Framework pentru dezvoltarea jocurilor de strategie

Constantin Ioan Nandra
Departamentul Calculatoare
Universitatea Tehnică din Cluj-Napoca
nandracosmin@yahoo.com

Dorian Gorgan
Departamentul Calculatoare
Universitatea Tehnică din Cluj-Napoca
Dorian.gorgan@cs.utcluj.ro

REZUMAT

Scopul acestui articol este de a propune și prezenta un model arhitectural care ar putea sta la baza dezvoltării jocurilor de strategie și, în general, a aplicațiilor care implică simularea interacțiunilor dintre un set de entități într-un mediu dat. Modelul propus constă dintr-un pachet de componente software care definesc arhitectura și mai multe seturi de componente interschimbabile care pot fi folosite direct în dezvoltarea potențialelor aplicații. Acest model are potențialul de a oferi ca alternativă la interacțiunea bazată pe invocarea funcționalității, caracteristică utilizării bibliotecilor software, un mod de interacțiune care oferă posibilitatea programatorului de a construi aplicația ca pe un agregat format din componente existente. În cadrul articolului vor fi scoase în evidență punctele forte ale acestei proiectări (flexibilitate, extensibilitate, modularizarea modelului), și se vor prezenta și demonstra beneficiile utilizării sale în dezvoltarea jocurilor de strategie.

Cuvinte cheie

Model arhitectural, proiectare modulară, componente reutilizabile, interacțiune alternativă.

Clasificare ACM

H5: Information interfaces and presentation (e.g., HCI), H.5.2: User Interfaces, H.5.m: Miscellaneous; D.1.5: Object-oriented programming

INTRODUCERE

Inițial apărute ca mijloace de recreere pentru programatori sau simple manifestări ale unor *hobby*-uri, astăzi, jocurile video reprezintă o industrie cu venituri anuale globale de 70 de miliarde de dolari, înregistrând cea mai semnificativă creștere din sectorul media(1). Această creștere se datorează, în mare măsură, apariției microprocesorului, computerului personal și creșterii exponențiale a puterii de calcul din ultimele decenii (2) (3). Un alt rol semnificativ în cadrul acestui fenomen de creștere a fost jucat de evoluția metodei de dezvoltare a acestor tipuri de software.

Este binecunoscut faptul că jocurile video reprezintă o categorie de produse software care tind să solicite mai multe resurse de calcul decât aplicațiile obișnuite, mai ales atunci când este vorba de simulări în timp real sau în cazul în care calitatea graficii joacă un rol important. Acesta este, probabil, unul dintre motivele pentru care în faza incipientă a industriei, aplicațiile de acest tip erau proiectate și implementate special pentru a folosi în mod optim resursele hardware ale platformelor – resurse grafice, constrângeri de memorie, etc (4). Această primă abordare presupune, desigur, o implicare în dezvoltarea tuturor componentelor și a eventualelor nivele de abstracție ale aplicației, de la nivelul managementului resurselor, până la nivelul dezvoltării componentelor abstracte ale aplicației. Evident, abordarea vine cu serioase dezavantaje. În primul rând, trebuie luat în considerare timpul necesar dezvoltării unei astfel de aplicații în integralitate. Un alt neajuns este reprezentat de posibilitatea destul de redusă a reutilizării codului în dezvoltarea unor aplicații similare.

Pentru a răspunde acestor neajunsuri, următorul pas avea să ducă la apariția unor sisteme specializate menite să îmbunătățească procesul de dezvoltare. Apărute, în general în cadrul companiilor mari, acestea încercau să ofere soluții software cât mai generale, seturi de componente reutilizabile precum și medii care să faciliteze procesul de dezvoltare. Aceste sisteme, denumite ulterior "Game engines" (aprox. "Motoare de jocuri"), au crescut în popularitate și număr ca urmare a introducerii elementelor de grafică 3D la începutul anilor '90 (5).

În zilele noastre, majoritatea oferă facilități ca: motoare de randare, detecția coliziunilor, inteligență artificială, sunet, management la nivelul *thread*-urilor și al memoriei etc (6). Toate aceste elemente aduc semnificative îmbunătățiri procesului de dezvoltare a jocurilor pentru PC și diverse alte platforme, promovând reutilizarea componentelor și oferind posibilitatea de adaptare a motorului de jocuri în scopul dezvoltării de noi aplicații.

OBIECTIVE

Acest articol își propune să prezinte un *framework*, o colecție de componente software care să faciliteze dezvoltarea jocurilor de strategie, asemănător motoarelor de jocuri menționate anterior. Departe de a fi o soluție pentru uz comercial sau de a furniza toate facilitățile considerate standard pentru acest tip de sisteme, modelul propus reprezintă un exemplu menit să ajute în explorarea și înțelegerea tipului de probleme pe care acest tip de software încearcă să le rezolve. Modelul arhitectural ce urmează a fi prezentat își propune să fie fundamentul unui sistem care să furnizeze două tipuri principale de facilități. În primul rând, va oferi o soluție generală care va încerca să cuprindă funcționalitatea comună tuturor jocurilor de strategie, oferind, în același timp, posibilitatea de extensie, de a crea componente specializate. Posibilitatea de adaptare a acestei soluții generale la cerințele utilizatorului reprezintă una dintre cele mai importante cerințe de proiectare, întrucât va permite simplificarea procesului de

dezvoltare a aplicațiilor în termeni de timp și cost. O altă cerință, la fel de importantă, o reprezintă implementarea soluției generale într-un mod cât mai transparent cu putință. Utilizatorul (dezvoltatorul de aplicații) nu trebuie să cunoască detalii despre structura internă sau implementare pentru a putea folosi această soluție. În acest fel este redus timpul necesar procesului de dezvoltare, dar și complexitatea soluției finale.

