utilizatorul în cazul eşecului.

1) Sistemul client constată invaliditatea datelor introduse sau primeşte un răspuns de eşec la cererea de procesare.
2) Sistemul afişează mesajele de eroare şi reactivează posibilitatea introducerii şi modificării datelor.
3) Utilizatorul corectează erorile de integritate a datelor şi retrimite formularul spre procesare.

**2.** *Denumire :* ***Realizarea operaţiilor CRUD în vederi tabelare***

Actori principali : **Utilizatorul**

**Interesele părţilor**

1) Utilizatorul : vizualizare şi administrarea datelor
2) Sistemul : actualizarea şi păstrarea integrităţii datelor

**Precondiţii**

Utilizatorul este autentificat în aplicaţie şi are drepturile necesare pentru vizionarea şi modificarea datelor.

**Postcondiţii**

Sistemul persistă în modificările efectuate.

**Scenariul principal de succes / Fluxul de bază**

1) Sistemul afişează o vedere tabelară.
2) Utilizatorul selectează articolele de interes.
3) Utilizatorul cere vederea de detaliu pentru articolele selectate
4) Sistemul procesează cererea şi deschide ferestrele de detaliu editabile în funcţie de cererea utilizatorului.
5) În cazul editării, fluxul continuă conform specificaţiei UC 1 pentru procesarea unui formular.

**Extensii (Fluxuri alternative)**

**3.a** Denumire : **Ştergerea**

1) Utilizatorul cere ştergerea articolelor selectate.
2) Sistemul procesează cererea şi notifică utilizatorul cu privire la rezultat.
3) Sistemul actualizează vederea.

### 5.2.2. Scenarii de aplicabilitate concrete *AirlineManager* cu mapare la frameworkul *jqCls*

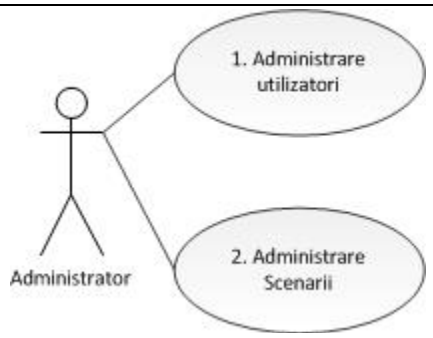



**Fig. 5.2 Vedere de ansamblu a scenariilor de aplicabilitate pentru administrator**

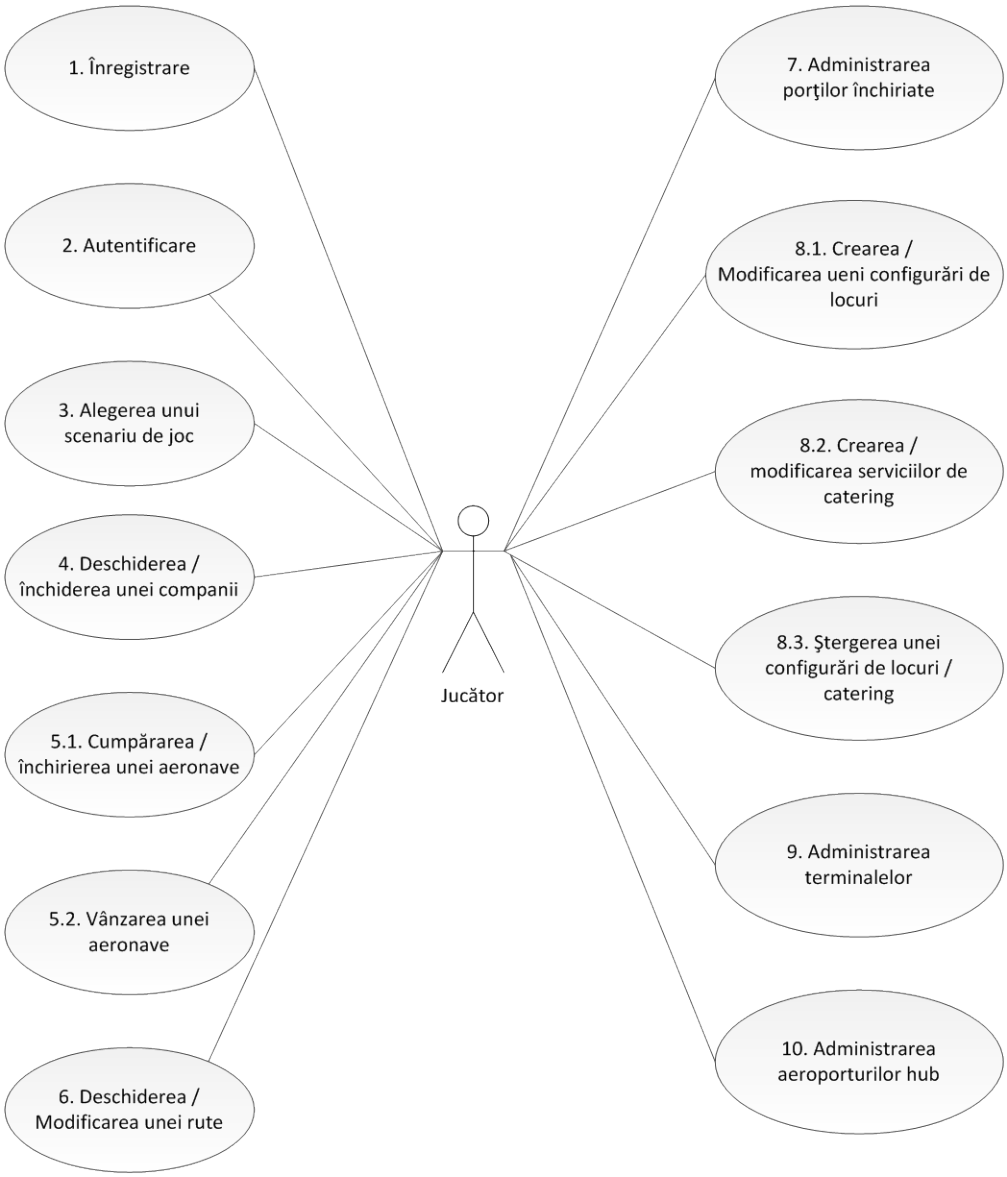



**Fig. 5.3 Vedere de ansamblu a scenariilor de aplicabilitate pentru un jucător**

Figurile 5.2 şi 5.3 conţin o prezentare de ansamblu a scenariilor de aplicabilitate identificate pentru aplicaţia demonstrativă propusă. După cum se observă, utilizatorii sistemului sunt de două tipuri : administratori şi jucători. Conform figurii 4.12, administratorii au sarcina de actualizare a listei jucătorilor şi de (UC 1) şi de pornire a scenariilor (UC 2). Activităţile jucătorilor, reprezentate prin diagrama din figura 4.13, pot fi grupate în două categorii după cum urmează : înregistrare şi autentificare (UC 1 şi UC2), respectiv acţiuni specifice desfăşurării simulării. Dintre ultimele, amintim : administrarea flotei (UC 5.1, 5.2), a rutelor (UC 6, 7, 9, 10) sau a serviciilor oferite la bord (UC 8.1, 8.2, 8.3).

Notăm că scenariile de aplicabilitate ale *Airline Manager* au fost alese pentru demonstrarea posibilităţilor de utilizare ale *jqCls*. Astfel, UC 1 – Administrare utilizatori, UC 3 – Alegerea unui scenariu de joc, UC 4 – Deschiderea şi închiderea unei companii, UC 5.2 – Vânzarea unei aeronave, UC 9 – Administrarea terminalelor şi UC 10 – Administrarea huburilor sunt exemplificări de integrare a funcţionalităţii fundamentale CRUD în vederi tabelare. Exemple enumerate anterior sunt dublate de cazuri cu complexitate crescută şi cu aplicabilitate de sinteză între componentele frameworkului. În ceea ce priveşte vederile tabelare, asemenea exemple sunt UC 5.1. Cumpărarea / închirierea unei aeronave pentru filtrare şi sortare, UC 6.1 Deschiderea şi modificarea unei rute prezintă posibilitatea utilizării în combinaţie cu formularele şi cu alte componente.

Pentru formulare exemplificăm funcţionalitatea de bază prin UC 2 Autentificare. Aplicabilitatea în cazuri complexe şi specifice este demonstrată în UC 1 Deschiderea unui scenariu de joc : integrarea câmpurilor de tip calendar, UC 1 Înregistrare : integrarea cu sisteme CAPTCHA, UC 6.1 Deschiderea şi modificarea unei rute : câmpuri intuitive, UC 8 administrarea serviciilor la bord : componente cu grad de interactivitate crescut.

### *Scenarii de aplicabilitate pentru utilizatori din categoria **administrator***

#### *1. Denumire : Administrare utilizatori*

Actori principali : **Administratorul**
**Interesele părţilor**

1) Administratorul : vizualizarea şi mentenanţa listei de utilizatori, acordarea sau revocarea privilegiilor

**Precondiţii**
Utilizatorul este autentificat în aplicaţie si are privilegii administrative.
**Postcondiţii**
Sistemul persistă modificările efectuate şi notifică utilizatorii în cauză.
**Scenariul principal de succes / Fluxul de bază**

1) Sistemul afişează vederea tabelară cu lista de utilizatori.
2) Administratorul efectuează modificări de privilegii.
3) Administratorul cere salvarea modificărilor.
4) Sistemul procesează cererea şi afişează rezultatul.

#### *2. Denumire : Deschiderea unui scenariu*

Actori principali : **Administratorul**
**Interesele părţilor**

1) Administratorul : pornirea unui fir de joc nou

**Precondiţii**

Utilizatorul este autentificat în aplicaţie şi are privilegiile necesare.
**Postcondiţii**
Sistemul efectuază cu succes operaţiunile pentru pornirea unui fir de joc nou.
**Scenariul principal de succes / Fluxul de bază**

1) Sistemul afişează formularul de configurare al unui fir de joc nou.
2) Administratorul completează parametrii de configurare referitori la : data de început şi de sfârşit din simulare, restricţii politice, situaţia iniţială predefinită a companiilor aeriene.
3) Administratorul cere pornirea simulării.
4) Sistemul înregistrează firul de joc şi notifică administratorul cu privire la succesul operaţiunii.

**Extensii (Fluxuri alternative)**

**3.a** Denumire : **Planificarea demarării firului de joc în viitor**

1) Administratorul alege data la care sistemul va porni firul de joc.
2) Sistemul planifică începerea simulării la data cerută de administrator.
3) Sistemul porneşte simularea la data planificată şi notifică administratorii

*Scenarii de aplicabilitate pentru utilizatori din categoria **jucător***

***1. Denumire :** Înregistrare*

Actori principali : **Jucătorul nou**
**Interesele părţilor**

1) Utilizatorul : crearea unui cont de utilizator pentru accesul în joc

**Precondiţii**
Nu sunt.
**Postcondiţii**
Sistemul validează contul de utilizator. Utilizatorul poate să se autentifice în joc.
**Scenariul principal de succes / Fluxul de bază**

1) Sistemul afişează vederea iniţială.
2) Utilizatorul deschide formularul de înregistrare.
3) Utilizatorul completează formularul de validare şi cere trimiterea formularului.
4) Aplicaţia client procesează datele local conform validărilor specificate pentru modelul de date.
5) Aplicaţia client trimite datele la server.
6) Serverul validează codul CAPTCHA, persistă datele şi trimite e-mailul de activare a contului.
7) Aplicaţia client afişează rezultatul şi notifică utilizatorul cu privire la trimiterea emailului de activare.
8) Utilizatorul confirmă contul prin click pe linkul de validare.
9) Sistemul activează contul.

**Extensii (Fluxuri alternative)**

**3.a.** Denumire : **Împrospătarea codului CAPTCHA**

Utilizatorul poate cere oricând împrospătarea codului CAPTCHA.

1) Utilizatorul apasă butonul de împrospătare a codului CAPTCHA.
2) Aplicaţia client cere generarea unei imagini noi modificând parametrul temporal din calea imaginii.
3) Aplicaţia server răspunde cu o imagine nouă.

4) Aplicaţia client afişează imaginea nouă.

### *2. Denumire : Autentificare*

Actori principali : **Jucătorul înregistrat**
**Interesele părţilor**
1) **Jucătorul înregistrat** : autentificarea în aplicaţie
2) **Sistemul** : determinarea utilizatorului şi restricţionarea accesului neautorizat

**Precondiţii**
Nu sunt.
**Postcondiţii**
Utilizatorul se autentifică în aplicaţie cu credenţialele alese la înregistrare.
**Scenariul principal de succes / Fluxul de bază**
1) Sistemul afişează vederea iniţială.
2) Utilizatorul completează formularul de autentificare.
3) Sistemul compară datele de autentificare cu cele existente.
4) În cazul reuşitei autentificării, sistemul afişează interfaţa cu funcţionalitate extinsă dedicată utilizatorilor.

### *3. Denumire : Alegerea unui scenariu de joc*

Actori principali : **Jucătorul înregistrat**
**Interesele părţilor**
1) **Jucătorul înregistrat** : alegerea unui mediu de simulare pentru sesiune
2) **Sistemul** : încărcarea resurselor necesare simulării

**Precondiţii**
Jucătorul este autentificat.
**Postcondiţii**
Sistemul încarcă în sesiune datele cu privire la scenariul ales. Acestea includ companiile aeriene ale utilizatorului şi parametrii generali ai simulării.
**Scenariul principal de succes / Fluxul de bază**
1) Sistemul afişează vederea tabelară a scenariilor disponibile şi a companiilor aeriene deschise.
2) Jucătorul alege compania aeriană.
3) Sistemul încarcă resursele pentru simulare

### *4. Denumire : Deschiderea şi închiderea unei companii aeriene*

Actori principali : **Jucătorul înregistrat**
**Interesele părţilor**
1) **Jucătorul înregistrat** : configurarea unei companii aeriene pentru intrarea în joc

**Precondiţii**
Jucătorul este autentificat.
**Postcondiţii**
Sistemul salvează datele de configurare a companiei aeriene.
**Scenariul principal de succes / Fluxul de bază**
1) Sistemul afişează vederea tabelară a scenariilor disponibile în care este posibilă înfiinţarea unei companii aeriene.
2) Jucătorul alege scenariul şi cere deschiderea unei companii aeriene.

3) Sistemul prezintă situaţia iniţială a scenariului şi formularul de configurare al companiei aeriene.
4) Utilizatorul alege numele şi sediul central al companiei aeriene dintr-un câmp intuitiv.
5) Sistemul salvează datele şi răspunde prin actualizarea vederii iniţiale.

### *5. Denumire : Administrarea flotei*

#### *5.1. Denumire : Cumpărarea / închirierea unei aeronave*

Actori principali : **Jucătorul înregistrat**
**Interesele părţilor**
1) **Jucătorul înregistrat** : cumpărarea / închirierea unei aeronave pentru definirea unei rute noi

**Precondiţii**
Jucătorul este autentificat.
**Postcondiţii**
Sistemul actualizează flota companiei şi situaţia financiară.
**Scenariul principal de succes / Fluxul de bază**
1) Utilizatorul deschide vederea de cumpărare şi închiriere.
2) Sistemul afişează lista cu ofertele de aeronave disponibile pentru comandă sau cumpărare imediată.
3) Utilizatorul alege aeronavele dorite şi modalitatea de utilizare : cumpărare, închiriere, leasing.
4) Utilizatorul confirmă operaţiunea.
5) Sistemul actualizează flota şi situaţia financiară a companiei.

**Extensii (Fluxuri alternative)**

**3.a.** Denumire : **Comandarea unui avion de la producător**

Jucătorul poate lansa comenzi de fabricaţie producătorilor de avioane.
1) Utilizatorul alege tipul de avion.
2) Sistemul afişează configurările de motoare şi accesorii disponibile.
3) Utilizatorul alege configurările şi plasează comanda.

### *6. Denumire : Administrarea rutelor de zbor*

#### *6.1. Denumire : Deschiderea şi modificarea unei rute*

Actori principali : **Jucătorul înregistrat**
**Interesele părţilor**
1) **Jucătorul înregistrat** : definirea rutelor de zbor pentru generarea profitului

**Precondiţii**
Jucătorul este autentificat.
**Postcondiţii**
Sistemul actualizează rutele şi reevaluează împărţirea călătorilor.
**Scenariul principal de succes / Fluxul de bază**
1) Jucătorul deschide vederea de administrare a rutelor.
2) Sistemul afişează rutele disponibile.
3) Jucătorul selectează ruta pentru care doreşte detalii.

4) Sistemul afişează vederea de detaliu care include aeroporturile, frecvenţele, reprezentarea pe hartă, distribuţia pasagerilor şi veniturile totale ale rutei.
5) Utilizatorul modifică datele rutei : modifică aeronavele sau frecvenţele (adăugare sau ştergere) şi confirmă acţiunea.
6) Sistemul validează modificările şi recalculează distribuţia pasagerilor între companiile concurente şi actualizează vederea cu rezultatele.

**Extensii (Fluxuri alternative)**

**3.a.** Denumire : **Jucătorul deschide o rută nouă**

Jucătorul poate să deschidă rute noi în limita restricţiilor scenariului.
1) Sistemul afişează vederea de detaliu a unei rute.
2) Utilizatorul defineşte ruta (plecare, escale, destinaţie).
3) Sistemul validează ruta conform restricţiilor politice ale scenariului şi afişează răspunsul.
4) Utilizatorul selectează aeronavele şi adaugă frecvenţe.
5) Sistemul validează aeronavele utilizate din punctul de vedere al capacităţii acestora de a efectua zborurile : condiţii de aterzare şi decolare – lungimea pistei, distanţă, timp de zbor. Sistemul validează numărul de frecvenţe în raport cu porţile închiriate.
6) Utilizatorul cere deschiderea rutei.
7) Sistemul continuă cu pasul 6.

### *7. Denumire : Administrarea porţilor închiriate*

Actori principali : **Jucătorul înregistrat**

**Interesele părţilor**
1) **Jucătorul înregistrat** : închirierea şi administrarea porţilor pentru operarea rutelor

**Precondiţii**

Jucătorul este autentificat.

**Postcondiţii**

Sistemul actualizează porţile închiriate şi timpii de întârziere conform gradului de utilizare al porţilor.

**Scenariul principal de succes / Fluxul de bază**
1) Jucătorul deschide vederea de administrare a porţilor.
2) Sistemul afişează porţilor disponibile după aeroport.
3) Jucătorul creşte sau scade numărul porţilor închiriate la un aeroport.
4) Sistemul validează modificările asigurând că numărul porţilor este suficient pentru efectuarea zborurilor definite.

### *8. Denumire : Administrarea serviciilor la bord*

#### *8.1. Denumire : Crearea / Modificarea unei configurări de locuri*

Actori principali : **Jucătorul înregistrat**

**Interesele părţilor**
1) **Jucătorul înregistrat** : administrarea configurărilor de locuri pentru operarea rutelor

**Precondiţii**

Jucătorul este autentificat.

**Postcondiţii**

Sistemul salvează configurările şi actualizează distribuţia pasagerilor pe rutele în cauză conform gradului de confort.

**Scenariul principal de succes / Fluxul de bază**

1) Jucătorul deschide vederea de administrare a locurilor.
2) Sistemul afişează lista de aeronave din flotă şi configurările de locuri definite.
3) Jucătorul selectează tipul de aeronavă şi cere definirea unei configurări noi sau modificarea unei existente.
4) Sistemul afişează formularul de configurare.
5) Jucătorul introduce numărul de locuri pentru clasele first, bussiness şi economy.
6) Sistemul validează datele conform capacităţii avionului selectat, afişează costul aplicării configurării de locuri şi punctajul de confort obţinut.
7) Jucătorul salvează configurarea şi alege aeronavele pentru care doreşte să aplice configurarea.
8) Sistemul verifică situaţia financiară a companiei aeriene, aplică configurările şi recalculează distribuţia pasagerilor pentru rutele afectate.

### 8.2. *Denumire : Crearea / Moficarea serviciilor de catering şi multimedia*

Actori principali : **Jucătorul înregistrat**

**Interesele părţilor**

1) **Jucătorul înregistrat** : administrarea serviciilor de catering şi multimedia puse la dispoziţia pasagerilor în timpul zborului

**Precondiţii**

Jucătorul este autentificat.

**Postcondiţii**

Sistemul salvează configurările şi actualizează distribuţia pasagerilor pe rutele în cauză şi gradul de satisfacţie dat de serviciile oferite.

**Scenariul principal de succes / Fluxul de bază**

1) Jucătorul deschide vederea de administrare a serviciilor în timpul zborului.
2) Sistemul afişează lista seturilor de servicii existente.
3) Jucătorul selectează setul de servicii pe care doreşte să îl modifice sau cere definirea unui pachet nou.
4) Sistemul afişează formularul de configurare.
5) Jucătorul selectează serviciile de catering şi multimedia oferite.
6) Sistemul calculează costurile serviciilor selectate şi punctajul pachetelui.
7) Jucătorul confirmă modificarea sau crearea pachetului.
8) Sistemul salvează modificările efectuate şi afişează formularul de aplicare al serviciilor.
9) Jucătorul selectează zborurile pe care doreşte aplicarea pachetului sau le defineşte generic în funcţie de distanţă.
10) Sistemul recalculează distribuţia pasagerilor.

# 6. Proiectare de detaliu şi implementare

jqCls este un framework open-source de sinteză organizat modular pe nivele de funcţionalitate care are ca scop principal facilitarea aplicării paradigmei obiectuale în JavaScript. În elaborarea frameworkului am avut în vedere următoarele principii generale : organizarea codului în pachete raportat la locaţia pe disc, respectiv spaţii de nume, clase şi interfeţe jqCls în JavaScript şi separarea funcţionalităţii conform şablonului arhitectural Model-View-Controller.

Integrarea soluţiilor existente într-un cadru complet şi general reprezintă contribuţia principală a frameworkului jqCls la dezvoltarea comunităţii open-source.

Din punct de vedere funcţional, considerăm că un framework JavaScript modern trebuie să trateze manipularea arborelui DOM, reprezentarea datelor, comunicarea transparentă printr-o interfaţa unitară şi consistentă cu furnizorii de persistenţă a datelor utilizând protocoalele şi tehnologiile existente, definirea şabloanelor de pagină. Combinaţia elementelor enumarate mai sus rezultă într-un set complet, general şi extensibil de componente abstracte pentru realizarea aplicaţiilor Web cu un grad ridicat de interactivitate. Nu în ultimul rând, pentru facilitarea abordării JavaScript în manieră obiectuală şi a programării bazate pe evenimente, vom pune la dispoziţia utilizatorilor implementări abstracte ale şabloanelor de design cu un spectru vast de valabilitate *Singleton* şi *Observer*.

## 6.1. Arhitectura conceptuală

jqCls este proiectat ca parte integrantă a soluţiilor arhitecturale de tip client-server orientate pe servicii şi în care componentele comunică prin formate de transport al datelor. În arhitetura aplicaţiilor Web client-server, frameworkul este localizat în mediul navigatorului şi rulează prin intermediul motorului JavaScript după cum am ilustrat în figura 6.1.

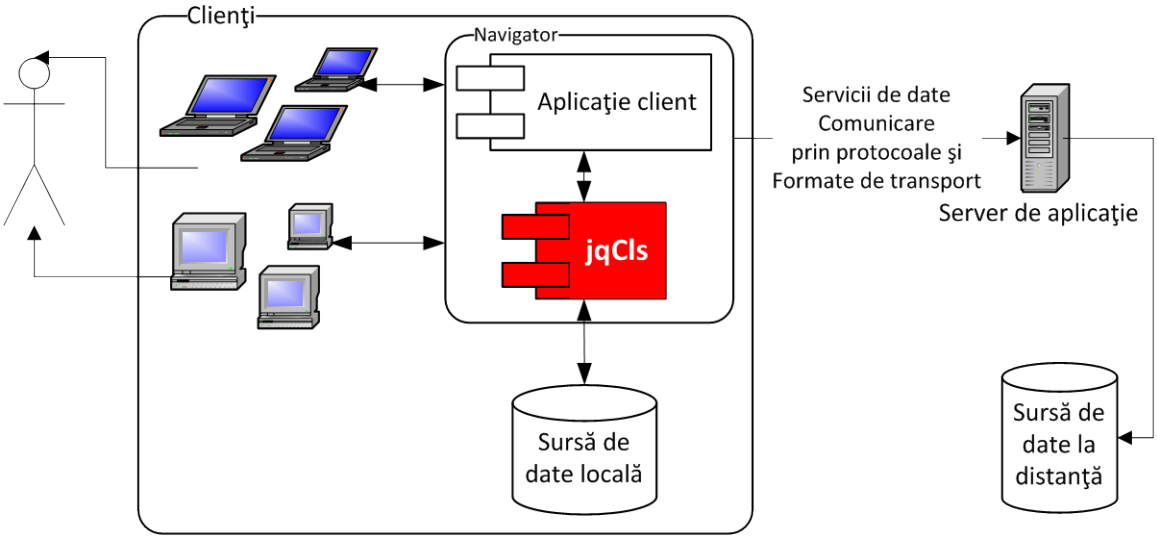



**Fig. 6.1 Poziţionarea jqCls în arhitectura aplicaţiilor Web**

Având în vedere aspectele de poziţionare în arhitectura aplicaţiilor Web şi clasificarea[1] produselor asemănătoare, jqCls se încadrează în categoria frameworkurilor JavaScript open-source cu scop general destinate navigatorului.

[1] Vezi Tabelul 3.3 Clasificarea librăriilor JavaScript

La nivel conceptual, frameworkul este împărţit în patru nivele de funcţionalitate bine delimitate.

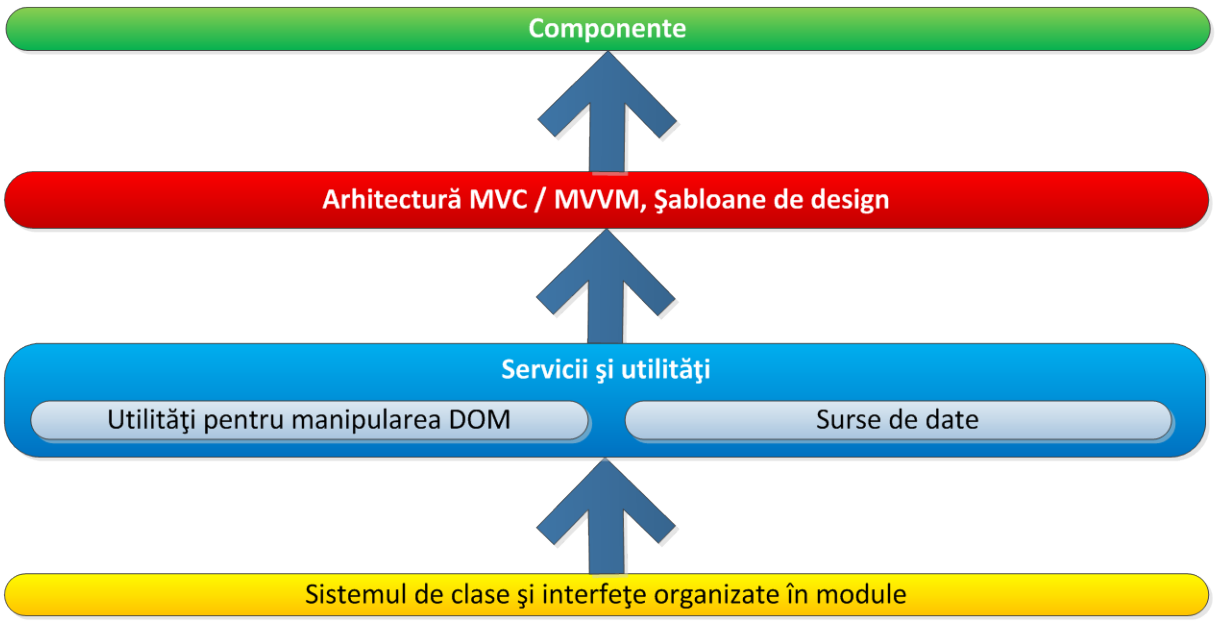



**Fig. 6.2 Arhitectura conceptuală a frameworkului jqCls**

Din figura 6.1 înţelegem că, jqCls este construit pornind de la implementarea infrastructurii necesare pentru crearea unui sistem modular ierarhic de clase şi interfeţe. Nivelul de servicii al frameworkului conţine utilităţi pentru operaţii cu arborele DOM, o interfaţă transparentă pentru fluxul de date în client şi mediul de stocare şi obiecte pentru reprezentarea colecţiilor de date. Etajul următor al jqCls este dedicat organizării codului şi implementării abstracte a şabloanelor de design. Componentele jqCls combină funcţionalitatea nivelurilor inferioare oferind o interfaţă de programare obiectuală unitară şi simplificată.

