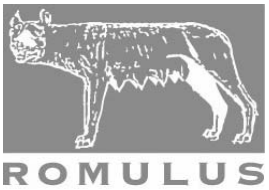
PROIECTARE ORIENTATĂ SPRE DOMENIU ŞI DEZVOLTARE ORIENTATĂ SPRE MASHUP BAZATE PE OPEN SOURCE JAVA METAFRAMEWORK PENTRU DEZVOLTAREA DE SOFTWARE WEB PRAGMATICĂ, FIABILĂ ŞI SIGURĂ:

METODOLOGIA ROMULUS PENTRU DEZVOLTAREA DE APLICAȚII OPEN SOURCE EXTENSIBILE



http://www.ict-romulus.eu, http://romulus.ici.ro

Rezumat: Această lucrare prezintă metodologia utilizată pentru elaborarea proiectului european Romulus. Scopul ei este acela de a descrie metodologia selectată şi lucrările efectuate pentru analiză, dezvoltare şi testare pentru a furniza o nouă versiune pentru Roma Meta-Framework şi noi plug-in-uri pentru Romulus.

Cuvinte Cheie: metaframework, Roma, plug-in, postare, implementare, portal, iterație, testare automată.

Abstract: The goal of this paper is to describe the methodology selected and the work made for the analysis, development and test to deliver Roma Meta Framework and the new Romulus plug-ins.

Keywords: metaframework, Roma, plug-in, post, implementation, portal, iteration, automat testing.

1. Introducere

Această lucrare are ca scop descrierea metodologiei utilizate şi a activităților efectuate pentru analiză, dezvoltare şi testare, pentru furnizarea unei noi versiuni pentru Roma Meta-Framework [RomaURL] versiunea 1.0.0 şi noul plug-in Romulus.

2. Metodologie

Această secțiune prezintă o metodologie cadru pentru Romulus. [Haw00] prevede că o metodologie unică nu poate lucra pentru întregul spectru de proiecte diferite; managementul de proiect identifică natura specifică a proiectului în speță şi, apoi, selectează metodologia de dezvoltare cea mai adecvată. Framework-ul Romulus are ca scop dezvoltarea metodologiei Agile pentru aplicații web.

Ca urmare, s-a utilizat o metodologie care se bazează pe următoarele principii:

- dezvoltare Agile. Agilitatea denotă "calitatea de a fi agil; pregătit pentru mişcare; agilitate, activitate, dexteritate în" mişcare, şi grupează un set de metodologii care încearcă să favorizeze comunicarea şi accelerarea dezvoltării;
- utilizarea instrumentelor de programare Open Source - Gratuit / Liber (Free / Libre);
- echipe mici-mijlocii de dezvoltare;
- dezvoltarea axată pe Domeniu (DDD - Domain Driven Development); principala tehnică de dezvoltare Romulus sugerează că, zona centrală care diferențiază aplicațiile, este concentrarea asupra domeniului.

3. Metodologiile Agile

În această secțiune se face o trecere în revistă a metodologiilor populare Agile [Abr02].

3.1 Extreme Programming

Extreme Programming (XP) [Bec99] este o metodologie Agile, cu idei şi practici deja trasate de către metodologiile existente. Aceasta este condusă de un set de practici principale şi concepte cheie:

- Planificarea jocului. Interacțiune strânsă între client şi programatori. Programatorii estimează efortul necesar pentru punerea în aplicare a "relatării privind sistemul actual" al clientului şi clientul decide apoi cu privire la domeniul de aplicare şi termenele de realizare.
- Realizări mici / scurte. Un sistem simplu este „produs" rapid - cel puțin o dată la două - trei luni. Noile versiuni sunt, apoi, actualizate zilnic sau cel puțin lunar.
- Metafora. Sistemul este definit de o metaforă / set de metafore între client şi programatori. Această poveste „comună" ghidează toată dezvoltarea prin descrierea modului în care funcționează sistemul.
- Proiectare simplă. Accentul se pune pe proiectarea celei mai simple soluții posibile, care este implementabilă în acest moment. Complexitatea inutilă şi codul suplimentar sunt eliminate imediat.
- Testarea. Dezvoltarea software-ului este axată pe testare. Testele sunt puse în aplicare înainte de scrierea codului şi sunt difuzate continuu. Clienții scriu testele funcționale.
- Refactorizarea. Restructurarea sistemului prin eliminarea duplicării, îmbunătățirea comunicării, simplificarea şi flexibilitatea adăugărilor.
- Asocierea de programare. Doi oameni scriu codul la calculator.
- Proprietate colectivă. Oricine poate modifica, în orice moment, orice parte a codului.
- Integrare continuă. O nouă piesă de cod este integrată în codul de bază, de îndată ce este gata. Astfel, sistemul este integrat şi construit de mai multe ori pe zi. Toate testele sunt rulate şi trebuie să fie trecute, astfel încât schimbările în cod să fie acceptate.
- 40 de ore pe săptămână. O săptămână are maxim 40 de ore de lucru. Nu sunt permise două săptămâni la rând, ore suplimentare. Dacă se întâmplă acest lucru, aceasta este tratată ca o problemă ce trebuie să fie rezolvată.
- Client. Clientul trebuie să fie prezent şi disponibil cu normă întreagă, pentru echipă.
- Standarde de codificare. Există reguli de codificare şi sunt urmate de programatori. Trebuie să fie subliniată comunicarea prin cod.
- Spațiu de lucru deschis. Este de preferat o cameră mare împărțită în compartimente mici. Programatori pereche ar trebui să fie plasați în centrul spațiului.
- Reguli. Echipa are propriile reguli care sunt urmate, dar care pot fi, de asemenea, schimbate în orice moment.

Modificările trebuie să fie convenite, în prealabil, şi impactul acestora trebuie să fie evaluat.

XP este destinată echipelor mici şi mijlocii şi ia în considerare, următoarii actori: programator, client, tester, depanator, antrenor, consultant şi manager (big boss).

Una dintre ideile fundamentale ale XP este că nu există nici un proces care se încadrează oricărui proiect ca atare, ci mai degrabă, practicile ar trebui să fie adaptate pentru a se potrivi nevoilor proiectelor individuale. Ca urmare, în conformitate cu principiile Agile, metodologia nu ar trebui să prevadă procese dificile care nu pot fi adaptate sau urmărite.