În al doilea rând, se urmărește proiectarea unei colecții de componente interschimbabile bazate pe acest model arhitectural. Această colecție va fi parte integrată a soluției ce urmează a fi furnizată utilizatorilor, și va promova în mod activ principiul de reutilizare a componentelor. Se urmărește pe această cale furnizarea unui tip de funcționalități adresată în principal dezvoltatorilor de aplicații. Este vorba despre o aplicație care, cu ajutorul unei interfețe simple și intuitive, să ofere posibilitatea de a crea specificația unui joc folosind seturi de componente existente, interschimbabile. În mod evident, acest tip de funcționalitate are potențialul de a reduce drastic timpul necesar și complexitatea dezvoltării de noi aplicații, datorită utilizării unor componente software extensiv testate și documentate.

Produsul finit se adresează, pe de o parte, programatorilor, oferind în acest sens un *framework* menit să ușureze considerabil efortul tipic necesar dezvoltării aplicațiilor de tipul jocurilor de strategie, iar pe de altă parte designerilor de jocuri, permițându-le acestora să realizeze prototipuri de aplicații în mod accesibil și rapid. Această posibilitate va fi oferită prin introducerea unui mod alternativ de interacțiune care va permite utilizatorului, în cazul de față designerului, să specifice structura modelului aplicației într-o formă grafică, ușor accesibilă și intuitivă, folosind în acest sens o colecție de componente implementate și testate ce vor fi livrate împreună cu framework-ul propriu-zis.

Construcția aplicației utilizând componente existente este un mod alternativ de interacțiune menit a fi folosit în conjuncție cu modul tradițional de interacțiune dintre *framework* și programator. Combinarea acestor două moduri de interacțiune va oferi un produs ce are potențialul de a permite realizarea atât a prototipurilor, cât și a aplicațiilor propriu-zise.

Implementarea *framework*-ului se va face utilizând tehnologia Java, datorită, în mare parte, portabilității și flexibilității soluțiilor oferite.

Cu ajutorul produsului finit se vor putea dezvolta jocuri de strategie, și, în general orice simulări care implică interacțiuni între colecții de entități suportate în cadrul unui mediu. Jocurile, în general, se vor limita la partide unu la unu între doi utilizatori, sau la interacțiuni relativ simple cu entități controlate de către computer. Va exista, desigur, posibilitatea extinderii în sensul adăugării unei inteligențe artificiale mai avansate, însă acest aspect nu va fi discutat în detaliu în cadrul articolului.

STUDIU BIBLIOGRAFIC

Conceptul motorului de jocuri

Dacă la începuturile industriei dezvoltarea jocurilor video implica lucrul cu resurse *hardware* și *software* specializate, produsele fiind adesea tratate ca simple jucării, în zilele noastre această activitate este la baza unei industrii de miliarde de dolari ce rivalizează *Hollywood*-ul în termeni de anvergură și popularitate (7).

Această dezvoltare nu ar fi fost posibilă fără apariția celebrelor motoare de jocuri *Quake* și *Doom* de la *id Software,* sau a *Unreal Engine* produs de *Epic Games*, care sunt, de fapt, kituri reutilizabile pentru dezvoltarea de *software* ce pot fi licențiate și utilizate pentru crearea a aproape oricărui tip de joc imaginabil (7) .

Tendința dezvoltării de astfel de kituri a prins contur odată cu apariția celebrului *Doom* la mijlocul anilor '90 (8). Joc video ce permite explorarea unui mediu prin perspectiva protagonistului, *Doom* aducea ca și noutăți pentru vremea respectivă o grafică 3D realistă și o arhitectură care separă funcționalitatea de bază a jocului (grafica 3D, detecția coliziunilor) de elementele specifice (reguli de joc, texturi, modele, etc.) (5) (7).

În zilele noastre, aceeași separare dintre funcționalitatea generală a unui joc și conținutul său specific stă la baza dezvoltării motoarelor de jocuri moderne. Prin reutilizarea acelei părți generale a funcționalității, timpul și costul necesare dezvoltării de noi aplicații sunt semnificativ reduse.

Soluții existente

La ora actuală pe piață se găsesc o varietate de sisteme care oferă o gamă variată de facilități pentru procesul de dezvoltare a jocurilor. De la motoare grafice, sisteme fizice și facilități de *scripting*, la inteligență artificială și *networking*, motoarele de jocuri stau la baza dezvoltării tuturor aplicațiilor majore de acest tip.

Există în acest moment o varietate de sisteme, de la celebrele *Quake și Unreal* a căror utilizare necesită costuri destul de ridicate, la motoare dezvoltate și utilizate intern de către marii producători din industria jocurilor video (9).

Topul motoarelor licențiate și utilizate pentru dezvoltarea de jocuri video de către părți terțe include nume sonore, ca *Anvil* folosit în dezvoltarea celebrelor francize *Assassin's Creed* și *Prince of Persia, RAGE,* cunoscut pentru contribuția adusă cunoscutului titlu *Grand Theft Auto (GTA),* sau *Unreal Engine,* unul dintre cele mai cunoscute și licențiate motoare de jocuri de pe piață (9).

Datorită evoluției actuale a jocurilor video în direcția fotorealismului, majoritatea motoarelor de jocuri sunt concentrate în jurul motoarelor grafice. În general, motoarele existente pe piață urmăresc să ofere funcționalitate orientată pe categorii de discipline.

Proiectul propus în cadrul acestui articol se axează pe îmbunătățirea procesului de dezvoltare mai degrabă decât pe furnizarea de funcționalitate specializată. Astfel, spre deosebire de majoritatea soluțiilor existente pe piață, proiectul propus pune accentul pe furnizarea unei baze de

componente *software* care să ofere sprijin în procesul de modelare a domeniului aplicației.

SOLUȚIA PROPUSĂ

Modelul arhitectural

Modelul propus în cadrul prezentului articol este bazat pe şablonul arhitectural cunoscut ca *"Model-View-Controller"* (MVC, aprox. Model – Prezentare - Controlor), un tipar care promovează separarea diferitelor tipuri de funcționalități (ex: logica modelului software față de cea responsabilă cu reprezentarea grafică). În acest mod, sunt promovate modularizarea proiectului şi posibilitatea de reutilizare a componentelor, acestea fiind două caracteristici de bază ale unei proiectări robuste.