## 6.2. Detalii arhitecturale

Pornind de la arhitectura conceptuală prezentată, în figura 6.3 detaliem implementarea nivelelor de funcţionalitate enumerate şi descrise generic mai sus. Astfel, nucleul frameworkului, *jqCls.core* creează infrastructura necesară sistemului ierarhic de clase şi interfeţe. Responsabilitatea organizării în module şi de administrare a dependinţelor revine requireJs. Considerăm important a nota că, în dezvoltarea frameworkului, a fost creat în scopul demonstrării conceptului şi un sistem propriu de încărcare dinamică a fişierelor JavaScript. Având în vedere temele adiacente pe care nu ni le-am propus în această lucrare : minifierea şi optimizarea codului, încărcătorul dinamic jqCls nu a fost dezvoltat până la capăt.

La nivelul serviciilor şi utilităţilor se situează pachetele dom, layout, list şi data, precum şi integrarea frameworkurilor underscore.js şi jQuery. Considerăm că seriviciile şi utilităţile pe care le oferă jqCls, se împart în două categorii fundamentale : operaţiile asupra arborelui DOM şi interfaţa pentru sursele de date. În acest sens, operaţiile DOM sunt tratate în pachetul cu acelaşi nume. Pachetul layout este un caz special în ceea ce priveşte operaţiile DOM având în vedere că clasele definite aici au rolul de a randa şabloanele de pagină şi conţinutul acestora.

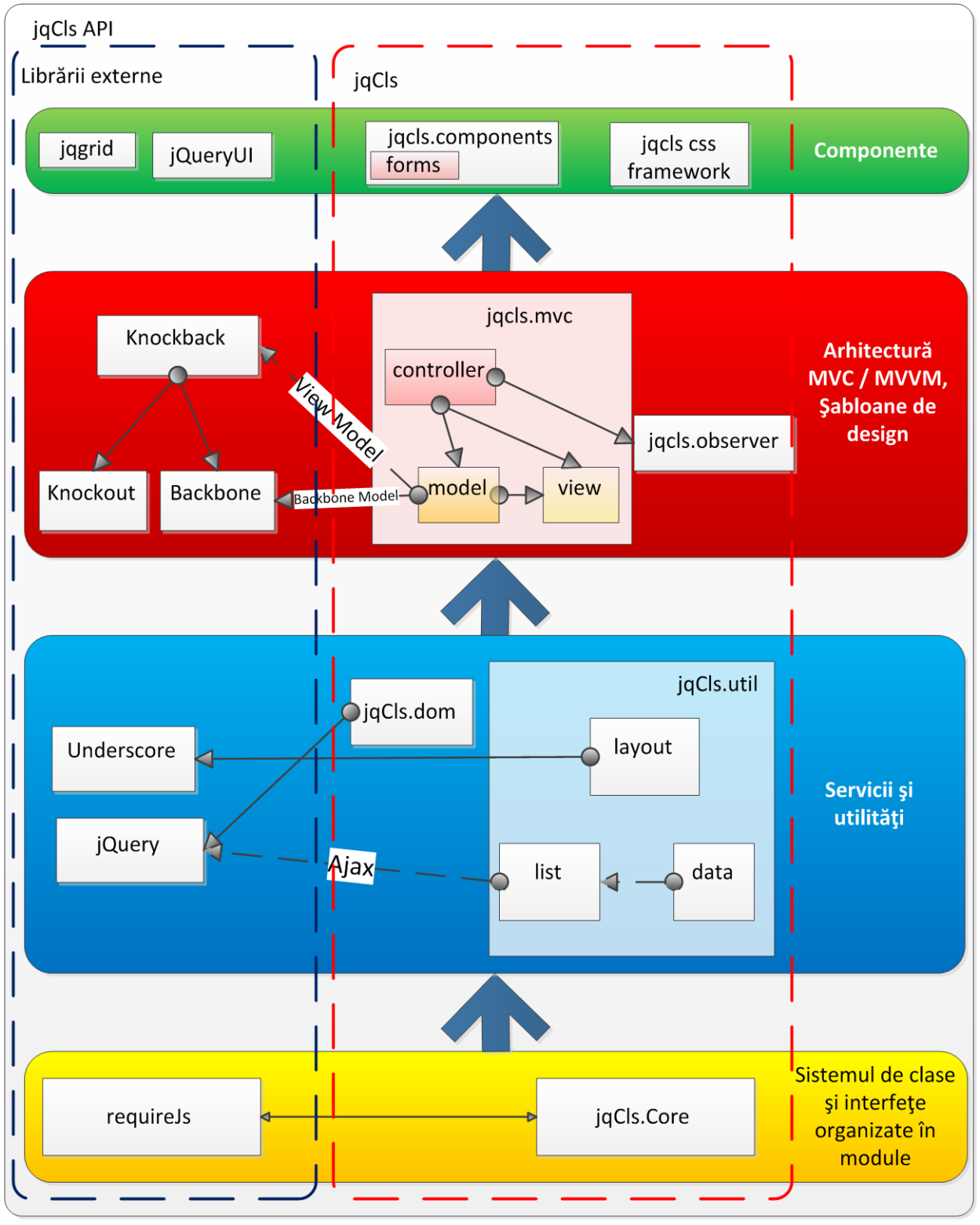



**Fig. 6.3 Diagrama de arhitectură detaliată**

Interfaţa de programare implementată în jqCls pentru operaţiile asupra seturilor de date este formată din modulele *list* şi *data*.

Rolul librăriei jQuery în ambele categorii de funcţionalitate la acest nivel este de a normaliza diferenţele de implementare între navigatoare.

Al treilea nivel al frameworkului conţine implementarea abstractă a modelelor arhitecturale şi de design prezentate în studiul teoretic. Astfel, jqCls oferă posibilitatea structurării codului după rigorile MVC. Prin librăriile Knockback (Konckout şi Backbone.js) pe care le situăm la acelaşi nivel, modelul MVC implementat în jqCls poate fi extins la Model-View-ViewModel. Modulul *observer* completează seria de funcţionalităţi cu scop de structurare a codului.

Ultimul nivel de arhitectură al frameworkului este compus din pachetul *components*. Utilizând serviciile de la nivelele inforioare, clasele din pachetul components reprezintă un set abstract de componente vizuale extensibile cu care putem realiza în mod obiectual aplicaţii client în paradigma Web. Menţionăm că, specific frameworkului propus, componentele vizuale sunt formate pe un schelet de soluţii proprii jqCls extins cu alte unităţi de funcţionalitate precum API-ul jQueryUI sau jqGrid împachetate în clase care fac posibilă abordarea problemelor în mod obiectual.

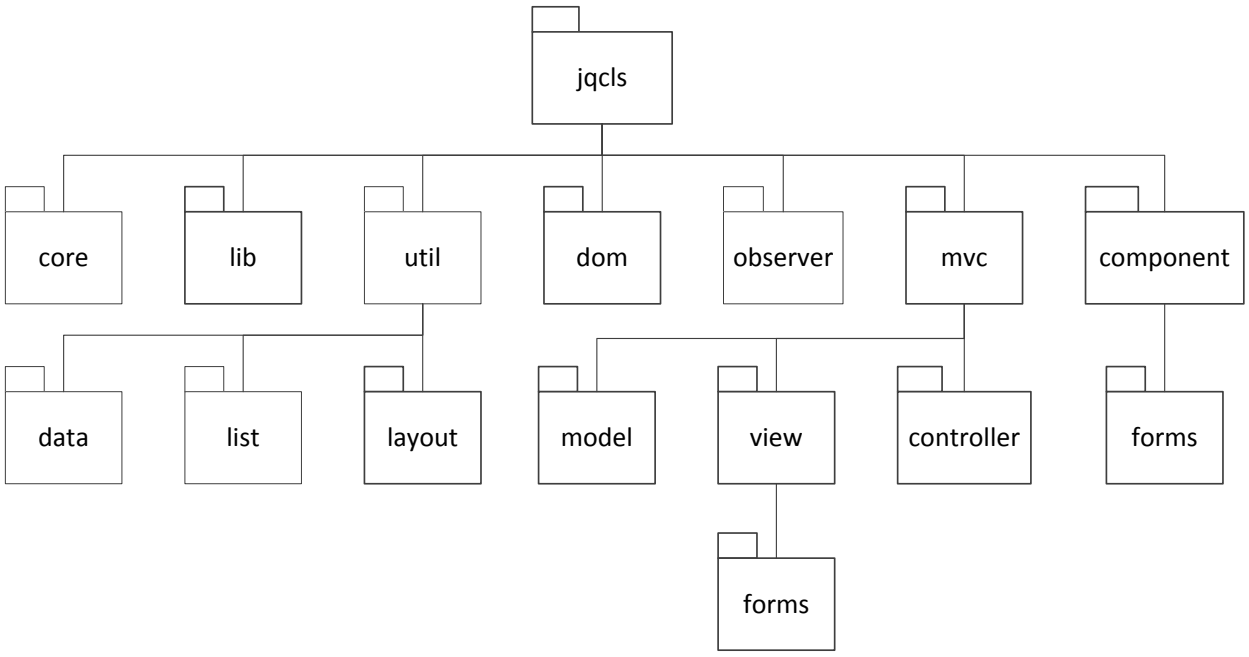


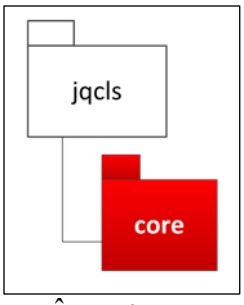

**Fig. 6.4 Diagrama de pachete**

Considerăm că figura 6.4 oferă o imagine de ansamblu clară asupra organizării frameworkului în pachete. Menţionăm că acestea au fost discutate anterior. Reţinem totuşi faptul că pachetul *lib* conţine librăriile externe integrate în jqCls şi face excepţie de la regula generală de mapare a pachetelor fizice pe spaţiile de nume şi modulele frameworkului. Pentru a nu extinde nejustificat şi pentru a nu polua spaţiul de nume jqCls, frameworkurile externe sunt integrate cu structura originală independentă şi sunt specificate ca dependinţe prin *requireJs*.

## 6.3. Implementare

Granularitatea, identificarea şi separarea corectă a categoriilor de funcţionalitate în pachete şi module au reprezentat un principiu important pe durata procesului de implementare al frameworkului jqCls. După cum vom observa în continuare, o regulă aplicabilă general pentru pachetele librăriei propuse este specificarea funcţionalităţii organizate într-o interfaţă.

### 6.3.1. Sistemul ierarhic de clase şi interfeţe organizate modular




Conform precizărilor, nucleul jqcls implementează în JavaScript un sistem de clase şi interfeţe organizate modular. Diagrama de clase care compun acest pachet este prezentată în figura 6.5. Observăm că în componenţa modului *core* intră clasele *Namespace*, *Class*, *Interface*, *Application*, interfaţa *IApplication* şi excepţia *MethodMissing*.

Clasa *Namespace* este responsabilă pentru iniţializarea şi definirea spaţiilor de nume. Acestea sunt reprezentate prin obiecte JavaScript standard în care proprietăţile definesc submodule.

În vederea organizării codului şi abstractizării conceptelor, jqCls implementează noţiunea de clasă conform aspectelor teoretice discutate în capitolele anterioare. Astfel, considerăm că o clasă jqCls este o entitate de reprezentare şi funcţionalitate bine delimitată. Obiectele *Class* şi *Interface* din modulul core asigură infrastructura necesară aplicării conceptelor de programare obiectuală utilizând jqCls. Notăm că definirea unei clase implică crearea spaţiului de nume aferent şi verificarea implementării interfeţelor declarate. Problema conceptelor OOP de abstractizare şi moştenire este abordată în jqCls în mod prototipic cu extensii şi posibilitatea de utiliza şi tehnica numită *mix-in*. Menţionăm că atât modelul de moştenire prototipică[1], cât şi tehnica *mix-in*[2] au fost prezentate în capitolul patru. Algoritmul ales pentru reutilizarea codului poate fi rezumat la definirea unei funcţii JavaScript cu rol de proxy, iniţializarea proprietăţii *prototype* a acesteia cu clasa extinsă şi verificarea conformităţii cu setul de metode din interfeţele declarate sau delegarea responsabilităţii în cazul definirii unei clase abstracte.

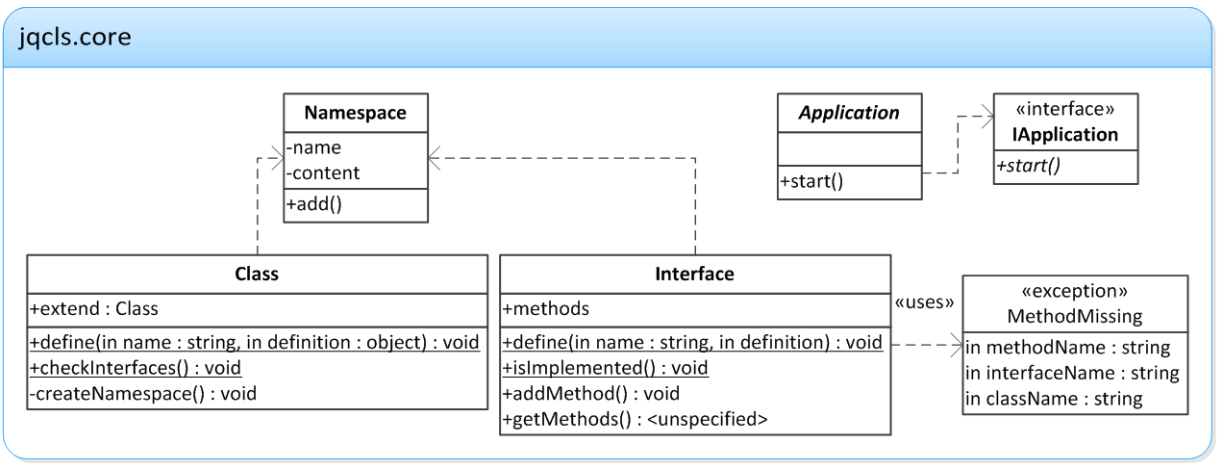



**Fig. 6.5 Diagrama de clase a pachetului core**

[1] Vezi Moştenirea prototipică, p. 29

[2] Vezi Alte metode de reutilizare a codului. Clase prin părţi. Împrumutarea metodelor. Clase abstracte. Moştenire multiplă, p. 37

Diagrama de secvenţă din figura Fig. 6.6 ilustrează paşii urmaţi în jqCls la apelul metodei statice `jqcls.Class.define(namespace, definitionObject)`.

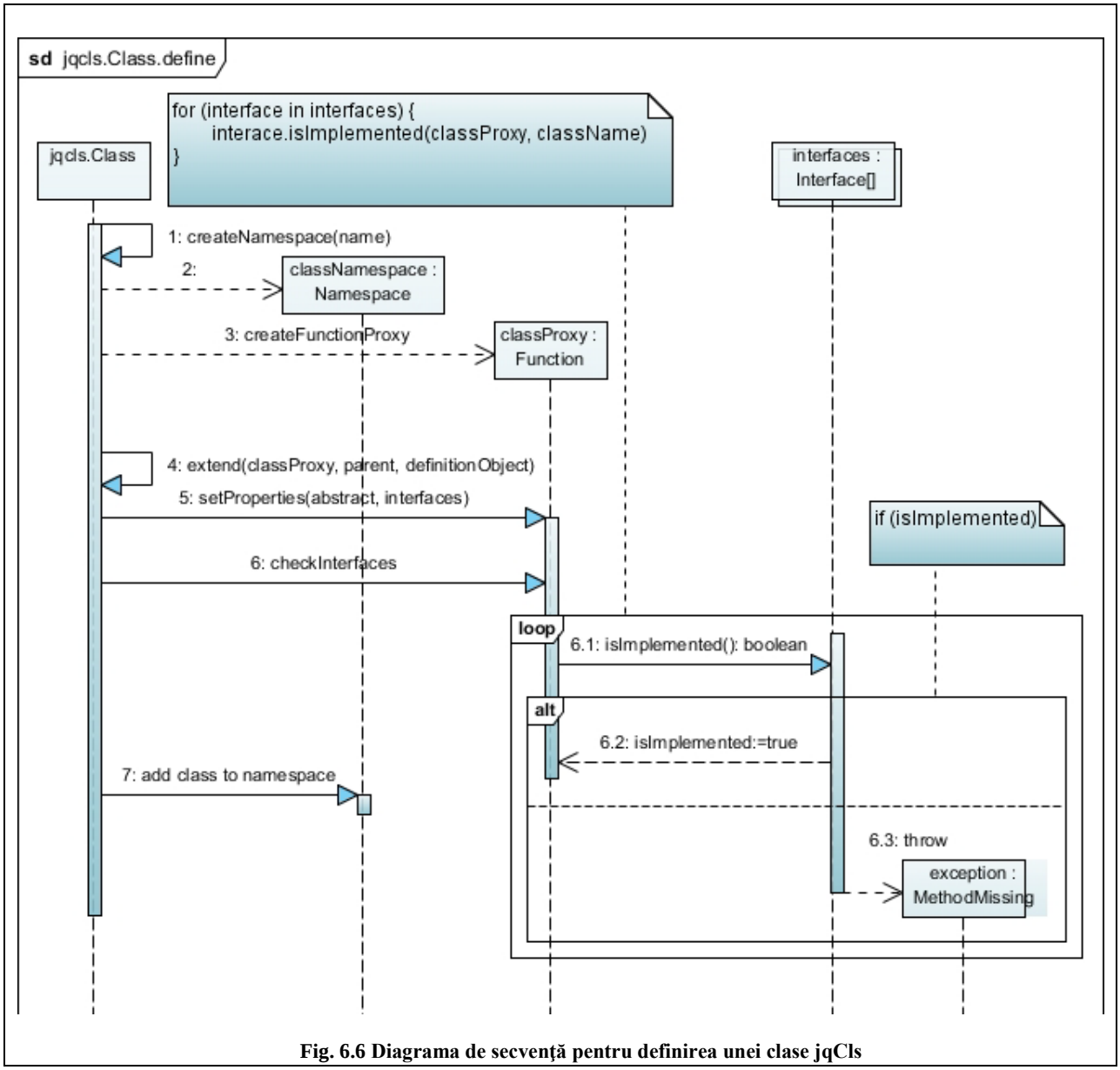



**Fig. 6.6 Diagrama de secvenţă pentru definirea unei clase jqCls**

Nivelul de implementare al conceptului de interfaţă în jqCls este *duck-typing*[1]. În acest sens, frameworkul validează implementarea unei interfeţe conform funcţionalităţii clasei. O interfaţă jqCls este reprezentată similar claselor. Metodele care alcătuiesc interfaţa sunt declarate sub forma unui vector de şiruri de caractere reprezentând semnătura funcţiilor. Similar altor limbaje de programare, omiterea unei funcţionalităţi din setul definit într-o interfaţă este posibilă prin definirea atributului *abstract* cu valoarea *true*. În caz contrar, frameworkul va semnala lipsa metodei aruncând o excepţie de tip *MethodMissing*.

[1] Vezi Emularea interfeţelor în JavaScript, p. 44

Amintim că, după cum am sugerat mai sus, clasele abstracte pot fi definite în jqCls utilizând o combinaţie între interfeţe şi entităţi de tip *Class*.

Pentru facilitarea operaţiilor de iniţializare ale unei aplicaţii, modulul core conţine clasa abstractă *Application* care implementează conform figurii 6.5 interfaţa *IApplication* care defineşte metoda *start*. Reţinem că utilitatea acesteia constă în execuţia dependentă de disponibilitatea tuturor resurselor necesare.

După cum am menţionat, sistemul de pachete al jqCls este implementat prin utilizarea API-ului *requireJs*. Notăm că jqCls distinge între pachete şi module. Astfel, pachetele reprezintă organizarea fişierelor pe disc pe parcursul procesului de dezvoltare, iar modulele sunt structura de spaţii de nume JavaScript. Regula generală este de mapare între module şi pachete, însă există şi excepţii. j*qcls.Class* şi jqcls.Interface sunt două dintre aceste excepţii. Având în vedere frecvenţa utilizării, excepţiile menţionate sunt motivate prin dorinţa de eficientizare a procesului de dezvoltare.

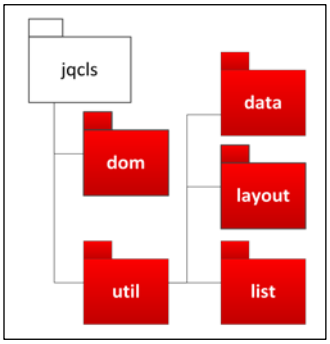
RequireJS este un încărcător de fişiere şi module JavaScript compatibil cu API-ul AMD (*Asynchronous Module Definition*) de încărcare asincronă a fişierelor şi modulelor JavaScript. RequireJS defineşte un modul ca fiind încapsularea unei bucăţi de cod într-o unitate utilă care îşi exportă capabilităţile.

Formatul AMD a luat fiinţă din dorinţa de a îmbunătăţi practica actuală de încărcare şi ordonare manuală a fişierelor JavaScript prin noduri HTML de tip *script*. Un avantaj important al formatului AMD este posibilitatea de specificare şi identificare a modulelor şi dependenţelor acestora printr-o denumire unică – asemănătoare cu cea a pachetelor Java – independentă de calea fizică a fişierului. Ideea de bază a formatului AMD este încapsularea definiţiei unui modul. Importanţa deosebită a API-ului AMD este dată de rezolvarea şi încărcarea automată a dependenţelor unui modul. Un alt punct forte al formatului AMD implementat de RequireJS este tratarea interdependenţei modulelor.

Nu în ultimul rând, menţionăm că legătura strânsă dintre RequireJS şi modulul de optimizare, minifiere şi concatenare a codului r.js a fost un factor important care a determinat alegerea acestui instrument în detrimentul soluţiei proprii. Notăm că instrumentul jqCls pentru încărcarea dinamică a resurselor îndeplineşte obiectivele fixate în această lucrare. Considerăm că sarcina de optimizare, minifiere şi concatenare este dincolo de scopul lucrării de faţă şi nerealistă în timpul avut la dispoziţie. Pe de altă parte, amintim că unul dintre obiectivele acestei lucrări este identificarea, utilizarea şi integrarea instrumentelor open-source disponibile.

Însumând, jqCls defineşte propriul sistem ierarhic de module, clase şi interfeţe şi se bazează pe funcţionalitatea RequireJS pentru definirea pachetelor şi tratarea interdependenţelor acestora.

### 6.3.2. Servicii şi utilităţi




Operaţiile pe arborele DOM şi cu seturile de date constituie obiectul principal de activitate al nucleului jqCls. Acestea sunt grupate la nivelul de utilităţi ale frameworkului jqCls în pachetele *dom* şi *util*. La rândul lui, pachetul *util* distinge între *data*, *list* şi *layout* conform diagramei de pachete prezentate. Ambele pachete majore de la acest nivel depind de normalizarea jQuery a comportamentului navigatoarelor.

Diagrama de clase detaliată a nivelului de servicii şi utilităţi este prezentată în figuraFig. 6.7.

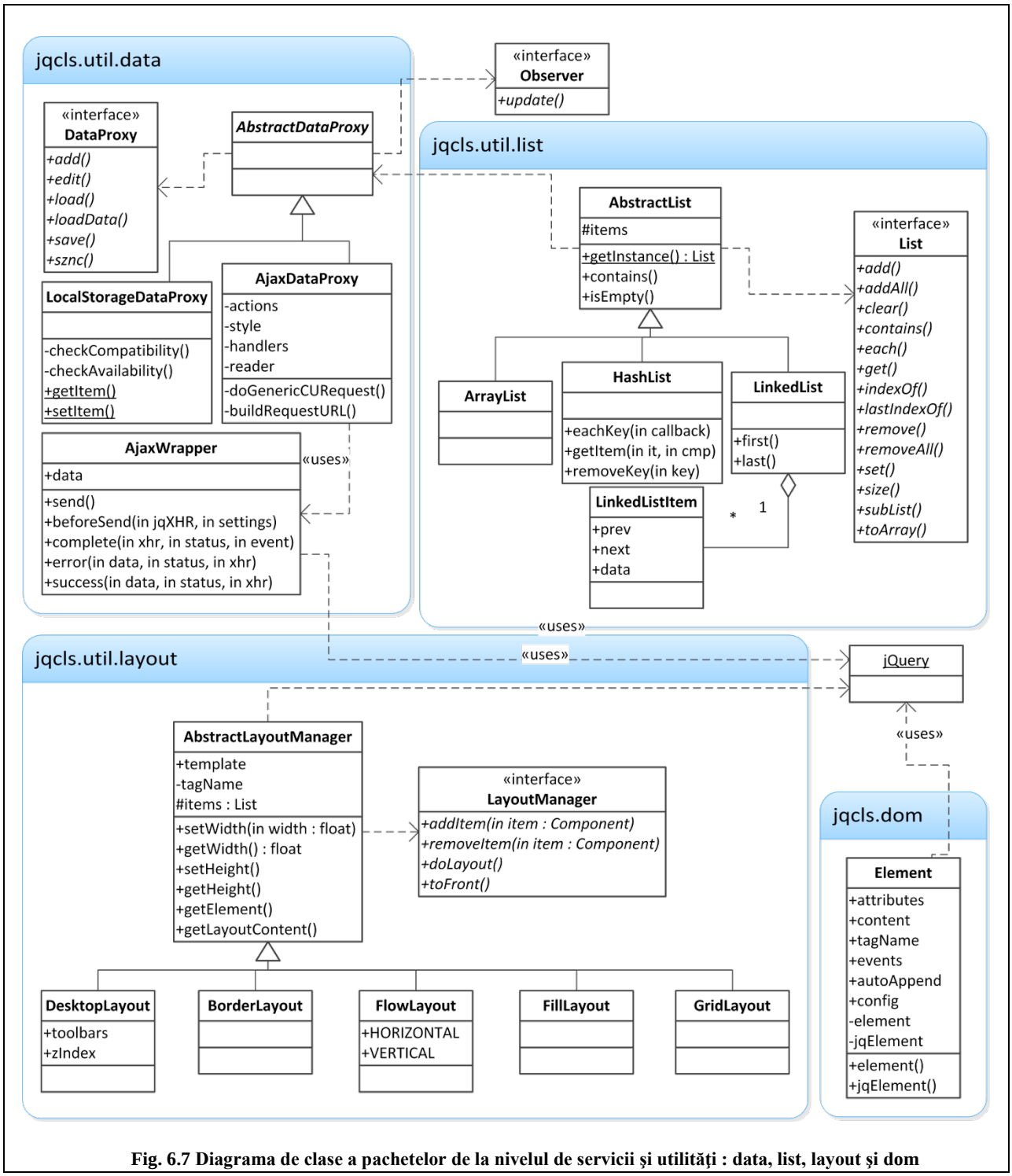



**Fig. 6.7 Diagrama de clase a pachetelor de la nivelul de servicii şi utilităţi : data, list, layout şi dom**

## *jqcls.dom*

Aşa cum rezultă din diagrama de clase pacheteul *jqcls.dom* conţine clasa *Element*. După cum sugerează şi numele, aceasta este responsabilă pentru reprezentarea JavaScript a elementelor DOM. În acest sens, *Element* se încadrează în categoria claselor de tip *învelitoare* (*wrapper*) pentru crearea şi accesul la elementele HTML sau la atributele acestora în mod obiectual prin

JavaScript. Specific claselor învelitoare, *Element* conţine atributele standard ale unui nod HTML. Dintre acestea amintim tipul – proprietatea *tagName* –, clasele css (*class*) sau proprietatea *ID*. Pentru reprezentarea atributelor unui element HTML este utilizată tehnica obiectului de configurare. Astfel, atributele unui element sunt înfăţişate într-un obiect standard JavaScript de configurare – *config* – cu rolul unei liste de perechi cheie-valoare.

