eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

UNIBERTSITATEKO MASTERRA:

TELEKOMUNIKAZIO INGENIARITZA UNIBERTSITATE MASTERRA

MASTER AMAIERAKO LANA

SOFTWARE-ENPRESA BATEN PRODUKZIOA GARAPEN-ARKITEKTURA JAKIN BATERA EGOKITZEKO INGURUNE BATEN DISEINUA ETA INPLEMENTAZIOA

Ikaslea *IRAZU, ECEIZA, IKER*
Zuzendaria *HUARTE, ARRAYAGO, MAIDER*
Saila *TELEMATIKA SAILA*
Ikasturtea *2017 / 2018*

Bilbo, 14, uztaila, 2018

Laburpena

Software aplikazioak garatzeak kudeaketa lan bat eskatzen du. Garatzen diren aplikazioen kantitatea geroz eta handiagoa izan, orduan eta handiagoa izaten da aplikazio horiek guztiak kudeatzeko egin beharreko lana. Kudeaketa hori errazteko, *software* aplikazioak garatzen dituzten enpresek garapen-arkitektura bat erabiltzea ohikoa da. Baina aldi berean, arkitektura hori betetzen dela ziurtatzeak lan gehiago ematen du. Proiektu honetan, aplikazioen garapena errazten duen garapen-ingurune bat diseinatu da, garapenerako erabiltzen den arkitekturan definitutako arauak betetzeaz arduratzen dena. Honi esker, programatzaileak aplikazioak garatzeaz arduratu beharko dira, arkitektura betetzen dela garapen-inguruneak ziurtatzen duelarik, modu garden eta automatizatu batean.

Gako-hitzak

Garapen-ingurunea, *Software* bertsioak egiteko sistema, Etengabeko integrazioa, *Software* proiektuen eraikuntza konfiguratzeko tresna.

Abstract

The development of a software application requires management tasks. The higher quantity of applications to be developed, higher is the work to be done to manage these applications. It is common in a company to define a development architecture to ease the management tasks. But, to assure that the applications follow the rules defined by the architecture requires more work. This project designs a development environment for applications that assures that the rules defined in an architecture are followed. Thanks to this, the developers can focus on development tasks, letting the management tasks to the development environment in a transparent and automatized form.

Key words

Development environment, Version Control Systems, Continuous integration, Tool for manage projects build.

Resumen

El desarrollo de una aplicación *software* requiere un trabajo de gestión. Si la cantidad de aplicaciones que se desarrollan se aumenta, el trabajo que se requiere para gestionar el desarrollo de todas esas aplicaciones también se incrementa. Para facilitar esa gestión, las empresas que desarrollan aplicaciones de *software* suelen usar una arquitectura de desarrollo. Al mismo tiempo, requiere más trabajo asegurarse de que se cumple esa arquitectura. En este proyecto, se ha diseñado un entorno de desarrollo que facilita el desarrollo de las aplicaciones, que se encarga de asegurar que se cumplan las normas definidas en la arquitectura. Esto permitirá a los desarrolladores ocuparse en tareas desarrollo de aplicaciones, asegurando que entorno de desarrollo cumpla las reglas de la arquitectura de forma transparente y automatizada.

Palabras claves

Entorno de desarrollo, Sistemas para versionar *software,* Integración continua, Herramienta para configurar la construcción de proyectos *software.*

Aurkibidea

Taulen zerrenda

Irudien zerrenda

Grafikoen zerrenda

Ekuazioen zerrenda

Akronimoen zerrenda

WAS: *WebSphere Application Server*

JDK: *Java Development Kit*

SVN: *Subversion*

IDE: *Integrated Development Environment*

JRE: *Java Runtime Environment*

SCCS: *Source Code Control System*

RCS: *Revision Control System*

CVS: *Concurrent Versions System*

CI: *Continuous Integration*

LMU: *lock-modify-unlock*

CMM: *copy-modify-merge*

POM: *Project Object Model*

JVM: *Java Virtual Machine*

SE: Sistema Eragilea

FIFO: *First In First Out*

LIFO: *Last In First Out*

MEMORIA

I. ZATIA

1. Sarrera

Proiektu honen bezeroa den enpresak aplikazio kantitate handiak garatzen ditu, hauetariko gehienak Java ingurunean izanik. Gaur egun, bezeroak garatzen edo mantentzen ari den aplikazio kantitatea 400 ingurukoa da. Kantitate horri aplikazio horiek funtzionatu ahal izateko behar dituzten beste 300 bat liburutegi ere gehitu behar zaizkio.

Liburutegien helburua aplikazio ezberdinetan errepikatzen diren funtzionalitateak biltzea da. Horrela, funtzionalitate bat aplikazio ezberdinetan behin eta berriz programatu beharrean, liburutegi bakarrean programatzen da funtzionalitate hori. Ondoren, aplikazioek liburutegi hori erabiliz.

Liburutegien kontzeptua ondorengo adibidearen bidez uler daiteke. Demagun bi aplikazio ezberdin ditugula eta bietan PDF fitxategiak sortzen direla. Aplikazio bakoitzak sortuko dituen PDF fitxategiak ezberdinak izango dira, baina PDF fitxategiak sortzeko prozesua bera izango da kasu bietan. Horregatik prozesu hori bi aplikazioetan programatu beharrean egokiagoa da PDF fitxategiak sortzen dituen liburutegi bat idaztea eta bi aplikazioek liburutegi hau erabiltzea PDFak sortzeko. Funtzionalitate komunak liburutegietan idazteak badu beste abantaila bat: prozesuaren inplementazioak edo algoritmoak akatsen bat badauka, zuzenketa behin bakarrik egin beharko da, hau da, liburutegian.

Aplikazio eta liburutegien kantitate handia dela eta, proiektu honen bezeroa den enpresa ez da aplikazio eta liburutegi horiek guztiak berak bakarrik garatzeko eta mantentzeko gai. Horregatik, lanaren parte handi bat beste enpresa batzuen laguntzarekin egiten du, hau da, azpikontratatu egiten du. Horregatik, beste enpresa askok parte hartzen dute garapenean. Ondorioz, aplikazio eta liburutegien garapenean pertsona askok parte hartzen dute. Eta honen guztiaren kudeaketa konplexua izaten da.

Bezeroak garatzen dituen proiektuak handiak eta luzeak dira. Horregatik, azpikontratatutako enpresekin egiten dituzten kontratuak ere luzeak izaten dira.

Hemendik aurrera, bezeroa soilik aipatzen denean proiektu honen bezeroa esan nahiko da. Aldiz, bezero honek azpikontratatutako enpresak aipatzeko kasu bakoitzean zehaztuko da. Hau da, bezeroa aipatzen denean, aplikazio eta liburutegien jabe edo arduraduna dena izango da.

Beraz, proiektu honetan, bezeroaren liburutegi eta aplikazio guztiak garatzeko kudeaketa errazten duen garapen-ingurune bideragarri bat diseinatu eta inplementatu da.

2. Testuingurua

Garapenean parte hartzen duten enpresa eta pertsonen kopuru handiak aplikazioen garapena kudeatzea zailtzen du. Horregatik produktibitatea hobetzen duen, mantentze-lanak erraztu eta bateragarritasuna bermatzen duen programazioko arkitektura bat definitzea ohikoa da software aplikazioak egiten dituzten enpresen artean.

Proiektu honen kasuan, proiektu honen bezeroa den enpresak definitua du arkitektura bat. Arkitektura honetan hainbat arau, baldintza eta teknologia zehazten ditu bezeroak liburutegi eta aplikazioen garapenerako.

2.1. Arkitektura

Arkitektura horretan ondorengoa definituta dago:

- Aplikazioak exekutatuko diren zerbitzaria zehazten du, *WebSphere Application Server* [1] (WAS) izeneko zerbitzaria hain zuzen. WAS zerbitzaria IBMk garatutako aplikazioen zerbitzari pribatu bat da. Pribatua izateak bere iturri-kodea itxia dela eta zerbitzaria produkzioan erabili ahal izateko lizentzia bat ordaindu behar dela esan nahi du. Arkitekturak WASen 8.5 [2] bertsioa zehazten du, izan ere, bertsio ezberdinak daude.

- Java plataforman web aplikazioak paketatzeko erabiltzen den fitxategien formatu ohikoa war da. Liburutegiak berriz jar formatuko fitxategietan paketatzen dira. Ez da derrigorrezkoa horrela izatea, baina bezeroak arkitekturan liburutegiak jar fitxategietan eta web aplikazioak war fitxategietan paketatu behar direla zehazten du. Jar [3] eta war fitxategiak izatez, zip formatuan konprimatutako fitxategiak dira, eta Java lengoaian idatzitako fitxategi exekutagarriak dira. Jar eta war fitxategien arteko ezberdintasun nagusia, war fitxategietan web-baliabideak sartzen direla da, hala nola, HTML, CSS, Javascript, Jsp, etab. Horrez gain, war fitxategi baten barnean web aplikazio batek exekutatzeko behar dituen liburutegiak (jar fitxategiak) ere izan ditzake.

  Web aplikazioak war fitxategietan paketatzen badira ere, bezeroak definitzen duen arkitekturan, war fitxategiak ez dira zuzenean WAS zerbitzarian instalatuko. War fitxategiak instalatu beharrean ear motako fitxategiak instalatuko direla zehazten du arkitekturak. Ear [4] fitxategiak Java plataformako beste formatu bat da, eta jar eta war-en antzera, zip formatuan konprimatuta dago. Ear formatuaren desberdintasun nagusia, fitxategi barnean hainbat modulu izan ditzakeela da: moduluak war fitxategiak nahiz jar fitxategiak izan daitezke besteak beste. Arkitektura honen arabera, ear bakoitzean war bakarra joango da, jar modulurik gabe.

- Arkitekturak aplikazioak eta liburutegiak konpilatzeko erabiliko den JDK (*Java Development Kit*) bertsioa zehazten du. Konpiladorea ez du arkitekturak zehazten zehazki, WAS zerbitzariak baizik. Izan ere, erabiliko den zerbitzaria WAS 8.5, eta zerbitzariaren 8.5 bertsioak Java 7 [5] eskatzen du exekuziorako.

  JDK Java programak garatzeko tresna multzoa eskaintzen duen softwarea da, eta tresna horien artean konpiladore bat dago. Java bertsio berri bakoitzak funtzionalitate berriak gehitzen dizkionez Java plataformari, funtzionalitate berri horiek garatzeko beharrezkoa da konpiladorearen bertsio berria ere erabiltzea.

- Aplikazio eta liburutegiak WAS zerbitzarian exekutatuko direnez, nahitaezkoa da aplikazioak eta liburutegiak Java lengoaian idaztea. Esan bezala, Java 7 bertsioa erabiliz.

- Arkitekturan aplikazioak exekutatzeko behar diren liburutegiak zerbitzari-mailan jarri behar direla zehazten da. Horrela, aplikazio bakoitzak bere liburutegiak war fitxategi barnean eduki beharrean, liburutegiak aplikazio guztien artean partekatzen dituzte.
  Ohikoena eta Java konpiladoreen lehenetsitako funtzionamendua aplikazio bakoitzak (war fitxategi bakoitzak) bere barnean erabili behar dituen liburutegiak (jar fitxategiak) izatea da. War fitxategien barnean, aplikazioa exekutatzeko behar diren jar fitxategiak /WEB-INF/libs/ direktorio barnean egoten dira. Baina bezeroaren zerbitzarietan ehunka aplikazio exekutatu behar direnez bakoitza ehunka liburutegirekin, liburutegiak zerbitzari mailan jarri behar direla zehazten du arkitekturak.

- Aplikazioek erabili ditzaketen liburutegiak zehazten ditu arkitekturak. Aurreko puntuan azaldu denez, liburutegiak zerbitzari mailan doaz aplikazioen barnean joan beharrean. Hori dela eta, aplikazio baten exekuzioa zerbitzarian instalatuta dauden liburutegien menpe dago, hau da, aplikazio batek liburutegi baten beharra badauka eta liburutegi hori zerbitzarian ez badago, aplikazioa ezin izango da exekutatu. Horregatik, programadoreek aplikazioak garatzerakoan arkitekturak zehazten dituen liburutegiak erabili beharko dituzte.

  Gainera, arkitekturak liburutegiak definitzeaz gain, liburutegien bertsioak ere definitzen ditu. Izan ere, liburutegiak garatuz doaz, bertsio berri bakoitzean funtzionaltasun berriak inplementatuz eta gehiago behar ez direnak kenduz. Horregatik, derrigorrezkoa da programatzaileek zerbitzarian instalatuta egongo diren liburutegiaren bertsioa jakitea, garatuko duen aplikazio edo liburutegia zerbitzariko liburutegiekin funtzionatuko duela ziurtatu ahal izateko.

- Aplikazio eta liburutegien fitxategiak zein kodifikaziorekin idatzi eta gorde behar diren zehazten du.
  Kodifikazioa errespetatzen ez bada, karaktere bereziak erabiltzerakoan (‘ñ’ hizkia adibidez) zerbitzariak edota aplikazioa erabiltzen ari den nabigatzaileak gaizki interpreta dezakete eta exekuzio erroreak agertu daitezke.

- Arkitekturak software proiektuen izenen formatua definitzen du.

Arkitektura hau dela eta, programatzaileek ezin dituzte aplikazioak eta liburutegiak nahieran programatu. Hau da, arkitekturak ezartzen dituen arauak errespetatu behar ditu programatzaileak.

Arkitekturarekin amaitzeko, aplikazioak exekutatzen diren inguruneak azaltzea geratzen da. Hiru ingurune existitzen dira: ingurune lokala, garapeneko ingurunea eta produkzioko ingurunea.

- Ingurune lokala programatzaile bakoitzak bere ordenagailuan duen inguruneari deitzen zaio. Programatzen duena momentuan probatzeko erabiltzen du, eta programatzailearen kontrolpean dago.

- Garapeneko ingurunea bezeroak duen zerbitzari batean dago. Ingurune honetan ingurune lokalean ontzat eman diren liburutegi eta aplikazioak probatzen dira. Liburutegi eta aplikazio guztiak bezeroaren zerbitzari honetatik pasa behar dira, bai bezeroak berak garatutakoak,

baita azpikontratatutako enpresek garatutakoak. Produkziora pasa aurretiko pausoa da, beraz, ingurune honetan ontzat ematen diren liburutegi eta aplikazioak produkzioko ingurunera pasatzen dira.

- Produkzioko ingurunean aplikazioen erabiltzaileek erabiliko duten zerbitzaria dago. Ingurune honetan jartzen diren aplikazio eta liburutegiak egonkortzat eman direnak dira, hau da, akatsik aurkitu ez zaienak eta bukatutzat eman direnak. Aplikazio edo liburutegi bat garapen ingurunean ontzat eman ondoren, produkzioko ingurunean instalatu daiteke. Produkzioko ingurunea, garapeneko ingurunea bezala, bezeroak duen zerbitzari batean dago.

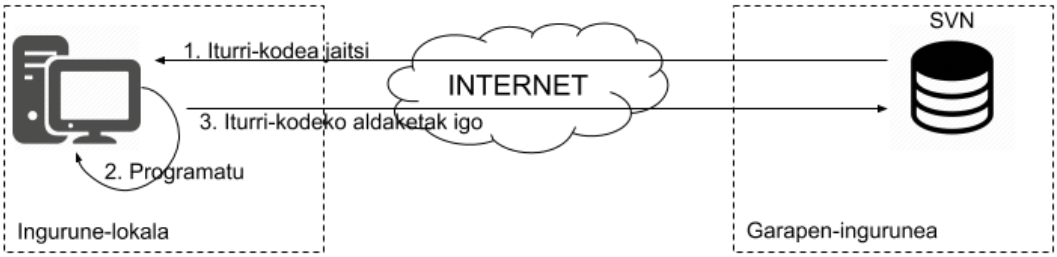
2.2. Hasierako egoera

Atal honetan, proiektu hau gauzatu aurreko egoeran bezeroak liburutegi eta aplikazioak nola garatzen dituen aztertuko da.

Aplikazio eta liburutegien kodea gordetzeko bezeroak biltegi bat dauka, *Subversion* [6] (SVN izenaz ere ezagutzen da) izenekoa. Izatez, *Subversion* softwarearen bertsioa kontrolatzeko [7] sistema bat da. Horrelako sistemek programen kodigoa biltegiratzen dute eta kodigoan egiten den aldaketa oro kontrolatzen dute. Kodigoan egiten diren aldaketa guztien historiko bat gordetzen dute, eta aldaketa bakoitza softwarearen bertsio berri bat gisa identifikatzen dute sistema hauek.