Componentele descrise în această prezentare generală sunt proiectate pentru a fi folosite pe post de şabloane. Scopul lor este de a captura acea funcționalitate care ar putea fi specifică tuturor componentelor de un anumit tip și de a da utilizatorului posibilitatea creării propriilor componente, cu funcționalitate specifică nevoilor sale, fără a fi nevoit să cunoască detalii legate de structura internă și implementarea componentelor de bază. Modelul va fi unul extensibil și flexibil, componente noi vor putea fi create, cu funcționalitate nouă adăugată, implementare diferită, dar compatibile cu cele vechi și interschimbabile. Motivul pentru care specificația acestui model constă dintr-un set de şabloane interdependente este, pe lângă promovarea extensibilității și reutilizării componentelor, modularizarea proiectului, proprietate care va permite crearea aplicațiilor folosind seturi de componente existente, livrate împreună cu *framework*-ul, sau chiar create și adăugate de către utilizator.

O schemă generală a proiectului propus este prezentată în Figura 1. După cum se poate observa în figură, modelul constă din şase elemente majore:

1. Joc – abstracție care stă la baza arhitecturii, are în general rolul de a oferi o interfață care să faciliteze accesul la model şi la prezentare către componentele din exterior. Este un agregat care are rolul de a controla accesul către componentele sale și de a păstra consistența între stările acestora.
2. Controlor – este o componentă cu rol de comandă și control. Comunicarea cu modelul jocului se realizează prin interfața expusă de componenta Joc. Poate consta dintr-un automat cu număr de stări finit (ASF). În general, în funcție de starea internă, de comenzile externe, și de starea modelului, va accesa diferite părți ale funcționalității expuse de către acesta.
3. Mediu – reprezintă componenta cu caracteristicile cele mai apropiate de Modelul din arhitectura MVC. Acesta este nucleul sistemului, iar ca structură, este un agregat constând dintr-un set de componente numite Entități care pot interacționa, pot influența Mediul sau pot fi influențate de către acesta. Desigur, starea internă a acestei componente, pe lângă caracteristicile specifice, constă din suma stărilor tuturor Entităților ce intră în alcătuirea sa.
4. Prezentare mediu – această componentă este responsabilă cu reprezentarea grafică a stării interne a Mediului. La fel ca și componenta pe care o reprezintă, este alcătuită ca un agregat, de data aceasta, de Prezentări (componente responsabile cu reprezentarea Entităților).
5. Entitate – reprezentare a obiectelor ce poate fi acomodată de către Mediu.
6. Prezentare – componentă ce are ca rol reprezentarea grafică a stării interne aparținând unei Entități. Între aceste două componente există o corespondență de unu la unu.

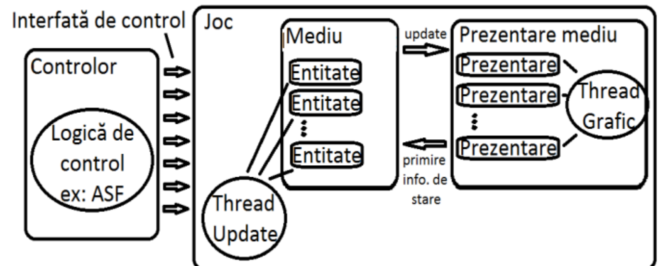

Figura 1. Prezentare generală a modelului arhitectural

Mecanisme interne

În continuare, vom examina modelul propus în detaliu, referindu-ne la subansamblul reprezentat de Mediu, Entități și interacțiunile dintre acestea.

O Entitate este compusă din patru elemente de bază, după cum se observă în Figura 2. În primul rând, în componența ei intră o structură cu funcționalitate de coadă, folosită pe post de container pentru comenzile primite ce urmează a fi executate. În orice moment dat, Entitatea poate avea cel mult o comandă curentă, comandă care se află în execuție. În momentul în care Entitatea primește comenzi noi, aceste comenzi vor fi stocate în coadă, și vor fi livrate spre procesare (execuție) în ordinea sosirii.

Prezentarea constituie componenta responsabilă cu reprezentarea grafică a Entității. Odată ce o Entitate este creată, Prezentarea acesteia este livrată componentei View (prezentarea mediului). Fiecare Entitate este responsabilă cu crearea propriei Prezentări.

Componenta numită Strategie funcționează ca o componentă de control (asemănător unui creier primitiv) a Entității. Are rolul de a controla comportamentul acesteia în cazul în care trebuie să acționeze singură. Strategia intră în acțiune doar în cazul în care nu există instrucțiuni implicite care să necesite execuție (coada pentru comenzi este goală și nu există comandă curentă). Modul de funcționare al Strategiei constă în trimiterea de comenzi către Entitate, în funcție de parametrii ca starea internă a acesteia, starea Mediului, sau a altor Entități cu care este nevoită să interacționeze. La fel ca și în cazul Prezentării, crearea Strategiei este o sarcină a Entității.

Toate interacțiunile care au loc în cadrul Mediului, sunt realizate prin intermediul componentelor de tip Comandă. Această componentă are rolul și structura asemănătoare cu ale componentei Comandă definite în specificația modelului de proiectare (*design pattern*) cu același nume. Rolul componentei este de a încapsula o serie de instrucțiuni spre a fi executate ulterior. Un alt rol, probabil la fel de important, este acela de de a captura serii de

instrucțiuni și a le cataloga în funcție de scop, acestea devenind astfel accesibile și ușor de refolosit.