Ciclul de viață al XP (figura 1) constă din cinci faze: explorare, planificare, repetări la lansare, producție, întreținere şi, în final, dispariția produsului:

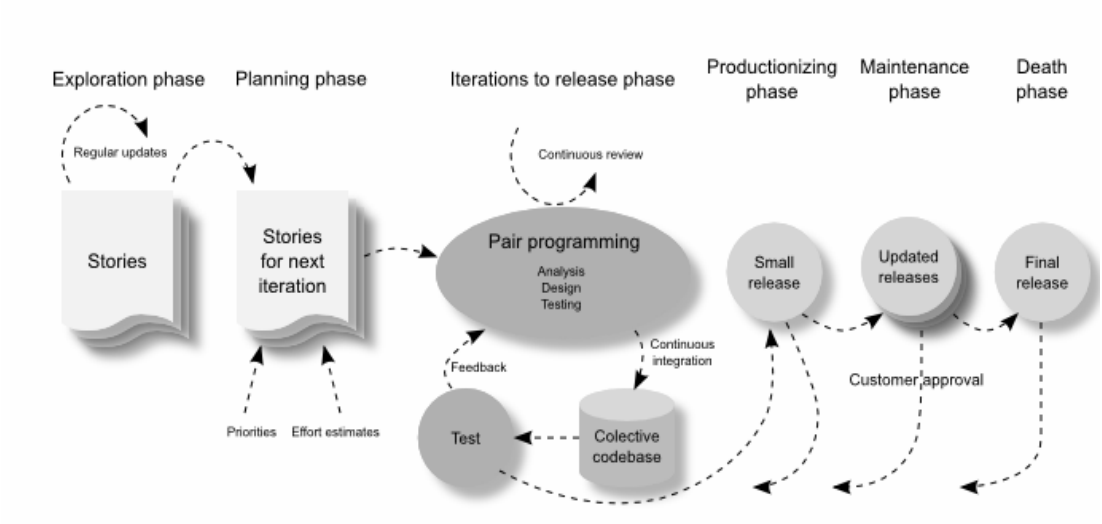


Figura 1. Extreme Programming - ciclul de viață

- Faza de planificare: se realizează un acord de conținut pentru prima variantă; sunt realizate, de asemenea: o estimare cu privire la efortul pentru fiecare caracteristică şi un calendar. Durata: câteva de zile.
- Lansări repetate de versiuni: include mai multe iterații ale sistemelor, înainte de prima lansare. Programul stabilit în etapa de planificare este defalcat la un număr de iterații. Testele funcționale create de client sunt rulate la sfârşitul fiecărei iterații. Durata: fiecare repetare va dura de la una, la patru săptămâni.
- Faza de producție: necesită teste suplimentare înainte ca sistemul să fie lansat. Ideile şi sugestiile: amânate, sunt documentate pentru punerea în aplicare ulterioară, în timpul fazei de întreținere. Durata: o săptămână pentru fiecare repetare.
- Faza de întreținere: după ce prima versiune este lansată pentru utilizarea clientului, conform XP, proiectul trebuie să păstreze sistemul în funcțiune în timp ce se produc, de asemenea, iterații noi.
- Faza de dispariție: atunci când clientul nu mai are nimic pus în aplicare.

3.2. Dezvoltare de software Open Source

Open Source Software (OSS) paradigmă [Fel00] sugerează că codul sursă ar trebui să fie disponibil pentru distribuire şi modificare gratuite, fără nici o taxă. Principiile de bază ale dezvoltării OSS sunt:

- Sistemele sunt construite de către un număr mare de voluntari.
- Sarcinile nu sunt atribuite; oamenii îşi aleg ei înşişi sarcinile care-i interesează.
- Nu există nici un sistem explicit la nivel de proiectare.
- Nu există nici un plan de proiect, calendar sau listă de rezultate.
- Sistemul este completat prin mici creşteri (incrementări).
- Programele sunt testate frecvent.

Principalii actori în dezvoltarea OSS sunt lideri de proiect, dezvoltatorii voluntari, raportorii de bug-uri şi creatorii de blog-uri. Din dezvoltarea OSS lipsesc multe dintre mecanismele tradiționale de coordonare a dezvoltării de software. În schimb, se impune libertatea de

contribuție, adesea ghidată de către şeful de proiect care încearcă să echilibreze obiectivele proiectului cu dorințele comunității pentru a concentra contribuțiile dezvoltatorului într-o singură direcție. De obicei, un proiect OSS constă din următoarele faze vizibile [Sha02]:

1. Descoperirea problemei;
2. Găsirea de voluntari;
3. Identificarea soluției;
4. Dezvoltarea şi testarea de cod;
5. Revizuirea schimbării de cod;
6. Realizarea codului şi a documentației;
7. Lansarea management-ului.

3.3 Dezvoltare axată pe Caracteristică (Feature Driven Development)

Feature Driven Development (FDD) este o abordare flexibilă şi adaptabilă pentru dezvoltarea de sisteme software care acoperă numai proiectarea şi construirea fazelor de dezvoltare de software, deşi a fost conceput pentru a funcționa cu alte activități ale unui proiect de dezvoltare software. În ansamblu, FDD încearcă să ofere o metodologie iterativă, combinată cu cele mai bune practici dovedite a fi eficiente în industrie. FDD constă din cinci procese secvențiale (figura 2) în care se efectuează proiectarea şi construirea sistemului.

- Elaborarea unui model de ansamblu. În acest stadiu, cerințele sunt folosite pentru a dezvolta un model de ansamblu. Domeniul de ansamblu este construit prin împărțirea acestuia în subdomenii, în care obiectele şi relațiile dintre ele sunt modelate şi specificate de echipe diferite;
- Construirea unei liste de caracteristici. Sunt prezentate funcții evaluate de clienți pentru a stabili seturi de aşa-numita caracteristică majoră, care grupează mulțimile de caracteristici obținute din cerințe şi din modelul de ansamblu;
- Planul de caracteristică. Este creat un plan general, în care seturile de caracteristici sunt secvențiale. Seturile de caracteristici şi clasele sunt alocate programatorilor şefi şi, respectiv, proprietarilor de clase;
- Proiectare după caracteristică şi construire prin caracteristică. Un grup mic de caracteristici este selectat şi echipele de lucru sunt formate de către proprietarii de clasă. Caracteristicile sunt puse în aplicare iterativ printr-un set de activități prezentate în figura 3.