Nu în ultimul rând, clasa conţine o referinţă spre obiectul jQuery învelitor pentru elementul HTML. Unul dintre avantajele acestei implementări este posibilitatea utilizării setului normalizat de evenimente DOM jQuery.

Notăm că librăriile externe *Knockout.js* şi *Underscore.js* adaugă funcţionalităţi pentru lucrul cu arbole DOM prin motoarele de procesare a şabloanelor HTML. Folosirea acestora va fi exemplificată în capitolul dedicat instalării şi utilizării frameworkului.

### *jqcls.util*

***jqcls.util.layout*** este un nivel superior de utilizare a arborelui DOM. În acest sens, clasele concrete din pachetul *layout* sunt responsabile pentru aranjarea componentelor jqCls în pagină. Funcţionalitatea comună a claselor din pachetul *layout* este grupată în clasa abstractă *AbstractLayoutManager* şi specificată de interfaţa *LayoutManager*. Metoda *doLayout* are o importanţă majoră fiind responsabilă pentru introducerea vederilor HTML în şablonul de pagină administrat. Din diagrama de clase prezentată în figura 6.7 notăm că tipurile administratorilor de compoziţie disponibile în jqCls sunt *DesktopLayout* – reţinem că acesta este folosit în cazul interfeţelor MDI –, *BorderLayout*, *FlowLayout*, *FillLayout* şi *GridLayout*. Considerăm că denumirea administratorilor de compoziţie este sugestivă şi nu vom detalia funcţionalităţile specifice acestora.

Combinaţia pachetelor ***jqcls.util.data*** şi ***jqcls.util.list*** alcătuieşte blocul de bază pentru opraţiile cu seturi de date. În acest sens *jqcls.util.list* conţine implementări ale unor cazuri diverse de reprezentare a colecţiilor de date. Interfaţa *List* defineşte comportamentul general al implementărilor de seturi de date. Corespondentul Java este în mare pare parte sursa de inspiraţie principală pentru funcţionalitatea specificată de *List* şi pentru implementările concrete.

Notăm că jqcls suportă trei tipuri de colecţii. În ordinea înfăţişării în diagrama de clase, acestea sunt *ArrayList*, *HashList* şi LinkedList. Implementarea *ArrayList* porneşte de la obiectele JavaScript de tip *Array* şi extinde sau mapează funcţionalitatea acestora prin decorare conform specificaţiei jqCls *List*. Suportul de date al acestei variante de implementare a colecţiilor jqcls este un vector JavaScript.

*HashList* acoperă nevoia colecţiilor de tip perechi cheie-valoare. În acest caz, suportul de date este asigurat de un obiect standard JavaScript ale cărui proprietăţi au forma necesară reprezentării dorite. jqCls extinde comportamentul *Object* cu funcţionalităţile specifice seturilor de date definite în *List*.

Ultima variantă de set de date implementat este *LinkedList*. Aceasta se bazează pe principiul bine cunoscut al listelor înlănţuite. Mai exact, abordarea jqCls este de listă dublu înlănţuită circulară.

Pachetul ***jqcls.util.data*** este construit după modelul de design *Proxy*. Proxy-urile de date jqCls au rol de intermediar între furnizorul de persistenţă şi seturile de date cu care lucrează o aplicaţie. Interfaţa *DataProxy* impune implementarea metodelor pentru operaţii CRUD *add*, *edit*, *load*, *save* şi *sync*. Frameworkul jqCls distinge între surse de date locale şi la distanţă specializând implementările în *LocalStorageDataProxy* pentru operaţiile cu mediul de stocare local HTML5 *LocalStorage* şi *AjaxDataProxy* pentru transportul de date la distanţă prin

*XMLHttpRequest* sau JSONP. În cazul surselor de date la distanţă, clasa *AjaxWrapper* este responsabilă pentru comunicarea efectivă cu serverul de aplicaţie prin AJAX. Aceasta învăluie în mod obiectual API-ul AJAX normalizat pus la dispoziţie de librăria jQuery[1].

Un fapt interesant cu privire la pachetul *data* este asemănarea claselor concrete implementate cu conceptul apărut recent numit *Promises* sau *Deferred Objects*[2]. Pe scurt, *Promises* este în JavaScript o abordare obiectuală a operaţiilor cu resurse a căror disponibilitate depinde de cereri asincrone. Astfel, după cum am arătat în diagrama de clase a pachetului, *AbstractDataProxy* implementează interfaţa *jqcls.observer.Observer* fiind notificat în momentul în care datele sunt disponibile. Comportamentul unui *Proxy* de date jqCls la notificări cu privire la starea cererii efectuate poate fi specificat prin apeluri *callback* la modul general sau specific pentru fiecare mesaj în parte.

În mod normal, obiectul *LocalStorage* HTML5 este o colecţie de perechi cheie-valoare de tip şir de caractere. *LocalStorageDataProxy* extinde posibilităţile de utilizare oferind opţiunea de serializare şi deserializare a obiectelor persistente cu ajutorul formatului JSON.

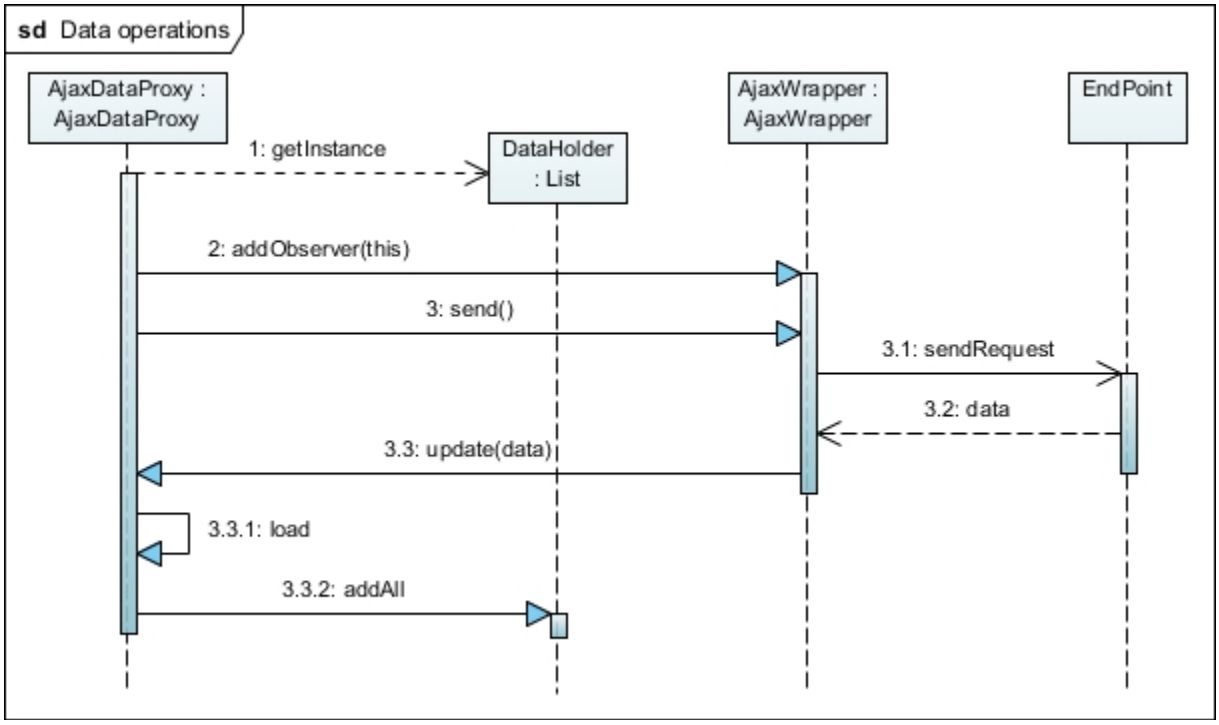
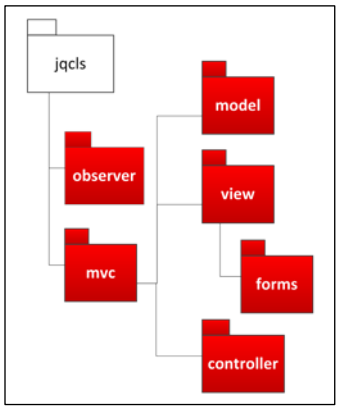



**Fig. 6.8 Diagramă de secvenţă generică pentru operaţii cu seturi de date**

Diagrama de secvenţă propusă în figura 6.8 explică modul generic în care jqCls tratează operaţiile cu seturi de date şi surse la distanţă. Menţionăm că diagrama tratează cazul specific al unei opraţii de încărcare. Având în vedere că în proces diferă doar denumirile metodelor, fluxul de mesaje poate fi aplicat universal.

[1] Vezi http://api.jquery.com/jQuery.ajax/
[2] Vezi http://api.jquery.com/category/deferred-object/

### 6.3.3. Şabloane arhitecturale şi de design




Al treilea nivel al stivei jqCls este destinat facilităţilor de construcţie a aplicaţiilor cu arhitectură complexă. La acest nivel se încadrează implementarea abstractă a modelului arhitectural MVC şi a şablonului de design *Observer* precum şi librăriile Knockback, Knockout şi Backbone. În diagrama din figura 6.13 detaliem structura pe clase a pachetelor *jqcls.mvc* şi *jqcls.Observer*.

#### *jqcls.observer*

*Observer* este un şablon fundamental al programării orientate pe obiecte şi se regăseşte în jqCls la diferete nivele şi cu implementări diferite. Dintre componentele care implementează modelul *Observer* amintim API-ul jQuery pentru tratarea evenimentelor şi librăriile Knockout sau Backbone. De asemenea jqCls conţine o implementare proprie a modelului *Obsever* asupra căreia ne vom concentra în cele ce urmează.

Abordarea jqCls a modelului *Observer* are în vedere elementele teoretice relevante expuse în studiul bibliografic. Reţinem că echilibrul în raport cu tehnicile extreme *push* şi *pull* a avut o importanţă majoră. Din diagrama de clase observăm că interfaţa *Observer* care defineşte metoda *update* şi clasa abstractă *Observable* stau la baza implementării jqCls.

După cum am precizat, echilibrului între extremele *push* şi *pull* a beneficiat de atenţie sporită, fapt demonstrat de implementarea metodei *notify* care, spre deosebire de echivalentul Java, este capabilă să declanşeze evenimente specifice. Notăm că, din punctul de vedere al implementării JavaScript, abordarea aleasă face uz de evenimente şi apeluri *callback* pentru declanşarea actualizării clienţilor. Această tehnică a fost descrisă în aprofundarea bibliografică a lucrării[1]. Notăm că maniera de implementare a şablonului în jqCls este conformă cu ceea ce numim *publish-subscribe*. Obiectele de tip *Observable* monitorizează lista de clienţi şi declanşează evenimente de modificare pentru fiecare în parte.

#### *jqcls.mvc*

Model-View-Controller este un şablon arhitectural utilizat la scară largă în industrie. JqCls facilitează utilizarea acestuia prin implementarea abstractă a claselor care stau la baza MVC. În ansamlu, un *Controller* jqCls este responsabil pentru un singur model de date şi poate răspunde la evenimentele mai multor vederi.

Observăm că variaţia MVC implementată în jqCls este *Active Model*. Conform descrierii MSDN[2], aceasta presupune schimbarea stării modelului fără implicarea controllerului şi reflectarea modificărilor în vederi. Urmând şablonul general, jqCls utilizează mecanismul *Observer* pentru a evita cuplarea modelului de clienţii interesaţi de schimbări. Având în vedere că un model poate conţine reprezentarea datelor pentru mai multe vederi, evenimentele de modificare sunt legate de proprietăţile observate. Pentru a obţine acest comportament, în jqCls a fost creată interfaţa *PropertyChangeListener*. Astfel, la detectarea unei schimbări, un obiect de tip *Model* declanşează procedura *firePropertyChange* pentru fiecare observator interesat să monitorizeze starea modelului dat. Menţionăm că mecanismul a fost gândit în primul rând pentru comunicarea dintre vederi şi model. Cum implementarea interfeţei *PropertyChangeListener* nu

[1] Vezi Observer, p. 48-49
[2] http://msdn.microsoft.com/en-us/library/ff649643.aspx (accesat 24 iunie 2013)

este predefinită în clasele de bază ale pachetului *view* – *AbstractView* şi *BlockView* – adoptarea acestei tehnici este o opţiune şi nu o constrângere a frameworkului.

În cele ce urmează, detaliem implementarea MVC de tip *Active Model* în jqCls prin diagrama de clase din figura 6.9 şi prin diagrama de secvenţă din figura 6.10. După cum am precizat, interfaţa *PropertyChangeListener* este o extensie a *Observer* cu menţiunea că, metoda *firePropertyCange* este mai specifică decât *update*. Concret, *firePropertyChange* transmite proprietatea modificată identificată prin nume, valoarea veche şi pe cea nouă. Comparativ, implementarea standard *Observer* este mult mai generică.

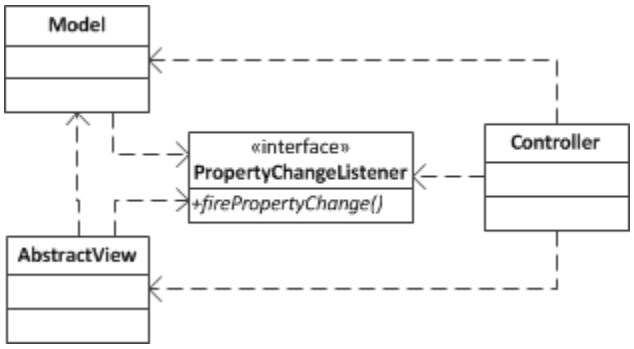



**Fig. 6.9 jqcls MVC Active Model**

Modele jqCls implementează metodele generice de scriere şi citire a proprietăţilor unui model care exploatează introspecţia facilă a obiectelor JavaScript datorată reprezentării acestora ca vectori asociativi (*hashmap*). Concret, metodele *get* şi *set* accesează proprietăţile unui obiect utilizând sintaxa de acces cu paranteze drepte (`obiect["proprietate"]`) specifică în general vectorilor. Avantajul acestei tehnici este confirmat, printre altele, de simplificarea execuţiei unei secvenţe de actualizare a vederilor.

Diagrama de secvenţă din figura 6.10 prezintă fluxul de mesaje declanşat de modificarea unei proprietăţi a modelului prin apelul metodei generice de scriere a proprietăţilor pe care am amintit-o anterior, *set*.

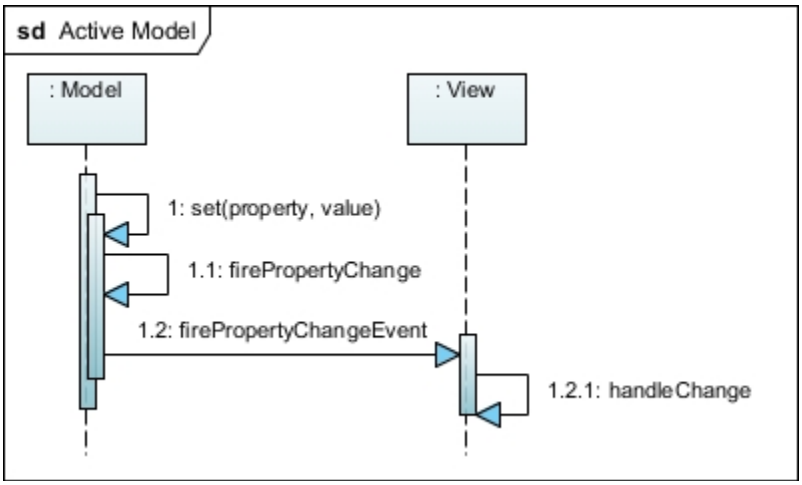



**Fig. 6.10 Diagrama de secvenţă MVC Active Model**

Diagrama de secvenţă din figura 6.10 prezintă fluxul de mesaje declanşat de modificarea unei proprietăţi a modelului prin apelul metodei generice de scriere a proprietăţilor pe care am amintit-o anterior, *set*. Implementarea acestei metode la nivelul modelelor jqCls constituie un punct central de intrare în procedura de actualizare prin apelul *firePropertyChange*. Efectul este iterarea listei de clienţi ai modelului şi declanşarea evenimentului *propertyChange*. Astfel, modificarea unei proprietăţi în modelul de date poate fi tratată individual, în funcţie de interes de către fiecare observator înregistrat în lista modelului.

Clasa *AbstractView* este baza vederilor jqCls. Aceasta extinde implementarea *Backbone* a vederilor şi oferă astfel posibilitatea utilizării funcţionalităţii *Backbone.View*. Menţionăm că vom reveni asupra subiectului librăriei *Backbone* în subcapitolul dedicat API-urilor şi frameworkurilor integrate.

Un aspect important şi general al vederilor jqCls în raport cu şablonul MVC, este respectarea principiului decompoziţiei în unităţi de reprezentare simple şi ierarhice. Pentru implementarea principiului menţionat a fost utilizată combinaţia de şabloane de design *Composite* şi *Decorator*. În acord cu rigorile modelului *Composite*, orice vedere jqCls poate fi formată din una sau mai multe alte subvederi. Detaliile de implementare ale combinaţiei *Composite* şi *Decorator* sunt documentate prin diagrama de clase din figura 6.11.

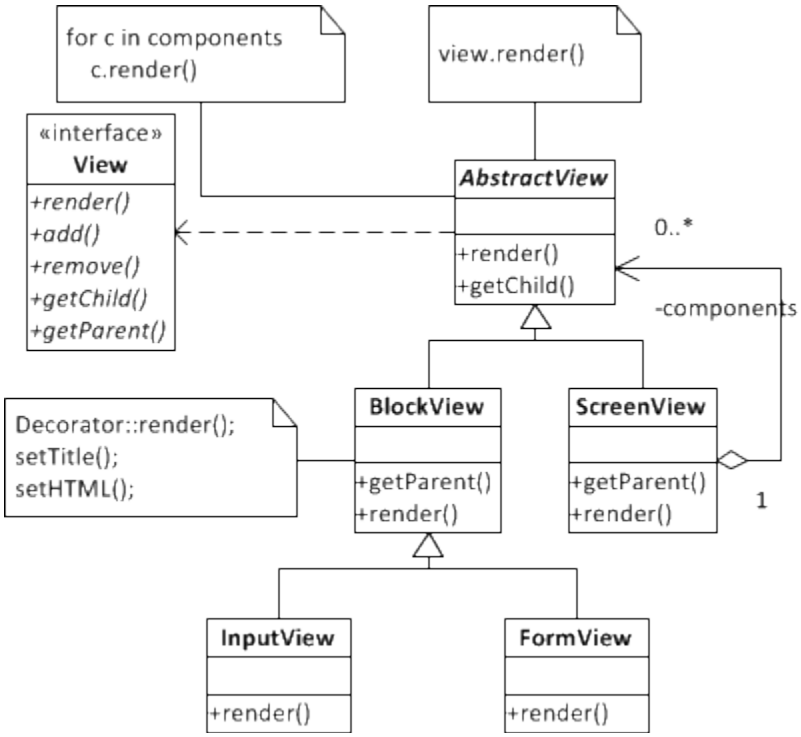



**Fig. 6.11 Implementarea combinaţiei Composite şi Decorator pentru vederi**

Aplicând definiţia teoretică a modelului *Composite* în diagrama de clase prezentată, reţinem că *View* şi *AbstractView* au rolul de a abstractiza comportamentul tuturor vederilor. *View* declară interfaţa obiectelor din compoziţie în timp de *AbstractView* implementează o parte a comportamentului comun. Din punctul de vedere al variaţiei modelului, notăm legătura bidirecţională între obiectul compus şi conţinutul său. *InputView*, *BlockView* sau *FormView* pot

fi tratate ca frunze şi implementează toate metodele definite în interfaţa unei vederi, dar pot reprezenta în acelaşi timp şi obiecte compozit. Prin definiţie, şi conform diagramei, *ScreenView* este un compozit şi reprezintă în frameworkul jqCls rădăcina componentelor afişate pe ecran. În general, specific modelului *Composite*, *ScreenView* va delega responsabilităţile spre obiectele din compoziţie.

După cum am precizat, afişarea elementelor vizuale în jqCls se face prin combinaţia dintre *Composite* şi *Decorator*. Observăm că *InputView* şi *FormView* extind *BlockView* care este la rândul ei o specializare a *AbstractView*. În acest sens, subclasele enumerate reprezintă obiecte de tip *Decorator* care modifică comportamentul orginial al unei metode din *AbstractView* prin suprascriere. Un exemplu sugestiv este adăugarea titlului în cazul clasei *BlockView*.

### *Variaţii ale şablonului MVC*

jqCls se doreşte a fi un framework flexibil şi agil care răspunde la nevoile specifice ale unei aplicaţii. Astfel, funcţionalitatea jqCls nu este condiţionată de utilizarea strictă a şablonului MVC. Acest fapt este confirmat de uşurina cu care au fost integrate API-uri şi frameworkuri adiţionale precum Knockout, Backbone şi combinaţia celor două, Knockback.

După cum rezultă din diagrama de clase din figura 6.13, jqCls integrează prin clasa învelitoare BackboneModel reprezentarea abstractă din frameworkul Backbone. Posibilitatea de interschimbare a modelelor clasice jqCls *Model* cu modele Backbone şi compatibilitatea vederilor de tip *AbstractView* cu oricare dintre soluţiile de reprezentare a datelor reprezintă unul dintre factorii de flexibilitate ai frameworkului.

Mai mult, prin integrarea cu Knockback este posibilă utilizarea variaţiei MVC Model-View-ViewModel care simplifică arhitectura unui sistem. Având în vedere faptul că funcţionalitatea MVVM este implementată de librăriile externe şi integrate cu frameworkul jqCls ne vom limita la o descriere sumară a caracteristicilor acestuia conform documentaţiei MSDN[1].

Blocurile fundamentale ale MVVM sunt *View*, *ViewModel* şi *Model*. Vederile transmit către componenta *ViewModel* evenimentele generate de acţiunile utilizatorului în interfaţa grafică. Clasele *ViewModel* intermediază scrierea şi citirea datelor între modelul de date al aplicaţiei şi vederi. De asemenea, în responsabilitatea claselor *ViewModel* este şi sarcina de a determina dacă o acţiune efectuată în interfaţă are efect asupra modelului şi trebuie propagată sau nu. Modelele MVVM seamănă cu cele MVC, însă sunt decuplate total de vederi. jqCls integrează şablonul Model-View-ViewModel utilizând vederile proprii combinate cu modele Backbone şi clase ViewModel Knockback după cum rezultă şi din figura 6.12.