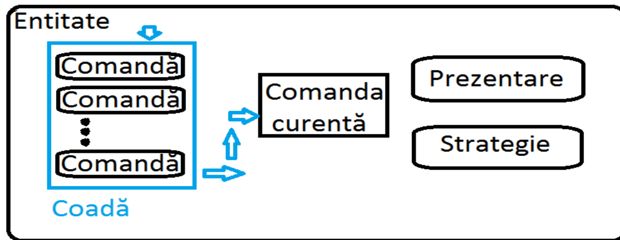

Figura 2. Prezentare generală a componentei Entitate

Beneficiile utilizării conceptului de transformare a acțiunilor în componente ale sistemului nu se opresc aici. Utilizând un consacrat model de proiectare numit "Compozit" (*Composite*), putem promova în mod activ reutilizarea componentelor și adăuga flexibilitate modelului. Un exemplu în acest sens, poate fi observat în Figura 3. Atât Mișcare, cât și Atac, sunt componente de tip comandă, cu funcționalitate specializată, specifică scopului pentru care au fost create. Din acest punct de vedere, sunt similare cu componenta ComandăCompusă. Însă această componentă este un agregat format din alte componente de tip Comandă. Componentele respective pot fi Mișcare, Atac, sau orice altă componentă specializată existentă sau definită ulterior. Ca agregat, funcționalitatea acestei componente se va rezuma pur și simplu la apelarea funcționalității constituenților. Acest artificiu de proiectare promovează reutilizarea componentelor, întrucât un agregat poate fi format din orice componente existente. Ba mai mult, se pot folosi și componente definite cu mult după ce conceptul de ComandăCompusă a fost proiectat și implementat, fără a fi nevoie ca această componentă să mai sufere vreo modificare. Se dovedește astfel flexibilitatea proiectării, dar și caracterul extensibil al său.

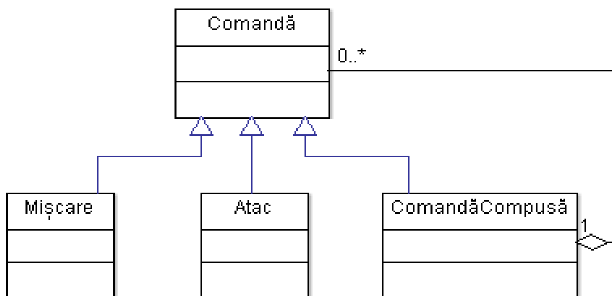

Figura 3. Exemplu pentru folosirea modelului de proiectare "Compozit"

Mod de funcționare

Structura generală a unui joc, indiferent de tipul său este reprezentată de o buclă infinită în care se execută codul ce actualizează starea modelului și cel responsabil de reprezentarea grafică. În cazul acestei arhitecturi, am ales separarea celor două din motive ce țin de frecvența cu care fiecare parte de cod ar trebui executată. Această separare este realizată utilizând thread-uri diferite.

Bucla de execuție a modelului acestei arhitecturi execută la infinit (până la primirea unui semnal din exterior; ex: pauză, stop) iterații ale Mediului. O iterație a componentei Mediu este un agregat compus din iterațiile setului de Entități componente, câte o iterație per componentă, executate în ordine (ordinea este prestabilită, spre exemplu, ordinea în care au fost adăugate componentele). Coborând încă un nivel, o iterație a fiecărei Entități constă din verificarea Comenzii curente (în cazul în care este finalizată, este înlocuită cu prima Comandă din coadă). Dacă, după verificare, încă mai există o acțiune curentă validă se execută o iterație a acesteia.

Structura internă și funcționarea Comenzii este crucială funcționării acestui mecanism. Aceasta componentă, pe lângă faptul că stochează instrucțiuni pentru execuția ulterioară și permite crearea de tipare comportamentale definite ca liste de comenzi dependente de stările interne ale Mediului sau Entității, prin proiectarea sa permite execuția comenzilor unui set de Entități fără a bloca execuția uneia până la finalizarea celei anterioare. Comanda menține o stare internă care poate fi afectată de fiecare iterație. De obicei, setul de instrucțiuni care trebuie executat de o comandă este de natură iterativă. Spre exemplu, o acțiune de mișcare în timp real constă din deplasarea Entității între locații adiacente până când se ajunge la destinație; un atac poate consta din scăderea în mod repetat a unei cantități de puncte dintr-un total, etc. Componenta Comandă este creată pentru a profita de această natură iterativă a acțiunilor. O iterație a componentei va corespunde, în funcție de necesitățile proiectului, uneia sau mai multor iterații din cadrul secvenței de instrucțiuni. Starea internă a componentei va reflecta progresul acțiunii după fiecare iterație. Se pot defini astfel metode de interogare pentru a afla dacă respectiva comandă este sau nu finalizată. Pentru tratarea excepțiilor, se va folosi o componentă specializată, o Comandă ce execută o singură iterație.

Această deciziei de a conferi componentelor Comandă natură iterativă, a fost luată, în principal, pentru a elimina blocajul care intervine în execuția comenzilor unei entități atunci când o alta, plasată înaintea primei entități, are de executat propria comandă. O decizie alternativă, cu rezultate mult mai slabe, ar fi fost plasarea execuției comenzilor fiecărei entități în propriul thread. Este ușor de anticipat faptul că această decizie ar fi dus la supraîncărcarea sistemului. Chiar și relativ modestele cerințe de a simula comportamentul a câtorva zeci de entități ar fi presupus crearea unui număr de tot atâtea thread-uri. O astfel de abordare nu ar fi fost practică, întrucât sistemele desktop obișnuite nu sunt potrivite pentru managementul unui număr mare de *thread*-uri.

IMPLEMENTARE

Prezentare generală

În cele ce urmează, vom prezenta implementarea modelului propus anterior, explicând modul în care am folosit facilitățile tehnologiei alese pentru a defini, accentua și pune în valoare acele caracteristici în general acceptate ca fiind specifice unei proiectări robuste și durabil.

În Figura 4 este prezentată o diagramă de clase a modelului propus, în care sunt reprezentate, în linii mari, dependențele dintre componente.