SVN biltegia bezeroaren sareko zerbitzari batean dago. Programatzaile batek liburutegi edo aplikazio bat programatu behar duenean, biltegi honetara konektatzen da Internet bidez eta proiektu horren iturri-kodea jaisten du programatzailearen ingurune lokalera (ingurune lokala programatzaile bakoitzaren ordenagailuari deitzen zaio). Liburutegia edo aplikazioa berria bada eta oraindik SVN biltegian ez badago, lehendabizi proiektu berriaren iturri-kodea SVN biltegira igo beharko du.

Ondoren, aldaketak egin eta berauek probatzen ditu bere ordenagailuan. Amaitzeko, aldaketak ontzat ematen dituenean, aldaketak berriz biltegira igotzen ditu. Prozesu hau ondorengo irudian azter daiteke:



Irudia 1: Iturri-kodea garatzeko prozesua SVNrekin

Programatzaile batek irudian ikus daitekeen bezala egiten du lan SVN biltegiarekin. Baina horrek ez du azaltzen zerbitzarietan instalatzen diren liburutegi eta aplikazioen fitxategi-bitarrak nola sortzen diren.

Aplikazio eta liburutegiak zerbitzarietan instalatzeko, hauen fitxategi-bitarrak sortu behar dira kodigoa konpilatuz. Gaur egun, fitxategi-bitar hauek ingurune lokalean sortzen dira eta ondoren bezeroari pasatzen zaizkio, honek garapen-inguruneko zerbitzarian instalatu ditzan. Prozesu hau programatzaile bakoitzak egiten du, eta ez dago inongo scriptekin automatizatuta. Beraz, programatzailearen eskuetan geratzen da arkitekturaren arauak betetzea, testak exekutatzea, etab. Eta bezeroak ez dauka azpikontratatutako programatzaileek hori dena bete duten ziurtatzeko modurik.

Izatez, bai bezeroaren enpresako programatzaileak eta baita azpikontratatutako enpresetakoak ere derrigortuta daude arkitekturako arauak betetzera. Baina aurretik aipatu den bezala, bezeroak garatzen dituen proiektuak handiak eta luzeak dira, ondorioz azpikontratatutako enpresekin egindako kontratuak ere luzeak dira. Adibidez, azpikontratatutako enpresa batekin 3 urteko kontratu bat egiten dute liburutegi eta aplikazio batzuk garatzeko. Baina azpikontratatutako enpresek ez dizkiote azken momentuan entregatzen liburutegi eta aplikazioak bezeroari. Izan ere, liburutegiak eta aplikazioak exekutatuko diren zerbitzariak bezeroak ditu, beraz probak egiteko bezeroari pasa behar zaizkio fitxategi-bitarrak. Hiru urte horietan zehar, egunen batzuk egongo dira fitxategi-bitarrik sortuko ez denik, gehienbat hasieran, baina orokorrean egunero fitxategi-bitar ugari sortuko dira, bezeroaren garapen-inguruneko zerbitzarian instalatu beharko direnak. Beraz, fitxategi-bitarra sortzen den bakoitzean arau guztiak betetzen direla egiaztatu behar dira. Arkitekturako arauak egiaztatzea eskuz egin beharreko zeregina izanik, denbora gehiegi xahutzen dute zeregin honetan.

Arkitekturako arauak ez betetzeak ez du esan nahi bezeroaren zerbitzarietan funtzionatuko ez duenik. Arauak betetzen ez dituzten liburutegi eta aplikazioek aurreikusi ezin daitekeen funtzionamendua eduki dezakete, arauak bete ez dituen liburutegia erabiltzen duten gainontzeko liburutegi eta aplikazioetan ere eragina edukiz. Arkitektura ez errespetatzeak izan ditzakeen ondorioak sakonago aztertuko dira 2.2.2. atalean.

2.2.1. Fitxategi-bitarrak sortzeko prozesua

Esan bezala, bezeroaren zerbitzarietan instalatzen diren softwareen fitxategi-bitarrak programatzaileek sortzen dituzte. Programatzaileek ingurune lokalean egiten dute lan, eta ingurune lokalaren kontrola programatzaile bakoitzak dauka. Ondorioz, ingurune lokalaren konfigurazioa beraien eskuetan geratzen da.

Ingurune lokalean lan egiteko Eclipse [8] softwarea erabiltzen da. Eclipse garapen-ingurune integratu (ingelesezko *Integrated Development Environment* edo IDE) bat da. IDE [9] bat softwarea garatzeko tresnak eta zerbitzuak eskaintzen dituen tresna edo aplikazioa da. Horretarako IDEak iturri-kodea idazteko editore bat, debuggera, auto-betetze adimendu bat, testak egiteko tresnak ditu besteak beste.

Eclipse Java programak idazteko oso aproposa da, nahiz eta beste lengoaia batzuetan idazteko ere balio duen. Zehazki esanda, Eclipse Javan idatzitako aplikazio bat denez, JRE (*Java Runtime Environment*) behar du exekutatzeko. Baina Java programak idazteko JREarekin ez da nahikoa. Java programak idazteko Eclipseri JDK (Java programak konpilatzeko behar diren liburutegiak dituen paketea) bat konfiguratu behar zaio, IDEak Java aplikazio eta liburutegiak konpilatzeko erabiliko duena.

JREa Java programak exekutatzeko beharrezko tresnak dituen software pakete bat da. Hau da, JDK Java programak konpilatzeko erabiltzen den bitartean, JREa konpilatutako programak exekutatzeko erabiltzen da. Aipatzekoa da JDKren barnean Java bertsio bereko JRE bat datorrela, JDK horrekin konpilatutako programak exekutatzeko balio duena.

2.2.2. Arazoak

Zerbitzarietan instalatzen diren aplikazio eta liburutegien fitxategi-bitarrak programatzaileen ingurune lokalean sortzeak arazo ugari ekartzen ditu.

Izan ere, bezeroak arkitekturaren arauak betetzen direla eta erabilitako teknologia arkitekturak zehaztutakoak direla ziurtatzeko modurik ez dauka. Ingurune lokala programatzaile bakoitzaren

menpe dagoenez, bere esku geratzen da JDK egokia erabiltzea, fitxategien kodifikazioa errespetatzea, eta oro har, arkitekturako arau eta baldintzak betetzea:

- Ezin da ziurtatu erabilitako kodifikazioa egokia dela.

- Ezin da ziurtatu konpilatzeko erabili den JDK bertsioa arkitekturak definitutakoa dela.

- Ezin da ziurtatu softwareak programatzeko eta konpilatzeko erabili diren liburutegiak arkitekturak definitutakoak direla.

- Ezin da ziurtatu fitxategi-bitarra sortu aurretik softwareak testak ondo pasa dituenik.

- Ezin da ziurtatu fitxategi-bitarraren kodigoa SVN biltegian dagoenarekin koherentea dela. Hau da, ezin da ziurtatu programatzaileak SVNra igo ez dituen aldaketak egin dituenik softwarean fitxategi-bitarra sortu aurretik.

- Bezeroak aplikazio bakoitza zerbitzarian exekutatu ahal izateko, zerbitzarian zein liburutegi instalatu behar diren jakiteko modu automatiko edo errazik ez dago.

- Ezin da ziurtatu aplikazio baten fitxategi-bitarra sortzen den bakoitzean, bere barnean ez dela liburutegirik sartzen.

Arau horiek ez betetzeak hainbat ondorio ditu:

- Kodifikazioaren eragina aplikazioaren exekuzioan zehar nabaritzen da. Adibidez, ‘ñ’ hizkia, azentu-markak, eta antzerako karaktereak gaizki agertu daitezke. Ondorio kritikoa ez bada ere, aplikazioaren kalitatea txartzen du honek.

- JDK bertsio egokia ez erabiltzeak aurreikusi gabeko eragina izan dezake. Adibidez, baliteke aplikazioa zerbitzarian arrankatu ere ez egitea eta ondorioz, bezeroa momentuan konturatzea aplikazioak ez duela funtzionatzen. Edo baliteke aplikazioaren zati zehatz bat exekutatu arte arazorik ez egotea (aplikazioaren gainontzeko zatiak JDK ezberdinarekin bateragarria delako), baina zati horrek ez funtzionatzea behar duen bezala edo zuzenean ez funtzionatzea. Izan ere, JDK bertsio batekin konpilatzen bada eta bertsio ezberdineko JRE batekin exekutatu, gerta daiteke liburutegi edo aplikazioan programatutako funtzionalitate bat exekutatzeko erabilitako JREan ez funtzionatzea.

- Arkitekturan definitutakoaren liburutegi ezberdinak erabiltzeak ere aurreikusi ezin daitekeen eragina izan dezake. Gutxi gorabehera JDK ezberdina erabiltzerakoan gerta daitekeen bera gerta daiteke.

- Testak ez pasatzearen ondorioa da, ondo funtzionatzen ez duen edo nahi ez den bezala funtzionatzen duen kodigoa zerbitzarietara pasatzea . Ondorioz, denbora asko galtzen da zerbitzarietan ondo ez dabiltzan liburutegiak eta aplikazioak instalatzen eta probatzen.

- Fitxategi-bitarra sortu aurretik SVNra igo ez diren aldaketak egiten badira iturri-kodean, ondoren erroreren bat agertzen bada ezin izango da iturri-kode hori berreskuratu zuzendu ahal izateko. Beraz, bezeroak ziur egon behar du zerbitzarietan instalatzen den software guztia SVNen igota dagoela eta berak uste duena dela. Hau ez betetzea segurtasun ikuspuntutik oso arriskutsua da, intentzio txarreko programatzaile batek nahi duen kodea instalatu baitezake bezeroaren zerbitzarietan.

- Arkitekturak definitzen duenaren arabera, liburutegiak ez doaz aplikazioen fitxategi-bitarren barnean. Ondorioz, aplikazioek behar dituzten liburutegiak banan bana instalatu behar dira zerbitzarian. Baina liburutegi hauen zerrenda lortzeko modu automatikorik edo errazik ez egoteak aplikazio bat instalatu behar den bakoitzean zerrenda hori lortzen denbora galtzea dakar. Eta gainera, zerrenda okerra lortuz gero edo liburutegiren bat ahaztuz gero, exekuzioko erroreak gerta daitezke, errorearen kausa zein den aurkitzeko denbora gehiago galduz.

- Aplikazioen fitxategi-bitarra sortzeko lehenetsitako prozesuan, liburutegiak aplikazioen barnean sartzen dira. Baina, arkitektura honen arabera, liburutegiak aplikazio barnean ezin dira joan. Beraz, aplikazio baten fitxategi-bitarra sortzen den bakoitzean, fitxategi-bitarraren barnean liburutegirik ez sartzeaz gogoratu behar da programatzailea. Hau ez betetzeak, beste behin ere, aurreikusi gabeko ondorioak ekar ditzake, bai zerbitzarian instalatzerakoan edota exekutatzerakoan. Liburutegiak zerbitzari-mailan jartzeak hainbat abantaila ditu:

    - Liburutegi batean aldaketa berri bat egiten bada, akats bat zuzentzeko adibidez, zerbitzarian liburutegia aldatu eta ondoren zerbitzaria berrabiaraztearekin nahikoa da. Aldiz, liburutegiak war fitxategien barnean badoaz, liburutegi hori erabiltzen duten aplikazio guztiak berreraiki (eraikitzea, labur azalduz, programa bat konpilatzea eta exekutagarria den jar, war edo ear formatuko fitxategi-bitarrean paketatzea da) eta berriz zerbitzarian instalatu behar dira akatsa zuzentzeko.

    - Liburutegiak zerbitzari-mailan jarrita, liburutegi bakoitza zerbitzarian behin kargatzen da. Liburutegiak war barnean sartuz gero, liburutegi bat erabiltzen duen aplikazio bakoitzeko kargatuko da zerbitzariaren memorian. Beraz RAM memoriaren kontsumo efizienteagoa lortzen da liburutegiak zerbitzari-mailan jarriz.

Fitxategi-bitarra sortzea software baten aldaketa berri bat inplementatu behar den bakoitzean errepikatu beharreko prozesua da, proiektu guztientzako berdina dena. Beraz, denbora asko galtzen dute programatzaileek prozesu errepikakor honetan.

Fitxategi-bitarra ondo ez sortzearen errua programatzaile bakoitzarena bada ere, prozesuak dituen arau eta baldintza ugariek prozesu konplexu batean bihurtzen dute. Eta aurretik aipatu den bezala, prozesu hau proiektu bakoitzeko egunean hainbat aldiz errepikatu daiteke. Ondorioz, oso ohikoa da akatsak maiz gertatzea. Honek ez du esan nahi, aplikazio eta liburutegi bakoitza bukatutzat ematen denean bezeroari akatsekin entregatzen zaionik. Baizik eta software bakoitza garapenean dagoen bitartean akats horiek asko zigortzen dutela garapen prozesua.

Beste arazo bat, ingurune lokalaren eta bezeroaren zerbitzariko ingurunearen artean dagoen desberdintasunak eragiten du. Inguruneak egoera ezberdinetan daudenez, gerta daiteke ingurune batean agertzen diren erroreak beste ingurune batean ez gertatzea.

Inguruneak egoera ezberdinetan egotearen arrazoi nagusia, programatzaileak garatzen ari diren liburutegien kantitatea da. Izan ere, garapenean daudenez, liburutegi asko etengabe aldatzen ari dira, ondorioz, liburutegi bat aldatzen den bakoitzean liburutegi hori erabiltzen duten liburutegi eta aplikazioak garatzen ari diren programatzaile guztiei abisatu eta bertsio berria pasa behar zaie. Hori dela eta, bi aukera daude bezeroak programatzaile guztiei liburutegi berriak pasatzeko maiztasunean:

1) Garapen-inguruneko liburutegietan aldaketa bat dagoen bakoitzean programatzaile guztiei abisatzea eta liburutegi berria pasatzea. Aukera hau bideraezina da, liburutegiak uneoro aldatzen ari direnez, programatzaileek uneoro ingurune lokala garapen-ingurunekoarekin sinkronizatzen egongo liratekeelako, programatzeko denborarik izan gabe.

2) Bigarren aukera, bezeroak programatzaileei noizbehinka (egunean behin edo bitan) garapen-inguruneko liburutegien zerrenda pasatzea da. Aukera honen abantaila da programatzaileek beren liburutegi eta aplikazioak programatzeko denbora izatea. Baina desabantaila gisa, ingurune lokalaren eta garapen-ingurunearen egoera denboraren zati handienean desberdina izatea suposatzen du, hau da, liburutegien bertsio berdinak ez izatea.

Bi aukera horien artean, bezeroak bigarrena erabiltzen du. Honen ondorioz, aurreikusi ezin diren akats asko ager daitezke bezeroaren zerbitzarietan, softwarearen garapen prozesua atzeratuz. Gainera, behin arazoa aurkituta dagoela, zaila da arazo hori ingurune lokalean erreproduzitzea, beraz zaila da jakitea noiz dagoen konponduta.

Amaitzeko, egoera honetan garatzen diren softwareen fitxategi-bitarren bertsiorik ez da egiten. Hau da, iturri-kodean egiten den aldaketa bakoitza SVN sisteman identifikatuta dago, baina iturri-kode horretatik sortzen den fitxategi-bitarra ez. Ondorioz, ezin daiteke jakin zein den zerbitzarian instalatuta dagoen fitxategi-bitarrari dagokion iturri-kodearen bertsioa. Beraz, ezin da jakin iturri-kodearen zein bertsiotan gertatu den erroreren bat eta zein dabilen ondo. Honek arazo ugari eragiten ditu: programatzaileek zaila dute zerbitzarian gertatutako erroreak erreproduzitzea beraien ingurune lokalean, edo aldaketa batean erroreren bat sartu bada software proiektu batean, zaila da berriz ere softwarearen aurreko bertsioa errekuperatzea. Fitxategi-bitar bat iturri-kode baten bertsioarekin erlazionatzeko, aukera erraz bat fitxategi-bitarraren izenean bertsioa edo identifikatzaile bat jartzea izango litzateke. Baina horrek ingurune lokaleko liburutegiak sinkronizatzen diren bakoitzean programatzaile bat garatzen ari den proiektu bakoitzaren Eclipse IDEko *classpath*a [10] birkonfiguratzea suposatuko luke. Horregatik, fitxategi-bitarrek beti izen bera dute.

3. Helburuak

Proiektu honetan enpresa baten software berriak garatzeko prozesua hobetzea da. Aurretik aipatu den bezala, enpresa honek arkitektura bat definituta dauka, zeinetan aplikazioak programatzerakoan arau batzuk errespetatu eta teknologia zehatz batzuk erabili behar diren exekutatuko diren ingurunearekin bateragarriak izan daitezen.