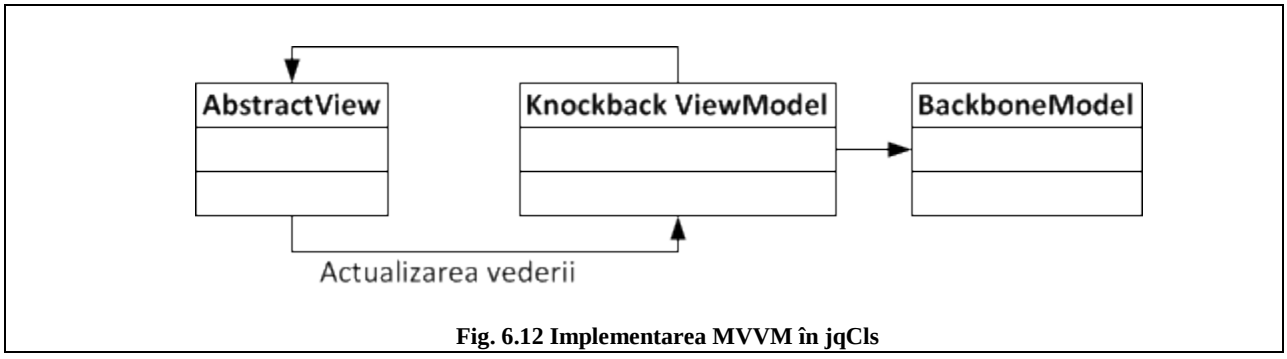



**Fig. 6.12 Implementarea MVVM în jqCls**

[1] http://msdn.microsoft.com/en-us/library/ff798384.aspx

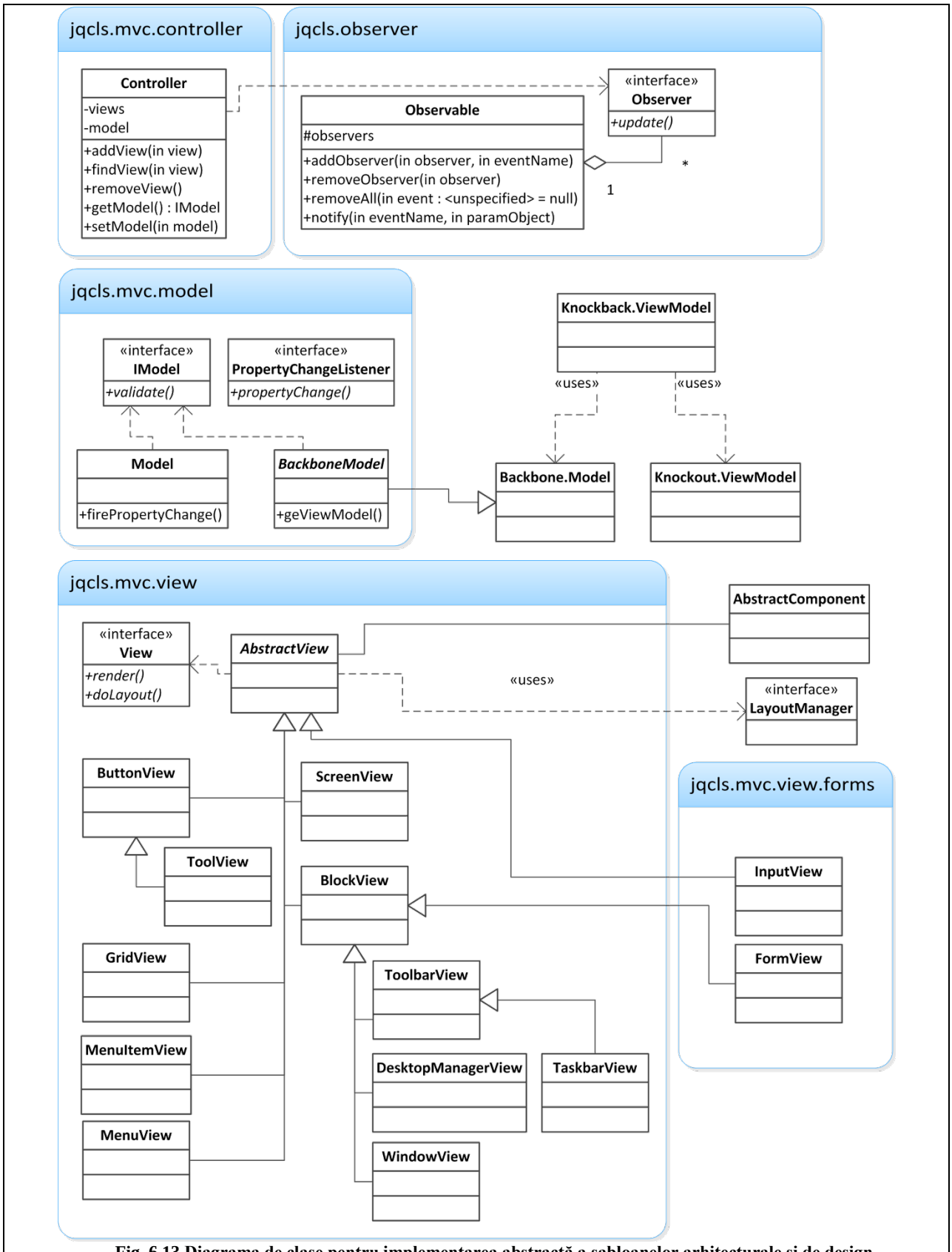
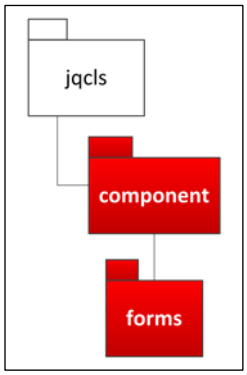



**Fig. 6.13 Diagrama de clase pentru implementarea abstractă a şabloanelor arhitecturale şi de design**

### 6.3.4. Componente




Nivelul superior al jqCls este cel al componentelor implementate pentru cazul general. Rolul acestui nivel în stiva jqCls este, pe de o parte, de a ascunde şi îmbina implementarea complexă din nivele inferioare. Pe de altă parte, o componentă jqCls face legătura între piesele utilizate în dezvoltarea fie cu MVC, fie cu MVVC, adică între vederi, *Controller* şi *Model* sau între *Model*, *ViewModel* şi *View*.

Notăm că, aşa cum am arătat în diagrama detaliată de arhitectură, nivelul superior al jqCls este format din pachetul *component* şi subpachetul forms. Acestea conţin unităţile funcţionale finale proprii şi integrate ale jqCls. În plan extern jqCls, se regăsesc librăriile jQueryUI şi plug-in-ul jqGrid care sunt integrate în framework prin clase învelitoare.

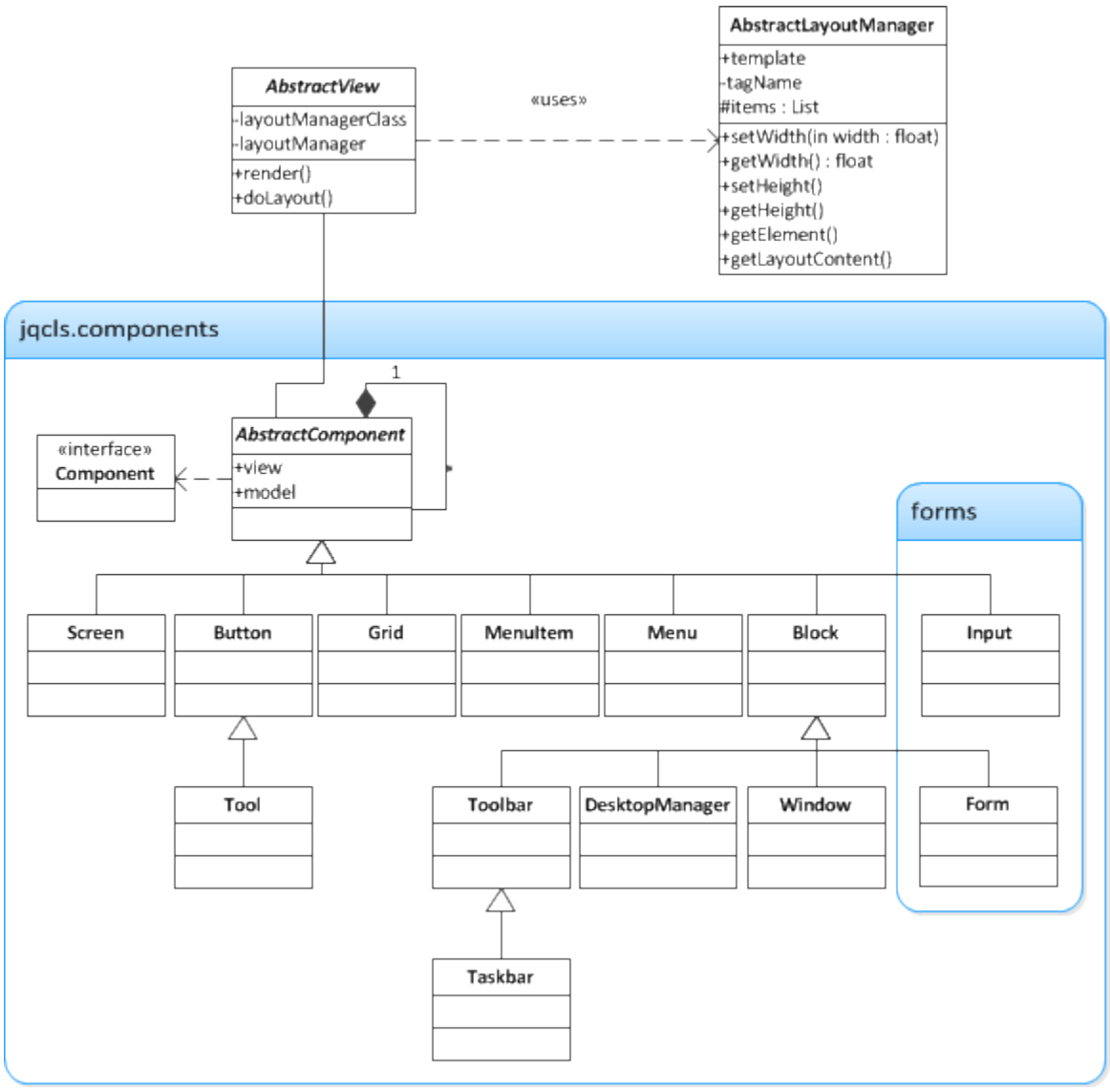



**Fig. 6.14 Diagrama de clase la nivelul componentelor jqCls**

În figura 6.14 ilustrăm diagrama de clase detaliată pentru acest nivel. Observăm că între vederile jqCls şi componente există o relaţie de unu la unu. La baza pachetului *component*, similar cu cazul *mvc.view* stau interfaţa *Component* şi clasele *AbstractComponent* şi *Block*. Acestea specifică şi implementează comportamentul comun tuturor unităţilor funcţionale ale frameworkului. Astfel, *Screen, Grid, Menu, MenuItem* şi *Input* extind direct *AbstractComponent*. *Tool* este o subclasă a lui *Button*. *Block* este specializat în *Toolbar, DesktopManager, Window* şi *Form*. *Taskbar* este o formă a componentei *Toolbar*.

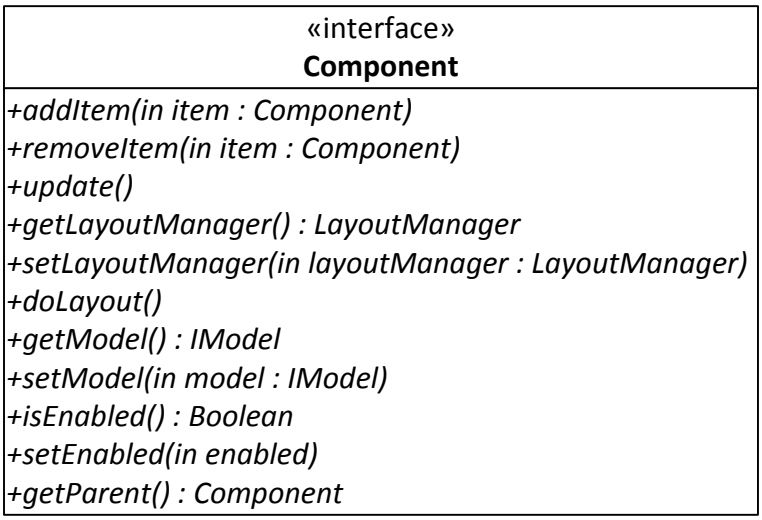



**Fig. 6.15 Interfaţa Component**

Diagrama interfeţei *Component* din figura 6.15 prezintă o vedere detaliată a capabilităţilor comune unei componente jqCls. Având în vedere că metodele implică şi funcţionalitate specifică vederilor, acestea sunt delegate. La nivelul *Component* sunt tratate doar aspectele care nu ţin de reprezentarea interfeţei grafice cum ar fi evenimente ale componentei sau logică internă.

Un detaliu general valabil pentru vederile componentelor jqCls este sarcina de instanţiere a administratorului de amplasare în pagină *LayoutManager*. Suntem de părere că exemplul este unul clasic pentru utilizarea modelului *Factory Method*. Un caz asemănător în jqCls este instanţierea suportului de date pentru obiecte intermediare de comunicare *DataProxy*.

Considerăm că denumirea componentelor este sugestivă. Prin urmare, vom prezenta succint elementele vizuale generale implementate în jqCls. După cum am precizat, nucelul componentelor jqCls se constituie în jurul *AbstractComponent* şi *Block*. Acestea sunt componentele generice ale frameworkului. Astfel, *AbstractComponent* abstractizează suma componentelor jqCls, în timp ce *Block* este o implementare generică a unei componente cu titlu şi conţinut.

Componentele realizate pe baza *AbstractComponent* şi *Block* reprezintă unităţi funcţionale concrete. Notăm că rolul clasei *Screen* este similar nodului *body* în HTML. Un obiect de tip *Screen* reprezintă nodul părinte al arborelui de componente dintr-o aplicaţie jqCls. *Button* învăluie funcţionalitatea *widget*-ului jQueryUI cu acelaşi nume pentru utilizarea în mod obiectual. Specializarea *Tool* a clasei *Button* reprezintă un buton ataşabil unui *Toolbar*. Diferenţele de comportament sunt minore.

Similar clasei *Button*, *Grid* integrează funcţionalitatea pluginului *jqGrid* cu sistemul de componente jqCls. Observăm că unul dintre avantajele utilizării frameworkului propus este reducerea operaţiunilor de configurare. În cazul grid, elementul HTML de tip *table* de la care pleacă jqGrid este generat automat. De asemenea componenta grid integrează o parte din configurările jqGrid care nu sunt implicite, dar pe care le considerăm consumatoare de timp cum ar fi specificarea modului de deserializare a datelor din format JSON. Menţionăm că prin învăluirea în *Grid* pluginul jqGrid poate fi utilizat atât cu implementarea implicită de cereri AJAX, cât şi în combinaţie cu listele şi intermediarii de date jqCls.

Pentru realizarea meniurilor în jqCls sunt implementate clasele *Menu* şi *MenuItem*. Un aspect interesant al acestora este posibilitatea de creare a meniurilor ierarhice specifice aplicaţiilor *desktop*.

Clasele *DesktopManager*, *Window* şi administratorul de aranjare în pagină *DesktopLayout* sunt fundaţia interfeţei multidocument jqCls. Având în vedere că jqCls este gândit pentru sisteme cu interfaţă multidocument considerăm importantă detalierea acestui aspect.

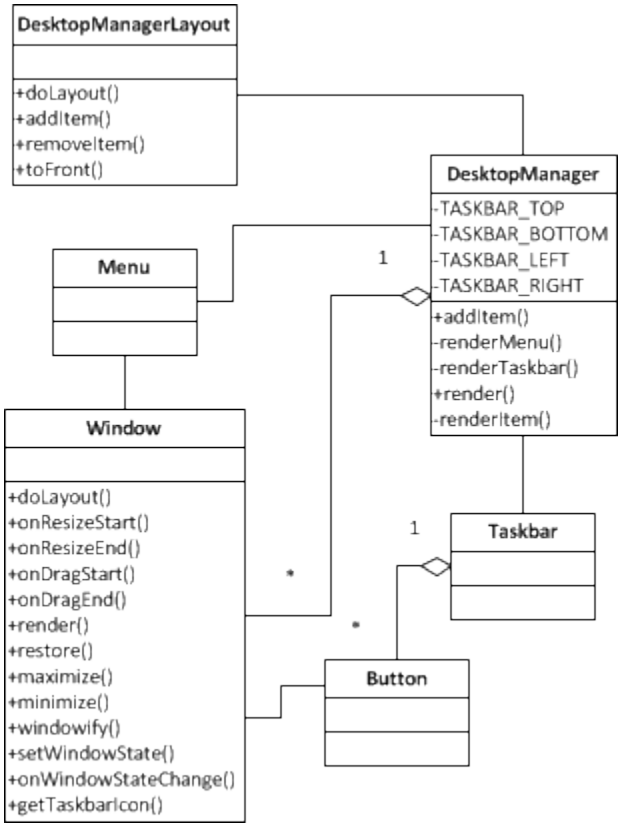



**Fig. 6.16 Sistemul multidocument jqCls**

Diagrama de clase care stă la baza sistemului MDI jqCls este prezentată în figura 6.16. Remarcăm că sistemul de ferestre jqCls are capabilităţile majorităţii soluţiilor similare *desktop*. În acest sens ferestrele jqCls pot fi fixe sau mobile din punctul de vedere al poziţiei şi al dimensiunii, asigură funcţionalitate de închidere, minimizare, maximizare şi restaurare şi pot să conţină meniuri. Implementarea *Window* porneşte de la *widget*-urile jQueryUI *draggable* şi *resizable.*

*DesktopManager* şi *DesktopManagerLayout* asigură operaţiile de afişare, dispunere a ferestrelor. De asemenea, clasele menţionate sunt responsabile pentru administrarea ierarhiei de afişare a ferestrelor pe care le conţin.

O altă funcţionalitate a sistemului multidocument jqCls este afişarea listei de ferestre deschise în componente de tip *Taskbar*. În acest sens, responsabilitatea creării butonului de afişat în bara de activităţi revine ferestrei.

Formularele sunt o componentă fundamentală pentru aplicaţiile interactive. jqCls acoperă acest domeniu prin clasele *Form* şi *Input* din pachetul *forms*. Pornind de la *Input*, în framework se dezvoltă o serie de specializări complexe cu interactivitate crescută menite să îmbunătăţească experienţa utilizatorilor finali cum ar fi *AutoComplete* pentru completarea intuitivă asistată, *UploadField, DynamicInput* pentru cazul în care utilizatorul poate introduce un număr nedefinit de valori, *PasswordField* sau *CaptchaInput* pentru securizarea formularelor publice împotriva completării automatizate.

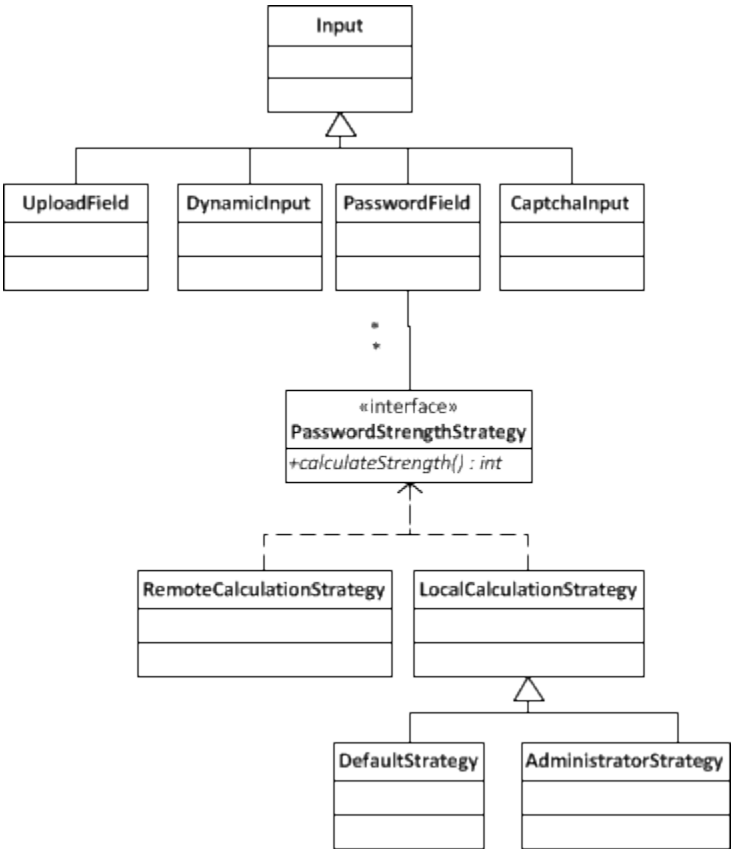



**Fig. 6.17 Specializări ale clasei Input**

În cazul câmpurilor de tip *AutoComplete* notăm că abordarea obiectuală rezolvă problema inserării câmpurilor adiţionale ascunse în arborele HTML în cazul valorilor identificate unic în baza de date. Un exemplu în acest sens ar putea fi un câmp pentru selecţia ţării de reşedinţă.

Apariţia AJAX a dus la schimbarea radicală a protocolului Web gândit iniţial pentru schimbul documentelor statice. În acest sens, pentru încărcarea unui fişier este nevoie de simularea trimiterii tradiţionale a unui formular de tip *multipart/form-data* pentru că standardul

actual JavaScript nu permite accesul direct la sistemul de fişiere. *UploadField* rezolvă această problemă în mod generic.

Funcţionalitatea adiţională pe care o oferă componentele *PasswordField* este de calcul al rezistenţei unei parole. Considerăm că acesta este un exemplu clasic de aplicare a modelului *Strategy*.

Procesarea, validarea datelor introduse de utilizator şi notificarea cu privire la rezultatul operaţiei este o problemă general valabilă în aplicaţiile interactive. Prin componenta *Form* jqCls implementează un mecanism aplicabil în mod general şi detaliat în diagrama de secvenţă din figura 6.18.

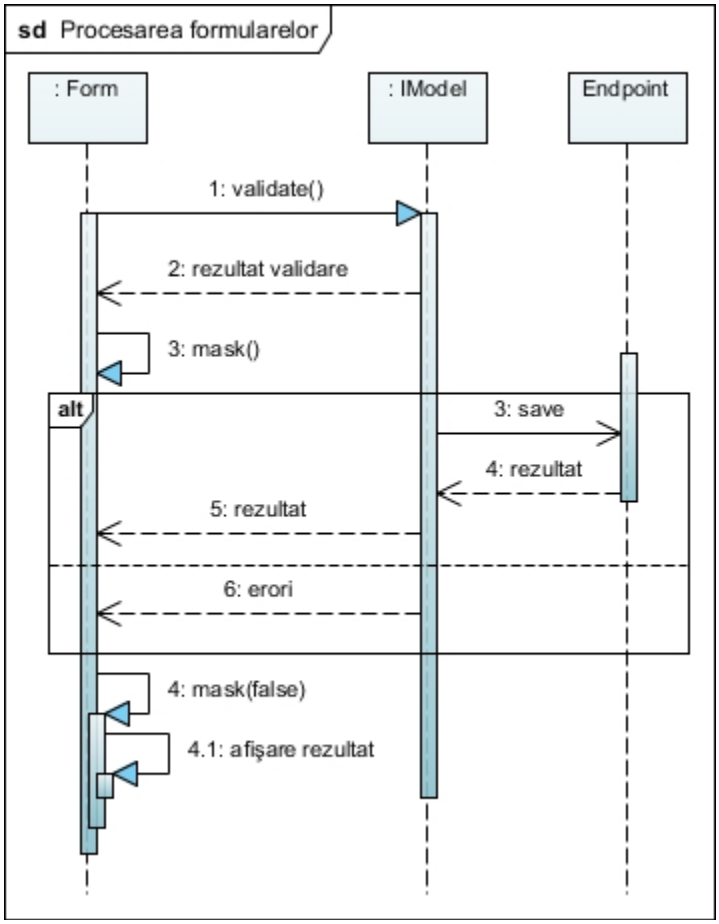



**Fig. 6.18 Diagramă de secvenţă pentru procesarea formularelor**

Însumând, considerăm că setul de componente general valabile pus la dispoziţie în frameworkul jqCls este un punct de plecare bun pentru dezvoltarea oricărei aplicaţii client Web.

## 6.4. Instrumente utilizate

Odată cu înmulţirea sarcinilor administrative cauzată de creşterea dimensiunilor proiectului exprimată în module, dependenţe, fişiere şi linii de cod, am considerat oportună automatizarea procesului de build prin instalarea şi configurarea unui server Jenkins.

Conform Wikipedia[1], Jenkins este un sistem de integrare continuă pentru dezvoltarea produselor software bazat pe un server rulat într-un container servlet şi poate executa proiecte bazate pe Apache Ant, Maven precum şi scripturi shell sau comenzi Windows.

În relaţie cu jqCls, serverul Jenkins este responsabil de rularea comenzilor necesare procesului de *build* în linia de comandă după cum urmează :

1. Descărcarea versiunii curente a codului sursă din sistemul de control al versionării SVN
2. Rularea modulului r.js în mediul Node.js pentru minifierea, concatenarea şi optimizarea scripturilor JavaScript :

```
node $WORKSPACE/checkout/trunk/js/r.js -o
$WORKSPACE/checkout/trunk/app.build.js out=$WORKSPACE/dist/$JOB_NAME-
$JQCLS_VERSION.$BUILD_NUMBER.$EXECUTOR_NUMBER/$JOB_NAME-
$JQCLS_VERSION.$BUILD_NUMBER.$EXECUTOR_NUMBER.js
```

3. Concatenarea surselor CSS prin modulul r.js

```
node $WORKSPACE/checkout/trunk/js/r.js -o
cssIn=$WORKSPACE/checkout/trunk/css/main.css
out=$WORKSPACE/dist/$JOB_NAME-
$JQCLS_VERSION.$BUILD_NUMBER.$EXECUTOR_NUMBER/$JOB_NAME-
$JQCLS_VERSION.$BUILD_NUMBER.$EXECUTOR_NUMBER.css
```

4. Rularea analizatorului yuidoc în vederea generării documentaţiei API

```
cd $WORKSPACE/checkout/trunk
yuidoc --project-version $JQCLS_VERSION.$BUILD_NUMBER.$EXECUTOR_NUMBER .
cp -a $WORKSPACE/docs/. /home/stefan/www/jqclsdocs/
```

5. Împachetarea fişierelor generate în formatul de distribuţie jqCls

```
rm -rf /home/stefan/www/jqcls
cp -a $WORKSPACE/dist/$JOB_NAME-
$JQCLS_VERSION.$BUILD_NUMBER.$EXECUTOR_NUMBER/. /home/stefan/www/jqcls
cp -a $WORKSPACE/checkout/trunk/css/themes/. /home/stefan/www/jqcls/themes
```

6. actualizarea distribuţiei curente *jqcls-current*

```
mv /home/stefan/www/jqcls/$JOB_NAME-
$JQCLS_VERSION.$BUILD_NUMBER.$EXECUTOR_NUMBER.js
/home/stefan/www/jqcls/jqcls-current.js
mv /home/stefan/www/jqcls/$JOB_NAME-
$JQCLS_VERSION.$BUILD_NUMBER.$EXECUTOR_NUMBER.css
/home/stefan/www/jqcls/jqcls-current.css
```

Considerăm că documentaţia API este un factor cheie în utilizabilitatea unui astfel de produs. În acest sens, pentru generarea documentaţiei API, jqCls se bazează pe instrumentul YUIDoc. Acesta face parte din suita YUILibrary şi este un instrument scris în JavaScript care poate rula pe platformă Rhino sau Node.js de generare a documentaţiei API. Instrumentul se bazează pe comentarii şi este compatibil cu o varietate de stiluri de programare. Fragmentul de mai jos ilustrează anotarea claselor jqCls cu etichetele pe care YUIDoc le înţelege.

```
/**
 * @namespace ns.mvc.view
 * @class DesktopManagerView
 * @submodule ns.mvc.view
 * @constructor
 *  @param [config] {Object}
 * @extends ns.mvc.view.BlockView
 **/
```

[1] http://en.wikipedia.org/wiki/Jenkins_(software)

După cum am precizat, codul jqCls este minifiat utilizând modulul r.js. Conform descrierii proiectului pe GitHub acesta este un instrument în linia de comandă de rulare a scripturilor JavaScript care folosesc API-ul AMD pentru declararea şi utilizarea modulelelor şi face parte din proiectul RequireJS. Pentru jqCls, configurarea acestui instrument este simplă şi se află în fişierul app.build.js al cărui conţinut îl prezentăm în continuare.

```
({
    name: 'Application',
    baseUrl: './js',
    "packages": [
        "components",
        "components/forms",
        "core",
        "dom",
        "lib",
        "mvc",
        "observer",
        "util"
    ],
    out: "../../build/jqcls-build.js",
    optimize: 'none'
});
```

Din configurările prezentate reţinem că pentru utilizarea instrumentului r.js este importantă specificarea pachetelor AMD de optimizat.

Suntem de părere că instrumentaţia prezentată în această secţiune aduce un argument important în ceea ce priveşte anvergura proiectului propus.

## 6.5. Detalii de implementare ale aplicaţiei demonstrative *Airline Manager*

Utilitatea frameworkului jqCls este demonstrată prin utilizarea acestuia în implementarea aplicaţiei Web client a unui joc cu tema simulării administrării companiilor aeriene. În această secţiune vom aborda sumar modul de implementare al acestei aplicaţii atât în ceea ce priveşte partea server, cât şi în legătură cu funcţionalitatea jqCls exploatată prin clientul Web.

### 6.5.1. Arhitectura conceptuală

La nivel conceptual, *Airline Manager* se încadrează în categoria aplicaţiilor de tip client-server în care gradul de decuplare a interfeţei utilizatorului de sistemul de procesare a datelor la distanţă este maxim. În cazul *Airline Manager* acest fapt este determinat atât de utilizarea frameworkului jqCls şi de capacitatea acestuia de a comunica cu aplicaţia server strict pentru transmiterea datelor de business, cât şi de implementarea serverului ca furnizor de servicii.