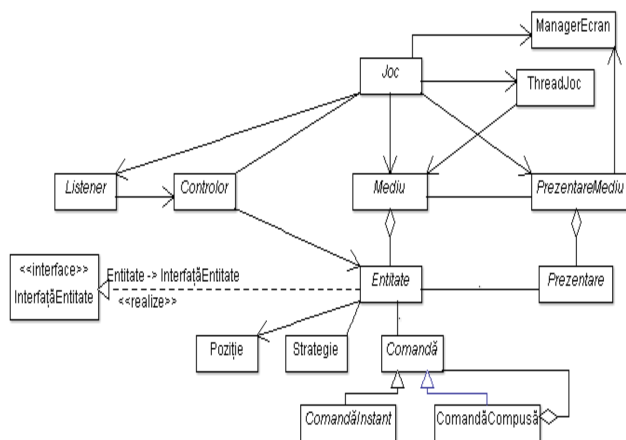

Figura 4. Diagrama de clase a modelului

Caracteristica principală, în cazul implementării modelului propus, este utilizarea claselor abstracte. Java oferă posibilitatea utilizării acestei noțiuni pentru a defini componente ce nu pot fi instanțiate. Această facilitate ar putea fi cel mai bine utilizată pentru a surprinde un set maximal de caracteristici care sunt comune tuturor componentelor de un anumit tip. Vom profita de mecanismul de moștenire pentru a insufla aceste caracteristici (componente și funcționalitate) atunci când vom defini componente specializate. O alternativă ar fi folosirea interfețelor ca substitut pentru clasele abstracte. Neajunsul acestei metode este că nu am reuși să ne folosim de implementări ale caracteristicilor comune. Prin natura lor, interfețele doar definesc, specifică funcționalitatea pe care obiectele ar trebui sa o implementeze. O interfața poate conține doar șablonul unei funcții (pe care conform convențiilor o vom numi în continuare metodă), nu și implementarea sa. Un alt beneficiu al folosirii claselor abstracte îl constituie posibilitatea de a forța definirea unor constructori care să conțină argumentele necesare inițializării potențialelor componente interne (pe care altfel utilizatorii nu le-ar putea recunoaște). Scopul este de a livra o colecție de clase care să permită utilizatorilor să le folosească in procesul de dezvoltare a aplicațiilor fără a avea cunoștințe despre structura lor internă.

ÎMBUNĂTĂȚIREA FLEXIBILITĂȚII

Identificarea problemei

După cum am precizat anterior, în implementarea modelului am folosit clase abstracte.

Rolul claselor abstracte este de a defini tiparul după care va fi construită fiecare componentă a sistemului. Prin definirea constructorilor se pot specifica dependințele dintre aceste componente. Spre exemplu, plasarea unei referințe de tip Entitate printre argumentele constructorului clasei Prezentare creează o relație de dependență între acestea. Definirea acestor relații cu ajutorul claselor abstracte le conferă un caracter general. Constructorul unei clase derivate din Prezentare poate primi ca argument orice tip de Entitate existent, și chiar tipuri definite ulterior. Un alt avantaj devine evident atunci când se dorește crearea unei Prezentări pentru o Entitate cu funcționalitate specială. Această funcționalitate (cerută de implementarea Prezentării) nu poate fi accesată dacă obiectul de tip Entitate este trimis cu o referință de tip Entitate, referința trebuie să corespundă tipului care implementează acea funcționalitate. Java permite redefinirea constructorilor unei sub-clase astfel încât aceștia să utilizeze sub-tipuri ale argumentelor definite inițial în super-clasă. Mai exact, se permite restrângerea domeniului de tipuri compatibile cu argumentele inițiale. În cazul exemplului precedent, o Prezentare specializată poate să specifice în cadrul constructorului său sub-tipuri ale tipului Entitate pentru a putea avea acces la funcționalitatea suplimentară expusă de către acestea.

Modelul are ca scop principal oferirea posibilității de a dezvolta aplicații bazate pe arhitectura prezentată folosind seturi de componente existente extensibile și interschimbabile Raportat la acest scop, avantajul dat de restrângerea tipurilor dependențelor compatibile cu clasele specializate se transformă într-un neajuns. Pentru a înțelege mai bine problema, să presupunem existența unui tip de componentă Comandă C1. În continuare, vom presupune că execuția acestui tip de comandă necesită un obiect de tip specializat de Entitate care să expună funcționalitatea compusă din metodele: a(), b() și c(), tip pe care îl vom numi E1. După același model, vom presupune existența altor două clase specializate: C2 și E2. C2 are nevoie ca entitatea pe care o controlează să expună funcționalitatea compusă din metodele: d() și e(), oferită doar de clasa specializată E2. Pentru a putea avea acces la funcționalitatea specificată, C1 și C2 trebuie să declare în cadrul constructorilor argumente de tip E1, respectiv E2. Acum devine destul de clar faptul că o componentă de tip C2 nu poate fi folosită în combinație cu una de tip E1, natura constructorului disponibil pentru C2 va împiedica orice încercare de combinare a celor două. Acesta este un exemplu reprezentativ pentru întregul model, întrucât dependențele dintre componente urmează același tipar. Dacă privim lucrurile din punctul de vedere al capacității acestui model de a suporta crearea și utilizarea unor componente complet interschimbabile care să poată fi folosite cu succes în dezvoltarea aplicațiilor, intervine o problemă: dependențele sunt prea rigide, componentele par a fi dezvoltate pentru niște nevoi prea specifice.

Soluția propusă

O soluție posibilă pentru problema rigidității componentelor ar putea fi găsită folosind conceptul de interfață, așa cum este definit și oferit în Java, pentru a decupla dependențele rigide dintre componente. Conceptul de interfață oferă posibilitatea de a crea specificația unei funcționalități fără a fi nevoie să o cuplăm cu o implementare. Un alt beneficiu major al utilizării interfețelor, este posibilitatea de atribuire a unui număr nelimitat de tipuri unei implementări date.

În cazul modelului prezentat anterior, utilizarea interfețelor în locul implementărilor pentru a specifica dependențele dintre componente ar putea fi un răspuns la problema cuplării excesive dintre componente.