Beraz, proiektu honen **helburua arkitektura hori errespetatzen duten aplikazioak garatzea ziurtatzen duen garapen-ingurune automatizatu bat diseinatzea** da.

> Garapen-ingurunea [11] software produktuak sortzeko prozesu eta programazio tresnen multzoa da. Garapen-inguruneak prozesu eta tresna horiek koordinatzen ditu kodigoa konpilatzeko, testak exekutatzeko eta softwarearen fitxategi-bitarrak sortzeko besteak beste. Funtzio nagusiak horiek izaten badira ere, garapen-ingurune batek beste hainbat gauza egin ditzake, adibidez, kodigoa aztertu errepikatutako kodigoa aurkitzeko.
>
> Garapen-ingurunea aipatzen denean ez da garapen-ingurune integratuarekin (IDE edo Integrated Development Environment [9]) nahastu behar. IDE bat, aurretik aipatu den bezala, programatzaile bakoitzak bere ingurune lokalean programatzeko erabiltzen duen softwarea da. Adibidez, aurretik aipatutako Eclipse, ingurune lokalean programatzaileek erabiltzen duten softwarea.
>
> Proiektu honetan diseinatuko den garapen-ingurunea ere ez da arkitekturak definitzen duen garapen-ingurunearekin nahastu behar. Arkitekturako garapen-ingurunea garatutako liburutegi eta aplikazioak probatzeko erabiltzen den zerbitzari bat da. Aldiz, proiektu honetan diseinatuko den garapen-ingurunea, liburutegi eta aplikazio horiek garatzea ahalbidetzen duen ingurune bat da.

Arkitektura errespetatzea esaten denean, garapen-inguruneak ondorengo puntuak bermatu behar dituela esan nahi da:

- WebSphere Application Server motako zerbitzari batean exekutatuko direnez garapen-ingurune honetan sortutako aplikazioak, aplikazioen kasuan ear motako fitxategi-bitarretan eta liburutegien kasuan jar motako fitxategietan paketatzen direla ziurtatu behar du inguruneak.

- Arkitekturak zerbitzariaren WAS 8.5 bertsioa zehazten duenez, honek funtzionatzeko Java 7 bertsioa behar da. Beraz, garapen-inguruneak bai aplikazioak eta baita liburutegiak JDK 7rekin konpilatu beharko ditu. Ondorioz, Javan idatzitako programak direla ere bermatu beharko du.

- Arkitekturan azaldu denez, liburutegiak zerbitzari mailan doaz (aplikazioen barnean joan beharrean). Garapen-inguruneak aplikazio bakoitzaren barruan liburutegirik sartzen ez dela ziurtatu behar du.

- Liburutegiak aplikazioen barnean ez doazenez, garapen-inguruneak aplikazio bakoitza exekutatu ahal izateko zerbitzarian instalatu behar diren liburutegiak eskaini behar ditu.

- Arkitekturak aplikazioek erabili ditzaketen liburutegiak zehazten dituenez, garapen-inguruneak ziurtatu behar du arkitekturak zehaztutako liburutegiak erabiltzen direla, eta ez beste edozein. Honi esker, aplikazio eta liburutegien arteko bateragarritasuna bermatuz.

- Garapen-inguruneak zerbitzarian instalatuta dauden liburutegien informazioa eskaini behar du: zein liburutegi diren eta liburutegi bakoitzaren bertsioa zein den. Honek, programatzaileek beraien ordenagailuetan zerbitzariko egoera ahalik eta berdinen erreproduzitzea errazten du.

- Proiektuetako fitxategi guztiek UTF-8 kodifikazioa erabiltzen dutela bermatu behar du garapen-inguruneak.

Garapen-ingurunea automatizatua izan behar dela aipatzen denean, programatzaileak softwareak idazteaz bakarrik arduratu behar direla esan nahi da. Ondorioz, ondorengo prozesuak automatikoki gauzatu behar ditu:

- Arkitekturako arauak betetzen direla ziurtatu. Garapen-inguruneak arkitekturako arauak betetzen diren egiaztatuko du softwarea eraikitzerakoan, programatzaileak arau bakoitza betetzen dela banan bana egiaztatu gabe.

- Testak exekutatu.

- Zerbitzarian instalatu daitezkeen garatutako liburutegi eta aplikazioen fitxategi-bitarrak sortu (jar eta ear fitxategiak).

- Sortutako fitxategi-bitarrak zerbitzarian instalatu.

- Software proiektuen bertsioak egitea. Hasierako egoeran ez da software proiektuen bertsiorik egiten, ondorioz, ezin da zehatz jakin zerbitzarian instalatutako liburutegi eta aplikazioen iturri-kodea zein den. Hau da, software proiektu bat zerbitzarian instalatu ondoren garatzen jarraitzen da, beraz, iturri-kodea aldatzen doanez, ezin daiteke jakin zein den zerbitzarian instalatuta dagoen iturri-kodea. Ondorioz, zerbitzarian proiektu baten aldaketa berriak instalatu eta errore batengatik aurreko bertsiora itzultzeko modurik ez dago (ezin da aurreko bertsioa errekuperatu).

Arkitektura bermatu eta automatizatua den garapen-ingurune bat garatzeaz gain, proiektu honek ondorengo helburuak ere baditu:

- Softwareen eraikuntza programatzaileen eskuetan ez geratzea. Izan ere, hasierako egoera zein den azaldu den moduan, gaur egun programatzaileek sortzen dituzte zerbitzarietan instalatuko diren softwareen fitxategi-bitarrak. Horrek testak pasatzea eta arkitekturako arauak errespetatzea beraien menpe uzten dute. Ondorioz, bezeroak ezin du ziurtatu zerbitzarian instalatuko dituen softwarea egokia den, ezta instalatzen duen softwarea biltegian gordetakoa den.

- Softwareen kodigoa biltegiratzeko sistema bat eskaini behar du. Gaur egun, bezeroak softwarearen kodigoa biltegiratzeko Subversion izeneko sistema bat dauka. Subversion [6] softwarearen bertsioa kontrolatzeko sistema bat da, hau da, softwareen iturri-kodea gordetzeaz gain, softwarean egiten diren aldaketa guztien historiko bat gordetzen duen sistema bat da. Bezeroak, biltegiratze sistema hau garapen-inguru nerako egokia den aztertzea nahi du, edo besteren bat egokiagoa den begiratzea.

Proiektu honen bezeroak garapen-ingurunea ez du aplikazio gutxi batzuk bakarrik garatzeko erabiliko, baizik eta 400 aplikazio eta 300 liburutegi inguru garatzeko erabiliko du. Horregatik, garapen-inguruneak software kantitate horrekin arazorik ez duela izango ziurtatu behar da. Gainera, software proiektu guzti horiek konfidentzialak dira, beraz iturri-kodea eta fitxategi-bitarrak biltegi publikoetan gordetzea saihestu behar da.

4. Onurak

Proiektu honen onurak bi taldetan sailkatu daitezke: onura teknikoak eta onura ekonomikoak.

4.1. Onura teknikoak

Proiektu honetan diseinatutako garapen-ingurunearen onura nagusiak teknikoak dira. Alde batetik, **software aplikazioak modu eraginkor batean garatzea** ahalbidetuko du. Izan ere, garapen-ingurunea automatizatua izateak programatzaileak softwareak garatzeaz bakarrik arduratzea eragingo du. Hau da, softwarea eraikitzeko prozesuan garapen-ingurunea ondorengo eginkizunetaz arduratuko da programatzailearen beharrik izan gabe:

- Testak exekutatu eta denak pasatzen dituela ziurtatu.
- Arkitekturako arauak eta teknologia errespetatzen direla ziurtatu.
- Zerbitzarian instalatu daitezkeen softwarearen fitxategi-bitarrak sortu.
- Fitxategi-bitarrak zerbitzarian instalatu.

Eginkizun horiek automatikoak izateak programatzaileari ardura kentzen dio. Ondorioz, software baten garapeneko prozesuak ez dira programatzailearen menpe geratzen. Honen onurak ondorengoak dira:

- Testak maizago exekutatzen eta pasatzen direla ziurtatuko da. Horren ondorioz, softwareak dituen akatsak lehenago identifikatuko dira. Akatsak geroz eta lehenago identifikatu, konpontzea errazagoa eta merkeagoa izango da, eta eragin txikiagoa izango du.
- Testekin gertatzen den bezala, arkitekturako arauak betetzen direla eta teknologia errespetatzen dela maizago egiaztatuko da. Modu berean, arazoren bat egotekotan, geroz eta lehenago identifikatu arazoa, eragina txikiagoa izango da, eta zuzentzeko errazagoa eta merkeagoa izango da.
- Aplikazio eta liburutegietan egindako aldaketa edo hobekuntzak maiztasun handiagoarekin instalatuko dira zerbitzarian. Horren ondorioz, alde batetik exekuzio probak egitea azkartu, sinplifikatu eta erraztuko da; eta bestetik, aplikazioak azken erabiltzaileentzat prest egotea azkartuko da.

Beste onura bat, garapen-inguruneak zerbitzariaren egoera zehatza jakitea ahalbidetzen duela da. Hau da, garapen-inguruneak zerbitzarian instalatuta dauden liburutegi guztien informazioa emango du: zein liburutegi den eta liburutegiaren zein bertsio den. Honi esker, software baten erroreren bat agertzen bada zerbitzarian exekutatzen ari den bitartean, programatzaileek beraien ekipo lokaletan errorea erreproduzitzea erraztuko du. Ondorioz, errorearen konponbidea azkartuko da.

4.2. Onura ekonomikoak

Proiektu honek ez ditu irabazi ekonomikoak zuzenean ekarriko. Baina arkitektura honekin, aplikazioak modu eraginkorragoan garatzea ahalbidetuko duenez, programatzaileak aplikazioa programatzeaz bakarrik arduratu beharko dira, aplikazioen garapeneko kudeaketa diseinatutako garapen-inguruneari utziz. Horrela, programatzaileek denbora eta esfortzu guztiak programatzen jarriko dituzte, denbora asko aurreztuz programatzea ez diren eginkizunetan. Ondorioz, hauen orduko kostuari etekin hobea atera ahalko dio bezeroak.

Gainera, garapen-ingurunea automatizatua izanik, arkitekturako zein bestelako edozein errore azkarrago identifikatu ahal izango dira. Ondorioz, errore horiek eragin dezaketen galera ekonomikoak minimizatzen dira. Izan ere, onura teknikoetan aipatu den bezala, erroreak gero eta lehenago identifikatu, hauek konpontzea askoz ere errazagoa eta merkeagoa da.

Onura ekonomikoekin amaitzeko, garapen-ingurune honen bidez aplikazioak zerbitzarietan modu automatiko batean instalatuko dira. Horri esker, alde batetik aplikazioen merkaturatze denbora bizkortuko da hasierako inbertsioa lehenago errekuperatuz. Bestetik, merkaturatutako aplikazio batean akatsak identifikatuz gero, konponketa ere berehala merkaturatuko litzateke, galera minimizatuz. Hau da, mantentze lanen kostua txikitzea lortuko da.

5. Artearen egoera

Atal honetan, software bertsioak egiteko sistemen, etengabeko integrazioa inplementatzen duten sistemen eta software proiektuen eraikuntza konfiguratzeko tresnen artearen egoera aztertuko da.

5.1. *Software* bertsioak egiteko sistemak

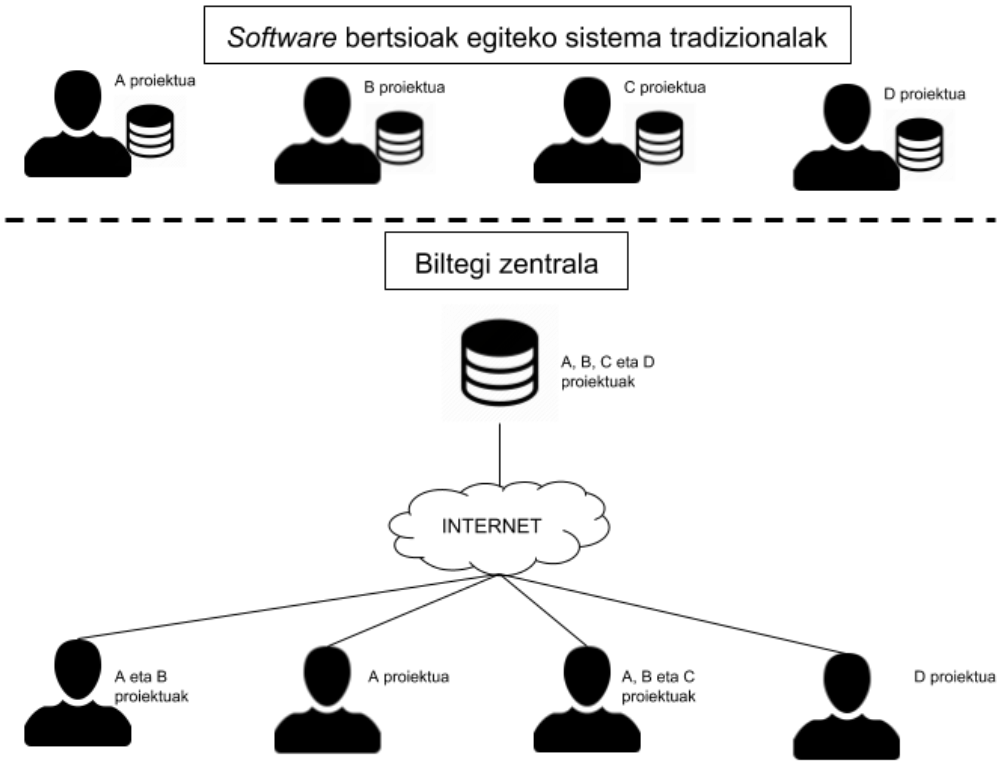
Software proiektuen bertsioak kudeatzearen arazoa 70. hamarkadaren hasieran agertu zen, programatzaileak bertsio ezberdinak eskuz kudeatzea bideraezina zela ohartu zirenean.

Arazo horri irtenbidea emateko, 1972. urtean *Source Code Control System* [12] edo *SCCS* sistema agertu zen. Sistema hura gai zen fitxategi indibidualak biltegiratzeko (proiektu osoak ezin ziren biltegiratu) eta horien kontrol sinplea egiteko: zein aldaketa egiten ziren eta nork egiten zituen gordetzen zen. Baina bazituen arazo batzuk: aldaketa bat egiten zenean zaila zen esatea zer aldatu zen eta noiz aldatu zen, eta biltegiratzeak bertsio bakoitzean aldaketak bakarrik gorde beharrean fitxategi osoak gordetzen zituenez, espazio asko behar zen. Arazo horiek izan arren, ondo inplementatutako soluzio bat zen eta horregatik gaur egun ere erabilia da.

SCCS sistemak 1982. urtera arte monopolioa izan zuen. Urte horretan monopolio horri aurre egiteko *Revision Control System* [13] edo *RCS* sistema agertu zen. SCCS sistemarekiko bazituen hobekuntza batzuk: erabiltzaile-interfaze errazago bat eskaintzen zuen eta azkarragoa zen. Baina desabantaila gisa, alde batetik ez zuen proiektu osoekin lan egiten uzten (fitxategi indibidualak) eta bestetik, ez zuen hainbat erabiltzaile fitxategi berarekin aldi berean lan egiten uzten.

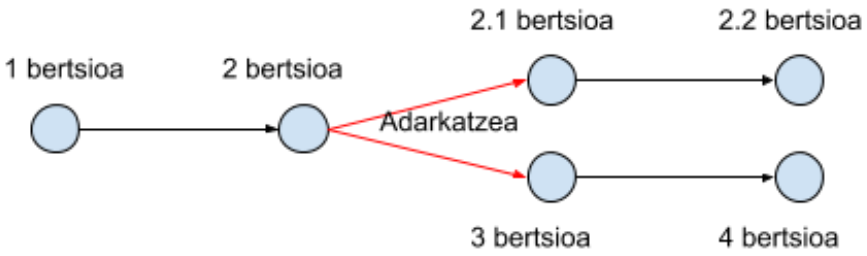
Aipatutako bi sistemek gabezia handiak dituzte: biltegiek fitxategi bakarra onartzen dute, proiektu osoak ezin dira biltegiratu, eta ez dute uzten fitxategi beraren gainean erabiltzaile bat baino gehiago aldi berean lanean aritzen. Hori dela eta, software bertsioak egiteko sistemen bigarren belaunaldia agertu zen.