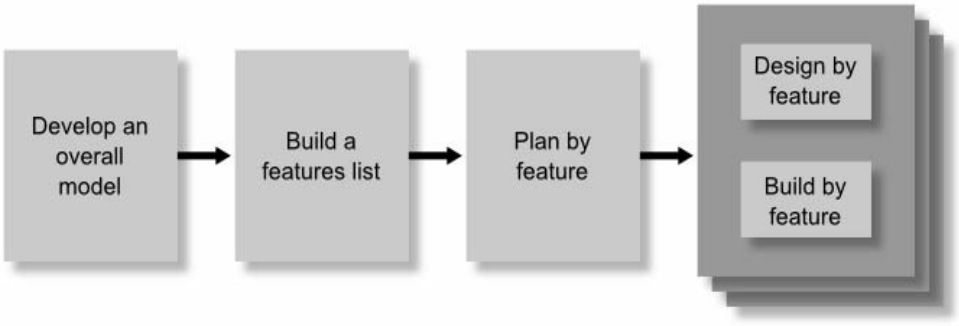


Figura 2. Procesele de dezvoltare pentru Feature Driven Development

După ce se realizează fiecare iterație, caracteristicile dezvoltate sunt salvate în "construcția" principală.

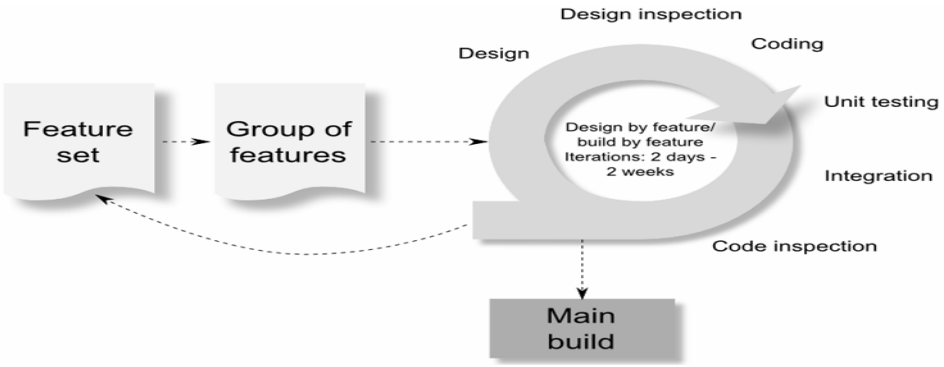


Figura 3. Impementarea (punerea în aplicare) a caracteristicilor iterative

Rolurile FDD sunt clasificate în rolurile principale (manager de proiect, arhitectul şef, director de dezvoltare, programator şef, proprietar de clasă şi experţi în domeniu), roluri suport (manager de versiune, avocat de limbaj / guru de limbaj, inginer „constructor", expert în instrumente şi administratorul de sistem) şi roluri suplimentare (testeri, „lansatori", scriitori tehnici). Un membru al echipei poate juca roluri multiple şi un rol unic poate fi partajat de mai multe persoane.

Această metodologie constă dintr-un set de bune practici:

- Modelare obiectelelor domeniului. Explorarea şi explicaţiile domeniului problemei. Rezultatul este un framework (cadru) în care se adaugă caracteristici.
- Dezvoltarea axată pe caracteristică. Dezvoltarea şi urmărirea progresului printr-o listă de funcţii descompuse din punct de vedere funcţional şi evaluate de client.
- Proprietatea Clasei individuale (Cod). Fiecare clasă are o singură persoană desemnată ca responsabilă pentru coerenţa, performanţa şi integritatea conceptuală a clasei.
- Caracteristicile echipelor. Se referă la echipe mici, formate dinamic.
- Inspecţie. Se referă la utilizarea mecanismelor cele mai cunoscute de detectare a defectelor.
- Construcţii obişnuite. Se referă la asigurarea faptului că există întotdeauna un sistem de rulare disponibil şi demonstrabil. În mod regulat se construieşte forma de bază, la care se adaugă caracteristici noi.
- Management de Configurare. Permite identificarea şi urmărirea istorică a celor mai recente versiuni ale fiecărui fişier complet de cod sursă.
- Raportarea Progresului. Progresul este raportat la toate nivelurile de organizare necesare, pe baza muncii complete.

3.4 Scrum

Scrum este o metodologie care vizează dezvoltarea generală a sistemelor. Ideea principală a metodologiei Scrum este aceea că variabilele de mediu, cum ar fi cerinţele, resursele şi tehnologiile, se pot schimba cu timpul, astfel încât este esenţial să se utilizeze un proces de dezvoltare care este în măsură să răspundă la schimbări. Scrum va încerca să realizeze acest lucru inclusiv prin includerea activităţilor de management ce au ca scop identificarea deficienţelor în procesul de dezvoltare [Ken06].

Scrum consideră următoarele roluri: master Scrum (utilizarea corespunzătoare a principiilor Scrum), proprietarul de produs, echipa de lucru, client şi manager. Procesul de dezvoltare constă în cele trei faze, după cum se arată în Figura 4.