Soluția propusă presupune, în primul rând, specificarea componentele interschimbabile din cadrul arhitecturii ale căror modificări de interfețe (funcționalitate adăugată) pot influența deciziile privind interfețele și/sau implementările

altor componente. Componentele interschimbabile din cadrul modelului sunt: Joc, Controlor, Mediu, PrezentareMediu, Entitate, Strategie și Comandă (inclusiv sub-clasele). Restul componentelor sunt fie auxiliare, fie folosite în cadrul aplicațiilor transparent față de utilizator. Dintre aceste componente interschimbabile, cele ale căror modificări de interfață (funcționalitate adăugată sub-claselor) pot influența alte componente, sunt:

- Joc – componentă de nivel înalt ce definește funcționalitatea modelului aplicației, funcționalitate de care depinde implementarea componentei Controlor.
- Mediu – poate avea funcționalitate variată, de care depind interfața și implementarea componentei Joc și implementare componentei PrezentareMediu.
- PrezentareMediu – funcționalitate variată ce poate influența interfața componentei Joc.
- Entitate – poate influența implementările componentelor: Strategie, Prezentare și Comandă.

Acestea vor fi numite, în continuare, componente critice. Celelalte componente sunt folosite de către sistem transparent față de utilizator, iar schimbările lor de interfață nu afectează alte componente.

Pentru cazul componentelor critice se propune utilizarea unor interfețe de bază care vor fi folosite pentru a defini funcționalitatea expusă de clasele abstracte ce le definesc. Cu alte cuvinte, clasele abstracte ce definesc cele patru componente identificate anterior vor deveni sub-tipuri ale unor tipuri definite cu ajutorul interfețelor. Toate referințele către aceste tipuri de componente din cadrul modelului vor fi înlocuite cu referințe de tipuri definite cu ajutorul interfețelor.

Pentru fiecare componentă specializată, de tipul celor critice, care va fi proiectată pentru a face parte din setul de componente interschimbabile folosite pentru dezvoltarea aplicațiilor, se va furniza câte o interfață folosită pentru a-i defini funcționalitatea. Interfața, în mod obligatoriu, va extinde, fie interfața de bază a componentei, fie una din sub-interfețele acesteia. Este recomandat ca fiecare clasă folosită pentru implementarea funcționalității să extindă clasa abstractă corespunzătoare interfeței alese pentru a beneficia de funcționalitatea comună. În cazul implementării componentelor specializate, referințele către fiecare componentă critică (referințele sunt de tipul specificat cu ajutorul interfețelor de bază) vor fi înlocuite cu referințe de tipul specificat cu ajutorul interfeței furnizate odată cu acea componentă. Cu alte cuvinte, referința de tip general aflată în semnătura constructorilor sau metodelor clasei abstracte, poate fi înlocuită cu o referință de tip specific în cadrul claselor specializate. Diferența față de implementarea simplă, în care am folosit doar clase abstracte, este că de această dată tipurile referințelor sunt definite cu ajutorul interfețelor. Componentele nu mai nu mai depind de implementări concrete, locul acestora fiind luat de interfețe.

Să considerăm un exemplu în acest sens. Entitate, fiind una dintre componentele critice, este o implementare a interfeței de bază numită InterfațăEntitate. Aceasta definește funcționalitatea pe care clasa abstractă, Entitate, o va implementa (parțial). În cazul în care am dori să dezvoltăm o entitate specializată care urmează să fie adăugată la setul de componente furnizat de *framework* (ce va fi folosit pentru dezvoltarea de aplicații), trebuie, în primul rând, să definim funcționalitatea acesteia cu ajutorul unei interfețe (o vom numi I1). Această interfață trebuie să extindă interfața InterfațăEntitate. Clasa care urmează să fie folosită pentru a implementa funcționalitatea trebuie să implementeze I1. În acest fel se garantează statutul de entitate al noii clase (deoarece implementează funcționalitatea definită de InterfațăEntitate), și în același timp primește statut de entitate specializată (implementează funcționalitatea definită de I1, care extinde InterfațăEntitate). Se recomandă ca noua implementare să extindă clasa abstractă Entitate pentru a beneficia de implementarea existentă a funcționalității comune. Astfel, noua clasă nu va fi nevoită să livreze implementări pentru toate metodele definite de către InterfațăEntitate, ci va moșteni implementarea clasei Entitate.

Mecanismul descris anterior oferă o soluție simplă și elegantă pentru problema incompatibilității dintre componente. Să presupunem că avem la dispoziție două componente specializate de tip Entitate, E1 și E2, care implementează interfețele I1, respectiv I2. Pe lângă acestea, mai dispunem și de o componentă de tip Comandă, C1, care este compatibilă doar cu entitatea E1 (C1 are definit un constructor ce primește ca argument o referință de tip I1). În cazul în care utilizatorul ar dori ca și comanda C1 să fie disponibilă pentru entitatea de tip E2, ar trebui să creeze o nouă clasă care să extindă E2 și, în plus, să implementeze I1. Noua clasă, E12 va moșteni integral implementarea lui E2 (un obiect de tip E12 va fi, practic, un obiect de tip E2) și va fi obligată să implementeze funcționalitatea descrisă de I1, E12 câștigând astfel posibilitatea de a putea folosi comanda de tip C1. Acest exemplu poate fi ușor generalizat, iar soluția, extinsă pentru a rezolva problemele de compatibilitate între alte tipuri de componente

MOD DE UTILIZARE

Să presupunem că un programator sau proiectant de aplicații are la dispoziție un set vast de componente proiectate și implementate după modelul descris în cadrul acestui articol, Pentru a construi o aplicație după modelul arhitectural descris în cadrul articolului, acesta ar trebui să urmeze o secvență de pași:

- Crearea unei componente de tip Joc
- Crearea unei componente Controlor specializată (sau alegerea uneia existente, caz în care componenta de tip Joc va implementa aceeași interfață care specifică tipul componentei Joc de care are nevoie componenta Controlor aleasă).
- Alegerea unei componente existente de tip Mediu.
- Se alege componenta PrezentareMediu. În cazul în care nu este compatibilă cu componenta de tip Mediu, este creată o sub-clasă a acestei componente ce implementează interfața care specifică tipului mediului din constructorul componentei PrezentareMediu.
- Se aleg tipurile de obiecte Entitate. Pentru fiecare entitate se aleg tipurile de Prezentare dorite. Acest

pas se realizează suprascriind metodele claselor specializate de tip entitate responsabile cu creare prezentărilor.