Bigarren belaunaldian, fitxategi-zuhaitzak biltegiratzea ahalbidetu eta biltegi zentralak agertu ziren. Fitxategi-zuhaitzak biltegiratzeak proiektu osoak biltegiratu daitezkeela suposatzen du, fitxategi bakoitza indibidualki biltegiratu beharrean. Biltegi zentralak, Internetez iritsi daitezkeen zerbitzarietan instalatzen diren biltegiak dira, horrela, sarbidea daukan edozein erabiltzailek erabil ditzake biltegi hauek. Biltegi zentralei esker, ordenagailu ezberdinetan dabiltzan programatzaileek proiektu bera aldi berean programatu dezakete. Baina biltegi hauetan eragiketak egin ahal izateko, biltegi zentralera konektatuta egon behar da. Lehen belaunaldiko software bertsioak egiteko sistemaren eta biltegi zentraleko sistemen arteko ezberdintasuna ondorengo irudian azter daiteke:



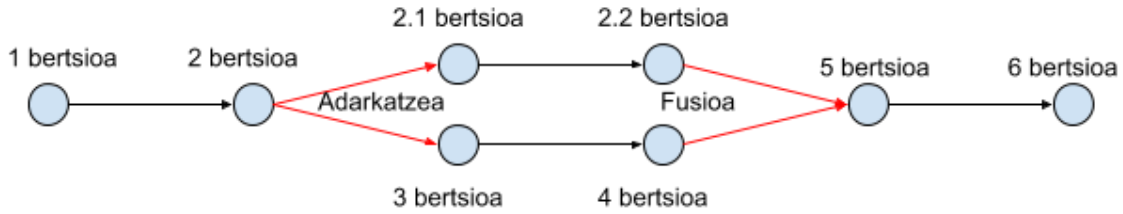
Irudia 2: Biltegi isolatuen eta biltegi zentralen eskemak

Bigarren belaunaldi honi 1990. urtean agertzen den *Concurrent Versions System* [14] edo *CVS* sistemak ematen dio hasiera. CVS sistemak biltegiratutako softwarearen bertsio ezberdinak modu efiziente batean kudeatzeko gai da, eta adarkatze (ezagunagoa den ingelesezko *branching*) eta fusio (ezagunagoa den ingelesezko *merging*) funtzionalitateak inplementatzen ditu. Adarkatzea biltegiratutako software bertsio batetik beste bide bat jarraituko duen "adar" bat ateratzea da, hau da, bertsio bat adarkatu ondoren software baten bi bertsio ezberdin daude, bakoitzak bere bidea izango duena, bere aldaketekin. Adarkatze prozesua ondorengo irudian azter daiteke:



Irudia 3: Adarkatze prozesua

Fusioa berriz, adarkatzearen kontrako prozesua da. Hau da, software baten bi bertsio ezberdin (noizbait adarkatutako softwarea) juntatzea bertsio berri bat sortuz. Aurreko irudiarekin jarraituz, ondorengo irudian azter daiteke fusio prozesua:

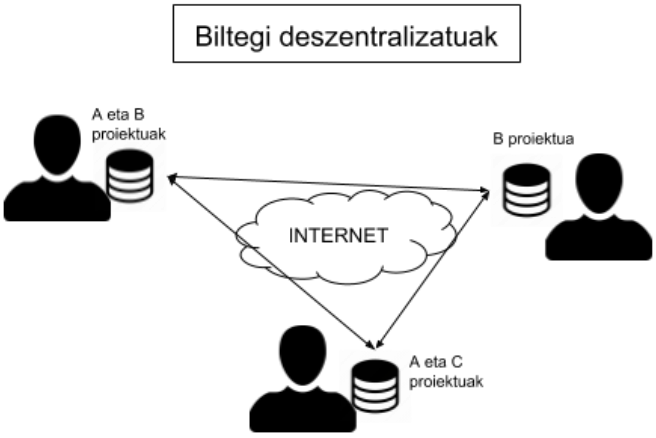


Irudia 4: Adarkatzea eta fusioa

Hala ere, CVS sistemak baditu hainbat eragozpen: aipagarriena eragiketa asko egiteko biltegia blokeatu behar duela da. Ondorioz, software proiektu berean hainbat erabiltzaile ari badira lanean, gainontzeko erabiltzaileak denbora batez lan egin ezin dutela uzteko arriskua dago.

Bigarren belaunaldi honetan aipagarria den beste sistema bat 2000. urtean agertu zen *Subversion* [6] edo *SVN* izeneko sistema da. SVNek horrelako sistemak modu masiboan erabiltzea eragin zuen programatzaileen artean. SVNren ekarpen nagusia aurreko sistemek zituzten akatsak konpontzea izan zen: biltegi zentralizatua erabiltzen zuen, baina fitxategi-zuhaitzak modu efizientean kudeatzen zituen, eta adarkatze eta fusio bezalako eragiketa konplexuak arazorik gabe erabil zitezkeen, hau da, eragiketa konplexuak biltegia gainontzeko erabiltzaileentzat blokeatu gabe egin zitezkeen.

Amaitzeko, hirugarren belaunaldi bat dago. Belaunaldi honetako desberdintasun nagusia biltegi zentralak izan beharrean, deszentralizatutako biltegiak [15] agertzen direla da. Deszentralizatutako biltegietan erabiltzaile bakoitzak bere biltegia dauka, beraz proiektu bera hainbat biltegitan egon daiteke. Programatzerakoan erabiltzaile bakoitzak bere biltegiarekin egiten du lan, software bertsioak egiteko eragiketa guztiak bere biltegian eginez. Aldaketa nahikoa egin ondoren eta hauek ontzat ematen direnean, gainontzeko erabiltzaileen biltegira igo daitezke aldaketak. Horrez gain, programatzaile batek beste erabiltzaile batek egindako aldaketak eduki nahi baditu, aldaketak beste erabiltzailearen biltegitik bere biltegira jaisteko aukera dauka. Biltegi deszentralizatuen adibide bat ondorengo irudian azter daiteke:



Irudia 5: Biltegi deszentralizatuen eskema

Hirugarren belaunaldi honi hasiera GNU Arch [16] izeneko sistemak eman bazion ere 2001. urtean, sistema deszentralizatuei zeresana Bazaar [17], Git [18] eta Mercurial [19] izeneko sistemek eman zioten.

Biltegi deszentralizatuen abantaila nagusia, lan egiteko edozein modutara egokitzen dela da. Software bertsioak egiteko eragiketa guztiak Internetera edo biltegi zentralera konektatu gabe egin daitezke. Eta biltegi zentralik egotea ez denez beharrezkoa, elkarri konektatuta (Internet bidez edo sare lokal batean) dauden bi programatzailek iturri-kodea arazorik gabe trukatu eta garatu dezakete.

Beste aukera bat biltegietako bat biltegi zentrala izatea da. Kasu honetan, programatzaile bakoitzak bere biltegia du bere ordenagailuan, baina biltegi horiek beste biltegi zentral batekin daude komunikatuta. Programatzaile bakoitzak software proiektu baten iturri-kodea biltegi zentral horretatik kopiatzen du bere biltegira, bere biltegian lan egiten du, eta nahi duenean egindako aldaketak biltegi zentralera igotzen ditu beste programatzaileek aldaketak jaso ditzaten.

5.2. Etengabeko integrazioa

Hiru praktika ezberdin daude, elkarren artean lotura handia dutenak: etengabeko integrazioa (*Continuous Integration*) [20], etengabeko entregatzea (*Continuous Delivery*) [21] eta etengabeko publikazioa (*Continuous Deployment*) [21].

5.2.1. Etengabeko integrazioa

Lehenengo praktika hau, software proiektu baten iturri-kodean egiten den aldaketa bakoitzean, automatikoki testak exekutatzean, bai banakako testak eta bai integraziokoak, eta softwarea eraikitzean oinarritzen da.

Etengabeko integrazioaren helburu nagusiak ondorengoak dira:

- Erroreak azkarrago aurkitu eta konpontzea
- Softwareen kalitatea hobetzea
- Softwareen eguneraketak balidatzeko eta publikatzeko denbora txikitzea

Ohikoa da software proiektu berdina garatzen ari diren programatzaileek bakoitzak bere aldetik lan egitea. Honek, normalean programatzaileek denbora luzez bakoitzak bere kabuz lan egitea eragiten du, bakoitzak bere lana amaitzean, denen lana elkarbanatzeko beharra egonik. Horrela lan egiteak, denen lana elkarbanatzea konplikatu eta luzatzea eragiten du. Arazoa handitu egiten da konpondu gabeko erroreak metatuz joaten direnean. Honek guztiak, software baten eguneraketak prestatzea eta bezeroei entregatzea zailtzen du.

Etengabeko integrazioan, programatzaile bakoitzak egindako aldaketak software bertsioak egiteko sistema bateko biltegira igotzen dira. Biltegira aldaketaren bat igotzean, etengabeko integrazioko sistemak aldaketa horiek detektatu eta automatikoki testak exekutatzen ditu, horrela errore funtzionalak edo integrazioko erroreak detektatzen dira berehala.

Aipatzekoa da, etengabeko integrazioan testak exekutatu ahal izateko, lehendabizi programatzaileek test horiek idatzi behar dituztela. Beraz, etengabeko integrazioaren bidez detektatzen diren erroreak programatutako testen araberakoak izango dira. Gauza bera gertatzen da etengabeko entregatzean eta etengabeko publikazioan.

5.2.2. Etengabeko entregatzea

Etengabeko entregatzea etengabeko integrazioa baino haratago doan praktika bat da. Praktika honekin, proiektu baten iturri-kodean aldaketak egiten direnean, automatikoki proiektua eraiki eta

testak exekutatzen dira etengabeko integrazioan egiten den bezala. Baina hori egin ondoren, eraikitzerakoan sortutako fitxategi-bitarra probako zerbitzari batean instalatu eta bertan test batzuk exekutatzen ditu, horrela, produkziorako prest utziz. Honi esker, software proiektuaren bertsio bat beti prest egoten da produkziorako.

Software proiektu bat produkziorako prest utzi aurretik, nahi adina probako zerbitzari ezberdinetan exekutatu daitezke testak. Zerbitzari bakoitza egoera ezberdinetan egon daiteke, horrela softwarea ingurune ezberdinetarako prest dagoela ziurtatzen da. Hala ere, praktika honetan, software bat produkzioko zerbitzari batean instalatu aurretik programatzaile batek onartu behar du, beraz hori ez dago automatizatuta.

Etengabeko entregatzearen onura, etengabeko integrazioaren aldean, testen automatizazioa banakako testak baino haratago doazela da, proiektu bat ingurune ezberdinetan testeatuz. Test hauek edonolakoak izan daitezke: integrazio-testak, kargako-testak, etab. Gainera, programatzaileek probetako ingurune batean liburutegia exekutagarri izaten dute probak egiteko. Praktika honi esker, software baten eguneraketak sakonago balioztatzen dira.

5.2.3. Etengabeko publikazioa

Etengabeko publikazioa, ia etengabeko entregatzearen berdina da. Desberdintasun bakarra, proiektu baten test guztiak exekutatu ondoren produkzioko zerbitzari batean instalatzeko programatzaile baten beharrik ez dagoela da. Hau da, software bat produkzioko zerbitzari batean instalatzeko prozesua ere automatizatuta dago.

Beraz, praktika honen bidez, software proiektu batean aldaketak egiten direnean, dena ondo joanez gero, proiektu hori produkzioko zerbitzarian instalatzen da.

5.3. *Software* proiektuen eraikuntza konfiguratzeko tresna

Software proiektu bat eraikitzearen helburu nagusia, iturri-kode batetik abiatuta makinek ulertu eta exekutatu dezaketen fitxategi-bitarra sortzea da. Helburu nagusia hori bada ere, eraikuntzan beste hainbat prozesu egiten dira, batzuk beharrezkoak direnak, eta beste batzuk aukerazkoak. Adibidez, iturri-kodea konpilatzea (beharrezkoa) eta testak exekutatzea (aukerazkoa).

Hasiera batean, eraikuntzako prozesu bakoitza exekutatzeko tresna bat erabili behar zen. Gainera, prozesu bakoitza exekutatzen zen bakoitzean modu ezberdin batean exekutatu zitekeen. Izan ere, ez zegoen prozesu horiek nonbait konfiguratzeko modurik beti modu berean exekutatu ahal izateko. Arazo hori konpontzeko sortu ziren software proiektuen eraikuntza konfiguratzeko tresnak.

Software proiektuen eraikuntza konfiguratzeko lehenengo tresna 1976. urtean agertu zen. Make [22] izena zuen tresnak, eta honek Makefile izeneko fitxategi batean software bakoitza nola konpilatu behar zen definitzen zuen. Software proiektuen tamaina hazi ahala, konpilatu beharreko fitxategi kantitatea ere hazi egin zen, ondorioz, konpilatze prozesua geroz eta konplexuagoa bihurtu zen. Horregatik agertu zen honelako tresnen beharra.

Make tresnaren funtzionalitateen artean ondorengoak dira nagusiak:

- Software baten azken erabiltzaileak softwarea instalatzea ahalbidetzen zuen, erabiltzaileak softwarea nola instalatu behar zen jakin gabe.

- Software baten eguneraketa bat dagoenean, aktualizatu behar diren fitxategiak zeintzuk diren kalkulatzeko gai da. Gainera, aktualizatu beharreko fitxategien ordena automatikoki zehazten

du. Honi esker, iturri-fitxategi batzuk aldatzen direnean ez du proiektu guztia konpilatzen, baizik eta aldaketa horiek jasaten dituzten fitxategiak bakarrik konpilatzen ditu.

- Make tresna ez da programazio lengoai bakarrera mugatzen.

- Make ez da softwarea eraikitzeko funtzioetara bakarrik mugatzen. Softwarea eraikitzeaz gain, paketeak instalatu, desinstalatu edota errepikatzen den edozein funtzio automatizatzeko balio du.

Urteak pasa ahala, software proiektuen eraikuntza konfiguratzeko tresnen aukera handituz joan da. Honekin batera, horrelako tresnek funtzionalitate gehiago inplementatu dituzte.

Proiektu honen bezeroak garatzen dituen aplikazio eta liburutegiak Java lengoaian garatzen direnez, Java lengoaian idatzitako proiektuak eraikitzeko tresnak zeintzuk diren ezagutzea beharrezkoa da. Horregatik, Java proiektuen tresnen artearen egoera aztertu da. Gaur egun, Java ekosistema hiru tresnek dominatzen dute: Ant + Ivy [23], Maven [24] eta Gradle [25].

5.3.1. Ant + Ivy

Ant software proiektuak eraikitzeko tresnen artean lehenengo tresna "modernoa" izan zen. 2000. urtean agertu zen Unix sistemetako Make tresna ordezkatzeko, izan ere, azken honek arazo asko zituen Unix sistemetan. Denbora laburrean Java proiektuak eraikitzeko tresna ezagunenean bilakatu zen. Aspektu askotan Make-ren antzekoa da, baina Ant Java lengoaian dago inplementatuta. Ondorioz, Java plataforma beharrezkoa du exekutatu ahal izateko. Ikasteko zailtasun kurba txikia dauka, ondorioz, edozeinek erabil dezake prestakuntza handirik gabe. Ant prozeduren programazioan [26] oinarritzen da. Urteen poderioz, tresna hobetua izan da pluginak onartzeko gaitasunari esker.

Ant tresnaren desabantaila nagusia eraikuntza-konfigurazioa XML formatuko fitxategietan egiten dela da. Izan ere, XML fitxategiek estruktura hierarkikoa izanik, ez dira egokiak prozeduren programaziorako. Gainera, Ant-en XML fitxategiak kudeatzeko oso zailak bihurtzen dira proiektuak txikiak ez direnean.

Urteak aurrera egin ahala, software proiektuek erabiltzen dituzten liburutegiak internetez lortzeko funtzionalitatea inplementatzeko beharra agertu zen. Hau da, orain arte proiektu batek behar zituen liburutegiak programatzaileak eskuz hornitu behar bazituen, orain programatzaileek proiektuek behar dituzten liburutegiak zerrendatu eta tresna bat arduratzen da Internetez deskargatzeaz eta proiektua hornitzeaz. Horretarako, Ant-ek Apacheren Ivy tresna inplementatu zuen 2004. urtean.