### 6.5.2. Detalii arhitecturale

În figura 6.19 este reprezentată diagrama de arhitectură conceptuală a aplicaţiei *Airline Manager*. O observaţie imediată pe baza arhitecturii este independenţa aplicaţiei client de implementarea server. De asemenea este de notat corespondenţa nivelelor arhitecturale. Astfel, nivelul vederilor aplicaţiei client corespunde serviciilor de date furnizate de server. Corelând diagrama de arhitectură cu cea de *deployment* înţelegem că nivelul serviciilor de date al aplicaţiei server reprezintă interfaţa de comunicare cu aplicaţia client.

Un avantaj evident al acestui model de dezvoltare este posibilitatea înlocuirii oricăreia dintre componente fie client, fie server fără ca sesizarea acestei schimbări de cealaltă parte. În acest

sens, considerăm că abordarea aderă principiului de programare bazat pe interfaţă şi nu pe implementare.

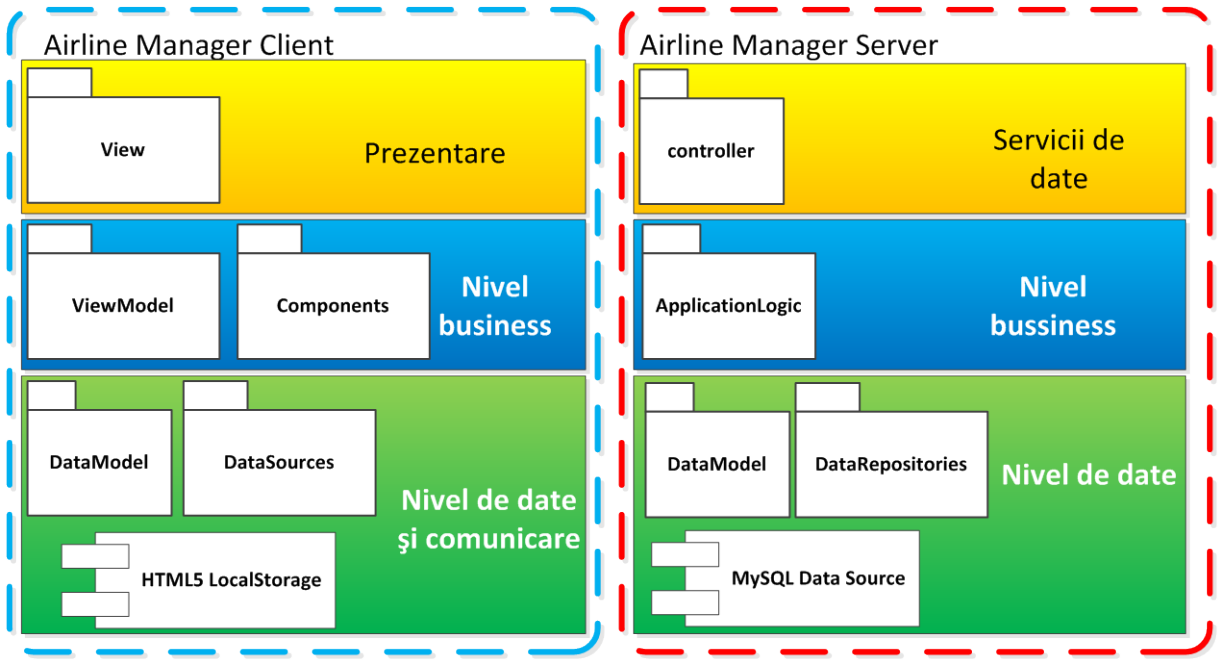



**Fig. 6.19 Diagrama de arhitectură conceptuală a Airline Manager**

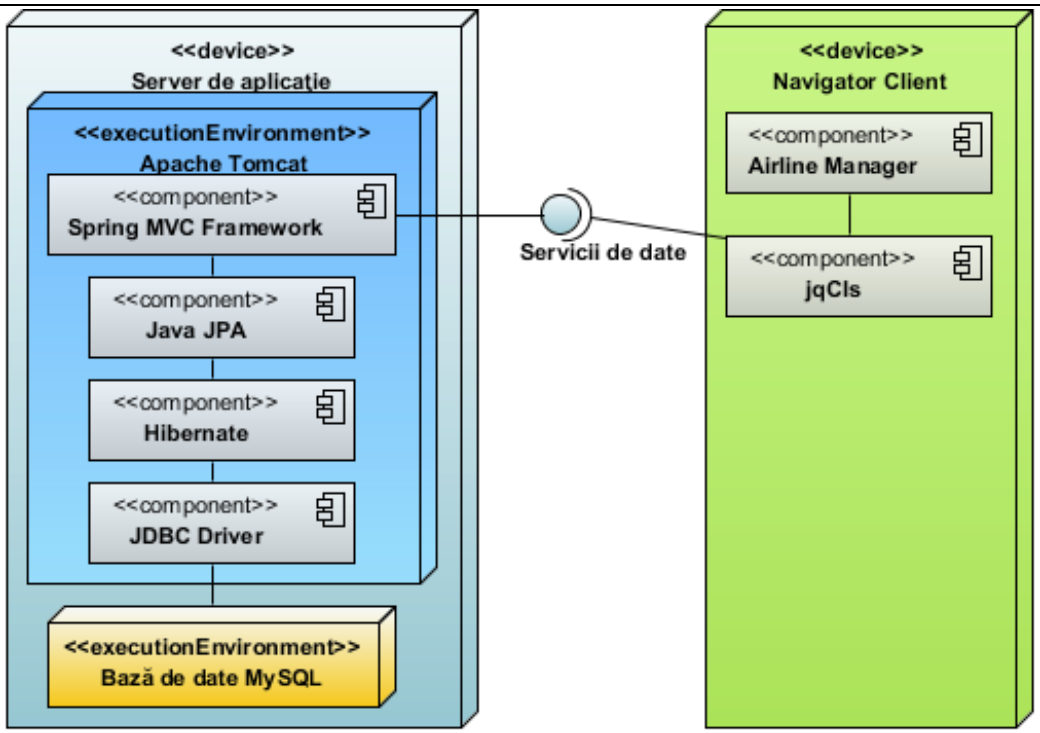



**Fig. 6.20 Diagrama de *deployment* a aplicaţiei Airline Manager**

Conform diagramei de *deployment* remarcăm că aplicaţia server este construită pe scheletul frameworkului *Spring MVC* 3 şi rulează pe un server de aplicaţie Apache Tomcat 7. Reprezentarea datelor la nivelul server este tratată utilizând specificaţia Java JPA 2.0 cu Hibernate ca furnizor de persistenţă. Sistemul de gestiune al bazelor de date ales este de tip MySQL cu motorul tranzacţional de stocare a datelor InnoDB.

### 6.5.3. Componente şi servicii

În figura 6.21 prezentăm în ansamblu componentele sistemului *Airline Manager*. Astfel, în ansamblul ei, aplicaţia client este considerată o componentă. Comunicarea între client şi server este realizată prin interfaţa generică a serviciilor de date expuse prin controllerele Spring MVC. Acestea sunt responsabile pentru comunicarea cu componentele de tip *JPARepository* şi cu nivelul de securitate. Componenta *JPARepositories* reprezintă o generalizare a unităţilor de acces la sistemul de gestiune al datelor.

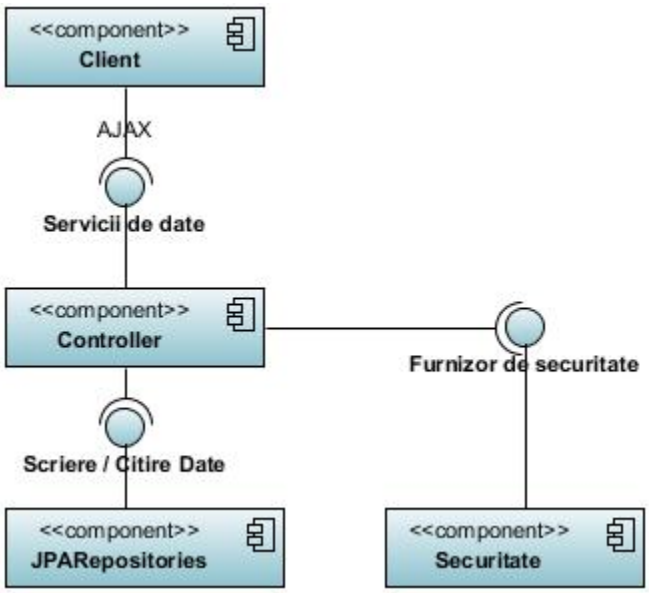



**Fig. 6.21 Diagrama de componente a sistemului Airline Manager**

Diagramele din figurile 6.22 şi 6.23 prezintă implementarea sistemului *Airline Manager* la nivel de componente concrete. Notăm că diagramele menţionate sunt valabile atât pentru aplicaţia server. Diferenţa între implementări constă în utilizarea Spring MVC ca bază de funcţionalitate pentru aplicaţia server şi jqCls cu acelaşi rol în ceea ce priveşte clientul Web. În acest sens, dublarea nivelului de *business* poate fi considerată un dezavantaj al arhitecturii pentru care este gândit jqCls. Acest neajuns poate fi depăşit prin implementarea unei metode de detectare şi generare automată a obiectelor aparţinând modelului de *business* al aplicaţiilor client jqCls.

După cum am precizat, diferenţa majoră între server şi client ţine de tehnologiile utilizate. În acest sens, observăm că nivelul de persistenţă în cazul aplicaţiei server este implementat utilizând Hibernate, Java JPA şi Spring *JPARepository*. Pe de altă parte, în cazul aplicaţiei client persistenţa datelor este asigurată local prin jqCls *LocalStorageDataProxy* şi obiectul JavaScript

*LocalStorage* iar comunicarea la distanţă cu serverul de aplicaţie se realizează prin obiecte de tip *AjaxDataProxy*.

Un aspect important de reţinut în ceea ce priveşte realizarea componentelor sistemului *Airline Manager* este maparea acestora în relaţie unu la unu pe cazurile de utilizare şi cerinţele funcţionale prezentate în capitolul precedent.

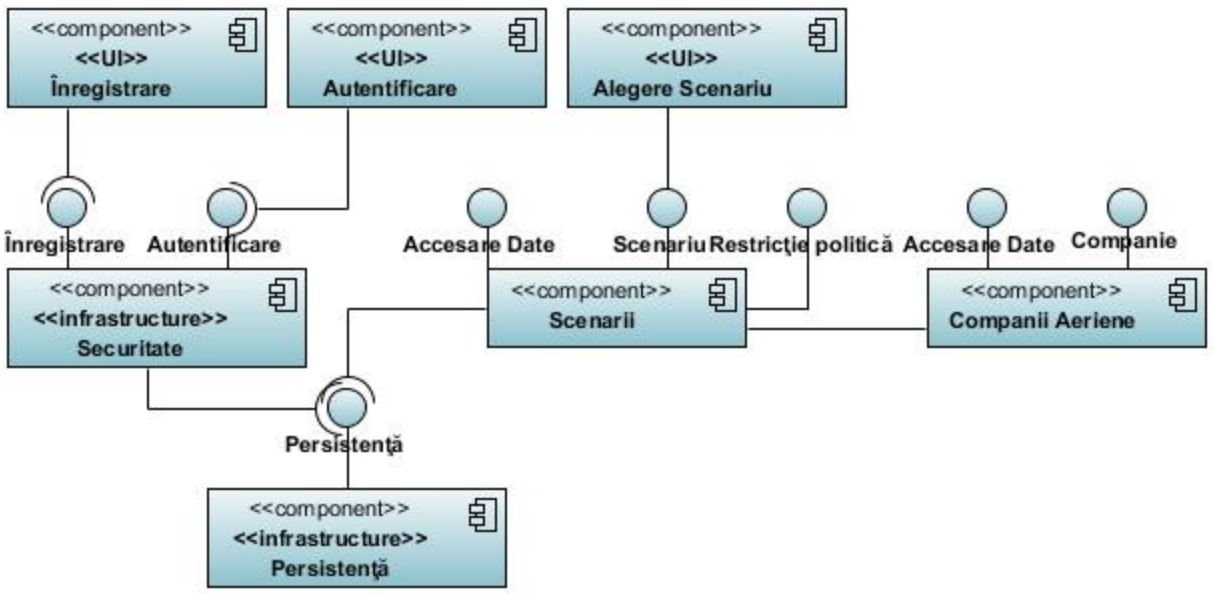



**Fig. 6.22 Diagrama de componente - Înregistrare, Autentificare, Alegere Scenariu**

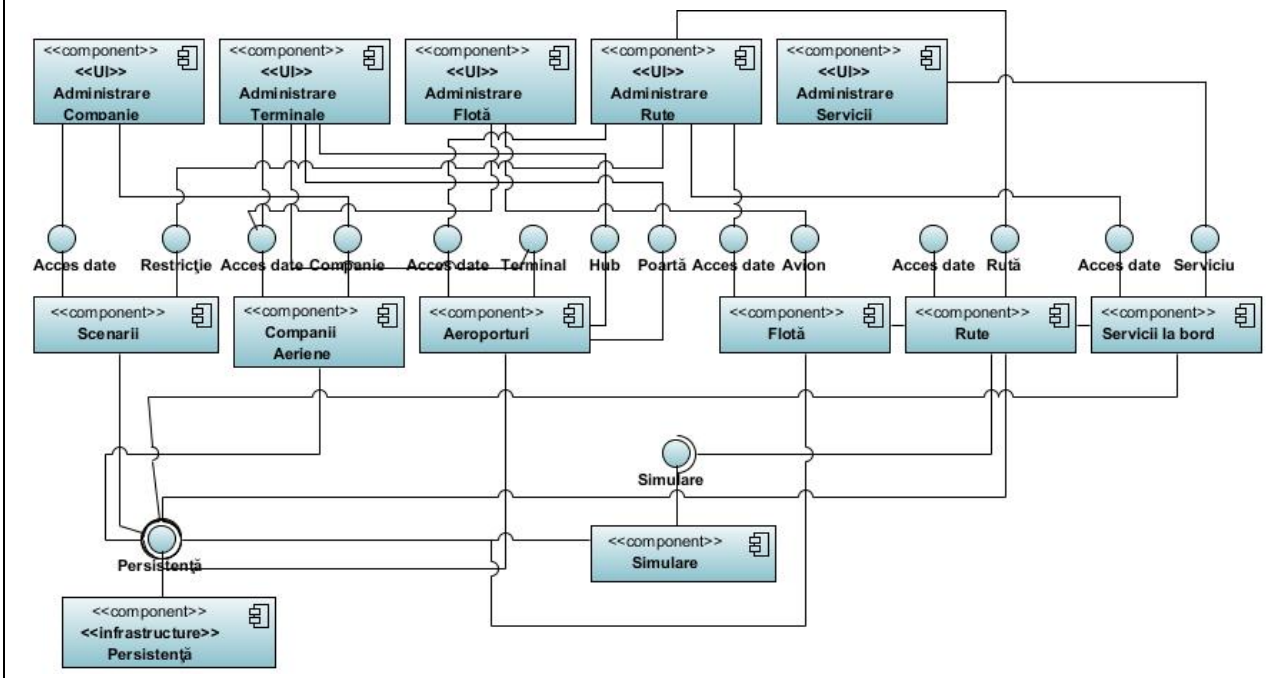



**Fig. 6.23 Diagrama de componente detaliată a simulării**

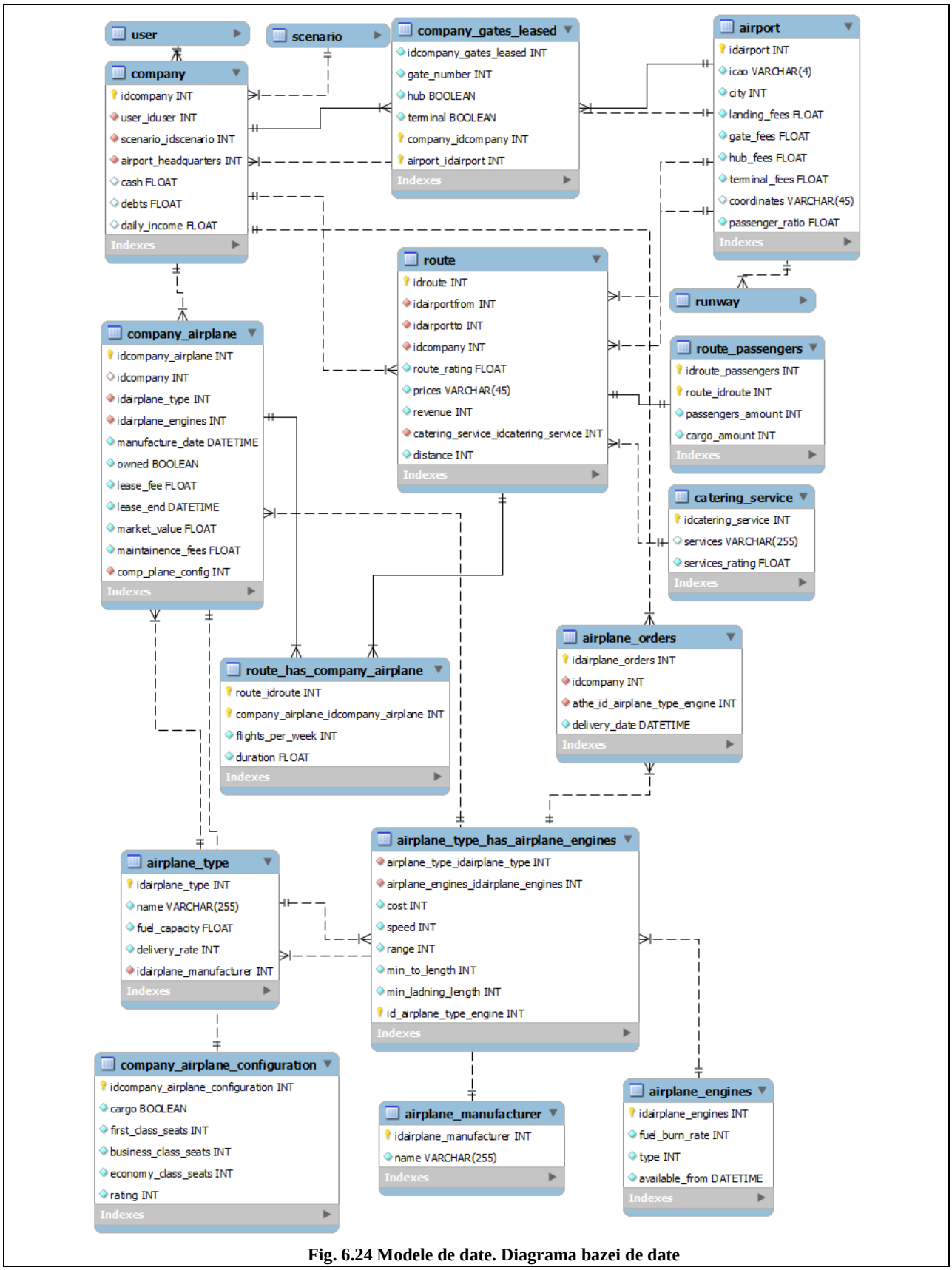



**Fig. 6.24 Modele de date. Diagrama bazei de date**

În figura 6.24 propunem diagrama bazei de date a sistemului *Airline Manager*. Tabelele au fost mapate în aplicaţia server utilizând instrumentele ORM puse la dispoziţie de frameworkul Hibernate. Pentru aplicaţia client au fost utilizate modele *BackboneModel* corespunzătoare formatului de date trimis de către server şi serializate prin frameworkul Google GSON.

Având în vedere că rolul aplicaţiei *Airline Manager* este de a valida funcţionalitatea frameworkului jqCls considerăm că informaţiile cu privire la implementare oferite sunt suficiente şi în măsură să formeze o imagine de ansamblu.

# 7. Testare şi validare

Testarea este un capitol deosebit de important în viaţa oricărui produs software. Precum conceptele de programare obiectuală, testarea aprofundată a aplicaţiilor JavaScript este adesea ignorată. După cum am precizat, calitatea reprezintă un principiu de bază de care am ţinut cont în dezvoltarea jqCls. În acest sens, funcţionalitatea jqCls a fost testată manual atent în timpul procesului de dezvoltare. De asemenea au fost realizate scenarii de testare automată cu caracter demonstrativ.

## 7.1. Scenarii de testare manuală *black-box*

Conform Wikipedia[1], testarea black-box este o metodă de testare software care examinează funcţionalitatea unei aplicaţii fără a intra în detaliile de implementare internă. Aceasta poate fi aplicată la orice nivel de testare.

Pentru realizarea scenariilor de testare manuală a fost urmată tehnica tabelelor decizionale având în vedere că acestea sunt o formă precisă şi compactă de reprezentare a logicii complexe. Un tabel decizional este format conform Wikipedia[2] din patru cadrane : condiţii, alternative, acţiuni şi intrări de acţiuni. Fiecare decizie corespunde unei variabile ale cărei valori sunt enumerate în alternative. O intrare precizează dacă acţiunea trebuie efectuată pentru setul de alternative căreia îi corespunde. Modelul unei astfel de specificaţii este prezentat în tabelul 7.1.

| Condiţii | Alternative |
|---|---|
| Acţiuni | Intrări |

**Tabelul 7.1 Modelul unui tabel decizional**

În continuare vom prezenta cu caracter demonstrativ două tabele decizionale cu nivele diferite de complexitate utilizate în testarea funcţionalităţii frameworkului jqCls. Menţionăm că secvenţele de cod necesare pregătirii condiţiilor de testare pot fi consultate în anexele lucrării.

Considerăm că testarea funcţionalităţii unei componente de tip *Toolbar* cu butoane reprezintă un caz simplu. Comportamentul vizat este aranjarea corectă pe verticală sau orizontală a conţinutului şi execuţia funcţiei *callback* de tratare a evenimentului de apăsare (*click*) al butonului. Secvenţa de cod utilizată se regăseşte în anexa 4.1. Scenariul de testare este specificat în tabelul 7.2.

Testarea meniurilor reprezintă un scenariu de complexitate medie. Comportamentul aşteptat este răspunsul la plasarea cursorului peste un element, afişarea ierarhiei şi aranjarea conform direcţiei specificate pe orizontală sau verticală. Secvenţa de cod prin care este configurat mediul de testare corespunzător poate fi consultată în anexa 4.2. Scenariul de testare pentru componentele de tip meniu este reprezentat în tabelul 7.3. Observăm că, în acest exemplu indiferenţa dată de exemplu de imposibilitatea situaţiei este marcată cu *A*.

[1] http://en.wikipedia.org/wiki/Black-box_testing
[2] http://en.wikipedia.org/wiki/Decision_table

| | | Reguli | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Condiţii | Dimensiuni fixe | Nu | Nu | Nu | Nu | Da | Da | Da | Da |
| | Afişare pe orizontală | Nu | Nu | Da | Da | Nu | Nu | Da | Da |
| | Butonul 2 a fost apăsat | Nu | Da | Nu | Da | Nu | Da | Nu | Da |
| Acţiuni / Rezultate | Toolbarul umple ecranul pe orizontală | Nu | Nu | Da | Da | Nu | Nu | Nu | Nu |
| | Toolbarul umple ecranul pe verticală | Da | Da | Nu | Nu | Nu | Nu | Nu | Nu |
| | Butoanele sunt afişate în ordine pe orizontală | Nu | Nu | Da | Da | Nu | Nu | Da | Da |
| | Butoanele sunt afişate în ordine pe verticală | Da | Da | Nu | Nu | Da | Da | Nu | Da |
| | Se tratează evenimentul de click pentru butonul 2 | Nu | Da | Nu | Da | Nu | Da | Nu | Da |

**Tabelul 7.2 Tabelul decizional pentru testarea componentei Toolbar**

| | | Reguli | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Condiţii | Submeniu ierarhic | Nu | Nu | Nu | Nu | Nu | Nu | Nu | Nu | Da | Da | Da | Da | Da | Da | Da | Da |
| | Cursorul este poziţionat peste meniu | Nu | Nu | Nu | Nu | Da | Da | Da | Da | Nu | Nu | Nu | Nu | Da | Da | Da | Da |
| | Cursorul iese din zona elementului de meniu | Nu | Nu | Da | Da | Nu | Nu | Da | Da | Nu | Nu | Da | Da | Nu | Nu | Da | Da |
| | Elementul de meniu a fost apăsat | Nu | Da | Nu | Da | Nu | Da | Nu | Da | Nu | Da | Nu | Da | Nu | | Nu | Da |
| Acţiuni / Rezultate | Elementul este distins prin CSS | Nu | Da | Nu | A | Da | Da | A | A | Nu | Da | Nu | A | Da | Da | A | A |
| | Afişarea submeniului | A | A | A | A | A | A | A | A | Nu | Da | Nu | A | Da | Da | A | A |
| | Ascunderea submeniului | A | A | A | A | A | A | A | A | Da | Nu | Da | A | Nu | Nu | A | A |
| | Tratarea evenimentului de apăsare | Nu | Da | Nu | A | Nu | Da | A | A | Nu | Da | Nu | A | Da | Da | A | A |
| | Evidenţierea elementului HTML | Nu | Da | Nu | A | Da | Da | A | A | Nu | Da | Nu | A | Nu | Da | A | A |

**Tabelul 7.3 Tabelul decizional pentru testarea comportamentului meniurilor**

Componentele jqCls au fost testate integral pe parcursul procesului de dezvoltare respectând acest tipar. Pentru exemplificarea procesului de testare considerăm cazurile prezentate în această secţiune ca fiind suficiente pentru demonstrarea rigurozităţii cu care a fost verificată funcţionarea corectă a frameworkului.

## 7.2. Scenarii de testare automată prin Qunit

Testarea automată a frameworkului jqCls a fost realizată cu ajutorul frameworkului QUnit. Conform descrierii proprii, Qunit este un cadru de testare automată a entităţilor funcţionale puternic şi uşor de utilizat. Autoritatea QUnit în testarea unităţilor funcţionale este confirmată de

utilizarea acestuia în validarea frameworkurilor utilizate la scară largă jQuery şi jQueryUI. Mai mult, QUnit este capabil, conform documentaţiei, să testeze orice cod JavaScript, inclusiv să se autoverifice.

În continuare, cu caracter demonstrativ vom prezenta un scenariu de test automat realizat cu QUnit. Considerăm că implementarea pachetelor de date reprezintă un punct deosebit de sensibil în orice framework. În consecinţă, scenariul de test ales are în centrul atenţiei implementarea listelor de tip ArrayList în jqCls. Codul integral pentru suita de testare *ArrayList* poate fi consultat în anexa 10.