- Se aleg componente de tip Strategie și Comenzi disponibile pentru fiecare entitate. În cazul în care entitățile nu implementează toate interfețele cerute de comenzile și strategia aleasă pentru fiecare, se creează sub-clase care le vor implementa.

Acest proces poate fi automatizat, deși nu vom intra în detalii pe acest subiect, folosind facilități de scriere în fișier (pentru a crea noi clase) și reflexie (pentru a obține informații despre clase: funcționalitate, interfețe implementate, constructori, argumente, câmpuri interne etc.). Pe baza acestei secvențe de pași se pot genera schelete de aplicații formate din componente existente. Deși aceste aplicațiile nu vor fi complete, o mare parte din implementare va fi furnizată prin intermediul mecanismului de moștenire. Procesul de implementare va fi cu mult simplificat, întrucât programatorilor le va reveni relativ simpla sarcină de a furniza implementări pentru metodele schelet generate automat în procesul de rezolvare a incompatibilităților.

Acest mod alternativ de interacțiune dintre dezvoltatorul de aplicații și *framework* are potențialul de a reduce costul, timpul și complexitatea dezvoltării aplicațiilor. De asemenea, s-ar putea dovedi un mod de interacțiune potrivit pentru designeri, care lucrează la nivel conceptual, fără a avea de-a face cu implementarea aplicațiilor. O astfel de funcționalitate le-ar permite să genereze proiecte schelet, cu arhitectura și conceptele de nivel înalt integrate. Odată generate, aceste aplicații schelet se pot finaliza într-un timp considerabil mai scurt decât în cazul aplicațiilor obișnuite.

Combinând acest tip de interacțiune cu tipul clasic (utilizarea unui *framework*), se pot obține aplicații utilitare destinate dezvoltatorilor și proiectanților de jocuri deopotrivă, având potențialul de a reduce costul, timpul și resursele alocate unei activități clasice de dezvoltare a acestor tipuri de aplicații.

CONSTRUCȚIA UNEI APLICAȚII FOLOSIND MODUL ALTERNATIV DE INTERACȚIUNE

În continuare, vom prezenta construcția modelului unei simple aplicații utilizând componente cu caracter general, ce vor fi livrate împreună cu *framework*-ul propriu-zis. Este vorba despre o reprezentare virtuală a unui joc de șah care permite vizualizarea mutărilor posibile și garantează respectarea regulilor. Aceasta va oferi și posibilitatea de a salva și încărca de pe *hard-disk* starea internă.

Se urmărește exemplificarea pașilor enumerați anterior care descriu modul de interacțiune alternativ în procesul de creare a jocurilor.

Primul pas din această secvență de interacțiune presupune definirea unei clase de tip Joc (sub-clasă a clasei abstracte cu același nume). Fiind vorba despre un mod de interacțiune destinat în principal designerilor de jocuri, acest aspect (crearea unei clase) este ascuns utilizatorului. În schimb, se va utiliza o interfață grafică pentru a reprezenta structura conceptuală a aplicației. În acest sens, se va folosi o structură arborescentă, asemănătoare celei din Figura 5, întrucât aceasta poate reprezenta destul de bine dependențele dintre componente. Interfața grafică va prezenta într-o manieră accesibilă informațiile referitoare la configurația componentelor aplicației (componente alese, categorii de componente ce trebuie adăugate, etc.).

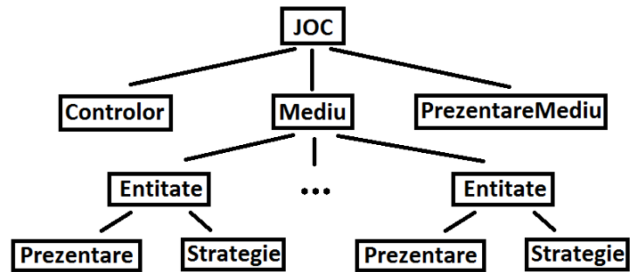

Figura 5. Reprezentarea conceptuală a unui joc bazat pe modelul arhitectural propus în cadrul articolului

Odată rădăcina fixată (specificarea numelui jocului), se va începe adăugarea elementelor de pe ramuri. Concret, trebuie selectate componente din categoriile Mediu, Prezentare Mediu și Controlor. În cazul de față, la fel ca în majoritatea cazurilor, se dorește crearea unei componente Controlor special concepută pentru noul joc. Asta nu oprește însă utilizatorul din a alege o componentă deja implementată în cadrul pachetului de *software* reutilizabil.

După crearea componentei de tip Joc, pe care o vom numi JocDeȘah și a celei de tip Controlor (ControlorȘah) , utilizatorul trebuie să aleagă componentele de tip Mediu, respectiv PrezentareMediu. Desigur, crearea unor componente noi în locul alegerii unora existente este întotdeauna o posibilitate.

În cazul nostru vom utiliza componenta MediuMatrice, un tip de Mediu cu dimensiuni reduse care oferă, printre altele, acces deplin la starea fiecărei poziții (ocupată sau nu de entități). În ceea ce privește prezentarea mediului, vom alege o componentă PrezentareȘablon care oferă utilizatorului posibilitatea de a defini scheme de culori sau imagini ce pot fi mapate pe pozițiile Mediului corespunzător.

Avansând un nivel mai jos, vom popula configurația mediului cu tipuri de entități ce pot fi susținute în cadrul acestuia. În acest scop vom crea un tip de Entitate nou, numit PiesăȘah. Specificația acestui tip nou de entitate va fi definită cu ajutorul componentelor pe care le va susține. O Entitate trebuie asociată obligatoriu cu o componentă Prezentare și un set de componente Comandă. Opțional, i se poate asocia și o Strategie. În cazul de față, noua Entitate va folosi componenta PrezentareSimplă furnizată în pachetul de componente reutilizabile. Această componentă oferă posibilitatea de a afișa imagini simple asociate entităților pe pozițiile pe care acestea le ocupă în cadrul mediului.

Comenzile pe care PiesăȘah le poate executa vor fi reprezentate de doua componente relativ simple, cu execuție instantanee: SchimbarePoziție și ÎnlocuireEntitate, ambele preluate din pachetul de componente reutilizabile.