5.3.2. Maven

Maven 2002. urtean agertu zen. Bere helburu nagusia programatzaileek Ant-ekin edukitzen zituzten arazo nagusiak konpontzea zen. Mavenek Antek bezala, XML formatuko fitxategiak erabiltzen ditu proiektuen eraikuntza konfiguratzeko. Baina Anten zeregin bat egiteko komandoak idatzi behar diren bitartean, Mavenen exekutatu beharreko zereginak jartzen dira ezer programatu gabe. Maveneko zereginei helburuak (ingelesezko "*goal*") deitzen zaie, eta besteak beste konpilatzeaz, testak exekutatzeaz eta fitxategi-bitarrak sortzeaz arduratzen dira. Hau da, Maveneko XML fitxategian helburu bat idazten denean, Mavenek helburu hori betetzeko plugin bat dauka, eta plugin hori arduratzen da helburu hori betetzeaz.

Mavenek ekarri zuen berrikuntza nagusia menpekotasunak (menpekotasunak proiektu batek erabiltzen dituen liburutegiak dira, eta ingelesez *dependency* esaten zaio) Internet bidez lortzea izan

zen, aurrerago Antek Ivy bitartez inplementatu zuena. Berrikuntza honek softwarea elkarbanatzeko modua aldatu zuen.

Mavenen beste abantaila bat bere bizi-zikloa [27] da. Mavenen bizi-zikloak hainbat fase ditu, fase bakoitzean aurretik aipatutako “helburuak“ exekutatzen direlarik. Bizi-ziklo honi esker, fase bat exekutatzeko lehendabizi aurreko fase guztiak exekutatzen dira. Adibidez, fitxategi-bitarra sortu nahi bada, aurretik beharrezkoa da iturri-kodea konpilatzea, beraz, *“package”* fasea (fitxategi-bitarra sortzeko fasea) exekutatu nahi badugu, Maven arduratzen da aurreko fase (eta ondorioz, “helburu”) guztiak exekutatzeaz. Honek, konfigurazioa asko sinplifikatzen du. Baina desabantaila gisa, konfigurazioan malgutasuna galtzen da.

5.3.3. Gradle

Gradle 2012. urtean agertu zen eta azkenengo urteetan indarra hartzen ari den tresna da. Adibide gisa, Googlek Gradle erabiltzen du Android OSrako lehenetsitako eraikuntza tresna gisa.

Gradlek ez du XML formatuko fitxategiak erabiltzen, baizik eta Groovyn [28] oinarritutako beste lengoaia bat erabiltzen du. Izan ere, helburuetako bat Maveneko XML fitxategiak atzean uztea du. Honi esker, Gradleko konfigurazio fitxategiak Mavenekoak eta Antekoak baino laburragoak eta garbiagoak dira. Gainera, Mavenen antzera, Gradlek ere bizi-ziklo bat dauka definituta, baina honen konfigurazioak Mavenekoak baino malgutasun handiagoa eskaintzen du.

Hasiera batean, Gradlek menpekotasunak ebazteko Ivy erabiltzen bazuen ere, aurrerago bere inplementazio propioa garatu du.

6. Diseinuaren lehenengo hurbilketa

Atal honetan, aurretik zehaztu diren arazoei irtenbidea eman eta helburuak nola beteko diren azalduko da. Horretarako, artearen egoeran egindako azterketa kontutan hartuko da.

6.1. *Software* bertsioak egiteko sistema

Softwareen iturri-kodea biltegiratzeko software bertsioak egiteko sistema bat erabiltzea proposatzen da. Orain arte SVN biltegi bat erabiltzen zen, baina proiektu honetan SVNrekin jarraitzea edo beste sistema modernoago bat erabiltzea egokiagoa den aztertu da. Azterketa hau alternatiben analisien atalerako utziko da.

6.2. Etengabeko integrazioa

Artearen egoera atalean, etengabeko integrazioan maila ezberdinak daudela ikusi da. Azkeneko maila etengabeko publikazioa izan da, baina proiektu honetarako ez da egokia. Izan ere, software bat oso ondo probatuta egon behar da produkzioko ingurunera joan aurretik. Proiektu honen bezeroaren kasuan, langile gutxi batzuk bakarrik daukate produkzioko ingurunean zein liburutegi eta aplikazio instalatuko diren erabakitzeko ardura.

Etengabeko entregatzeari dagokionez, proiektu honen testuingurua erreparatuz, software bera garatzen aldi berean programatzaile asko egon daitezke. Honen ondorioz, proiektu baten biltegira aldaketa ugari denbora tarte labur batean igotzen dira. Horregatik, aldaketa bat igotzen den bakoitzean ez da praktikoa softwarea arkitekturan definitutako garapeneko inguruneko zerbitzarira igotzea. Horregatik, batzuetan etengabeko entregatzea egin nahiko da, baina beste batzuetan etengabeko integrazioarekin nahikoa izango da.

Horregatik, bien arteko nahasketa bat egin da: lehenetsitako moduan etengabeko integrazioa erabiliko da, eta nahi izanez gero, etengabeko entregatzea erabiltzeko aukera eskainiko da. Kasu guztietan, produkzioko ingurunean instalatzeko prozesua proiektu honetatik kanpo geratuz.

6.2.1. Zein arazo konpon ditzake etengabeko integrazioak

Behin etengabeko integrazioa zer den eta bere onurak zeintzuk diren ikusita, zein arazo konpontzen dituen eta proiektu honetan zein helburu betetzeko erabiliko den zehaztuko da.

Etengabeko integrazioak konpontzen duen arazo nagusia **softwareen eraikuntza automatizatzea** da. Hau da, software bertsioak egiteko sistema batean biltegiratuta dagoen proiektu baten iturri-kodea jaitsi eta eraiki egiten du. Horri esker, programatzaileek ezin dute eskurik sartu proiektuaren eraikuntzan, eta ondorioz, etengabeko integrazioan sortzen diren fitxategi-bitarrak software bertsioak egiteko sisteman biltegiratutako iturri-kodearekin koherentea dela ziurtatzen da.

Eraikuntza automatizatzeaz gain, proiektu bakoitzaren iturri-kodearekin nahi dena egitea ahalbidetzen du etengabeko integrazioak. Horri esker ondorengo helburuak bete daitezke:

- Behin software baten iturri-kodea jaitsita, eraiki aurretik proiektuak arkitekturako arauak betetzen dituela ziurtatu daiteke. Baina horretarako, beharrezkoa izango da arau horiek balidatzeko sistema bat sortzea.

- Softwarearen fitxategi-bitarra sortu ondoren, fitxategi-bitar horren bertsioa egin eta zerbitzarian instalatzea. Etengabeko integrazioa hau egiteko gai ere bada, baina horretarako tresnak garatu behar zaizkio.

6.2.2. Etengabeko integrazioa inplementatzea

Garapen-ingurune batean etengabeko integrazioa inplementatzeko modu nagusia aplikazio bitartez da. Hainbat aplikazio existitzen dira garapen-ingurunean muntatu daitezkeenak, adibidez: Jenkins [29] / Hudson [30], Travis CI [31], etab. Software hauek alternatiben analisian aztertuko dira.

6.3. Software proiektuen eraikuntza konfiguratzeko tresna

Software proiektuen eraikuntza konfiguratzeko tresna batekin software baten eraikuntza nolakoa izan behar den deskribatzen da eta ondoren, tresnak softwarea deskribapen horren arabera eraikitzen du.

Deskribapena edo konfigurazioa nola egiten den tresnaren arabera aldatzen da, baina normalean software barneko fitxategi batean deskribatzen dira software horren eraikuntzan exekutatu behar diren prozesuak. Horrela, konfigurazio fitxategi horrekin eraikitzen diren proiektu guztiak era berean eraikitzea ziurtatzen da.

Software baten eraikuntzan konfiguratu daitekeenaren artean ondorengoak daude besteak beste:

- Konpilatzeko erabiliko den konpiladorea.
- Testak exekutatzea.
- Zein liburutegi erabiliko diren konpilatzeko eta testak exekutatzeko.
- Fitxategi-bitarrean zer sartu behar den. Honen bidez, fitxategi-bitarra sortzerakoan iturri-kodeko fitxategi batzuk baztertu daitezke. Adibidez, .jar fitxategiak ez sartzea .war fitxategien barnean.
- Zein motatako fitxategi-bitarra sortu behar den. Adibidez, Java proiektuen kasuan, jar, war edo ear.

Horrez gain, tresnaren arabera, beste gauza asko egin eta konfiguratu daitezke. Eta proiektu baten eraikuntza prozesuan egin nahi dena existitzen ez bada, tresna askok plugin berriak garatzea ahalbidetzen dute, aukerak infinituak izanik.

Horrelako tresna baten onurak ondorengoak dira:

- Software baten eraikuntzako prozesuak konfiguratzea ahalbidetzen du. Hala nola, testak exekutatu behar diren ala ez, jar, war edo ear motako fitxategiak sortu behar diren, zein konpiladore erabili behar den, etab.
- Software bat beti modu berean eraikitzea ahalbidetzen du.

Horrelako tresnak, besteak beste, ondorengoak dira: Maven [24], Gradle [25] eta Ant + Ivy [23].

6.3.1. Zein arazo konpon ditzake software proiektuen eraikuntza konfiguratzeko tresnak

Atal honetan, software proiektuen eraikuntza konfiguratzeko tresnak, proiektu honetan zehaztutako zein arazo konpondu eta zein helburu betetzeko erabiliko den aztertuko da.

Konpontzen dituen arazoak ondorengoak dira:

- Konpilatzeko erabili den konpiladorea arkitekturak zehaztutakoa izatea.

- Arkitekturak definitutako liburutegiak erabiltzea softwarea konpilatzeko eta testak exekutatzeko.

- Konpilatzerako orduan arkitekturak definitutako kodifikazioa erabiltzea.

- Testak exekutatzea ziurtatzea.

- Softwarea exekutatzeko zerbitzarian instalatu behar diren liburutegiei buruzko informazioa eskaintzea.

- Software baten eraikuntza guztiak berdinak izatea, programatzailearen eskutan utzi gabe eraikuntza prozesua.

- Zerbitzarian instalatu beharreko fitxategi-bitarra sortzea: jar motako fitxategiak liburutegien kasuan eta ear motako fitxategiak aplikazioen kasuan.

- Fitxategi-bitarraren barnean liburutegiak ez sartzea.

- Bezeroaren zerbitzarian instalatu behar diren liburutegien informazioa eskaintzen duenez, informazio hori erabili daiteke programatzaileek bezeroaren zerbitzarien egoera erreproduzitzeko beraien ingurune lokaletan.

- Eraikitzen den software bakoitzari bertsio bat ematea.

Hala ere, kontutan eduki behar da, nahiz eta software proiektuen eraikuntza konfiguratzeko tresna bat erabili, bezeroaren zerbitzarietan instalatzen diren fitxategi-bitarrak programatzaile bakoitzak bere ingurune lokalean eraikitakoak izaten jarraitzen duen bitartean, bezeroak ezin duela ziurtatu berak zehaztutako konfigurazioa erabili denik fitxategi-bitarra sortzeko.

Horregatik, beharrezkoa da tresna hau etengabeko integrazioa inplementatzen duen aplikazioan erabiltzea. Hau da, helburuak ondo betetzeko, ondorengo prozesua jarraitu beharko da:

1. Proiektu baten iturri-kodean aldaketa bat egon denean, etengabeko integrazioko aplikazioak detektatu eta eraikuntza prozesua hasiko du.
2. Eraikuntza prozesuan, etengabeko integrazioko aplikazioak proiektuaren iturri-kodea jaitsiko du.
3. Iturri-kodea jaitsita, software proiektuen eraikuntza konfiguratzeko tresnaren bidez proiektua eraikiko du aplikazioak.

6.4. Software bertsioak egiteko mekanismoa

Software proiektuetako iturri-kodearen aldaketa ezberdinak identifikatzeko bertsioak egiteko edo izendatzeko mekanismo bat proposatzen da atal honetan.

Bi bertsio mota ezberdinduko dira: ***snapshot*** bertsioak eta ***release*** bertsioak. Garapen egoeran dauden softwareek *snapshot* bertsioa izango dute, eta aldaketa multzo bat ontzat ematen denean softwareari *release* bertsioa emango zaio. Hau da, *release* bertsioak software baten egoera egonkorra dela esan nahiko du. Beraz, zerbitzarian instalatzen den software proiektu guztiak *release* bertsioa izan beharko dute.

Adibidez, programatzaile bat software berri bat garatzen hasten denean, softwareari 1.0.0-SNAPSHOT bertsioa emango dio. Bertan nahi adina aldaketa egingo ditu, eta egin dituen aldaketak ontzat ematen dituenean eta zerbitzarian instalatu nahi dituenean, 1.0.0 bertsioa amaitu duela suposatuko da. Beraz, 1.0.0-SNAPSHOT bertsiotik 1.0.0-RELEASE bertsioa aterako du. Ondoren, softwarea garatzen jarraitu nahi badu, hurrengo bertsio batean egin beharko du, adibidez, 1.1.0-SNAPSHOT. Izan ere, aipatu den bezala, *release* bertsioak egonkorrak dira eta ondorioz, ezin da aldaketarik egin bertsio hauetan.

Azaldu da bertsioak egiteko mekanismo honetan *snapshot* eta *release* bertsioak nola kudeatuko diren. Baina adibideko bertsioen numerazioan hiru zenbaki [32] agertu dira. Jarraian, bertsioan hiru zenbaki erabiltzea zergatik proposatzen den azalduko da.

Lehenengo zenbakiari ***major*** deitzen zaio. Zenbaki hau software proiektu batek aurreko bertsioarekin bateraezina den APIaren aldaketak egiten dituenean aldatzen da.

Bigarren zenbakiari ***minor*** deitzen zaio. Eta zenbaki hau software proiektu bati aurreko bertsioarekin bateragarria den aldaketak egiten direnean aldatzen da.

Major eta *minor* zenbakien adibide gisa, A aplikazio batek B liburutegiaren 1.0.0-RELEASE bertsioa erabiltzen duela suposatuko da. B liburutegiaren bertsio berri bat agertzen da, 1.1.0-RELEASE. *Minor* zenbakia aldatu denez, ezin da ziurtatu liburutegiak lehen bezala funtzionatzen duenik, baina aurretik zuen APIarekin bateragarria dela esan nahiko du. Beraz, A aplikazioak B liburutegiaren bertsio berria erabiltzeko gai izango da. Ondoren, B liburutegiaren beste bertsio berri bat agertzen da: 2.0.0-RELEASE. Kasu honetan *major* zenbakia aldatu da, ondorioz, aurreko APIarekin ez da bateragarria eta ezin da ziurtatu A aplikazioa B erabiltzeko gai izango denik aldaketa handirik egin gabe. Aldaketa hauek, B liburutegiak behar dituen beste liburutegi batzuk aldatzea ere suposatu dezake.

Azken zenbakiari ***patch*** deitzen zaio, eta produkzio-ingurunean dagoen bertsio batean aurkitutako arazoak konpontzen direnean aldatzen den zenbakia da. Adibide gisa, suposatu produkzioan aplikazio baten 1.0.0-RELEASE bertsioa dagoela eta bertan errore garrantzitsu bat aurkitzen dela. Errore horrek ezin du itxaron garapenean dagoen hurrengo bertsioari, izan ere, garapenean dagoen bertsioa baliteke oraindik amaitu gabe egotea edo produkziora pasatzeko oraindik balidatu gabe egotea. Horregatik, zuzenketak zuzenean produkzioan dagoen bertsioaren gainean egin behar dira, zuzenketa hauek garapen-ingurunetik pasako ez direlarik. Kasu honetan, zuzenketa eginda *patch* bertsioa aldatuko litzateke, 1.0.1-RELEASE hain zuzen. Amaitzeko, zuzenketa garapenean dagoen bertsioan ere aplikatu behar da, bestela, garapenean dagoen bertsioa produkziora pasatzen denean, berriz ere errore berdina agertuko da.

Proiektuaren soluzioa diseinatzeko erabiliko diren sistema, teknika eta mekanismoak zehaztuta, ondorengo ataletan horien artearen egoera eta alternatiben analisia egingo da. Ondoren, azterketa horiei esker proiektu honen diseinu bat proposatuko da.

7. Aukeren analisia

Atal honetan, proiektu honen helburuak bete ahal izateko diseinuaren lehenengo hurbilketan zehaztu diren tresna eta sistema ezberdinak aztertuko dira. Kasu bakoitzean proiektu honetako helburuak betetzeko eta arazoak konpontzeko egokiena dena aukeratuz.

Beraz, atal honetan ondorengoen alternatiben analisia egin da: softwarea bertsioak egiteko sistemak, etengabeko integrazioa inplementatzen duten sistemak eta software proiektuen eraikuntza konfiguratzeko tresnak.