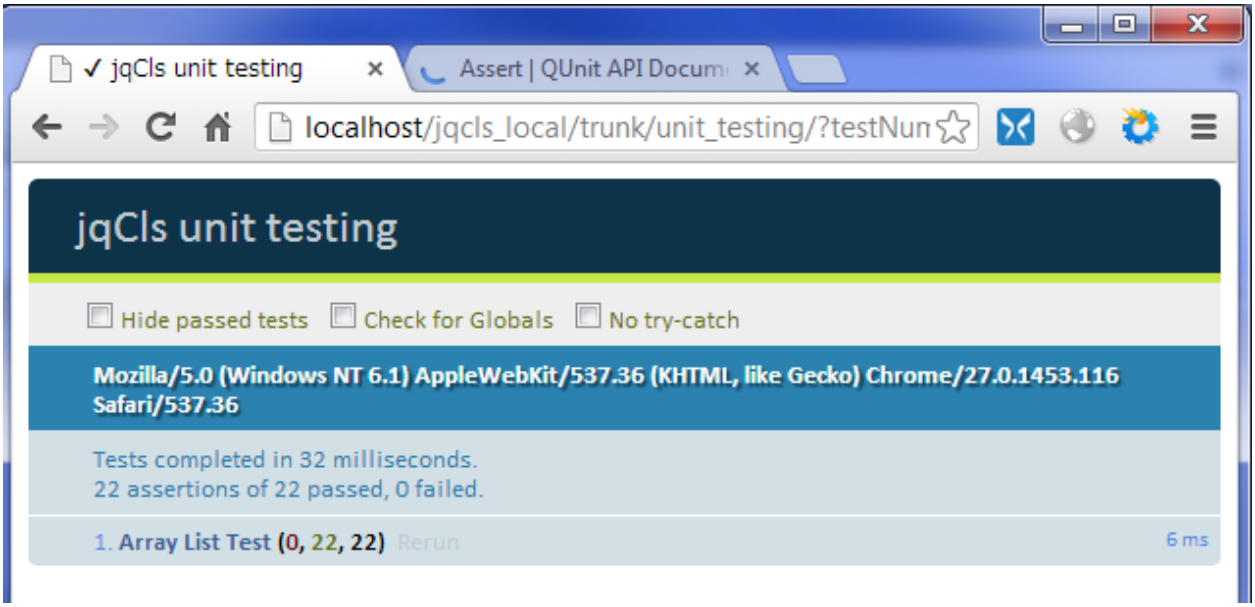



**Fig. 7.1 Rularea suite de test Array List Test**

În crearea suitei de test am avut în vedere abordarea fiecărei metode a clasei *ArrayList* în parte şi a cazurilor speciale cum ar fi ştergerea primului şi ultimului element sau golirea unei liste care nu are elemente. Această afirmaţie este susţinută de suita de test anexată lucrării.

## 7.3. Asigurarea calităţii codului

Calitatea codului este un factor esenţial. În JavaScript şi în general în limbajele slab tipizate, considerăm că această caracteristică joacă un rol mult mai important decât în cele cu tipizare strictă şi este măsura directă a mentenabilităţii codului. În acest sens, abuzată, libertatea de exprimare şi utilizare a limbajului se poate transforma uşor într-o capcană cu efecte dezastroase în ceea ce priveşte mentenabilitatea unui produs software.

Având în vedere cele precizate mai sus, în dezvoltarea jqCls recomandările generale şi specifice de programare au fost urmate cu stricteţe. JSHint este un instrument deosebit de practic în ceea ce priveşte respectarea standardelor de cod JavaScript. Integrat în editorul de cod, JSHint permite configurarea standardelor de cod de urmat la nivelul proiectului şi le urmăreşte în timp real. Instrumentul poate fi configurat prin comentarii JavaScript şi este capabil să verifice următoarele aspecte ale codului :

- asumarea funcţionalităţilor unui mediu JavaScript : BOM, Node.js, Rhino
- prezenţa instrucţiunilor *alert* sau *console.****

- respectarea condiţiilor de asignare
- avertizarea cu privire la operatorii pe biţi
- sintaxa ECMAScript 5
- evitarea funcţiei *eval*
- filtrarea parcurgerilor *for ... in* în cazul obiectelor
- scrierea numelui unei funcţii constructor cu majusculă
- utilizarea i = i + 1 în locul i++ care este mai puţin lizibilă
- parametri neutilizaţi
- utilizarea indicaţiei *use strict*
- respectarea modelului *variable on top* (o singură declaraţie *var* la nivelul superior al unei funcţii)
- utilizarea conformă a caracterelor de spaţiere
- indentarea şi lungimea maximă a unei linii de cod

Menţionăm că sursele jqCls au fost scrise în acord cu regulile JSHint enumerate mai sus.

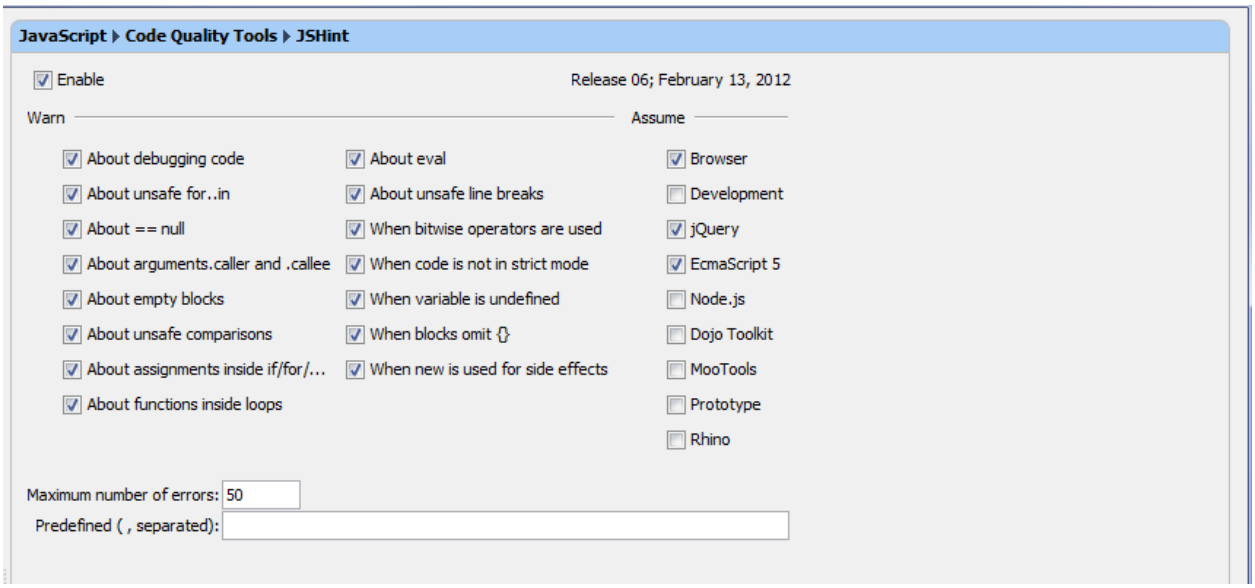



**Fig. 7.2 Configurarea instrumentului JSHint**

# 8. Instalare şi utilizare

În acest capitol propunem o prezentare a resurselor necesare şi a modului de utilizare al frameworkului jqCls.

## 8.1. Resurse necesare

jqCls este un framework JavaScript. Ca atare, resursele necesare pentru utilizarea jqCls sunt tipice produselor software care se încadrează în această categori. Categoria principală de utilizare a frameworkului sunt aplicaţiile care rulează în mediul navigatorului însă aceasta nu este o restricţie. Frameworkul funcţionează atât în mediul navigatorului, cât şi în orice alt motor JavaScript cum ar fi Rhino sau Node.js.

Pentru rularea în condiţii optimite este recomandat un navigator care implementează specificaţia HTML5 şi standardul ECMAScript 5.1 cum ar fi Mozilla Firefox, Goolge Chrome sau Internet Explorer 10+. În absenţa funcţionalităţii HTML5 şi ECMAScript 5.1, jqCls se degradează controlat şi poate fi utilizat conform limitărilor impuse de sistem. Pentru identificarea compatibilităţii cu navigatorul în cazuri specifice recomandăm pagina web *Can I use* (www.caniuse.com) şi comparaţia amplă oferită Wikipedia[1].

Din punctul de vedere al platformei, specific produselor JavaScript, jqCls este independent.

În privinţa resurselor hardware, jqCls nu impune restricţii adiţionale celor recomandate de fabricanţii navigatoarelor majore.

## 8.2. Instalare

Punem la dispoztia utilizatorilor două metode de instalare a frameworkului. Fie prin descărcarea manuală de pe Web a unui pachet de distribuţie jqCls, fie prin utilizarea versiunii curente accesibile la adresa http://jqcls.sieraindiagolf.com/jqcls-current

În ambele variante menţionate este necesară includerea resurselor jqCls JavaScript şi CSS în nodul *head* al documentului prin adăugarea instrucţiunilor generice exemplificate mai jos.

```
<head>
...
<script type="application/javascript"
src="<CALEA_SPRE_DIRECTORUL_JQCLS>/jqcls-<VERSIUNE>.js"></script>
<link type="text/css" rel="stylesheet" media="screen"
href="<CALEA_SPRE_DIRECTORUL_JQCLS>/jqcls-<VERSIONE>.css"/>
...
</head>
```

Astfel, parametrii *CALEA_SPRE_DIRECTORUL_JQCLS* şi *VERSIUNE* reprezintă locaţia fizică a resurselor jqCls şi identificatorul de versiune. Exemplificăm în cazul utilizării versiunii curente :

```
<head>
<script type="application/javascript"
src="http://jqcls.sieraindiagolf.com/jqcls-current.js"></script>
<link type="text/css" rel="stylesheet" media="screen"
href="http://jqcls.sieraindiagolf.com/jqcls-current.css"/>
</head>
```

[1] http://en.wikipedia.org/wiki/Comparison_of_web_browsers

Formatul standard de împachetare a unei distribuţii a frameworkului jqCls este prezentat în figura 8.1. Notăm că, resursele pentru utilizarea frameworkului se află în directorul *dist*, documentaţia API poate fi consultată din *docs*, iar codul sursă este livrat în *sources*.

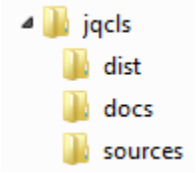



**Fig. 8.1 Structura de directoare în distribuţiile jqCls**

## 8.3. Utilizare

Potrivit descrierii implementării interne, entităţile funcţionale jqCls sunt grupate în module ierarhice. Figura 8.2 prezintă diagrama de module. Conţinutul acestora a fost descris amplu în capitolul destinat detaliilor de implementare. Pornind de la diagrama de pachete şi conceptele fundamentale de programare oritentată pe obiecte implementate în framework vom prezenta în continuare modul utilizare al funcţionalităţilor jqCls.

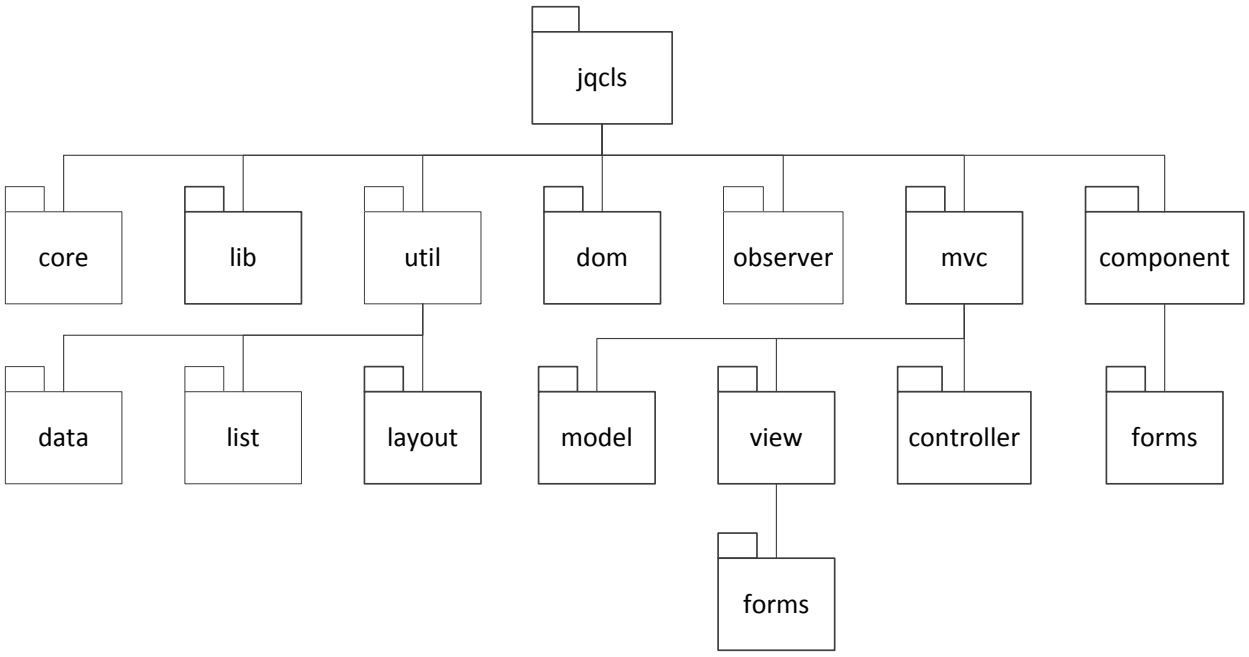



**Fig. 8.2 Diagrama de pachete**

După cum am precizat, jqCls introduce conceptul de aplicaţie. Abordarea este similară cu cea a aplicaţiilor Java. În acest sens, interfaţa Iapplication defineşte metoda *start* care poate fi apelată precum *main* în Java pentru rularea unei aplicaţii. Exemplificăm acest pas în utilizarea jqCls prin următorul extras de cod din aplicaţia demonstrativă.

```
<script src="./resources/lib/jqcls-0.5.1.339.0.js" type="text/javascript">
</script>
<script type="text/javascript">
        /**
        * Configurari requireJs
        * baseUrl - Locatia fisierelor javascript ale Airline Manager
        **/
            require.config({
                baseUrl: '/resources/js',
                shim: {
                    'jqcls': {
                        exports: 'jqcls'
                    }
                }
            });
        /**
        * Cererea de incarcare a fisierului ./resources/js/index/index.js
        * Apelarea metodei start.
        **/
            require(['index/index'], function (index) {
                new jqcls.ns.airlineManager.AirlineManager().start();
            });
</script>
```

Definirea propriu-zisă a unei aplicaţii jqCls este realizată prin extinderea clasei Application după cum urmează.

```
//define([modul1, modul2, LISTA_DEPENDENTE], function (mod1, mod2,
DEPENDENTE) { ... return MODUL }
define(['Application', <ALTE_RESURSE>], function (Application) {
    jqcls.Class.define('<SPATIU_DE_NUME>.<NUMELE_APLICATIEI>',{
        //Subclasarea jqcls.Application
        extend:Application,
        start: function () {
            //Cod de initializare
            //Crearea unei componente de tip screen
            //Apelul metodei suprascrise

jqcls.ns.airlineManager.AirlineManager.prototype.parent.start.apply();
        }
    });
    //Returnarea modulului definit
    return jqcls.ns.airlineManager;
});
```

Sistemul ierarhic de module al jqCls este realizat prin requireJS. Reţinem că pentru definirea unui modul este utilizată funcţia *define* în timp ce pentru încărcare este apelată funcţia *require*. Din exemplele prezentate pentru configurarea unei aplicaţii jqCls observăm că lista de parametri a funcţiei *define* conţine dependenţele de care depinde modulul şi apelul *callback* de definiţie. Modulele de care depinde cel definit sunt transmise ca parametri funcţiei de definiţie. La rândul ei, funcţia *callback* returnează un obiect JavaScript care reprezintă modulul definit.

Conceptele de clasă şi interfaţă jqCls pornesc de la implementările jqcls.Interface şi jqcls.Class. O clasă sau o interfaţă sunt declarate prin apelul metodelor jqcls.Class.define respectiv jqcls.Interface.define descrise în secvenţa de cod următoare.

```
define([<LISTA_DEPENDENTE>], function ([OBIECTE DE CARE DEPINDE CLASA]) {
    jqcls.Class.define('<SPATIU_DE_NUME>.<NUMELE_CLASEI>',{
        //Subclasarea jqcls.Application
        [extend: PARINTE,] //Implicit jqcls.Class
        [singleton: true || false]
        [abstract: true || false] //Implicit false
        [interfaces: [LISTA_INTERFETE],] //Implicit []
        [constructor: function (ARGUMENTE) {

<SPATIU_DE_NUME>.<NUMELE_CLASEI>.prototype.parent.constructor.apply(SCOP,
ARGUMENTE);
        },]
        METODE_SI_ATRIBUTE
    });
    //Returnarea clasei
    return <SPATIU_DE_NUME>.<NUMELE_CLASEI>;
});

define([<LISTA_DEPENDENTE>], function ([OBIECTE DE CARE DEPINDE CLASA]) {
    jqcls.Interface.define('<SPATIU_DE_NUME>.<NUMELE_INTERFETEI>',{
        //Subclasarea jqcls.Application
        [extend: PARINTE,] //Implicit jqcls.Interface
        methods: [LISTA_METODE]
    });
    //Returnarea interfetei
    return <SPATIU_DE_NUME>.<NUMELE_INTERFETEI>;
});
```

Observăm că similar *super* în Java pentru apelarea constructorului sau a metodei părinte, în jqCls se utilizează `CLASĂ.prototype.parent.METODA.apply(SCOP, ARGUMENTE);`. Nu în utilmul rând, notăm utilizarea atributului *singleton* pentru utilizarea modelului *Singleton*.

Specificarea modelului de date este posibilă în jqCls prin implementarea proprie frameworkului a clasei *Model* sau prin utilizarea modelelor Knockback integrate. În acest sens menţionăm că implementarea proprie jqCls a clasei model constă în obiecte de tip POJO (*Plain Old JavaScript Objects*). Modelul Knockback este mai complex şi poate conţine metodele de validare predefinite în plug-inul *Knockback.Validation*. Dintre tipurile de date pentru care este implementată validarea amintim tipurile *required*, *email* sau *numeric*. De asemenea, jqCls adaugă posibilitatea de validare la distanţă pe serverul de aplicaţie şi de verificare a egalităţii între atribute. Ultima poate fi deosebit de utilă pentru reprezentarea parolelor sau colectarea adresei de e-mail în procesul de înregistrare. Notăm că validările specifice unui model se definesc prin atributul de configurare *validations* reprezentat ca obiect JavaScript. În cazul unui model cu câmpul *attr1* validările aferente acestuia se specifică în atributul de configurare al validărilor utilizând acelaşi nume şi un obiect care conţine denumirea metodelor de validare, de exemplu `{ required : true }`.

Considerăm că listele şi sursele de date sunt următorul pas logic în descrierea utilizării jqCls. În acest sens jqCls implementează trei variante de liste pentru utilizarea conform cerinţelor de reprezentare a datelor *ArrayList*, *HashList* şi *LinkedList*. Popularea acestora poate fi realizată prin utilizarea unui proxy de date – fie la distanţă de tip *AjaxDataProxy*, fie local *LocalStorageDataProxy*, fie local cu cererea iniţială a datelor de pe serverul de aplicaţie. Ultima variantă reprezintă o combinaţie între *LocalStorageDataProxy* şi *AjaxDataProxy*. Vom

exemplifica utilizarea intermediarilor de date locale şi la distanţă prin obiectele de configurare aferente.

Astfel, pentru un proxy AJAX avem la dispoziţie următoarele opţiuni *baseURL*, *style*, *handlers* şi *reader* după cum rezultă şi din exemplificarea de mai jos. Considerăm că majoritate sunt autoexplicative. Atributul style defineşte modul în care este construită adresa la care se trimit cererile la distanţă. În modul *URL*, tipul de adresă generată este `baseURL/ACTIUNE/[PARAMETRI]`

Pentru *ACTION* este construit formatul clasic de tip `baseURL?action=ACTIUNE[[&param=val] /* pentru cereri de tip GET */ ]`

Următoarea secvenţă prezintă sintaxa obiectelor de configurare a proxyurilor de date AjaxDataProxy.

```
   {
       baseURL: LOCATIA_DE_BAZA_A_SERVICIULUI //
http://aplicatie.ap/utilizatori
       style: 'url' // Defineste forma adresei pentru cereri AJAX
       handlers: {
       //Apeluri callback pentru adaugare, modificare, incarcare si salvare
           [add: callback]
           [,edit: callback]
           [,load: callback]
           [,save: callback]
       },
       reader: //Definirea formatului json de citit
       ajax : //Configurari specifice jQuery.ajax,
       dataSourceType: //Tipul listei administrate
    }
```

Configurarea unui proxy de tip *LocalStorageDataProxy* este similară, cu menţiunea că se adaugă atributele *remoteDataSource* care specifică dacă intermediarul are şi o sursă de date la distanţă şi *syncRemote*, marcajul pentru sincronizarea datelor printr-un *AjaxDataProxy*. O altă caracteristică importantă a surselor de date este instanţierea listelor aferente în regim *FactoryMethod* prin specificarea tipului dorit în atributul *dataSourceType*.

Şablonul de design *Observer* este utilizat prin extinderea clasei *Observable* şi implementarea interfeţei *Observer* care defineşte metoda *update*. La adăugarea unui client în lista unui obiect *Observable* poate fi specificat numele evenimentului care prezintă interes. În mod implicit, evenimentul pentru care este înregistrat un observator este *notify*.

Modelul arhitectural MVC implementat în jqCls este simplu şi utilizează clasele tipice *Model*, *View* şi *Controller*. Tratarea evenimentelor în *Controller* este realizată utilizând implementarea jQuery *.on*. Pentru observarea modificărilor la nivel de model, este utilizată interfaţa *PropertyChangeListener*.

Integrarea cu frameworkul *Knockback* pentru utilizarea MVVM are la bază modele de tip *BackboneModel*. Astfel pentru obţinerea unui ViewModel se defineşte şi instanţiază un model *BackboneModel* şi se apelează metoda *getViewModel* pe instanţa obţinută.

Clasele jqCls de tip *Component* reprezintă locul de îmbinare a structurilor de date *Model* sau *BackboneModel* cu elementele vizuale şi cu obiectele de control de tip *Controller* sau *ViewModel*.

Pentru realizarea vederilor, jqCls conţine un set de administratori de aranjare în pagină. Funcţionalitatea este inspirată din Java Swing. Dintre administratorii de pagină implementaţi amintim *FillLayout* pentru elemente care trebuie să umple spaţiul pus la dispoziţie de părinte,

*FlowLayout* orizontal şi vertical şi *GridLayout* pentru arajarea tabelară. *DesktopLayout* şi *BorderLayout* reprezintă implementări complexe ale unui administrator de pagină. Astfel, *BorderLayout* împarte ecranul în cinci regiuni *TOP*, *LEFT*, *CENTER*, *RIGHT* şi *BOTTOM*. Poziţionarea acestora este exemplificată în figura 8.3. *CENTER* reprezintă regiunea principală şi este dimensionată pentru a umple spaţiul în funcţie de dimensiunile celorlale. De asemenea, regiunile *TOP* şi *BOTTOM* vor umple spaţiul pe axa orizontală. O componentă poate fi plasată într-o regiune prin specificarea atributului *region TOP*, *LEFT*, *CENTER*, *RIGHT* sau *BOTTOM* în obiectul de configurare dat ca argument constructorului acesteia.

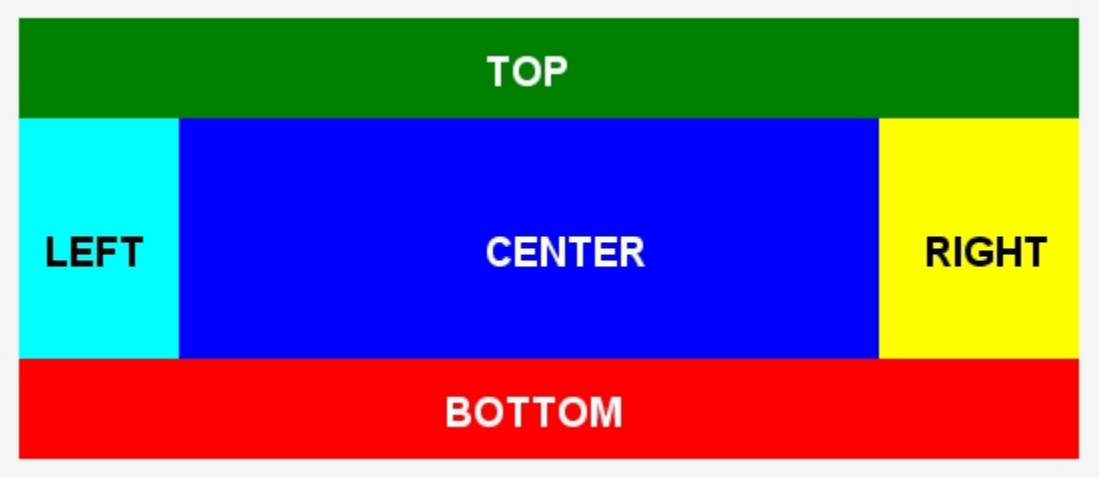



**Fig. 8.3 Regiunile administratorului de pagină BorderLayout**

Administratorul *DesktopLayout* este destinat sistemului de interfaţă multi-document. Sarcina principală a acestuia este de controlare a ordinii ferestrelor utilizând atributul CSS *zIndex* după cum sugerează.

Menţionăm că, similar cazului intermediarilor de date, instanţierea unui administrator de aranjare în pagină este tratată prin modelul *factory* după cum vom arată în exemplificarea utilizării componentelor jqCls pe care le vom discuta în continuare.

Ierarhia de componente jqCls are la bază clasele *AbstractComponent* şi *Block* care implementează comportamentul comun general şi sunt specializate în unităţi funcţionale concrete cum ar fi *Screen*, *Menu*, *Window*, *Form* sau *Grid*. În acest sens, recomandăm subclasarea *Block* sau *AbstractComponent* pentru implementarea componentelor definite de utilizator.

Pornind de la realizarea ecranului de autentificare şi înregistrare al aplicaţiei *AirlineManager*, vom aborda câteva aspectele generale de utilizare a sistemului de componente jqCls. Codul sursă integral pentru definirea acestui ecran poate fi consultat în anexa 6. Extrasele relevante ale acestei secvenţe de cod vor fi prezentate mai jos.

```
define(['Application', 'index/components/RegistrationForm'], function
(Application) {
    jqcls.Class.define("ns.airlineManager.AirlineManager", {
        extend: Application,
//Definirea functiei callback de executat dupa incarcarea resurselor necesare
        start: function () {
            var screen = new jqcls.ns.components.Screen({
```

```
                components: [/*** Construirea initiala a componentelor**/]
            });
// Apelul metodei clasei parinte
jqcls.ns.airlineManager.AirlineManager.prototype.parent.start.apply();
        }
    });
    return jqcls.ns.airlineManager.AirlineManager;
});
```

O observaţie imediată este iniţializarea unei componente de tip *Screen* în metoda *start* a clasei *jqcls.ns.airlineManager.AirlineManager*. După cum am precizat, o instanţă *Screen* repreztină nodul părinte al componentelor unei aplicaţii jqCls şi este asociat unei vederi de tip *ScreenView*. Remarcăm că în structura aplicaţiilor jqCls o instanţă *Screen* este suportul principal de conţinut şi are sarcina de a declanşa procesul de construcţie iniţială a interfeţei grafice. Ierarhia de componente care alcătuiesc o unitate funcţională în aplicaţiile jqCls se constituie în atributul *components* al obiectului de configurare dat ca parametru constructorului acesteia. Aplicând aspectele practice descrise asupra definiţiei modelului *Composite* constatăm că realizarea obiectului *compozit* se face prin atributul *components* menţionat. Următoarea secvenţă de cod ilustrează specificarea componentelor ecranului *Screen* care conţine formularele de autentificare şi înregistrare. Notăm că atributul *components* poate fi format din instanţierea directă prin utilizarea constructorului cum este cazul ferestrei pentru formularul de autentificare, apeluri de funcţii care returnează o componentă sau variabile care conţin referinţe spre un obiect de tip *Component*. Pentru modularitate şi lizibilitate recomandăm utilizarea ultimelor două variante.

```
components: [
    //Initializarea formularului de inregistrare
    new jqcls.ns.airlineManager.RegistrationForm(),
    //Initializarea ferestrei care contine formularul de autentificare
    new jqcls.ns.components.Window({
        title: 'Log in',
        /**
        * Alte proprietati specifice
        * ...
        */
        components: [
            //Initializarea formularului de autentificare
            new jqcls.ns.components.forms.Form({
                region: jqcls.ns.util.layout.BorderLayout.CENTER,
                //Specificarea proprietatilor elementului HTML
                element: {
                    attributes: {"class": 'login-form'}
                },
                components: [/*** Câmpurile formularului de autentificare*/]
            })
        ]
    })
]
```

jqCls porneşte de la considerentul că orice interfaţă grafică poate fi abstractizată prin trei concepte fundamentale : formulare pentru interacţiunea cu utilizatorul, vederi tabelare pentru prezentarea colecţiilor de date şi vederi de detaliu. În implementarea jqCls conceptele enunţate

sun reflectate de clasele *Form* şi *Input* pentru realizarea formularelor şi *Grid* în ceea ce priveşte prezentarea colecţiilor de date.