Se poate observa în cadrul acestui exemplu cum, utilizând o serie de componente cu funcționalitate generală, livrate împreună cu *framework*-ul propriu-zis, se poate realiza o

specificație formală a structurii unui joc. Cea mai mare parte a modelului aplicației este construită pe baza acestei specificații.

În cadrul acestui exemplu, după cum se poate observa din Figura 6, cinci din cele opt componente de bază ale modelului aplicației sunt componente reutilizabile aparținând pachetelor ce vor fi furnizate utilizatorului odată cu implementarea modelului arhitectural. Procentajul componentelor reutilizabile din structura unei aplicații create prin intermediul acestui mod de interacțiune poate varia în funcție de complexitatea proiectului și de nevoile utilizatorului.

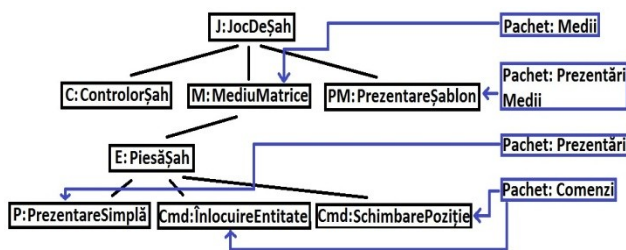

Figura 6. Reprezentarea conceptuală modelului noii aplicații

Specificația formală a aplicației va fi folosită pentru crearea unui proiect Java care să înglobeze componentele reutilizabile specificate și să le adapteze (după exemplul prezentat anterior, prin sub-clasare) în cazul în care există incompatibilități de interfață între acestea. Proiectul propriu-zis va consta doar din componentele create de către utilizator (Joc, Controller) și din clasele create implicit pentru adaptarea componentelor cu interfețe incompatibile. Componentele reutilizabile vor fi făcute disponibile prin includerea bibliotecilor corespunzătoare în contextul noului proiect.

Acest proces poate fi automatizat folosind facilități de scriere în fișier și tehnica de Reflexie oferită de Java pentru a examina structura internă și interfețele claselor existente. Se pot construi astfel proiecte constând din clase schelet și componente reutilizabile, care vor furniza un subset al funcționalității respectivelor proiecte. Prin furnizarea de implementări metodelor schelet, utilizatorul va obține aplicații funcționale într-un timp mai scurt și cu mai puțin efort decât în cazul proceselor normale de dezvoltare a aplicațiilor.

CONCLUZII

În cadrul articolului de față am prezentat un model arhitectural care oferă o serie de facilități pentru dezvoltarea jocurilor de strategie și, în general, a aplicațiilor care încearcă să simuleze interacțiunile dintre o populație de entități aflate în interiorul uni mediu dat. S-a demonstrat faptul că acest proiect este unul robust, flexibil și modular. Structura sa generală promovează reutilizarea componentelor software și permite crearea de componente noi, interschimbabile.

Modelul furnizează o soluție generală, oferind utilizatorului, în acest caz programatorului, posibilitatea de adaptare a proiectului la nevoile personale. Din acest punct de vedere, modelul este unul flexibil și promovează conceptul de reutilizare a componentelor software. Mai mult, se permite extinderea funcționalității existente, fapt ce reduce timpul necesar dezvoltării de noi componente.

Pe lângă faptul că a fost gândit și implementat pentru a conferi programatorilor un mecanism care să le permită dezvoltarea aplicațiilor bazate pe un model arhitectural prestabilit într-un mod complet transparent (funcționalitate tipică unui *framework*), acest model merge un pas mai departe, încercând să ofere proiectanților și programatorilor deopotrivă posibilitatea de a crea aplicații folosind exclusiv componente existente, livrate împreună cu acest *framework*.

Pentru a putea exploata la maxim natura reutilizabilă a modelului, și mai ales a setului de componente interschimbabile ce pot fi dezvoltate pe baza acestuia, s-ar putea dezvolta o aplicație care să permită un mod alternativ de dezvoltare a aplicațiilor. Această aplicație ar permite utilizatorului un tip de interacțiune alternativ în procesul de creare a jocurilor. Cu ajutorul unei interfețe intuitive, un utilizator fără cunoștințe de programare ar putea dezvolta un joc, la nivel conceptual, selectând componentele dorite dintr-un set furnizat împreună cu aplicația. Acest tip de interacțiune s-ar putea derula conform secvenței de pași descrisă anterior, și ar putea fi utilizat pentru a reduce drastic efortul de dezvoltare al aplicațiilor în termeni de resurse, cost și timp alocat.

REFERINŢE

1. **Reuters.** Factbox: A look at the $65 billion video games industry. *uk.reuters.com.* [Interactiv] 6 June 2011. [Citat: 14 June 2013.] http://uk.reuters.com/article/2011/06/06/us-videogames-factbox-idUKTRE75552I20110606.

2. **Zackariasson, P. și Wilson, T.L.** *The Video Game Industry: Formation, Present State, and Future.* New York : s.n., 2012.

3. Player 3 Stage 6: The Great Videogame Crash. 1999.

4. **Moore, Michael E. și Novak, Jeannie.** *Game Industry Career Guide.* s.l. : Delmar: Cengage Learning, 2010. ISBN 978-1-4283-7647-2.

5. **Lily, Paul.** Doom to Dunia: A Visual History of 3D Game Engines. *maximumpc.* [Interactiv] 2009. http://www.maximumpc.com/article/features/3d_game_engines.

6. **Ward, Jeff.** What is a Game Engine? *GameCareerGuide.com.* [Interactiv] 2008.

7. **Gregory, Jason.** *Game Engine Architecture.* Wellesley, Massachusetts : A K Peters, Ltd., 2009. ISBN 978-1-56881-413-1.

8. **Rihal, Dharamjit.** The History of First-Person Shooters. *msu.edu.* [Interactiv] 2007.

9. **Stead, Chris.** The 10 Best Game Engines of This Generation. *ign.com.* [Interactiv] 15 July 2009. http://www.ign.com/articles/2009/07/15/the-10-best-game-engines-of-this-generation.