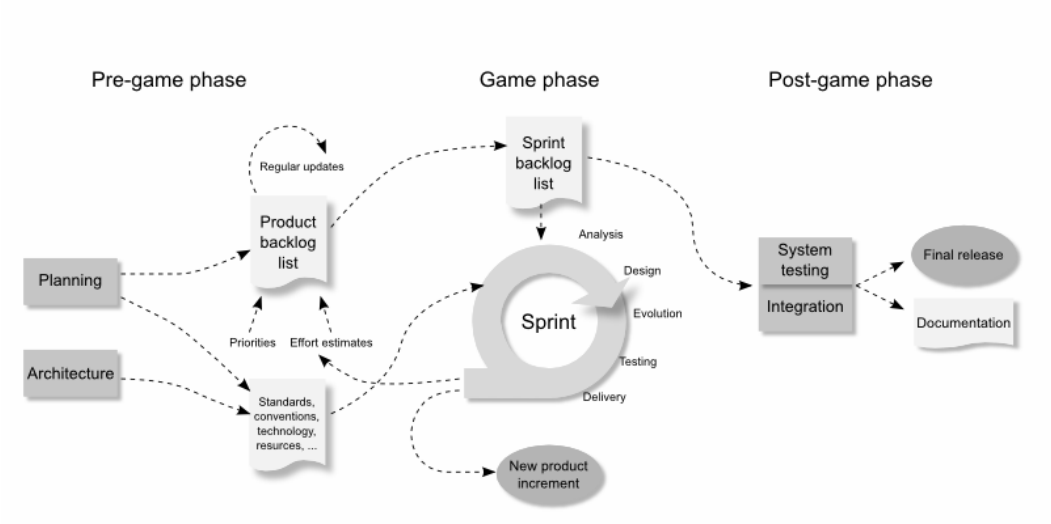


Figura 4. Faze conform metodologiei Scrum

3.5. Metoda de dezvoltare a sistemelor dinamice

Metoda de dezvoltare a sistemelor dinamice (DSDM) este un framework (cadru) gratuit, pentru dezvoltare de RAD-uri (Rapid Application Development - Dezvoltări rapide ale aplicaţiilor).

Ideea fundamentală a DSDM este următoarea: este de preferat să se stabilească timpul şi resursele necesare realizării unui produs şi apoi să se ajusteze, în consecinţă, funcţionalitatea în loc de stabilirea funcţionalităţii, iar apoi să se ajusteze timpul şi resursele necesare realizării unui produs având această funcţionalitate.

DSDM defineşte 15 roluri pentru utilizatori şi dezvoltatori. Cele mai dominante roluri sunt cele de: dezvoltator, coordonator tehnic, utilizator implicat în dezvoltare, utilizatorul vizionar (care ştie cel mai clar obiectivele proiectului) şi sponsorul executiv. DSDM constă din cinci faze:

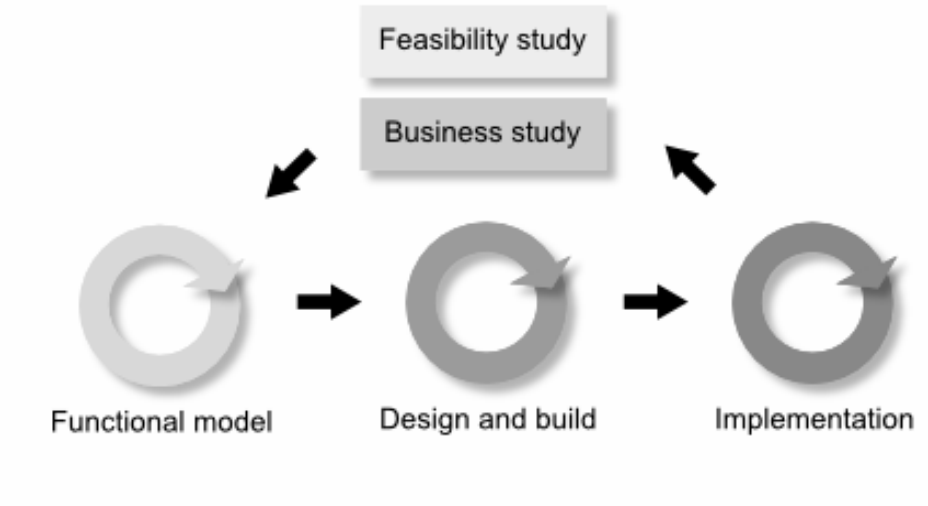


Figura 5. Fazele Metodei de Dezvoltare Dinamică a Sistemelor

Modelarea Agile

Modelarea Agile (AM) este o metodologie de dezvoltare axată pe practici şi principii culturale. Modelarea Agile propune cinci valori: comunicare, simplitate, reacţie, curaj şi „umilinţă".

Principiul de bază al modelării Agile constă în promovarea importanţei funcţionării

software-ului ca principal scop al modelării.

Din punctul de vedere general al dezvoltării de software, modelarea Agile nu este suficientă în sine. Deoarece aceasta acoperă doar modelarea, este nevoie de metode suport.

Programare pragmatică

Programarea pragmatică (PP) este un set de cele mai bune practici în programare, publicat în revista „The Pragmatic Programmer" (Programator pragmatic) de către Andrew Hunt şi Thomas David, ambii implicaţi în mişcarea Agile şi Manifestul Agile.

Familia de Metodologii Crystal

Familia de metodologii Crystal include o serie de metodologii diferite pentru alegerea metodei celei mai potrivite pentru fiecare proiect individual. Fiecare membru al familiei de Crystal (figura 6) este marcat cu o culoare care să indice „greutatea" metodologiei, adică o culoare mai închisă indică o metodologie mai complicată. Simbolurile caracter indică o pierdere potenţială cauzată de o eroare de sistem (adică nivelul critic, care poate fi Comfort -C, bani discreţionar -D, bani esenţial -E şi Life -L-), în timp ce numărul indică numărul maxim de persoane implicate în proiect.

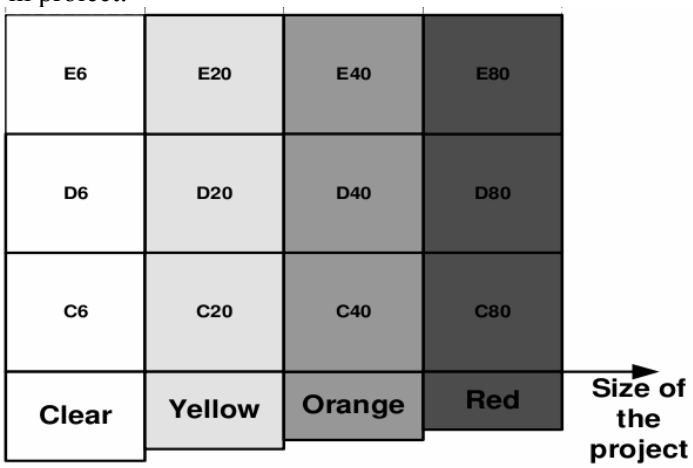


Figura 6. "Greutatea" metodologiei

Proiectele folosesc întotdeauna cicluri de dezvoltare incrementale, cu o lungime de creştere de maximum patru luni, dar de preferinţă între una şi trei luni. Accentul se pune pe comunicarea şi cooperarea între oameni. Metodologiile Crystal nu limitează practicile de dezvoltare, instrumentele sau produsele de lucru, permiţând în acelaşi timp, adaptarea la practicile Scrum. În prezent există trei metodologii principale Crystal: Crystal Clear, Crystal Orange şi Crystal Orange Web.