7.1. *Software* bertsioak egiteko sistema

Gaur egun software proiektuetan aurkitzen den arazo handienetako bat bertsioak kudeatzea izan ohi da. Ez proiektua pertsona desberdinek gauzatu behar dutenean soilik, baizik eta pertsona bakoitzaren erakunde eta egitura mailan ere. Arazo honi aurre egiteko bertsioak kontrolatzeko sistemak existitzen dira [3]. Sistema hauek software proiektuaren osagai guztiak gordetzea, bakoitza zaharberritzea eta bakoitzaren aldaketa bakoitza gordetzen duen historikoa edukitzea ahalbidetzen dute.

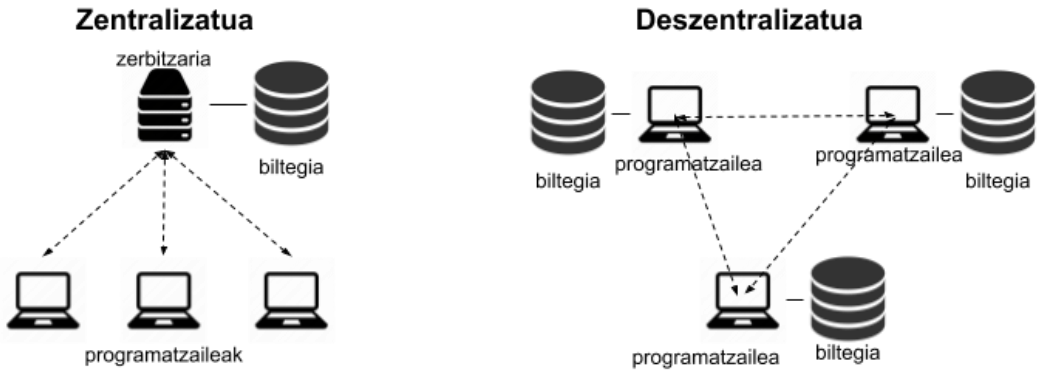
Gaur egun software bertsioak egiteko sistema ugari existitzen dira, eta horietako batzuk aztertuko dira, hasiera batetik interesgarrienak diruditenak edota interneten gehien erabiltzen direnak. Gainera, horrelako sistema berri bat garatzearen aukera hasieratik baztertuko da, proiektu honen helburuetatik irteten baita. Hori dela eta, aurre-filtro honen ondorioz Subversion [6], Git [18] eta Mercurial [19] sistemak aztertuko dira.

7.1.1. Aukeraketa irizpideak

Analisi honetan ondorioak atera ahal izateko eta aukeraketa bat egiteko ondorengo irizpideak jarraitu dira.

Hasteko, sistema ezberdinen informazio teknikoa erabili da, biltegi mota zehaztuz eta eragiketa ezberdinak nola egiten dituzten zehaztuz.

- Biltegi motari dagokionez, aurretik aipatu den bezala, **zentralizatutako** edo **deszentralizatutako** sistema den aztertu da. Sistema zentralizatu batean biltegi nagusi bat dago zeinetara gainontzeko guztiak konektatzen diren. Honen eraginez, programatzaile batek biltegian egiten dituen aldaketa guztiak gainontzekoei eragiten diete eta beharrezkoa da sarera konektatuta egotea iturri-kodean aldaketak egin ahal izateko. Deszentralizatutako sistemetan aldiz, ez dago biltegi bakarra, baizik eta programatzaile bakoitzak biltegiaren kopia bat edukitzen du bere ordenagailuan. Horrela, eragiketa guztiak bere ordenagailuan bertan egin ditzake, sarera konektatzeko beharrik izan gabe. Sarera konektatzeko beharra aldaketak gainontzekoekin konpartitu nahi direnean soilik dago. Honi esker, sistema honetan egiten diren eragiketak oso azkarrak dira, eta beste programatzaile batek egindako aldaketak ez dute eraginik norberaren biltegian, aldaketa horiek norberaren biltegira jaistea erabakitzen den arte. Gainera, biltegiaren hainbat kopia egon daitezkeenez, iturri-kodea galtzea zailagoa da, biltegi bakoitzean iturri-kode osoa gordetzen baita.



Irudia 6: Biltegi zentralizatu eta deszentralizatuen arkitekturak

- Eragiketak nola egiten diren aztertzerakoan, hainbat atal aztertu dira. Alde batetik, erabiltzaile bat aldaketak egiten ari denean sistema blokeatzen den (*lock-modify-unlock* edo LMU) edo erabiltzaile ezberdinek aldi berean aldaketak egin ditzaketen (*copy-modify-merge* edo CMM) aztertu da [33]. Bestetik, fitxategien aldaketen historiala biltegiratzerakoan egindako aldaketak bakarrik gordetzen diren (changeset), edo aldaketa egin aurreko eta ondorengo fitxategi osoa gordetzen den (snapshot) aztertu da. Hau da, aldaketa egin aurreko eta ondorengo fitxategi osoak gordetzen diren, edo bi fitxategiren arteko ezberdintasuna bakarrik gordetzen diren aztertu da. Aztertu den beste alderdi tekniko bat, fitxategiak bakarrik edo fitxategien-zuhaitzak gordetzen dituzten izan da. Eta alderdi teknikoekin amaitzeko bertsio ezberdinak identifikatzeko hash funtzio kriptografikoak edo zenbaki arruntak erabiltzen dituzten ikusi da.

Sistema	Subversion	Git	Mercurial
Biltegi mota	Zentralizatua	Deszentralizatua	Deszentralizatua
LMU/CMM	LMU/CMM	CMM	CMM
Aldaketen historia	Changeset/Snapshot	Snapshot	Changeset
Aldaketen irismena	Zuhaitza	Zuhaitza	Zuhaitza
Bertsioen identifikazioa	Zenbakiak	SHA-1 hash	SHA-1 hash

Taula 1: Software bertsioak egiteko sistemen ezaugarriak

Alderdi teknikoak aztertu ondoren, sistema bakoitzaren funtzionalitate espezifikoak aztertu dira:

- **Commit atomikoak:** aldaketa guztiak egitea edo arazoren bat egonez gero aldaketa bat ere ez egitea ziurtatzen du.

- **Fitxategi eta direktorioen berrizendapena:** ea sistema fitxategi edo direktorio bat mugitu edota kopiatzean bere historiala mantentzeko gai den.

- **Berrizendatutako fitxategien fusioa:** ea sistema fitxategi edo direktorio bat berrizendatzean bere historiala mantentzeko gai den.

- **Biltegien erreplika:** ea sistema gai den biltegiaren kopia oso bat egiteko.

- **Biltegiaren baimenak:** sistema fitxategi eta direktorio ezberdinetarako baimen ezberdinak definitzeko gai den zehazten du.

- **Kodifikazioa:** ea sistema gai den kodifikazio ezberdinak dituzten fitxategien gainean eragiketak egiteko.

Sistema	Subversion	Git	Mercurial
Commit	Bai	Bai	Bai
Berrizendapena	Bai	Partzialki	Bai
Fusioa	Ez	Bai	Bai
Erreplika	Ez	Bai	Bai
Baimenak	Ez	Bai	Bai
Kodifikazioa	Bai	Partzialki	Ez

Taula 2: Software bertsioak egiteko sistemen funtzionalitate espezifikoak

Amaitzeko, sistema bakoitzean egin daitezkeen eragiketak edo exekutatu daitezkeen komandoak ikertu dira:

- **Init**: biltegi berri bat sortzeko.

- **Clone**: existitzen den biltegi baten kopia zehatza egiteko.

- **Pull**: urruneko biltegi batean dauden aldaketak lokalera jaisteko.

- **Push**: lokalean egindako aldaketak urruneko biltegira igotzeko.

- **Branch**: adar bat sortzeko.

- **Checkout**: biltegi lokal bat sortzeko.

- **Update**: biltegi lokala eguneratzeko urruneko biltegi batetik abiatuz.

- **Lock**: biltegi bateko fitxategiak blokeatu beste erabiltzaile batek ez aldatzeko.

- **Add**: hurrengo commit-ean biltegira gehitu behar diren fitxategiak markatzeko.
- **Remove**: hurrengo commit-ean biltegitik ezabatu behar diren fitxategiak markatzeko.
- **Move**: hurrengo commit-ean biltegiko beste direktorio batetara mugitu behar diren fitxategiak markatzeko.
- **Copy**: hurrengo commit-ean kopiatu behar diren fitxategiak markatzeko.
- **Merge**: fitxategi baten bi bertsioen arteko desberdintasunak aplikatu edota fusionatzeko.
- **Commit**: aldaketak biltegian gordetzeko.
- **Revert**: kopia lokala berrezartzeko biltegitik abiatuz.

Sistema	Subversion	Git	Mercurial
Init	Bai	Bai	Bai
Clone	Bai	Bai	Bai
Pull	Bai	Bai	Bai
Push	Bai	Bai	Bai
Branch	Ez	Bai	Bai
Checkout	Bai	Bai	Bai
Update	Bai	Bai	Bai
Lock	Bai	Bai	Bai
Add	Bai	Bai	Bai
Remove	Bai	Bai	Bai
Move	Bai	Bai	Bai
Copy	Bai	Bai	Bai
Merge	Bai	Bai	Bai
Commit	Bai	Bai	Bai
Revert	Ez	Bai	Bai

Taula 3: Software bertsioak egiteko sistemek exekutatu ditzaketen komandoak

7.1.2. Emaitza

Emaitza bat lortzeko, sistemen arteko ezberdintasunetan erreparatu da, ezaugarri bakoitzari garrantzi bat emanez. Ondorengo taulan azter daiteke ezaugarri bakoitzari emandako garrantzia:

Ezaugarria	Garrantzia	Subversion	Git	Mercurial
Biltegi mota	Deszentralizatuak 15	0	15	15
LMU/CMM	CMMk 10	10	10	10
Berrizendapena	10	10	5	10
Fusioa	10	0	10	10
Erreplika	10	0	10	10
Baimenak	10	0	10	10
Kodifikazioa	10	10	5	0
Branch	10	0	10	10
Revert	10	0	10	10
TOTALA	95	30	85	85

Taula 4: Software bertsioak egiteko sistemen konparaketaren emaitza

Egin daitezkeen eragiketetan eta funtzionalitateetan erreparatuz gero, argi ikusten da Mercurial eta Git, Subversion baino pauso bat aurrerago daudela. Izan ere, bi horiek Subversion baino berriagoak dira. Gainera bi hauek deszentralizatutako sistemak dira, horregatik aukeraketa bi hauen artean geratzen da.

Git eta Mercurialen arteko ezberdintasunak oso txikiak dira, beraz, bat edo bestea aukeratzean ez da aldaketa handirik egoten. Baina bietako bat aukeratu behar denez, Git suertatzen da garaile. Honen zergatia, programatzaileen artean zabalduago dagoela eta merkatuaren joera Git-era jotzea delako da.

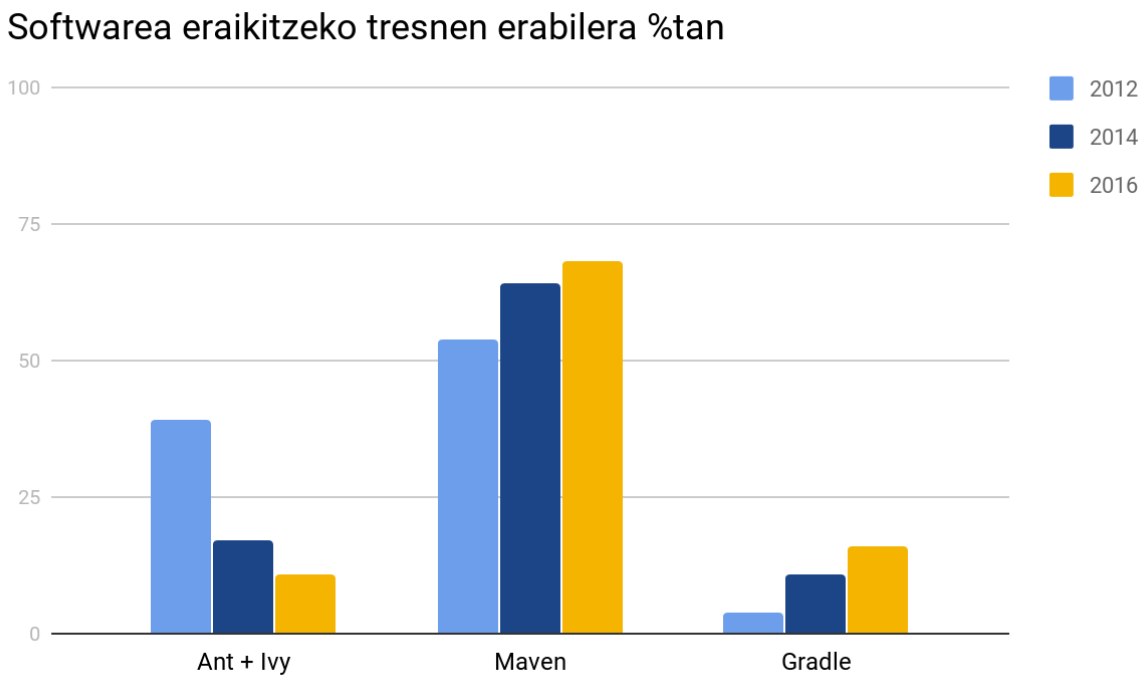
Behin software bertsioak egiteko sistema aukeratuta, hurrengo pausoa sistema hori inplementatzen duen biltegi bat aukeratzea izango da. Kasu honetan, Git inplementatzen duen biltegi bat izan behar da.

Biltegi hau garapen-ingurunean instalatuko da, izan ere, proiektuak pribatuak izanik, ezin dira proiektu hauek biltegi publiko batetara igo, ezta Interneten hirugarren enpresa baten eskuetan dagoen biltegi pribatuetara. Biltegi honek, bezeroaren proiektu guztien iturri-kodea biltegiratzeko balioko du. Hortaz, programatzaileek proiektuen iturri-kodea biltegi honetatik jaitsi eta egindako aldaketak biltegi honetara igo beharko dituztelarik. Hau da, programatzaileek elkarrekintza zuzena izango dute biltegi honekin.

Baldintza eta arrazoi horiek kontutan hartuta, garapen-ingurune honetarako biltegirik egokiena Gitlab [34] dela ondorioztatzen da. Beraz, garapen-ingurune honetan Gitlab instalatuko da.

7.2. *Software* proiektuen eraikuntza konfiguratzeko tresna

Gaur egun, artearen egoera aztertu den atalean esan den bezala, Java proiektuak eraikitzeko tresnen artean hiru tresnek menperatzen dute merkatua: Ant + Ivy, Maven eta Gradle. Hori dela eta, proiektu honetan eraiki behar den garapen-ingurunea Java proiektuentzako zuzendua dagoenez, hiru horien artean egin da azterketa, proiektu honetako helburuak hobekien betetzen dituena aukeratuz.



Grafikoa 1: Softwarea eraikitzeko tresnen erabilera (https://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-2016-trends/)

Tresna bakoitzaren aurkezpena eginda dago jada Artearen egoera atalean (5.3. atala). Beraz, atal honetan, zuzenean erabakitzeko irizpide batzuk aukeratu dira eta horren arabera elkarren artean konparatu dira. Horrela, erabaki zuzen bat hartu ahal izan da amaieran.

Horretarako, tresna bakoitzaren errendimendu datuak *zeroturnaround.com* [35] orrialdeko azterketa batetik hartu dira. Izan ere, azterketa egoki bat egiteko behar den informazio guztia lortzeko egin beharko liratekeen probek denbora handiegia eskatzen dute, proiektu honen helburutik kanpo geratzen delarik. Beraz, atal honetan egin den azterketa orrialde horretan egindakoan oinarritzen da, betiere garrantzia proiektu honetarako egokiena denari ematen eta modu berean, ondorioak egokituz.

7.2.1. Aukeraketa irizpideak

Tresna bat edo bestea aukeratzeko ondorengo ezaugarriak hartu dira kontutan:

- Ikasteko zailtasun kurba
- Software proiektu bat eraikitzeko behar duen denbora, fitxategi-bitarra sortuz

- Konfigurazio fitxategia egiteko eta mantentzeko konplexutasuna
- Zenbat plugin existitzen diren eta plugin berriak sortzeko zailtasuna
- Daukan dokumentazioa eta atzean duen komunitatea
- Programatzaileen tresna ezberdinekin (IDE, aplikazioen zerbitzariak eta etengabeko integrazioko sistemak) integratzeko gaitasuna

Ezaugarri horiek kontutan hartuz, ondorengo taula lortu da, balio handiagoak ezaugarri hobea adierazten dutelarik. Puntuaziorik altuena 5 da eta baxuena 0.