Pornind de la implementarea formularului de înregistrare în *Airline Manager* vom exemplifica realizarea formularelor prin frameworkul jqCls. În figura 8.4 prezentăm formularul obţinut prin utilizarea API-ului jqCls. Secvenţele de cod aferente acestui exemplu pot fi consultate integral în anexa 7.

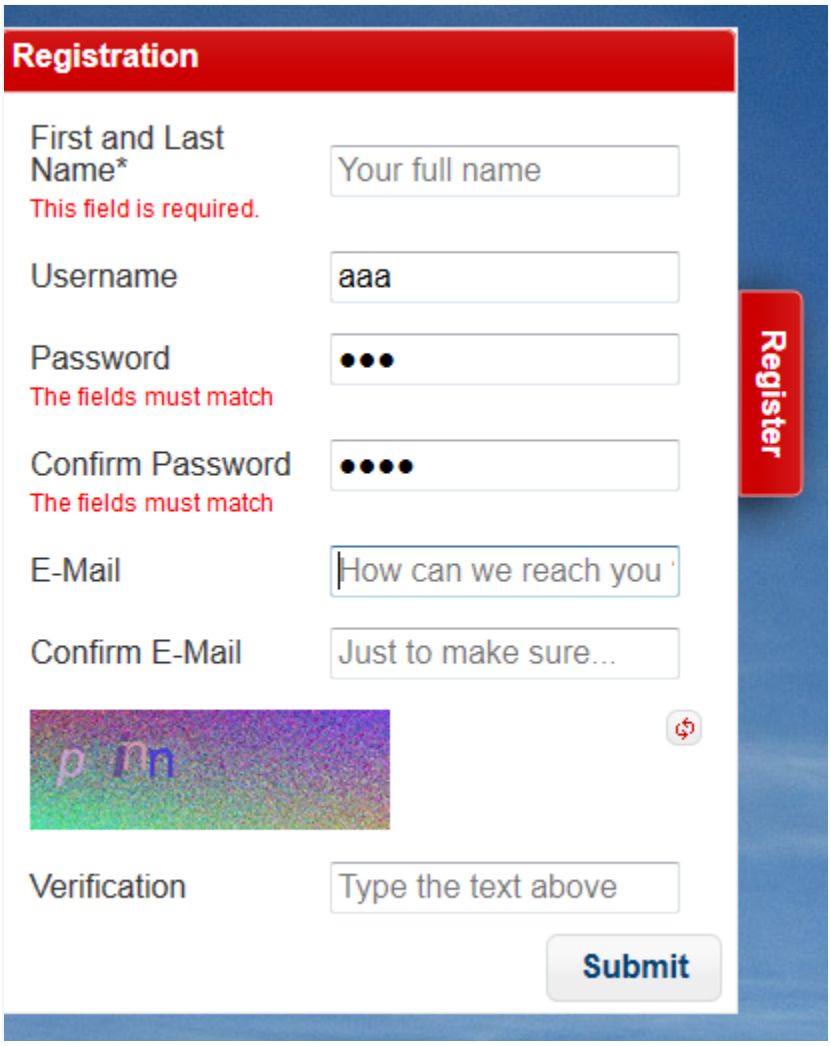



**Fig. 8.4 Formularul de înregistrare în aplicaţia Airline Manager**

Pentru realizarea formularului au fost definite componenta *RegistrationForm* care extinde *Block*, vederea *RegistrationFormView* pe baza *FormView* şi modelul *RegistrationModel* – o specializare a clasei *BackboneModel*. Observăm că, din punctul de vedere al modelului arhitectural, este exploatată integrarea cu frameworkul *Knockback*. *ResitrationForm* îmbină partea de vedere cu modelul, fiind responsabilă de crearea componentei de tip *Form*, instanţierea *RegistrationFormView* şi a modelului de date *RegistrationModel*. De asemenea, *RegistrationForm* creează componenta de tip *ViewModel* prin apelul metodei *getViewModel* a instanţei modelului *RegistrationModel*.

Definirea modelului de date conţine două părţi esenţiale : configurarea validărilor şi specificarea atributelor. Acestea sunt exemplificate în următoarea secvenţă de cod.

```
validations: {
    firstLastName: {required: true},
    username: {required: true, uniqueUser: true},
    password: {required: true, mustEqual: 'passwordConfirmation'},
    passwordConfirmation: {required: true, mustEqual: 'password'},
    email: {email: true, mustEqual: 'emailConfirmation'},
    emailConfirmation:{email: true, mustEqual: 'email'},
    captcha:{required: true}
},
constructor: function () {
    var me = this;
    me.config = {};
    me.config.attributes = {
        firstLastName: "",
        username: "",
        password: "",
        passwordConfirmation: "",
        email: "",
        emailConfirmation: "",
        captcha: ""
    };
    me.options = {url: './users/register.json'};
jqcls.ns.airlineManager.RegistrationModel.prototype.parent.constructor.apply(
      me, [me.config.attributes, me.options]);
}
```

Astfel, modelul de date conţine câmpurile *firstLastname*, *username*, *password*, *email*, *captcha* şi *passwordConfirmation*. Observăm constrângerile de integritate aplicate modelului prin câmpul *validations*. Astfel, modelul asigură completarea tuturor câmpurilor. Mai mult, numele de utilizator trebuie să fie unic prin specificarea constrângerii *uniqueUser*, iar pentru parolă şi email este verificată potrivirea cu câmpul de confirmare prin *mustEqual*.

Componentele formularului de înregistrare sunt declarate în *RegistrationForm* conform modului valabil la nivel general în jqCls. Considerăm că următoarea secvenţă de cod este sugestivă pentru modul de utilizare al intrfeţei de programare a pachetului *forms* în vederea definirii câmpurilor unui formular HTML.

```
new jqcls.ns.components.forms.Input({
    label: 'Confirm E-Mail',
    inputConfig: {
        attributes: {
            name: 'passwordConfirmation',
            "data-bind": 'value: emailConfirmation',
            placeholder: "Just to make sure...",
            type: 'email'
        }
    }
}),
new jqcls.ns.components.forms.CaptchaInput({
    label: 'Verification',
    inputConfig: {
        attributes: {
            name: 'captcha',
            "data-bind": 'value: captcha',
```

```
            placeholder: 'Type the text above'
        }
    },
    imgSource: './captcha/image'
}),
new jqcls.ns.components.Button({
    element: {
        attributes: {"class": 'align-right'}
    },
    jQueryUiButtonConfig: {label: 'Submit'}
})
```

Efectul secvenţei prezentate este generarea câmpurilor pentru confirmarea adresei email, afişarea şi introducerea codului CAPTCHA şi a butonului pentru trimiterea formularului. Observăm utilizarea proprietăţii *data-bind* în configurarea atributelor HTML ale câmpurilor generate. Aceasta reprezintă legătura între vedere şi modelul de date.


**Fig. 8.5 Grilă jqGrid integrată cu sistemul de ferestre jqCls**

Alături de formulare, grilele reprezintă un concept fundamental în realizarea interfeţelor grafice. JqCls tratează problema prin integrarea API-ului jqGrid. În figura 8.5 exemplificăm o astfel de componentă. Pentru instanţierea unei grile, în jqCls se utilizează clasa *Grid*. Aceasta învăluie API-ul jqGrid. În anexa prezentăm un exemplu complet de utilizare din care considerăm importantă următoarea secvenţă demonstrativă pentru instanţierea şi configurarea unei grile.

```
new jqcls.ns.components.Grid({
    jqGrid: {
        width: 200,
        datatype: 'local',
        data: jqgridData,
        colNames: ['Nume', 'Prenume'],
        colModel: [
            {name: 'col1', index: 'col1', width: 55},
            {name: 'col2', index: 'col2', width: 90}
```

```
        ],
        sortname: 'col1',
        sortorder: 'desc',
        caption: 'Exemplu vedere grila'})
```


**Fig. 8.6 Interfaţa MDI**

Frameworkul jqCls implementează un sistem de realizare a interfeţelor multidocument uşor de utilizat. Figura 8.6 prezintă componentele de bază ale interfeţei MDI jqCls bara de activităţi şi componentele de tip *Window*. În utilizarea unei interfeţe MDI pornim de la clasele *DesktopManager* şi *Window* prin care este implementat conceptul în jqCls. Anexa 9 conţine un exemplu complet de configurare a administratorului de ferestre jqCls.

În concluzie, menţionăm că, din gama de funcţionalitate implementată, am ales pentru fiecare subiect abordat punctele culminante în scopul de a crea o imagine de ansamblu asupra capabilităţilor frameworkului jqCls. Notăm că informaţiile cu privire la întreaga interfaţă de programare pusă la dispoziţie de frameworkul jqCls sunt accesibile uşor prin documentaţia API. Însumând, suntem de părere că alături de detaliile de implementare, exemplele de utilizare discutate confirmă îndeplinirea obiectivelor practice fixate iniţial.

# 9. Concluzii şi dezvoltări ulterioare

În încheierea acestei lucrări propunem evaluarea obiectivelor fixate şi a gradului de îndeplinire al acestora, evidenţierea contribuţiei proprii şi descrierea posibilităţilor ulterioare de dezvoltare.

Astfel, din punct de vedere teoretic, ne-am propus aprofundarea cunoştiinţelor de JavaScript, prin prezentarea aspectelor specifice de limbaj, a standardelor existente în vigoare şi studiul şi exemplificarea metodelor de aplicare ale conceptelor clasice de programare obiectuală.

Din punct de vedere practic, în lucrarea de faţă, elementele teoretice enumerate mai sus sunt aplicate în realizarea prototipului unui framework JavaScript *open-source* cu domeniu general de utilizare pentru dezvoltarea aplicaţiilor Web client în arhitecturi SOA. Cerinţele de bază impuse frameworkului sunt interfaţa de programare unitară dar transparentă în raport cu soluţiile utilizate la nivelele inferioare şi cu un grad de cuplare minim între module, identificarea şi integrarea soluţiilor *open-source* existente, implementarea unui nivel consistent de reprezentare a datelor, utilizarea şi abstractizarea şabloanelor arhitecturale şi de design şi a unui set de componente vizuale reutilizabile, extensibile şi scalabile. Nu în ultimul rând, ne-am propus validarea frameworkului prin utilizarea acestuia în construirea unei aplicaţii demonstrative.

## 9.1. Realizări

Considerăm că obiectivele teoretice şi practice enumerate mai sus au fost îndeplinite. În vederea **aprofundării cunoştiţelor de JavaScript**, în capitolul trei, au fost prezentate amplu caracteristicile principale ale limbajului. Dintre acestea amintim : standardele actuale în vigoare, reprezentarea datelor, aplicarea modelului *Prototype*, conceptele de scop şi închidere, mediul specific navigatorului sau modelul de propagare al evenimentelor.

**Studiul metodelor de aplicare ale conceptelor clasice POO şi ale şabloanelor de design** a fost efectuat riguros în capitolul patru şi, reiterând concluziile acestuia, a demonstrat că abordarea aplicaţiilor *Web* în maniera reutilizabilă şi modulară, propusă la începutul lucrării, este o consecinţă a programării prin paradigma orientată pe obiecte în JavaScript.

Cu privire la aspectele practice ale acestei lucrări, frameworkul jqCls este **construit modular şi implementează conceptele fundamentale de programare obiectuală** în JavaScript. Această afirmaţie se bazează pe sistemul ierarhic de clase şi interfeţe organizate în module şi pe implementarea abstractă a modelelor de design *Observer* şi *Singleton*. Nu în ultimul rând, detaliile de implementare ale frameworkului demonstrează utilizarea altor şabloane fundamentale cum ar fi *Factory Method*, *Composite* şi *Decorator*.

Integrarea frameworkurilor *Backbone, Knockout* şi *Underscore* demonstrează flexibilitatea şi **cuplarea minimă** între module având în vedere că în locul modelului MVC implementat în jqCls poate fi utilizată varianta MVVM. Amintim că vederile jqCls sunt independente de soluţia arhitecturală aleasă. Nu în ultimul rând, prin integrarea librăriilor menţionate, considerăm îndeplinit obiectivului de **împachetare a soluţiilor *open-source* existente într-o interfaţă de programare care tratează problemele aplicaţiilor Web în ansamblu**.

Exemplele atât generale, cât şi specifice din capitolul anterior confirmă implementarea unui nivel complex de abstractizare a datelor şi a proceselor de *bussiness* specifice unei aplicaţii, cât şi construirea unui set de componente vizuale, care pot fi utilizate la scară largă.

## 9.2. Dezvoltări ulterioare

Implementarea practică a acestei lucrări este un prototip şi trebuie tratată ca atare. Prin urmare, o îmbunătăţire imediată ţine de creşterea performanţei şi îmbunătăţirea legăturilor între module. Ca orice soluţie de anvergura unui framework, jqCls se află în faza de maturizare. În acest sens, arhitectura utilizată trebuie stabilizată.

Strict legat de implementare şi de extinderea funcţionalităţii, amintim semnalarea sau interzicerea redefinirii unei entităţi, fie spaţiu de nume, clasă sau interfaţă.

O direcţie interesantă de dezvoltare este implementarea unor structuri arborescente – de exemplu: arbori binari – şi a algoritmilor de căutare şi sortare răspândiţi în alte limbaje de programare.

Ţinând cont de dublarea nivelului *business* în client prin utilizarea unei arhitecturi orientate pe servicii, aşa cum este gândită aceasta la nivelul jqCls, o posibilitate de dezvoltare practică este maparea automată a reprezentării modelului similar cu modul în care este rezolvată problema în cazul serviciilor Web prin specificaţia WSDL.

Nu în ultimul rând, în privinţa compatibilităţii înapoi cu navigatoare mai vechi, amintim asigurarea degradării controlate pentru ansamblul de componente jqCls.

În ceea ce priveşte conţinutul teoretic, suntem de părere că această lucrare întruneşte condiţiile pentru a fi dezvoltată într-un material educativ introductiv pe tema programării aplicaţiilor *Web* utilizând JavaScript.

## 9.3. Contribuţie proprie

Lucrarea de faţă contribuie în mod real la tratarea problemelor actuale cu care se confruntă domeniul programării aplicaţiilor Web. Din punct de vedere teoretic, sunt abordate subiectele de importanţă majoră ale programării aplicaţiilor JavaScript : diferenţele faţă de limbajele larg răspândite cu care este asemănat adesea în mod eronat. În acest sens, lucrarea de faţă conturează o imagine de sinteză a tehnicilor de programare obiectuală în JavaScript şi oferă argumente cu privire la avantajele utilizării acestora.

Din punctul de vedere al implementării practice, diferenţa majoră între jqCls şi alte produse asemănătoare este construcţia frameworkului propus în această lucrare atât prin implementări proprii  cât şi pe baza soluţiilor punctuale existente şi validate în comunitatea *open-source*.

Însumând, jqCls este un framework care abordează JavaScript strict prin prisma paradigmei obiectuale şi facilitează utilizarea acesteia. Având în vedere principiile urmate şi setul de funcţionalităţi oferite, suntem de părere că jqCls este un punct de pornire adecvat pentru dezvoltarea aplicaţiilor Web client conform cerinţelor actuale.

## 10. Bibliografie


Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerland, şi Michael Stal, *Pattern-Oriented Software Architecture - A System of Patterns.* Vol. I. Chichester: John Wiley & Sons Ltd., 2001.

Crockford, Douglas. *JavaScript: The Good Parts.* Sebastopol: O'Reilly Media, Inc., 2008.

Ecma International. „Standard ECMA-262 5.1 Edition." Standard, Geneva, 2011.

Flanagan, David. *JavaScript The Deifinitive Guide.* 6th. Sebastopol: O'Reilly Media inc., 2011.

Flesler, Ariel. „Chapter 8 : Events." În *jQuery Cookbook*, de Cody Lindley, 171-191. Sebastopol: O'Reilly, 2010.

Fowler, Martin. *Patterns of Enterprise Application Architecture.* Boston: Addison-Wesley, 2002.

Gamma, Erich, Richard Helm, Ralph Johnson, şi John Vlissides. *Design Patterns : elements of reusable object-oriented software.* Boston: Addison-Wesley, 1995.

Harmes, Ross, şi Dustin Diaz. *Pro JavaScript Design Patterns.* Apress, 2007.

ISTQB. *http://istqbexamcertification.com.* fără an. http://istqbexamcertification.com/what-is-decision-table-in-software-testing/ (accesat 6 26, 2013).

Joldoş, Marius. „Programare orientată pe obiecte : Note de curs, Cursul 1." *Programare orientată pe obiecte.* 1 ianuarie 2011. http://users.utcluj.ro/~jim/OOPR/Resources/Lectures/OOP01r4.pdf (accesat august 2, 2012).

Joldos, Marius. „Programare orientată pe obiecte : Note de curs, Cursul 5." *Programare orientată pe obiecte.* 1 ianuarie 2011. http://users.utcluj.ro/~jim/OOPR/Resources/Lectures/OOP05r4.pdf (accesat iulie 23, 2012).

jQuery Foundation. *Introduction to Unit Testing.* 2013. http://qunitjs.com/intro/ (accesat iunie 30, 2013).

—. *QUnit API Documentation.* 2013. http://api.qunitjs.com/ (accesat iunie 30, 2013).

—. *QUnit Cookbook.* 2013. http://qunitjs.com/cookbook/ (accesat iunie 30, 2013).

Lindley, Cody. *jQuery Cookbook.* Sebastopol: O'Reilly, 2010.

MacCaw, Alex. *JavaScript Web Applications.* Sebastopol: O'Reilly, 2011.

Microsoft. *MSDN.* fără an. http://msdn.microsoft.com/en-us/library/ff649643.aspx (accesat 3 14, 2013).

—. *MSDN.* fără an. http://msdn.microsoft.com/en-us/library/ff798384.aspx (accesat 3 15, 2013).

Mozilla Developer Network. *Docs - bind : Mozilla Developer Network.* 6 iulie 2012. https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Function/bind (accesat august 1, 2012).

—. *Mozilla Developer Network Docs - apply.* 26 iunie 2012. https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Function/apply (accesat iulie 24, 2012).

—. *Mozilla Developer Network Docs - call.* 26 iunie 2012. https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/Function/call (accesat august 1, 2012).

RequireJS. „RequireJS." *How to Use RequireJS with JQuery.* fără an. http://requirejs.org/jquery.html (accesat 10 13, 2012).

—. „RequireJS." *Asynchronous Module Definition API.* fără an. http://requirejs.org/whyamd.html (accesat 13 10, 2012).

—. „RequireJS." *Web Modules.* fără an. http://requirejs.org/why.html (accesat 10 13, 2012).

—. „RequireJS." *RequireJS API Documentation.* fără an. http://requirejs.org/docs/api.html (accesat 10 13, 2012).

Stefanov, Stoyan. *JavaScript Patterns.* Sebastopol: O'Reilly, 2010.

—. *Object-Orieted JavaScript.* Birmingham: Pakt Publishing, 2008.

Wikipedia. *Class (computer programming).* fără an. http://en.wikipedia.org/wiki/Class_(computer_programming) (accesat 7 15, 2012).

—. *Prototype-Based Programming.* fără an. http://en.wikipedia.org/wiki/Prototype-based_programming (accesat 7 15, 2012).

World Wide Web Consortium. *Document Object Model (DOM).* 19 ianuarie 2005. http://www.w3.org/DOM/#what (accesat iulie 17, 2012).

—. *http://www.w3.org.* 13 noiembrie 2000. http://www.w3.org/TR/DOM-Level-2-Events/events.html (accesat iulie 18, 2012).

—. *www.w3.org.* 28 iunie 2012. http://www.w3.org/TR/selectors-api/ (accesat iulie 17, 2012).

# Anexe

## Anexa 1 : Lista de tabele



## Anexa 2 : Lista de figuri

## Anexa 3 – Secvenţele de cod utilizate pentru testarea comportamentului moştenirii prototipice

```
/*
    Cazul 1 - Comportamentul obiectelor compuse initializate static in
relatie cu metoda mostenirii prototipice
*/
var mostenirePrototipica = (function(){
    var Proxy = function(){},
        functieExtindere = function(obj){
            Proxy.prototype = obj;
            return new Proxy();
        };
    return functieExtindere;
})();

var MijlocDeTransportAerian = {
        areMotoare : false,
        motoare : []
    };

MijlocDeTransportAerian.__proto__.decoleaza = function(){};
MijlocDeTransportAerian.__proto__.zboara = function(sursa, destinatie) {};
MijlocDeTransportAerian.__proto__.aterizeaza = function() {};

var Avion = mostenirePrototipica(MijlocDeTransportAerian),
    Planor = mostenirePrototipica(MijlocDeTransportAerian),
    AirbusA320 = mostenirePrototipica(Avion),
    Boeing737 = mostenirePrototipica(Avion),
    OZN = mostenirePrototipica(MijlocDeTransportAerian.__proto__);

AirbusA320.motoare.push("A320 Motor 1");
AirbusA320.motoare.push("A320 Motor 2");
Boeing737.motoare.push("B737 Motor 1");
Boeing737.motoare.push("B737 Motor 2");

Avion.areMotoare = true;
Planor.areMotoare = false;

/*
    Cazul 2 - Comportamentul obiectelor compuse in cazul reinitializarii
membrilor descenti ai Object
*/
var mostenirePrototipica = (function(){
    var Proxy = function(){},
        functieExtindere = function(obj){
            Proxy.prototype = obj;
            var obiectNou = new Proxy();

            /* Reinitializarea membrilor descenti din Object */
            obiectNou.init();
            return obiectNou;
        };
    return functieExtindere;
})();

var MijlocDeTransportAerian = {
```

```
        areMotoare : false,
        motoare : []
    };

/* Definirea functiei de initializare a campului motoare */
MijlocDeTransportAerian.__proto__._initializareMotoare = function() {
            this.motoare = new Array();
        }

/* Definirea functiei de initializare a obiectului compus
MijlocDeTransportAerian */
MijlocDeTransportAerian.__proto__.init = function() {
    this._initializareMotoare();
}

MijlocDeTransportAerian.__proto__.decoleaza = function(){};
MijlocDeTransportAerian.__proto__.zboara = function(sursa, destinatie) {};
MijlocDeTransportAerian.__proto__.aterizeaza = function() {};

/* Initializarea obiectului MijlocDeTransportAerian */
MijlocDeTransportAerian.init();

var Avion = mostenirePrototipica(MijlocDeTransportAerian),
    Planor = mostenirePrototipica(MijlocDeTransportAerian),
    AirbusA320 = mostenirePrototipica(Avion),
    Boeing737 = mostenirePrototipica(Avion),
    OZN = mostenirePrototipica(MijlocDeTransportAerian.__proto__);

AirbusA320.motoare.push("A320 Motor 1");
AirbusA320.motoare.push("A320 Motor 2");
Boeing737.motoare.push("B737 Motor 1");
Boeing737.motoare.push("B737 Motor 2");

Avion.areMotoare = true;
Planor.areMotoare = false;
```

## Anexa 4 – Secvenţe de cod utilizate pentru testarea manuală

### Anexa 4.1 – Testarea componentelor de tip *Toolbar*

```
/*
 * Copyright (c) 2013 Stefan Ioan Gencărău. All rights reserved.
 */

/**
 * Created with JetBrains WebStorm.
 * User: stefan
 * Date: 1/2/13
 * Time: 4:56 PM
 * To change this template use File | Settings | File Templates.
 */
/*global define:false */

define(
    ["js/Application"],
    function (Application) {
        "use strict";
```

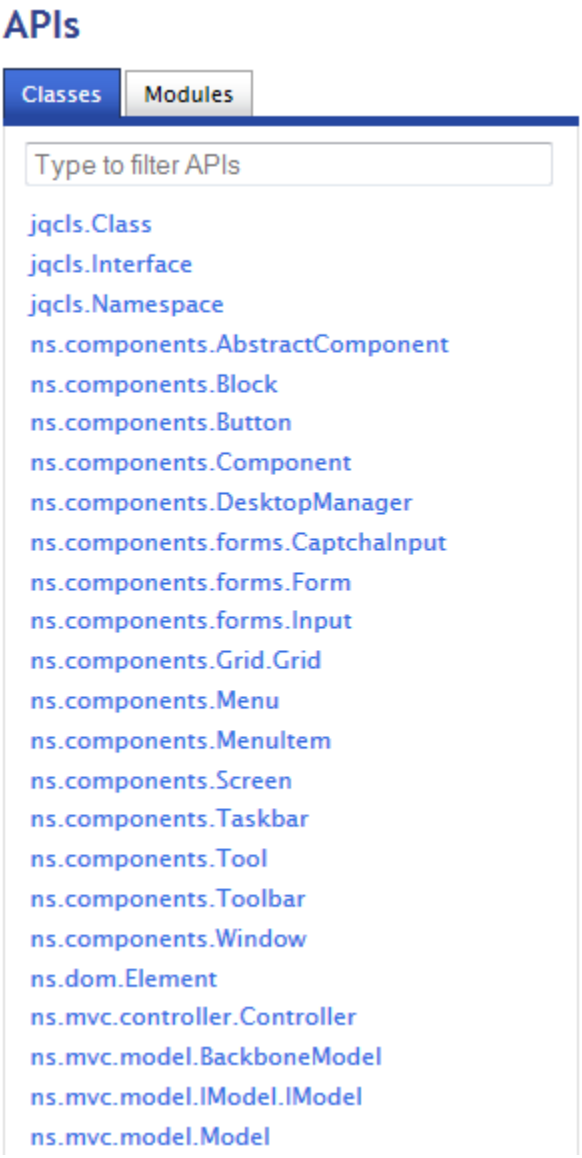
```
        jqcls.Class.define('ns.MyApp', {
            extend:jqcls.ns.Application,
            abstract:false,
            constructor:function (config) {
                var me = this;
                me.parent.constructor.apply(me, [config]);
            },
            start: function () {}
        });

        return new jqcls.ns.MyApp(
            {
                define:[],
                require:[
                    [],
                    function (Application) {
                        var screen = new jqcls.ns.components.Screen({
                            components: [
                                new jqcls.ns.components.Toolbar({
                                    components: [
                                        new jqcls.ns.components.Tool({
                                            jQueryUiButtonConfig: {
                                                label: 'Test 1'
                                            }
                                        }),
                                        new jqcls.ns.components.Tool({
                                            jQueryUiButtonConfig: {
                                                label: 'Test 2'
                                            },
                                            click: function () {
                                                alert("Test 2 clicked!");
                                            }
                                        })
] })]});}]});});
```