4. Metodologia utilizată în proiectul Romulus

Aşa cum s-a menţionat anterior, metodologia Romulus este menită să ofere o metodă de dezvoltare Agile pentru a construi aplicaţii web de către echipe mici / mijlocii prin utilizarea de instrumente Open Source Software OSS şi dezvoltarea axată pe domeniu.

Deşi metodologiile nu sunt legate de utilizarea tehnologiilor specifice, instrumentelor sau limbajelor, este nevoie ca acestea să fie adaptate pentru utilizatorii lor ţintă şi pentru pregătirea acestora. Dezvoltatorii OSS sunt un grup ţintă mare. De multe ori, dezvoltatorii OSS sunt reticenţi să urmeze o metodologie, astfel încât, pentru a ajunge la acest grup de dezvoltatori este nevoie de un efort de comunicare mai mare.

Metodologia Romulus este inspirată din Extreme Programming, Programarea Pragmatică şi alte metodologii Agile; la acestea se adăugă abordarea dezvoltării axate pe domeniu (DDD -

Domain Driven Development) pentru proiectarea de software.

Principiile sale de bază privind echipa de elaborare a proiectului, sunt descrise în continuare:

- Programare pereche. Doi oameni scriu cod. Unul se concentrează pe detalierea codului, în timp ce celălalt revede codul şi se concentrează pe imaginea de ansamblu. Acest lucru duce la un cod de calitate mai bună şi un mediu de lucru mai dinamic. Perechile sunt modificate şi rolurile sunt schimbate frecvent, astfel încât comunicarea este continuă şi toată lumea ştie toate părţile sistemului.
- Dezvoltare iterativă (săptămânal). Sistemul este dezvoltat cu creşteri mici pentru a identifica riscurile şi problemele. O nouă piesă de cod este integrată în codul de bază, de îndată ce este gata. Toate modificările intervenite în timpul dezvoltării sunt reversibile. Astfel, sistemul este integrat şi construit de mai multe ori pe zi.
- Testare. Testarea este integrată în întregul ciclu de viaţă de dezvoltare. Calitatea software-ului este verificată prin testare în timpul fiecărei iteraţii (repetări). Toate testele trebuie să fie trecute pentru ca modificările de cod să fie acceptate. Scrierea de teste se face înainte de punerea în aplicare. În cele din urmă, se scrie un caz de testare automată pentru fiecare bug întâlnit.
- Livrările de software. Urmărirea progresului se bazează pe lansări de versiuni de software şi decizii majore mai degrabă decât pe documente scrise.

Metodologia proiectului Romulus constă din trei faze (figura 7), care sunt acoperite continuu în construirea iterativă a unui produs software:

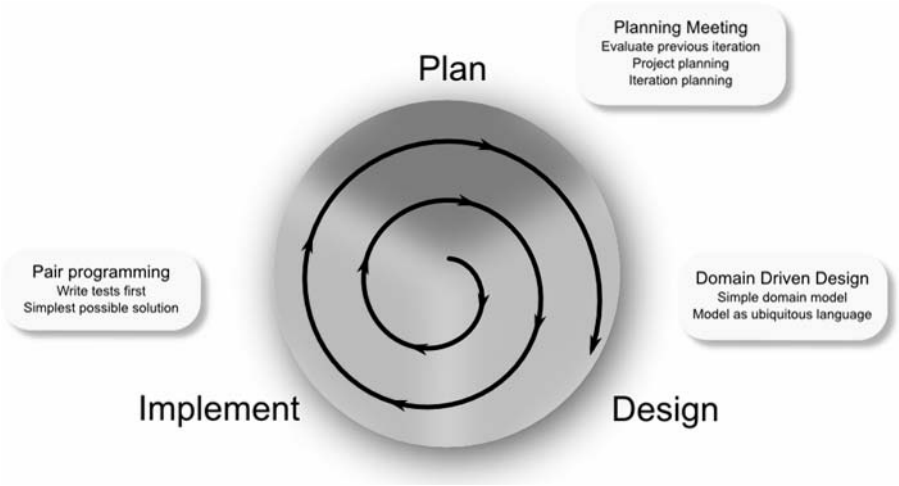


Figura 7. Fazele metodologiei Romulus

Plan

Faza de plan este inspirată din structura de planificare a metodologiei Extreme Programming. Faza plan pentru metodologia Romulus a constat în conferinţe de planificare în care au fost discutate proiectul, obiectivele repetitive şi sarcinile. Toţi membrii echipei, inclusiv utilizatorii, au fost prezenţi la reuniune. Întrucât faza de plan este prezentă în fiecare repetare, Reuniunea Planificare a avut loc săptămânal. Durata reuniunii a fost mai mare prima oară (în zile diferite), în cazul în care cerinţele utilizatorilor au fost incomplete şi instabile.

Proiectarea

De obicei, aplicaţiile prezintă funcţionalităţi similare, definiţia domeniului fiind una dintre sarcinile cele mai complexe de dezvoltat. În această metodologie, pentru a efectua această activitate este recomandată proiectarea axată pe domeniu. Proiectarea axată pe domeniu este o evoluţie a modelării domeniului orientată pe obiect, în care un set de modele de proiectare şi de

bune practici este definit în scopul de a facilita întreținerea software-ului şi procesul de testare.

Proiectarea axată pe domeniu se compune din trei etape:

- În primul rând, adoptarea unui model simplu al domeniului;
- În al doilea rând, utilizarea modelului domeniului ca limbaj de comunicare (numită limbaj omniprezent);
- Şi în al treilea, păstrarea, la zi, a modelului domeniului.

Modelul simplu al domeniului

Un model al domeniului reprezintă o parte a lumii reale cu care utilizatorii de software doresc să interacționeze. Un model al domeniului constă dintr-un set de entități fundamentale, precum şi relațiile dintre acestea. Acesta este doar un mecanism de a construi software-ul unde nu trebuie modelate toate detaliile.