Ezaugarriak	Ant + Ivy	Maven	Gradle
Ikasteko zailtasun kurba	3	3	4
Eraikitzeko denbora	3.5	4.5	4.5
Konplexutasuna	3	1.5	4.5
Pluginak	3	4	3
Dokumentazioa eta komunitatea	2	3	5
Garapen tresnekin integrazioa	4	5	3
TOTALA	18.5	21	24

Taula 5: Softwarea eraikitzeko tresnen ezaugarriak

7.2.2. Emaitza

Aukeraketa hau egiteko erabilitako ikerketan kasu ezberdin batzuk aztertu dira. Hau da, tresna bakoitza zein inguruneta erabiltzen denaren arabera azterketa bat egin da. Kasu hauek ondorengoak dira: tresna hobby gisa programatzeko erabiltzen denean, tresna Open Source proiektuak garatzeko erabiltzen denean, tresna tamaina ertaineko enpresetan erabiltzen denean, eta amaitzeko, tresna enpresa handiek erabiltzen dutenean.

Kasu ezberdinak direla eta, kasu bakoitzean ezaugarri bakoitzari garrantzi handiagoa edo txikiagoa eman zaio, ezaugarriak garrantzi handia duenean x2 , ertaina duenean x1, eta garrantzi txikia x0.5 eginez.

Proiektu honen bezeroaren kasua aztertuz, kontutan edukiz programatzen diren liburutegi, aplikazio eta horretarako lanean aritzen diren programatzaile kopuruak (bai bezeroarenak, baita azpikontratatzen dituenak ere), enpresa handitzat hartu da. Ondorioz, ezaugarri bakoitzari kasu horren araberako garrantzia eman zaio, ondorengo taula lortuz:

Garrantzia	Ezaugarriak	Ant + Ivy	Maven	Gradle
Txikia (2.5)	Ikasteko zailtasun kurba	1.5	1.5	2
Handia (10)	Eraikitzeko denbora	7	9	9
Ertaina (5)	Konplexutasuna	3	1.5	4.5
Ertaina (5)	Pluginak	3	4	3
Txikia (2.5)	Dokumentazioa eta komunitatea	1	1.5	2.5
Handia (10)	Garapen tresnekin integrazioa	8	10	6
35	TOTALA	23.5	27.5	27

Taula 6: Softwarea eraikitzeko tresnen konparaketaren emaitza

Taulak erreparatuz, gaur egun tresnarik egokienetakoa Gradle erabiltzea badirudi ere, enpresa handi baten ikuspuntutik, Maven ateratzen da garaile. Horrek ez du esan nahi Gradle aukera txarra izango litzatekeenik, baina oraingoz ez da Mavenek eman dezakeen egonkortasuna emateko gai.

Beraz, proiektu honetarako aukeratu den tresna Maven izan da.

7.3. Etengabeko integrazioa inplementatzen duten sistemak

Atal honetan, proiektu honetan diseinatuko den garapen-ingurunean etengabeko integrazioa inplementatzeko sistema aukeratu da. Etengabeko integrazioa inplementatzen duten sistema ugari dago merkatuan, baina denak ikertzea bideraezina denez, proiektu honetan, ezagunenak, erabilienak edota interes berezia pizten dutenak aztertu dira. Betiere, aurreko ataletan aukeratu diren tresna eta sistemekin bateragarriak direnak aztertu dira, gainontzekoak baztertuz.

Beraz, aztertu diren sistemak ondorengoak izan dira:

- Jenkins [29]
- Travis CI [31]
- Gitlab CI [36]

Hiru horiez gain, Hudson [30] izeneko beste sistema ezagun bat existitzen da. Baina kontutan ez hartzeko arrazoi batzuk daude. Izan ere, Jenkins, Hudsonen *fork* bat da, hau da, hasiera batean Hudson proiektua existitzen zen eta momentu batean proiektuak bi bide hartu zituen, bide horietako bat Jenkins izanik eta bestea Hudson. Honen xehetasun gehiago bibliografian jarritako Jenkinseen orrialdean aurkitu daiteke [37]. Bi proiektuak banandu zirenetik, biak antzekoak izan arren, Jenkins gailendu da, iturri-kode irekia, erabiliagoa eta aukera handiagoak eskaintzen dituena izanik.

7.3.1. Aukeren aurkezpena

7.3.1.1. Jenkins

Merkatuan dagoen sistemarik ezagunenetarikoa eta erabilienetarikoa da. Horrez gain, iturri-kode irekia duen etengabeko integrazioko sistemarik handienetakoa ere bada.

Sistema honekin, proiektuen eraikuntza, testen exekuzioa eta zerbitzarietan publikatzea automatizatu daiteke. Jenkins, besteak beste, Windows, Mac OSX, hainbat Unix eta Linux sistema edota JRE bat dagoen edozein sistematan instalatu daiteke aplikazio gisa [38].

Jenkinsek proiektu baten iturri-kodean aldaketak egiten diren bakoitzean, proiektua eraikitzea eta eraikuntza amaitu ondoren eraikuntzaren emaitza programatzaileari jakinaraztea ahalbidetzen du. Horrela, programatzaileak proiektuaren egoera ezagutu dezake.

Jenkinsen abantaila nagusia bere pluginen eskaintza da. Merkatuan dagoen ia edozein sistema, tresna edo zerbitzu bakoitza Jenkinsekin integratu ahal izateko plugin bat dago. Gainera, iturri-kode irekia izanik, plugin berriak garatu daitezke edozein beharretara egokitzeko. Ondorioz, sistema egokia da edozein egoerarako: bai enpresa txikientzat, baita enpresa handientzat ere.

Beste abantaila bat, doakoa dela da. Abantaila guzti hauek (iturri-kode irekia, pluginak eta doakoa izatea) komunitate handi batek babestea eragin du.

7.3.1.2. Travis CI

Travis CI merkatuan dagoen beste sistema ezagunenetariko bat da. Hasiera batean iturri-kode irekiko proiektuentzako bakarrik erabil bazitekeen ere, gaur egun proiektu pribatuekin ere erabil daiteke.

Travis CI etengabeko entregatzera eta etengabeko publikaziora bideratuta egon baino, gehienbat etengabeko integraziora dago bideratuta. Proiektu baten iturri-kodean aldaketak egiten direnean, Travisek aldaketak detektatu eta proiektua eraikitzen du eta ondoren, aldaketa zuzena izan den edo ez jakinarazten du.

Programatzaileek Travis CI erabil dezakete testak exekutatzen diren bitartean hauen emaitza aztertzeko eta hainbat test aldi berean exekutatzeko. Gainera, sistema ezberdinekin integratu daiteke eraikuntza zuzena ez denean jakinarazteko, adibidez, email bidez.

Hala ere, etengabeko integraziora bideratuta dagoenez (eta ez etengabeko entregatzera edo etengabeko publikaziora), sistema honekin integratu daitezkeen sistemen kopurua mugatua da.

Prezioari dagokionez, Githubeko iturri-kode irekiko proiektuentzako doakoa da. Baina proiektu pribatuekin erabiltzeko 69 $ - 489 $ bitarteko prezioa [39] dauka hilabetean.

Amaitzeko, aipatzekoa da Githuben biltegiratuta dauden proiektuentzako bakarrik balio duela.

7.3.1.3. Gitlab CI

Gitlab berez software bertsioak egiteko sistema edo biltegia da. Baina biltegi-sistema hori atera ondoren, talde berak etengabeko integrazioko sistema bat garatu zuen. Beraz, software bertsioak egiteko aukeratutako sistema Gitlab izanik, talde berdinaren etengabeko integrazioko sistema aztertu da atal honetan.

Proiektuen testak exekutatzeaz gain eta eraikitzeaz gain, sistema honek proiektuak zerbitzarietan instalatu ditzake. Honi esker, etengabeko entregatzea edota etengabeko publikatzea egiteko aukera ematen du.

Normala den bezala, oso ondo integratzen da Gitlab biltegiarekin. Eta Jenkins eta Travis CI sistemekin gertatzen den bezala, proiektu bakoitza eraiki ondoren, sistemak programatzaileari jakinarazten dio ea eraikuntza ondo joan den ala ez.

Prezioari dagokionez, *Community Edition* doakoa da. Baina bertsio hobetuek 4 $ - 99 $ bitarteko prezioa dute hilabetean erabiltzaileko. Gainera, Gitlabekin batera datorrenez, sistema ugaritan instalatu daiteke [40].

Amaitzeko, Gitlab eta Gitlab CI iturri-kode irekiko sistemak dira. Ondorioz, Jenkins bezala, nahieran aldatu daiteke bakoitzaren beharretara egokitzeko.

7.3.2. Aukeraketa irizpideak

Kontutan hartuko diren irizpideak ondorengoak dira:

- Etengabeko integrazioa zein mailatara eramateko gaitasuna eta horretarako eskaintzen duten erraztasuna. Hau da, etengabeko integrazioa, etengabeko entregatzea eta etengabeko publikazioa egiteko zein aukera eskaintzen duten.

- Mavenekin integratzeko gai diren.

- Plugin kantitatea. Izan ere, plugin aukera geroz eta handiagoa izan, orduan eta errazagoa izango da konfigurazio ezberdinak egitea.

- Bestelako sistemekin integratzeko aukera. Adibidez, proiektu honen bezeroa den enpresak etorkizunean eskakizun berri bat behar badu eta horretarako sistema berri batekin integratu behar bada.

- Sistema ezberdinetan instalatzeko aukera. Hau da, online dagoen aplikazio batean dagoen (bezeroaren proiektuak konfidentzialak direla kontutan hartu behar da, beraz, online dagoen aplikazio batean iturri-kodea eraikitzea saihestu nahi da), JRE batean exekutatu daitekeen aplikazio bat den (aplikazioa JRE batean exekutatu badaiteke, edozein sistemetan muntatu daiteke), edota zein sistema eragilerekin funtziona dezakeen.

- Prezioa

- Konfigurazio aukerak eta plugin berriak garatu daitezkeen. Hau da, iturri-kode irekikoa den edo sistema itxi bat den.

Ezaugarri horiek haintzat hartuz, ondorengo taula osatu da, 3 puntuaziorik altuena izanik eta 0, txikiena:

Ezaugarria	Jenkins	Travis CI	Gitlab CI
Etengabeko integrazioaren aukerak	3	1	2
Mavenekin integrazioa	3	3	3
Plugin kantitatea	3	1	1
Sistema ezberdinekin integratzeko aukera	3	1	1
Instalazio aukerak	3	0	3
Prezioa	2	1	2
Konfigurazio aukerak	3	1	3

Taula 7: Etengabeko integrazioa inplementatzen duten sistemen ezaugarriak

7.3.3. Emaitza

Emaitza lortzeko, ezaugarri bakoitzak zenbateko garrantzia duen adierazi da. Horretarako, garrantzi handia duten ezaugarriei x2, ertaina dutenei x1, eta garrantzi txikia dutenei x0.5 egin zaie. Ondorioz, ondorengo taula lortu da:

Garrantzia	Ezaugarria	Jenkins	Travis CI	Gitlab CI
Ertaina (3)	Etengabeko integrazioaren aukerak	3	1	2
Handia (6)	Mavenekin integrazioa	6	6	6
Handia (6)	Plugin kantitatea	6	2	2
Txikia (1.5)	Sistema ezberdinekin integratzeko aukera	1.5	0.5	0.5
Ertaina (3)	Instalazio aukerak	3	0	3
Ertaina (3)	Prezioa	2	1	2
Handia (6)	Konfigurazio aukerak	6	2	6
28.5	TOTALA	27.5	12.5	21.5

Taula 8: Etengabeko integrazioa inplementatzen duten sistemen konparaketaren emaitza

Puntuazioei erreparatuz, Travis CI beste bien oso atzetik geratu dela azter daiteke. Honen arrazoi nagusia, sistema hau Githubekin erabiltzeko pentsatuta dagoelako da, gehienbat pribatuak ez diren proiektuekin. Eta ez enpresa handiko proiektuentzat.

Gitlab CI eta Jenkinsen artean ere aldea dago. Honen zergatia, Jenkins programatzaileen komunitatean oso zabalduta dagoelako da. Ondorioz, Jenkinsek eskaintzen dituen aukerak amaigabeak dira, eta horretarako dagoeneko garatuta dituen pluginak ere oso ugariak dira.

Horregatik, hiru aukeren artean, proiektu honetara ondoen egokitzen den sistema Jenkins dela ondorioztatu da.

8. Arriskuen analisia

Atal honetan proiektuaren arriskuak identifikatuko dira, ondoren arrisku horien aurrean plan bat prestatuta edukitzeko. Plan hauen helburua, arrisku bakoitzaren inpaktua ahal den heinean minimizatzea da.

8.1. Arriskuen identifikazioa

Arriskuak bi taldetan sailkatzen dira iturriaren arabera. Arriskua proiektu bertatik badator, barneko arrisku bat izango da. Bestela, arriskua kanpokotzat joko da.

8.1.1. Barneko arriskuak

1. Proiektuaren kostua igotzea

2. Proiektua atzeratzea

3. Errendimendu-eskakizunak ez betetzea

8.1.2. Kanpoko arriskuak

1. Bezeroaren arkitekturako arauak aldatzea

2. Programatzaileentzat garapen inguruneko tresna berriak erabiltzen ikasteko zailtasuna handia izatea: Git, Maven, Jenkins

3. Garapen-ingurunearen helburuak hobeto betetzen dituzten sistema, tresna edota aplikazio berriak agertzea.

8.2. Arriskuen probabilitatea eta inpaktua

Atal honetan aurreko atalean identifikatutako arriskuak gertatzeko probabilitatea eta izan dezaketen inpaktua aztertuko da.

8.2.1. Barneko arriskuak

1. Proiektuaren kostua igotzea

Gerta daiteke proiektuaren exekuzioan zehar proiektuaren kostua igotzen duten faktoreak agertzea. Proiektuaren kostua gehien handitu dezakeen faktorea proiektua atzeratzea da. Hala ere, proiektuaren kostua igotzeko probabilitatea **txikia** da, proiektuaren aurrekontua kontu handiz egin baita eta proiektuaren plangintza egiterakoan, ataza bakoitzari denbora soberan eman baitzaio. Arrisku hau gertatzeak izango duen inpaktua **ertaina** dela aurreikusten da.

2. Proiektua atzeratzea

Proiektu baten exekuzioan zehar arazo batzuk agertu daitezke, proiektuaren iraupena hasiera batean kalkulatutakoa luzatzen dutenak. Arrisku hau gertatzeko probabilitatea nahiko **txikia** da. Izan ere, aurretik aipatu den bezala, proiektuaren plangintzako ataza bakoitzari denbora nahikoa eta soberan eman zaie. Hala ere, proiektua atzeratzeko arrisku handiena errendimendu-eskakizunak betetzen direla egiaztatzeko egin behar diren probetan dago.

Arrisku honek duen inpaktua **handia** da, arrisku hau betetzeak kanpoko arriskuetatik hirugarrena gauzatzeko probabilitatea handitzen duelako. Hala ere, helburuak betetzeko sistema edo aplikazio

berriago eta hobeagoak agertzeak ez du esan nahi egindako diseinuak balio ez duenik. Horrez gain, proiektu bat atzeratzeak proiektuaren kostua igotzea dakar. Batik-bat, proiektuaren aurrekontuko alderik garestiena barne-orduak direlako, eta proiektua atzeratuz gero, barne-ordu gehiago sartu behar direla suposatzen baitu.

3. **Errendimendu-eskakizunak ez betetzea**

Proiektu honen helburuetako bat garapen-inguruneak errendimendu-eskakizun batzuk betetzea da. Hori bermatzeko, nahitaezkoa izango da proba ugari egitea. Izan ere, ia ezinezkoa da lehenengoan errendimendu-eskakizun horiek betetzea. Horregatik, hau gertatzeko probabilitatea **handia** da.

Proiektuaren plangintzan aurreikusita dago errendimendu-eskakizun horiek lehenengoan ez betetzea, hala ere, inpaktu **handia** izango luke. Izan ere, errendimendu-eskakizunak bete ahal izateko, garapen-ingurunearen diseinuan aldaketak egin behar izatea eragin dezake. Horrek, proiektua atzeratzea ekar dezake, eta ondorioz, proiektuaren kostua ere igotzea.