## Anexa 4.2 Testarea meniurilor

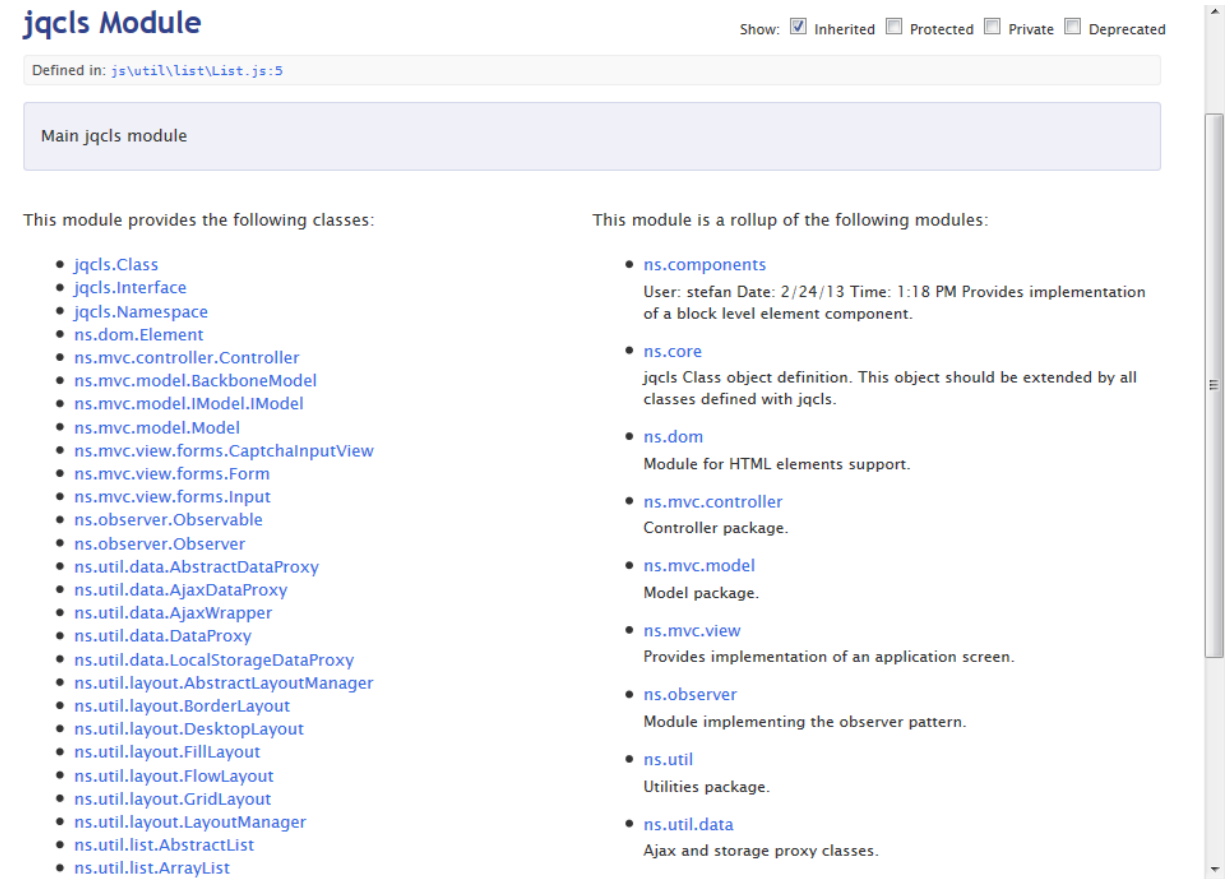
```
/*
 * Copyright (c) 2013 Stefan Ioan Gencărău. All rights reserved.
 */

/**
 * Created with JetBrains WebStorm.
 * User: stefan
 * Date: 1/2/13
 * Time: 4:56 PM
 * To change this template use File | Settings | File Templates.
 */
/*global define:false */

define(
    ["js/Application"],
    function (Application) {
        "use strict";
        jqcls.Class.define('ns.MyApp', {
            extend:jqcls.ns.Application,
```

```
            abstract:false,
            constructor:function (config) {
                var me = this;
                me.parent.constructor.apply(me, [config]);
            },
            start: function () {}
        });
        return new jqcls.ns.MyApp({
                define:[],
                require:[ [],
                    function (Application) {
                        var screen = new jqcls.ns.components.Screen({
                            components: [
                                new jqcls.ns.components.Menu({
                              direction: jqcls.ns.components.Menu.HORIZONTAL,
                                    components: [
                                        new jqcls.ns.components.MenuItem({
                                            html: 'Menu 1'
                                        }),
                                        new jqcls.ns.components.MenuItem({
                                            html: 'Menu 2',
                                            components: [
                                               new jqcls.ns.components.Menu({
                                                    components: [
                                                new jqcls.ns.components.MenuItem({
                                                            html: 'Submenu 2.1',
                                                               components: [
                                                    new jqcls.ns.components.Menu({
                                                     components: [
                                                new jqcls.ns.components.MenuItem({
                                                     html: 'Submenu 2.1.1'
                                                                                }),
                                                new jqcls.ns.components.MenuItem({
                                                     html: 'Submenu 2.1.2'
                                                                                })
                                                                            ]
                                                                        })
                                                                    ]
                                                                }),
                                                            new jqcls.ns.components.MenuItem({
                                                                html: 'Submenu 2.2'
})]})]})]})]});}]});});
```

## Anexa 5 Documentaţia API a frameworkului jqCls




**Lista de clase şi module a documentaţiei API**



## jqcls Module

Show: Inherited Protected Private Deprecated

Defined in: js\util\list\List.js:5

Main jqcls module

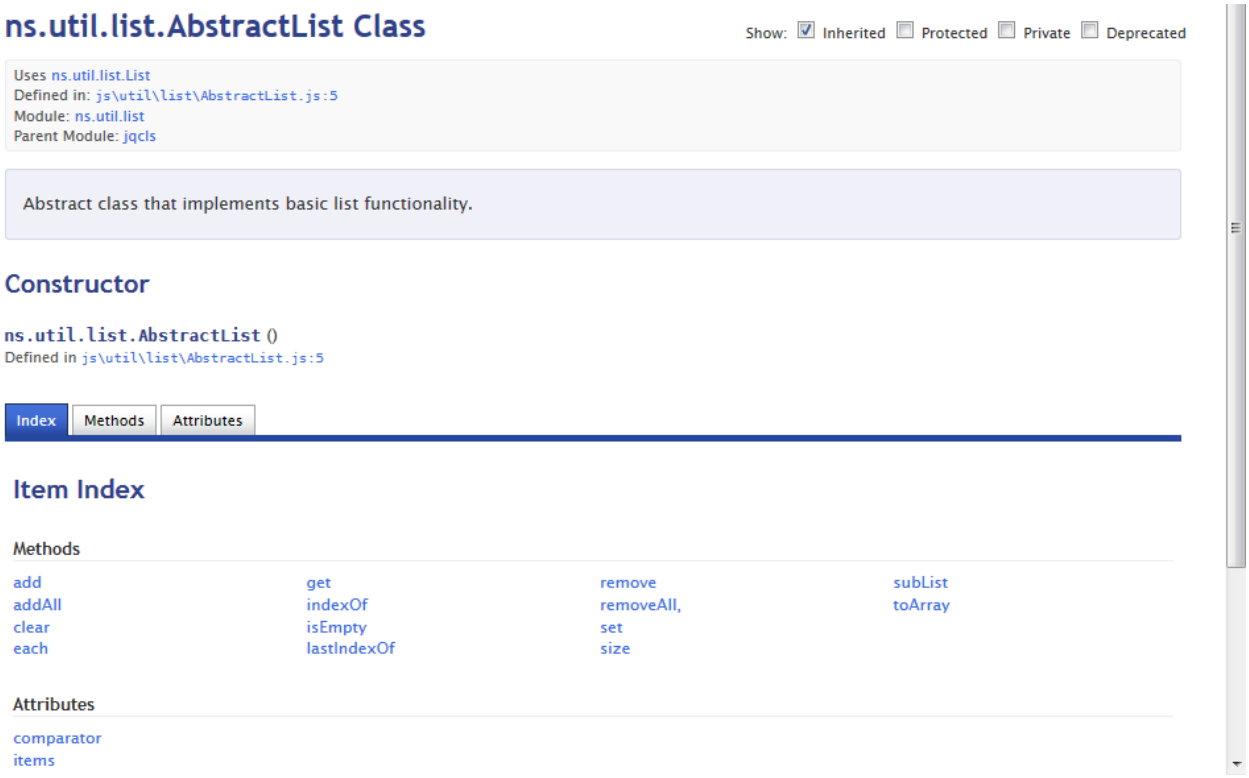
This module provides the following classes:

- jqcls.Class
- jqcls.Interface
- jqcls.Namespace
- ns.dom.Element
- ns.mvc.controller.Controller
- ns.mvc.model.BackboneModel
- ns.mvc.model.IModel.IModel
- ns.mvc.model.Model
- ns.mvc.view.forms.CaptchaInputView
- ns.mvc.view.forms.Form
- ns.mvc.view.forms.Input
- ns.observer.Observable
- ns.observer.Observer
- ns.util.data.AbstractDataProxy
- ns.util.data.AjaxDataProxy
- ns.util.data.AjaxWrapper
- ns.util.data.DataProxy
- ns.util.data.LocalStorageDataProxy
- ns.util.layout.AbstractLayoutManager
- ns.util.layout.BorderLayout
- ns.util.layout.DesktopLayout
- ns.util.layout.FillLayout
- ns.util.layout.FlowLayout
- ns.util.layout.GridLayout
- ns.util.layout.LayoutManager
- ns.util.list.AbstractList
- ns.util.list.ArrayList

This module is a rollup of the following modules:

- ns.components
  User: stefan Date: 2/24/13 Time: 1:18 PM Provides implementation of a block level element component.
- ns.core
  jqcls Class object definition. This object should be extended by all classes defined with jqcls.
- ns.dom
  Module for HTML elements support.
- ns.mvc.controller
  Controller package.
- ns.mvc.model
  Model package.
- ns.mvc.view
  Provides implementation of an application screen.
- ns.observer
  Module implementing the observer pattern.
- ns.util
  Utilities package.
- ns.util.data
  Ajax and storage proxy classes.

**Detalierea unui modul**



## ns.util.list.AbstractList Class

Show: Inherited Protected Private Deprecated

Uses ns.util.list.List
Defined in: js\util\list\AbstractList.js:5
Module: ns.util.list
Parent Module: jqcls

Abstract class that implements basic list functionality.

### Constructor

**ns.util.list.AbstractList** ()
Defined in js\util\list\AbstractList.js:5

Index | Methods | Attributes

### Item Index

#### Methods

add, addAll, clear, each, get, indexOf, isEmpty, lastIndexOf, remove, removeAll, set, size, subList, toArray

#### Attributes

comparator
items

**Vederea de detaliu a unei clase**

## Anexa 6 Definirea ecranului de autentificare şi înregistrare al aplicaţiei *Airline Manager*

```
define(['Application', 'index/components/RegistrationForm'], function
(Application) {
    jqcls.Class.define("ns.airlineManager.AirlineManager", {
        extend: Application,
        start: function () {
            var screen = new jqcls.ns.components.Screen({
                components: [
                    new jqcls.ns.components.DesktopManager({
                        title: 'Virtual Airline Manager',
                        hasTaskbar: false,
                        components: [
                            //Initializarea formularului de inregistrare
                            new jqcls.ns.airlineManager.RegistrationForm(),
//Initializarea ferestrei care contine formularul de autentificare
                            new jqcls.ns.components.Window({
                                title: 'Log in',
                                /**
                                * Alte proprietati specifice
                                * ...
                                */
                                components: [
//Initializarea formularului de autentificare
                                    new jqcls.ns.components.forms.Form({
                                        region:
jqcls.ns.util.layout.BorderLayout.CENTER,
//Specificarea proprietatilor elementului HTML
                                        element: {
                                            attributes: {
                                                "class": 'login-form'
                                            }
                                        },
                                        components: [
//Specificarea campurilor formularului
                                        new jqcls.ns.components.forms.Input({
                                                label: 'Username',
                                                inputConfig: {
                                                    attributes: {
                                                        name: 'username'
                                                    }
                                                }
                                            }),
                                        new jqcls.ns.components.forms.Input({
                                                label: 'Password',
                                                inputConfig: {
                                                    tagName: 'input',
                                                    attributes: {
                                                        type: 'password',
                                                        name: 'password'
                                                    }
                                                }
                                            }),
                                            new jqcls.ns.components.Button({
                                                element: {
                                                    attributes: {
                                                       "class": 'align-right'
```

```
                                                    }
                                                },
                                                jQueryUiButtonConfig: {
                                                    label: 'Submit'
                                                }
})]})]})]})]});
// Apelul metodei clasei parinte
jqcls.ns.airlineManager.AirlineManager.prototype.parent.start.apply();
}});
    return jqcls.ns.airlineManager.AirlineManager;
});
```

## Anexa 7 Secvenţe de cod utilizate pentru realizarea formularului de înregistrare

### RegistrationForm.js

```
/**
 * Created with IntelliJ IDEA.
 * User: stefan
 * Date: 5/25/13
 * Time: 10:47 AM
*/
define(['components', 'index/components/RegistrationFormView',
'index/components/RegistrationModel'], function () {
    function createForm () {
        return new jqcls.ns.components.forms.Form({
            title: 'Registration',
            components: [
                new jqcls.ns.components.forms.Input({
                    label: 'First and Last Name*',
                    inputConfig: {
                        attributes: {
                            name: 'name',
                            "data-bind": "value: firstLastName",
                            placeholder: "Your full name"
                        }
                    }
                }),
                new jqcls.ns.components.forms.Input({
                    label: 'Username',
                    inputConfig: {
                        attributes: {
                            name: 'username',
                            "data-bind": 'value: username',
                            placeholder: 'The desired username'
                        }
                    }
                }),
                new jqcls.ns.components.forms.Input({
                    label: 'Password',
                    inputConfig: {
                        attributes: {
                            name: 'password',
                            type: 'password',
                            "data-bind": "value: password",
```

```
                            placeholder: "Something safe..."
                        }
                    }
                }),
                new jqcls.ns.components.forms.Input({
                    label: 'Confirm Password',
                    inputConfig: {
                        attributes: {
                            name: 'passwordConfirmation',
                            "data-bind": 'value: passwordConfirmation',
                            placeholder: "Just to make sure...",
                            type: 'password'
                        }
                    }
                }),
                new jqcls.ns.components.forms.Input({
                    label: 'E-Mail',
                    inputConfig: {
                        attributes: {
                            name: 'email',
                            "data-bind": 'value: email',
                            placeholder: "How can we reach you ?",
                            type: 'email'
                        }
                    }
                }),
                new jqcls.ns.components.forms.Input({
                    label: 'Confirm E-Mail',
                    inputConfig: {
                        attributes: {
                            name: 'passwordConfirmation',
                            "data-bind": 'value: emailConfirmation',
                            placeholder: "Just to make sure...",
                            type: 'email'
                        }
                    }
                }),
                new jqcls.ns.components.forms.CaptchaInput({
                    label: 'Verification',
                    inputConfig: {
                        attributes: {
                            name: 'captcha',
                            "data-bind": 'value: captcha',
                            placeholder: 'Type the text above'
                        }
                    },
                    imgSource: './captcha/image'
                }),
                new jqcls.ns.components.Button({
                    element: {
                        attributes: {"class": 'align-right'}
                    },
                    jQueryUiButtonConfig: {label: 'Submit'}
                })
            ]
        });
    };
```

```
    jqcls.Class.define('ns.airlineManager.RegistrationForm', {
        extend: jqcls.ns.components.Block,
        config: {
            width: 390,
            layoutManagerClass: jqcls.ns.util.layout.FlowLayout,
            element: {
                tagName: 'div',
                attributes: {
                    'class': 'block ui-widget registration-component'
                }
            },
            title: 'Register'
        },
        constructor: function (config) {
            this.model = new jqcls.ns.airlineManager.RegistrationModel();
            var form = createForm();
            this.config.components = [form];
            this.view = new
jqcls.ns.airlineManager.RegistrationFormView(this.config);
jqcls.ns.airlineManager.RegistrationForm.prototype.parent.constructor.apply(
      this, [this.config]
);
            var me = this;
            this.model = this.model.getViewModel();
            form.model = this.model;
            $(jqcls).on('appStart', function () {
                jqcls.findMe = me.model;
                jqcls.ko.applyBindings(me.model, me.view.element.element());
            });
        }
    });
    return jqcls.ns.airlineManager.RegistrationForm;
});
```

### RegistrationFormView.js

```
define(
    ['mvc'],
    function () {
        var defaultConfig = {};
        jqcls.Class.define('ns.airlineManager.RegistrationFormView', {
            extend: jqcls.ns.mvc.view.BlockView,
            state: 0,
            render: function () {
jqcls.ns.airlineManager.RegistrationFormView.prototype.parent.render.apply(
      this, []);
                var me = this;
                me.element.jqElement().find('.ui-widget-
header:first').click({scope: me}, me.toggle);
            },
            toggle: function (e) {
                var me = e.data.scope,
                    left;
                if (me.state ===
jqcls.ns.airlineManager.RegistrationFormView.EXPANDED) {
                    left = -370;
```

```
                    me.state =
jqcls.ns.airlineManager.RegistrationFormView.COLLAPSED;
                } else {
                    left = 0;
                    me.state =
jqcls.ns.airlineManager.RegistrationFormView.EXPANDED;
                }
                me.element.jqElement().animate({
                    left: left
                },
                1000,
                'easeInOutExpo')
            }
        });
        jqcls.ns.airlineManager.RegistrationFormView.EXPANDED = 1;
        jqcls.ns.airlineManager.RegistrationFormView.COLLAPSED = 0;
        return jqcls.ns.airlineManager.RegistrationFormView;
    });
```

### RegistrationModel.js

```
define(
    ['mvc', 'knockout'],
    function (mvc, ko) {
        var validationUrl = '/users/isUsernameAvailable.json';
        var validationClient = new jqcls.ns.util.data.AjaxWrapper({
            dataType: 'json',
            url: validationUrl
        });
        jqcls.Class.define('ns.airlineManager.RMUsernameValidator', {
            implement: [jqcls.ns.observer.Observer],
            update: function (event, xhr, data) {
                this.callback(event, xhr, data);
            }
        });
        var validationObserver = new
jqcls.ns.airlineManager.RMUsernameValidator();
        validationClient.addObserver(validationObserver, 'success');
        ko.validation.rules['uniqueUser'] = {
            async: true,
            validator: function (val, otherVal, callback) {
                validationClient.url = validationUrl + '/' + val;
                validationObserver.callback = function (event, xhr, data) {
                    callback(data.data.isAvailable);
                }
                validationClient.send();
            },
            message: 'Username already taken. Try an other one.'
        };
        jqcls.ko.validation.registerExtenders();
        jqcls.Class.define('ns.airlineManager.RegistrationModel', {
            extend: jqcls.ns.mvc.model.BackboneModel,
            abstract: false,
            validations: {
                firstLastName: {
                    required: true
                },
```

```
                username: {
                    required: true,
                    uniqueUser: true
                },
                password: {
                    required: true,
                    mustEqual: 'passwordConfirmation'
                },
                passwordConfirmation: {
                    required: true,
                    mustEqual: 'password'
                },
                email: {
                    email: true,
                    mustEqual: 'emailConfirmation'
                },
                emailConfirmation:{
                    email: true,
                    mustEqual: 'email'
                },
                captcha:{
                    required: true
                }
            },
            constructor: function () {
                var me = this;
                me.config = {};
                me.config.attributes = {
                    firstLastName: "Abcd",
                    username: "",
                    password: "",
                    passwordConfirmation: "",
                    email: "",
                    emailConfirmation: "",
                    captcha: ""
                };
                me.options = {
                    url: './users/register.json'
                };
jqcls.ns.airlineManager.RegistrationModel.prototype.parent.constructor.apply(
      me, [me.config.attributes, me.options]);
            }
        });
        return jqcls.ns.airlineManager.RegistrationModel;
    });
```

## Anexa 8 : Utilizarea grilei jqGrid în jqCls

```
define( ["js/Application"],
    function (Application) {
        "use strict";
        var jqgridData = [], obj, names = ['Cristina', 'Stefan', 'Ion',
'Roxana', 'Mihai', 'Andrei', 'Oana', 'Florin'], index;
        jqgridData = [
            {"col1": "Stefan", "col2": "Ioan"},
            {"col1": "Cristina", "col2": "Anca"},
            {"col1": "Andrei", "col2": "Gheorghe"},
```

```
            {"col1": "Mihai", "col2": "Bogdan"}
        ];
        jqcls.Class.define('ns.MyApp', {
            extend: jqcls.ns.Application,
            abstract: false,
            constructor: function (config) {
                var me = this;
                me.parent.constructor.apply(me, [config]);
            },
            start: function () {
                var screen = new jqcls.ns.components.Screen({
                    components: [
                        new jqcls.ns.components.DesktopManager({
                            components: [
                                new jqcls.ns.components.Window({
                                    title: '',
                                    width: 300,
                                    height: 200,
                                    components: [
                                          new jqcls.ns.components.Grid({
                            region: jqcls.ns.util.layout.BorderLayout.CENTER,
                                        jqGrid: {
                                            width: 200,
                                            datatype: 'local',
                                            data: jqgridData,
                                            colNames: ['Nume', 'Prenume'],
                                            colModel: [
                                        {name: 'col1', index: 'col1', width: 55},
                                        {name: 'col2', index: 'col2', width: 90}
                                            ],
                                            sortname: 'col1',
                                            sortorder: 'desc',
                                          caption: 'Exemplu grilă'
                                        }
})]})]})]});}});
        return new jqcls.ns.MyApp().start();
    });
```

## Anexa 9 : Utilizarea interfeţei MDI

```
define(
    ['./js/Application'],
    function (Application) {
        "use strict";
        jqcls.Class.define('ns.MyApp', {
            extend: jqcls.ns.Application,
            abstract: false,
            constructor: function (config) {
                var me = this;
            jqcls.ns.MyApp.prototype.parent.constructor.apply(me, [config]);
            },
            start: function () {
                var screen = new jqcls.ns.components.Screen({
                    components: [
                        new jqcls.ns.components.DesktopManager({
                            hasTaskbar: true,
```

```
                            components: [
                                new jqcls.ns.components.Window({
                                    title: 'Window 3',
                                    components: [],
                                    width: 800,
                                    height: 500
                                }),
                                new jqcls.ns.components.Window({
                                    title: 'Window 4',
                                    width: 300,
                                    height: 200
                                }),
                                new jqcls.ns.components.Window({
                                    title: 'Window 5',
                                    width: 300,
                                    height: 200
                                })
                            ]
                        })
                    ]
                });}});
        new jqcls.ns.MyApp({}).start();
});
```

## Anexa 10 Suita de testare automată *Array List Test*

### Index.html

```
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" href="qunit-1.12.0.css">
    <script type="text/javascript" src="../js/lib/require.js" data-
main="TestApp.js"></script>
    <script type="text/javascript" src="./qunit-1.12.0.js"></script>
    <title>jqCls unit testing</title>
</head>
<body>
    <div id="qunit"></div>
</body>
</html>
```

### TestApp.js

```
define(
    ['../js/Application', './ArrayListTest'],
    function (Application, qunit) {
        "use strict";
        console.log("Depine MyApp", qunit, test);
        jqcls.Class.define('ns.MyApp', {
            extend: jqcls.ns.Application,
            abstract: false,
            constructor: function (config) {
                var me = this;
                console.log("Call app constructor");
```

```
                jqcls.ns.MyApp.prototype.parent.constructor.apply(me,
                  [config]);
            },
            start: function () {
                var test = new jqcls.ns.test.ArrayListTest();
                test.run();
            }
        });
        new jqcls.ns.MyApp({}).start();
    }
);
```

### ArrayListTest.js

```
define( ['../js/util/list/main'],
    function (list) {
        jqcls.Class.define('ns.test.ArrayListTest',{
            input: [0,1,2,3,4,5,6,7,8,9,0],
            iterations: 0,
            run: function () {
                var me = this;
                this.list = new list.ArrayList();
                test("Array List Test", function () {
                    me.testAdd();
                    equal(me.list.items[0], 0);
                    me.testAddAll();
                    equal(me.list.size(), 12);
                    me.testClear();
                    equal(me.list.items.length, 0);
                    me.testEach();
                    equal(me.iterations, 11);
                    deepEqual(me.eachOutput, me.input);
                    equal(me.testGet(), 0);
                    me.testSet();
                    equal(me.list.get(0), 99);
                    equal(me.list.isEmpty(), false);
                    me.list.clear();
                    equal(me.list.isEmpty(), true);
                    equal(me.testIndexOf(1024), -1);
                    equal(me.testIndexOf(0), -1);
                    equal(me.testIndexOf(0, true), 0);
                    equal(me.testLastIndexOf(1024), -1);
                    equal(me.testLastIndexOf(0, true), 21);
                    deepEqual(me.testRemove(1), {
                        initial: 22,
                        length: 21
                    });
                    deepEqual(me.list.items,
                        [0,2,3,4,5,6,7,8,9,0,0,1,2,3,4,5,6,7,8,9,0]);
                    deepEqual(me.testRemove(1024), {
                        initial: 21,
                        length: 21
                    });
                    deepEqual(me.list.items,
                        [0,2,3,4,5,6,7,8,9,0,0,1,2,3,4,5,6,7,8,9,0]);
                    me.testRemoveAll(0);
```

```
                    deepEqual(me.list.items,
                        [2,3,4,5,6,7,8,9,1,2,3,4,5,6,7,8,9]);
                    equal(me.testSize(), me.list.items.length);
                    deepEqual(me.list.subList(0,1), [2]);
                    deepEqual(me.list.toArray(), me.list.items);
                })
            },
            testAdd: function () {this.list.add(0);},
            testAddAll: function () {this.list.addAll(this.input);},
            testClear: function () {this.list.clear();},
            testEach: function () {
                var me = this;
                this.iterations = 0;
                this.eachOutput = [];
                this.list.addAll(this.input);
                this.list.each(function (i, v) {
                    me.iterations = me.iterations + 1;
                    me.eachOutput.push(v);
                });
            },
            testGet: function () {return this.list.get(0);},
            testSet: function () {this.list.set(99,0); },
            testLastIndexOf: function (item, init) {
                init = init || false;
                if (init === true) {
                    this.list.addAll(this.input);
                }

                return this.list.lastIndexOf(item);
            },
            testIndexOf: function (item, init) {
                init = init || false;
                if (init === true) {
                    this.list.addAll(this.input);
                }

                return this.list.indexOf(item);
            },
            testRemove: function (item) {
                var initialLength = this.list.size();
                this.list.remove(item);
                var length = this.list.size();
                return {
                    "initial": initialLength,
                    "length": length
                }
            },
            testRemoveAll: function (item) {this.list.removeAll(item);},
            testSize: function () {return this.list.size();}
        });
        return jqcls.ns.test.ArrayListTest;
    });
```