Dar modelul nu este doar o schemă de date, aceasta este cheia pentru a rezolva o problemă complexă; modelele şi recomandările expuse în abordarea de proiectare bazată pe domeniu trebuie să fie luate în considerare pentru a obține un model simplu care, în acelaşi timp, colectează toate informațiile necesare pentru punerea lor în aplicare în aplicație. Modelul va evolua în timpul dezvoltării aplicației; împreună cu principiile Agile, simplitatea este o prioritate, iar acest lucru trebuie să fie păstrat până la sfârşitul proiectului.

În definirea modelului, utilizatorii şi dezvoltatorii trebuie să lucreze împreună pentru a obține cea mai bună soluție, studiind diferitele opinii şi scenarii de utilizare.

Un limbaj omniprezent

Modelul obținut stabileşte un limbaj comun. Scopul său este ameliorarea comunicării între dezvoltatori şi utilizatori. Construirea unui glosar de termeni pe baza modelului poate ajuta la folosirea acestuia. De asemenea, aceasta facilitează gestionarea eterogenității între paradigme diferite.

Deşi proiectarea orientată pe obiect este paradigma cea mai utilizată, în unele cazuri este necesar să se lucreze cu paradigme diferite, cum ar fi paradigme relaționale (folosirea de baze de date relaționale). Deoarece bazele de date gestionează persistența obiectelor, acestea sunt mai legate de modelul de obiecte decât alte componente, fiind necesar să se acorde o atenție deosebită paradigmei bazei de date.

Actualizarea modelului

Modelul domeniului va fi utilizat în toate fazele metodologiei: planificarea, proiectarea şi implementarea. În fiecare repetare (iterație) vor exista schimbări ale modelului şi, prin urmare, limbajul de comunicare va evolua.

În faza de plan, vor fi stabilite noile sarcini care trebuie dezvoltate şi acestea vor fi reflectate în model în timpul fazei de proiectare. Modelul trebuie să fie păstrat actualizat şi simplu, astfel încât în fiecare repetare, în faza de proiectare, se va face o revizuire în acest scop. În plus, în faza de punere în aplicare pot exista modificări asupra modelului şi acestea trebuie incluse.

Codul evoluează în timpul procesului de dezvoltare, deoarece sunt puse în aplicare noi funcționalități sau îmbunătățiri ce au fost realizate. Îmbunătățirea codului, păstrând în acelaşi timp comportamentul său, se numeşte Refactorizare. Scopul Refactorizării poate fi îmbunătățirea calității codului, care este, de obicei, atins la utilizarea de modele şi instrumente automate care se aplică acestor modele, sau pentru a îmbunătăți lizibilitatea şi inteligibilitatea codului, deoarece definirea modelului de domeniu devine mai clară.

D.1 - proiectare Inițială

În această activitate, se face proiectarea inițială. Este construită o diagramă inițială a claselor, bazată pe indicațiile utilizatorului, care reprezintă toate elementele necesare pentru sarcina privind progresele înregistrate. Mai multe instrumente pot fi folosite pentru a reprezenta această diagramă, dar în orice caz, utilizatorii trebuie să fie capabili să o înțeleagă şi să folosească termenii în model pentru a facilita, în viitor, comunicarea cu dezvoltatorii de iterații.

D.2 – Refactorizarea proiectării

În această activitate proiectarea inițială este refactorizată. După obținerea unui proiect inițial, modele DDD şi cele mai bune practici trebuie să fie aplicate pentru a îmbunătăți modelul domeniu, păstrând funcționalitatea inițială. În continuare, este prezentat un set de măsuri în scopul de a refactoriza modelul domeniului.

1. Analizarea asocierilor: O asociere stabileşte o relație între două sau mai multe obiecte. Modelul trebuie să fie cât mai simplu posibil, iar evitarea de asocieri complicate este un mecanism bun pentru a realiza acest lucru. Unele idei pentru simplificarea asocierilor sunt: impunerea unei direcții, adăugarea unui calificativ, reducerea multiplicității şi eliminarea de asocieri neesențiale. De exemplu, într-o aplicație blog există funcții postare (publicare) şi comentarii. Fiecare postare are mai multe comentarii şi fiecare comentariu aparține unei postări speciale. Putem reprezenta această asociere cum este descrisă sau putem simplifica prin impunerea unei direcții de la postări la comentarii şi o multiplicitate, atâta timp cât o postare poate avea zero sau mai multe comentarii.

2. Entitățile, obiectele cu valoare şi serviciile: Modelul trebuie să fie clar, făcând distincție între entități şi obiectele cu valoare. O entitate este un concept fundamental: este necesar un atribut unic de identificare, deoarece starea sa se poate schimba în timpul executării produsului software. Un obiect cu valoare descrie doar o caracteristică a domeniului. Un obiect cu valoare nu are identitate şi poate fi partajat. În unele cazuri, caracteristicile imuabile îmbunătățesc punerea în aplicare. În aplicația de blog, mesajele sunt entități, pentru că fiecare postare trebuie să fie unică; comentariile sunt obiecte cu valoare, acestea nu au identitate şi acelaşi comentariu poate apărea în două postări. În cele din urmă, într-un model de domeniu nu există servicii. Acestea reprezintă procese care nu sunt responsabile de orice entitate sau obiect cu valoare. Dar, uneori, serviciile sunt suprasolicitate şi funcțiile care corespund logicii de afaceri a unui obiect sunt puse în aplicare ca un serviciu. O bună practică pentru a evita această situație este aceea de a utiliza un verb ca nume de serviciu.

3. Utilizarea agregărilor: Agregarea constă dintr-un set de obiecte asociate, dar numai unul dintre acestea (rădăcina) poate fi referit de obiectele care sunt în afara agregării. Obiectele implicate într-o agregare acționează ca o unitate, facilitând gestionarea asocierilor complexe dintre obiecte. În domeniul unei aplicații de blog, postările şi comentariile formează o agregare în care postările sunt rădăcină şi controlează accesul la comentarii.

4. Selectarea arhivelor: Un alt şablon pentru a obține un model bun al domeniului este utilizarea arhivelor. Un depozit gestionează stocarea unui tip concret de obiecte. Aceasta implică punerea în aplicare a operațiunilor tipice pe o bază de date cum ar fi adăugarea, editarea şi eliminarea elementelor şi interogarea facilităților. Arhivele sunt: obiecte din domeniu asociate cu un agregat care gestionează stocarea datelor şi regăsirea lor, precum şi mecanisme de abstractizare a persistenței necesare pentru a efectua aceste operații. În domeniul blog-ului, de exemplu, vom folosi un depozit pentru a obține o postare sau o listă de postări stocate în baza de date. De obicei, fiecare framework (cadru) de dezvoltare web oferă o modalitate de a pune în aplicare persistența. Este important să se analizeze dacă framework-ul oferit este cel mai potrivit pentru aplicație sau dacă este cerută dezvoltarea unei soluții specifice.

5. Utilizarea „metodei fabricației” (Factory method)[1] pentru a crea obiecte: Un alt aspect ce se ia în considerare este crearea de obiecte şi posibilitatea de a utiliza „fabrici” în acest scop. O „fabrică” defineşte procesul de creare a unui obiect. Folosirea unei „fabrici” este utilă numai atunci când obiectul de creat este complex. În restul cazurilor, utilizarea acesteia poate complica procesul şi este mai bine să se utilizeze un simplu constructor. De exemplu, „crearea unei postări” este un proces uşor, aşa că se va folosi un constructor. Dar dacă vrem să verificăm dacă o postare există, în multe site-uri de la distanță, înainte de a o crea, este necesar să se utilizeze o „fabrică” pentru a ascunde această operație. Pentru un obiect cu valoare, „fabrica” nu reprezintă acelaşi lucru ca şi pentru o entitate. În primul caz, fabrica va defini complet procesul de creare, deoarece obiecte cu valoare sunt imuabile şi trebuie să fie pe deplin descrise încă de la crearea lor; starea unei entități se poate schimba în timpul executării software-ului şi fabrica sa trebuie să definească numai unele caracteristici ale acesteia. Mai ales, nu putem uita atributele sale de identificare.

6. Validarea modelului: În acest moment, este o bună practică pentru a continua cu securitatea utilizarea mai multor scenarii, pentru a confirma că modelul se potriveşte cerințelor. Trebuie să ne asigurăm că activitățile ce se dezvoltă în această repetare pot fi puse în aplicare.

7. Divizarea modelului în module: În dezvoltarea unei aplicații de întreprindere, este necesară colaborarea cu mai multe persoane care lucrează în paralel, fiind necesar să se împartă modelul de domeniu într-un set de module. Pentru a menține integritatea sistemului, fiecare modul trebuie să fie definit cu un context delimitat. Trebuie să existe o integrare continuă şi, de asemenea, este recomandabil să se reprezinte relațiile dintre modelele implicate în sistem, într-un context de hartă.

Implementare (Punerea în aplicare)

În faza de implementare, programatorii pereche vor scrie şi testa codul necesar pentru a executa sarcinile în care sunt implicați, în conformitate cu modelul domeniului definit în faza precedentă. Punerea în aplicare trebuie să fie simplă şi uşor de înțeles, de exemplu, oricine ar trebui să poată citi codul şi să înțeleagă cum este modelat şi ce funcționalitate este destinată să îndeplinească, în scopul de a facilita refactorizarea sa şi extinderea în iterații viitoare.

Paşii pentru a dezvolta modelul domeniului urmează metodologia prezentată în [Ric06]. Interfața este implementată cu ajutorul unei abordări Test Driven Development (TDD - Dezvoltare condusă de testare), în care testele sunt puse în aplicare înainte de scrierea codului, facilitând executarea lor după dezvoltarea de cod, pentru a se asigura că cerințele sunt îndeplinite cu respectarea sarcinii care este în curs de dezvoltare.

Activitățile care sunt efectuate în faza de implementare sunt prezentate în Figura 8.

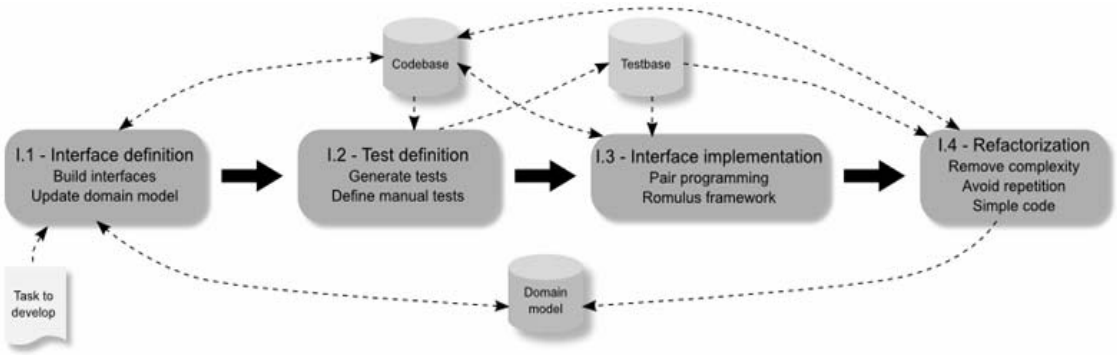


Figura 8. Activități în faza de implementare

[1] **Factory method** se referă la procesul de creare a obiectelor prin abstractizarea unui constructor (e tipic pt. Java). Există câte o metodă pentru fiecare tip de obiect care poate fi creat; în programare este cunoscut ca „factory method pattern”.

1 - Interfață - definire

În această activitate, proiectarea informațiilor este adaptată la tehnologia / punerea în aplicare specifică, respectiv, framework-ul Romulus şi sunt definite interfețele sau clasele vide. Intrarea pentru această activitate este modelul domeniului şi sarcina de implementare (punere în aplicare).

Interfața fiecărei clase este compusă din atribute şi definiții ale metodelor. Aceasta poate fi definită în două etape:

a) Clasele identificate, care corespund entităților şi obiectelor de valoare cu atributele lor şi relațiile, care sunt definite în modelul de domeniu.

b) Comportamentul se adaugă la modelul de domeniu prin definirea responsabilităților şi colaborărilor, în funcție de sarcina curentă în curs de dezvoltare. Responsabilitățile determină ce clase trebuie să facem şi sunt puse în aplicare prin una sau mai multe metode. Colaborările indică ce clase sunt solicitate de către o altă clasă pentru a realiza responsabilitățile sale.

2 - Definirea Testului

Cele mai multe cazuri de testare vor fi generate automat din clasa interfețe. Generatorul de test Romulus va folosi adnotări pentru a finaliza definirea testelor. În unele cazuri, testele nu pot fi pe deplin generate automat şi dezvoltatorii trebuie să le completeze cu funcționalitățile dorite.

În această activitate, ar trebui să fie definită o suită de teste care acoperă un număr rezonabil de cazuri. Aceasta ar trebui să includă, cel puțin, cazuri de testare generate automat şi cazuri de testare generate semi-automat, umplute manual.

3 - Implementarea (punerea în aplicare a) Interfeței

În această activitate, clasele sunt completate cu codul corespunzător, astfel încât suita de teste să se execută fără erori. Framework-ul Romulus permite dezvoltarea de aplicații Java. El este compus dintr-un set de Aspecte care gestionează caracteristicile necesare pentru a dezvolta aceste tipuri de aplicații, cum ar fi persistența sau vizualizarea. Fiecare Aspect este pus în aplicare prin unul sau mai multe module, permițând alegerea între diferite tehnologii care stau la bază.

Folosind Romulus, efortul de dezvoltare se axează pe punerea în aplicare a domeniului. Prin urmare, după definirea sarcinii necesare pentru punerea în aplicare a relatării utilizatorului şi de definire, în faza de proiectare, a modelului simplu de domeniu, acesta este momentul de punere în aplicare prin intermediul lui POJO (Plain Old Java Object). POJO-urile sunt clase simple Java care sunt axate pe probleme de afaceri. Folosind POJO, dezvoltarea este mai uşoară, mai rapidă şi independentă tehnologic.

Ca orice clasă Java, POJO-ul constă dintr-un set de atribute şi metode. Ca cele mai bune practici de proiectare, aceste atribute şi metode trebuie să reprezinte informații care sunt prezente în modelul domeniului; această abordare este utilizată în Romulus. Cu toate acestea, anumite informații necesare pentru a face să meargă aplicația nu sunt prezente în domeniu. Aceste informații se numesc preocupări „cross-cutting”[2]. În Romulus, preocupările „cross-cutting” sunt modelate prin Aspect-e. Aspect-ele reprezintă punctele de vedere independente de aplicație care afectează diferite unități logice sau de domeniu. Pentru a realiza acest lucru, POJO-urile au un set de adnotări Java asociate care sunt conforme cu metadatele claselor. Aceasta permite definirea fiecărui Aspect.

Modulele pun în aplicare unul sau diferite Aspect-e ale unei aplicații. Acest lucru permite Romulus să fie un metaframework: aplicațiile sunt definite într-un mod independent tehnologic

[2] **Cross-cutting** este o tehnică de editare, cel mai des folosită în filme, pentru a stabili acțiuni care apar în acelaşi timp în două locații diferite

în funcție de diferite Aspect-e. Diferite module pot fi conectate în intrare sau ieşire, mai târziu, şi arhitectura Romulus va permite ca implementarea aplicației să rămâne neatinsă. Exemple de module disponibile sunt Persistența-JPOX (care acoperă Aspect-ul persistence), View-Echo2 (view Aspect) sau modulul Users (authentication Aspect).

Pentru a rezuma, într-o aplicație Romulus conceptele de domeniu sunt definite prin POJO-uri. Adnotările sunt incluse în aceste POJO pentru a defini detaliile Aspect. În cele din urmă, modulele sunt selectate pentru a pune în aplicare aceste Aspect-e.

5. Concluzii

Această lucrare furnizează dezvoltatorului o metodologie integrată şi recomandări de bune practici pentru dezvoltarea de aplicații, într-un mod principial.

În plus, documentul prezintă noile module Roma, precum şi arhitectura noului plugin. Ca rezultat, metaframework–ul Roma oferă un set complet şi flexibil de funcționalități pentru dezvoltarea rapidă a proiectelor.

Datorită activității desfăşurate în Romulus, au fost dezvoltate, în conformitate cu funcționalități definite de cerințele Romulus, unele noi module adăugate, datorită componentei Roma, la arhitectura sa expandabilă de plug-in-uri, cum ar fi View Aspect, Semantic Aspect, module de scripting server side, logging sau flow management.

BIBLIOGRAFIE

[RomaURL] Roma Meta Framework project, http://www.romaframework.org

[FreURL] Freemarker project, http://freemarker.org/

[FrmURL] FreeMarker manual, http://freemarker.org/docs/index.html

[Abr02] Abrahamsson, P., Salo, O., Ronkainen, J. and Warsta, J. Agile software development methods. VTT Publications, 2002.

[Amb02] Ambler, S. Agile Modeling: Effective Practices for Extreme Programming and the Unified Process. New York, John Wiley & Sons, Inc., 2002.

[Aqu08] Dell'Aquila L., Garulli L., and Maestro G., Roma <Meta> Framework Handbook, 2008.

[Bec99] Beck, K. Extreme programming explained: Embrace change. Reading, Mass., Addison-Wesley, 1999.

[Cro02] Crowston, K. and Scozzi, B. Open source software projects as virtual organisations: competency rallying for software development. IEE Proceedings – Software 149(1): 3–17, 2002.

[Fel00] Feller, J. and Fitzgerald, B. A Framework Analysis of the Open Source Software Development Paradigm. 21st Annual International Conference on Information Systems, Brisbane, Australia, 2000.

[Haw00] Hawrysh, S. and Ruprecht, J. Light Methodologies: It's Like Déjà Vu All Over Again. Cutter IT Journal. 13: 4–12, 2000.

[Ken06] Ken Schwaber, 2006, SCRUM Development Process, Advanced Development Methods, http://jeffsutherland.com/oopsla/schwapub.pdf

[Kru00] Kruchten, P. The Rational Unified Process: an Introduction. Addison-Wesley, 2000.

[Mar98] Martin, R. C. The Process. Object Oriented Analysis and Design with Applications. Addison Wesley: 93–108, 1998.

[Ric06] Richardson, C. POJOs in Action, Developing Enterprise Applications with Lightweight Frameworks. Manning Publications, 2006.