8.2.2. Kanpoko arriskuak

1. **Bezeroaren arkitekturako arauak aldatzea**

Proiektu honetan garatzen den garapen-ingurunearen helburuetako bat, garapen-ingurune horretan garatzen diren proiektuek, bezeroaren arkitekturako arauak betetzen dituztela bermatzea da. Arau horiek orokorrean egonkorrak dira, hau da, oso arraroa da arkitekturan aldaketak egotea. Horregatik, hau gertatzeko probabilitatea oso **txikia** da.

Hala ere, arauak aldatuko balira, aldatzen den arauaren araberakoa izango da inpaktua, baina inpaktu hori **handia** izan daiteke.

2. **Programatzaileentzat, garapen inguruneko tresna berriak, erabiltzen ikasteko zailtasuna handia izatea: Git, Maven, Jenkins**

Garapen-inguruneak hainbat elementu berri ditu programatzaileentzat. Hauen erabilera ez da guztiz gardena, beraz programatzaileek nahitaezkoa dute ikasketa prozesu kbat igarotzea garapen-ingurunea modu egoki batean erabili ahal izateko. Prozesu horretan, gerta daiteke programatzaile batzuentzat garapen-inguruneko elementuak erabiltzen ikastea zailagoa izatea.

Hori gertatzeko probabilitatea **txikia** da, alternatiben analisian, tresna bakoitza erabiltzen ikasteko zailtasuna maila aukeraketa egiteko irizpideetako bat izan baita. Eta kasu bakoitzean, izango duen eragina **txikia** izango da.

3. **Garapen-ingurunearen helburuak hobeto betetzen dituzten sistema, tresna edota aplikazio berriak agertzea**

Alternatiben analisian egin den azterketari esker, garapen-inguruneak izango dituen aplikazio, tresna eta sistemak aukeratu dira. Aukeraketa hori egiteko irizpide batzuk jarraitu dira, kasu bakoitzean, azterketa egiterako momentuan zegoen aukera egokiena aukeratuz. Hala ere, gerta daiteke proiektu hau gauzatzen den bitartean elementu horien bertsio berriak edota alternatiba berriak agertzea, proiektu honen helburuak betetzeko egokiagoak direnak.

Arrisku hau gertatzeko probabilitatea **ertaina** da, baina honen eragina **txikia** da. Izan ere, nahiz eta helburuak hobeto edo errazago betetzen dituen sistema, tresna edo aplikazio bat agertu, horrek ez du esan nahi dagoeneko aukeratuta dagoen sistema, tresna edo aplikazioa ere horretarako ez denik egokia.

8.3. Inpaktu-probabilitate matrizea

Aztertu diren arriskuak matrize batean kokatu dira, inpaktua ekidin edo txikiagotzeko lehentasuna zeintzuk duten jakiteko. Horrela, arrisku bat matrizean geroz eta kolore gorritik hurbilago egon, arrisku horrekiko kontingentzia plan bat egiteko behar handiago bat egongo da. Aldiz, kolore gorritik urrunduz kolore berdera hurbilduz, kontingentzia plan bat prest edukitzeko beharra txikituko da.

Matrizean arriskuak identifikatzeko zenbakiak erabili dira, zenbaki bakoitzak ondorengo arriskua identifikatzen duelarik:

1. Proiektuaren kostua igotzea
2. Proiektua atzeratzea
3. Errendimendu-eskakizunak ez betetzea
4. Bezeroaren arkitekturako arauak aldatzea
5. Programatzaileentzat garapen inguruneko tresna berriak erabiltzen ikasteko zailtasuna handia izatea: Git, Maven, Jenkins
6. Garapen-ingurunearen helburuak hobeto betetzen dituzten sistema, tresna edota aplikazio berriak agertzea.

	Inpaktua			
Probabilitatea		Txikia	Ertaina	Handia
	Txikia	5	1	2, 4
	Ertaina	6		
	Handia			3

Taula 9: Arrisku bakoitza gertatzeko probabilitatea eta bakoitzak duen inpaktua erlazionatzen dituen matrizea

8.4. Kontingentzia plana

Kontingentzia planari esker identifikatutako arriskuak nola saihestuko diren zehaztu da. Atal honetan proposatutako planak lan gehigarri bat suposatuko du, baina honi esker proiektuaren garapena zuzena izatea lortuko da.

Inpaktu eta gertatzeko probabilitate txikia duten arriskuentzako kontingentzia planik ez da egingo, ez baitu merezi. Ondorioz, 1, 2, 3, 4 eta 6 arriskuen kontingentzia plana egin da.

1. Proiektuaren kostua igotzea

Arrisku hau saihesteko % 10 gehitu zaio proiektuaren aurrekontuari.

2. Proiektua atzeratzea

Arrisku hau saihestearren % 10eko marjina eman zaie kritikoak diren ataza bakoitzaren iraupenari.

3. **Errendimendu-eskakizunak ez betetzea**

Errendimendu-eskakizun handiegiak lortu behar izateak proiektu honen helburuak inoiz bete ahal ez izatea eraman dezake. Eta ondorioz, proiektua ezin bukatu ahal izatea. Hori saihesteko, proiektu honen errendimendu-eskakizunak kontu handiz definitu dira, lor daitezkeen balioak definituz. Gainera, balio horiek lortu ahal izateko aukera asko ematen dituzten teknologia eta aplikazioak aukeratu dira. Adibidez, Jenkinsek *master-slave* arkitektura onartzen du, Jenkins aplikazioak exekutatu behar dituen zereginak beste makina batek exekutatu ditzan, eta horrela, Jenkins aplikazioa lan-zama txikituz.

Beraz, errendimendu-eskakizunak bete ahal izateko, hardwarea handitu ahal izango da, kasuan-kasu, bertikalki eskalatuz edo horizontalki.

4. **Bezeroaren arkitekturako arauak aldatzea**

Kasu honetan bi gauza gertatu daitezke: bezeroak arau berriak gehitzea edo aurretik definituta dagoen arau bat aldatzea. Bi kasuetan, inpaktua arauaren araberakoa da, baina handia izan daiteke. Ondorioz, proiektu honetan hasieran definitutako arauak beteko dira. Arauetan egindako aldaketak edota arau berriak, berriz, proiektu honen gehigarria izan daitekeen proiektu berri baterako utziko dira, proiektu honen mantentze-lanez arduratuko dena.

5. **Garapen-ingurunearen helburuak hobeto betetzen dituzten sistema edota aplikazio berriak agertzea**

Aurretik aipatu den bezala, arrisku honen inpaktua txikia da. Izan ere, alternatiben analisian aukeratu diren sistema, tresna eta aplikazioek proiektu honen helburuak betetzeko egokiak direlako aukeratu dira. Horregatik, kasu honetan ere, proiektu hau garatzen ari den bitartean ager daitezkeen sistema edota aplikazio berrien azterketa proiektu honen gehigarria izan daitekeen proiektu berri baterako utziko da.

9. Diseinua

Goi mailako diseinuaren atalean aztertu da proiektu honetan konpondu nahi diren arazoak eta bete nahi diren helburuak lortzeko erabili behar diren tresna eta aplikazioak zeintzuk diren. Atal honetan berriz, atal horretan ikusitakoari garapen-ingurune baten forma emango zaio diseinu baten bidez. Horretarako, alternatiben analisian aukeratutako tresnez baliatuz.

9.1. Gitlab

Hasteko, software proiektuak biltegiratzeko, software bertsioak egiteko sistema bat erabiliko da. Alternatiben analisian aztertu den bezala, sistema hau Gitlab izeneko aplikazioaren bidez inplementatuko da, bezeroaren zerbitzari batean instalatuko dena.

Programatzerako orduan, programatzaileek garapen inguruneko elementu honekin zuzenean komunikatuko dira. Programatzaile batek software proiektu bat garatu behar duenean, lehenengo aldian pauso batzuk jarraitu beharko ditu. Pauso hauek jarraitzeko beharra, Gitlabek Git sistema erabiltzen duelako da, eta Git sistema deszentralizatu bat delako da.

9.1.1. Proiektu bat garatzeko lehenengo pausoak

Lehenengo aldian jarraitu behar diren pausoak proiektua dagoeneko existitzen den edo berria denaren arabera aldatzen da.

9.1.1.1. Proiektua berria bada

Proiektua berria bada, proiektu berriaren iturri-kodearekin ingurune-lokalena biltegi-lokal bat sortu behar da. Behin biltegi-lokala sortuta dagoelarik, hurrengo pausoa, garapen-inguruneko Gitlab sisteman proiektu berri horrentzako biltegi bat sortzea da.

Bi biltegiak sortuta daudelarik, azken pausoa, bi biltegiak erlazionatzea eta biltegi-lokalean dagoen proiektuaren iturri-kodea Gitlabeko biltegira igotzea da. Honi esker, beste edozein programatzailerentzako eskuragarri egongo da.

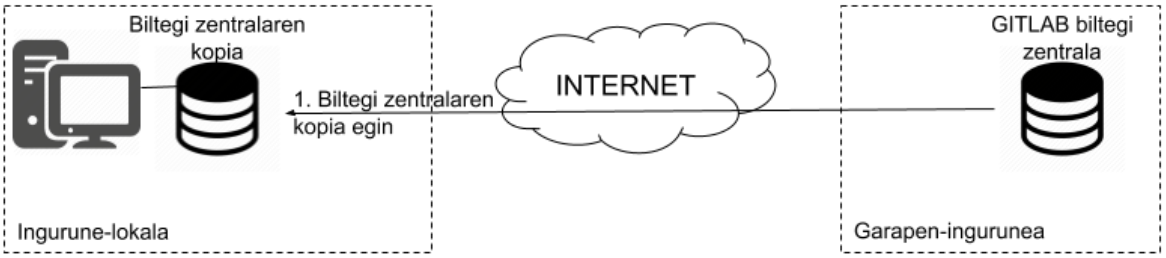
Kontutan eduki behar da Gitlabeko biltegiari ematen zaion izena. Izan ere, bezeroaren arkitekturak proiektuen izenen formatua definitzen du. Eta aurrerago ikusiko den bezala, garapen-inguruneak ondo funtziona dezan, garrantzitsua izango da Gitlabeko proiektu baten biltegiak proiektuaren izen bera edukitzea. Batez ere, Gitlab Jenkinsekin komunikatu dadin.

Aipatzekoa da, pauso hauek (biltegiaren izena kontutan edukitzea izan ezik) Giteko edozein sistemetan jarraitu beharreko pausoak direla, eta ez direla proiektu honetarako espezifikoak. Hala ere, bezeroak orain arte SVN biltegiak erabiltzen zituenez, Git biltegietan jarraitu beharreko pausoak SVN biltegikoekin alderatuz ezberdinak direlako azaldu da prozesua.

9.1.1.2. Proiektua dagoeneko Gitlaben existitzen bada

Proiektua dagoeneko Gitlabeko biltegian existitzen bada, jarraitu beharreko pausoak ez dira proiektu honetarako diseinatutakoak, hau da, edozein Git sistematan jarraitu beharrekoak dira. Horregatik, labur-labur azalduko da, gehiegi sakondu gabe.

Ingurune-lokalean Giteko komandoak erabiliz, biltegi lokalera jaitsi nahi den proiektuaren biltegiaren kopia bat egin behar da. Behin hori eginda, proiektuaren iturri-kodea ingurune-lokalean egongo da, programatzaileak proiektua garatzeko prest.



Irudia 7: Gitlabeko proiektu baten biltegiaren kopia egin ingurune-lokalean

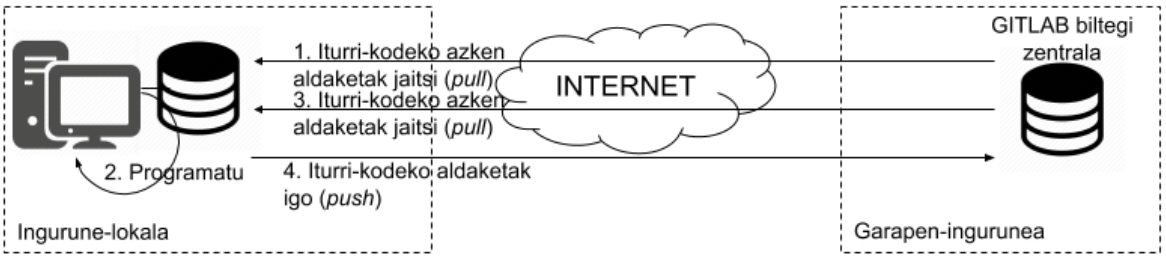
9.1.2. Dagoeneko ingurune-lokalean biltegiratuta dagoen proiektu bat garatzea

Behin garatu nahi den proiektuaren biltegiaren kopia ingurune-lokalean izanda, lehenengo pausoa beti Gitlabeko biltegi zentralean dauden aldaketa berriak ingurune-lokaleko biltegira jaistea izango da, Giteko komandoetan *pull* izenez ezagutzen dena. Honen zergatia, proiektu bera beste programatzaile batzuk ere garatzen aritu daitezkeelako da, beraz, beharrezkoa izango da gainontzeko programatzaileek biltegi zentralera igotako aldaketak bakoitzaren biltegi-lokalera jaistea.

Behin iturri-kodeko aldaketa guztiak ingurune-lokalean edukita, proiektua garatzeko prest egongo da. Horrela, programatzaileak proiektua garatu eta egindako aldaketak probatu ahalko ditu bere ingurune-lokalean.

Ondoren, egindako aldaketak programatzileak ontzat ematen dituenean, aldaketa horiek Gitlabeko biltegi zentralera igo beharko ditu. Horretarako, beste behin ere, biltegi zentralean aldaketa berririk dagoen begiratu eta jaitsi egin beharko ditu. Aldaketarik badago, aldaketa horiek programatzaileak egindakoekin bateratu beharko ditu. Bateratu ondoren edo aldaketarik ez badago, programatzaileak egindako aldaketak biltegi zentralera igo beharko ditu, Giteko *push* izenez ezagutzen den komandoa erabiliz.

Prozesu hau ondorengo irudian azter daiteke:



Irudia 8: Proiektu bat ingurune-lokalean garatzea eta Gitlabeko biltegi zentralera igotzea

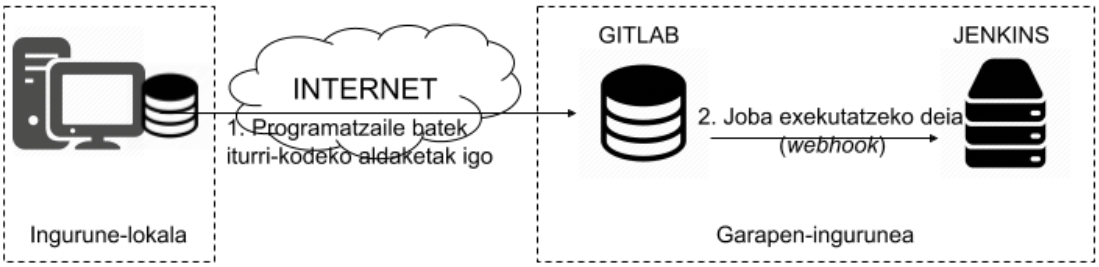
Programatzeko modu hau bezeroarentzat berria bada ere, Git sistemetan proiektu bera hainbat programatzaileek garatzeko prozesu ohikoa da. Hau da, ez da proiektu honentzako espezifikoa. Baina garrantzitsua da prozesu hau nolakoa den zehaztea, programatzaileek garapen-ingurunearekin izango duten harreman nagusia adierazten baita.

9.1.3. Proiektuak automatikoki eraikitzeko deia

Orain arte, programatzaileek Gitlab sistemarekin izango duten harremana aipatu da. Baina proiektu honen helburuetan aipatu den bezala, proiektu honetako garapen-ingurunea automatizatua izan behar da. Garapen-ingurunea automatizatzeko, Gitlabeko proiektu bakoitzaren biltegian *webhook*ak konfiguratu daitezke.

Webhook [41] bat gertaera bat gertatzen denean gertaera horren erantzun gisa exekutatzen den HTTP eskaera da.

Proiektuak eraikitzeko Jenkins aplikazioa erabiliko da, beraz, Gitlabeko proiektu bakoitzaren biltegian iturri-kodeko aldaketa bat jasotzen den bakoitzean *webhook* bat exekutatzea konfiguratu behar da. *Webhook* horrek Jenkinseko URL batetara egingo HTTP POST eskaera bat, eta eskaera horretan Gitlabek bidaltzen duen informazioa erabiliko da Jenkinsek dei horrekin zer egin behar duen jakiteko.



Irudia 9: Proiektu bat Jenkinsen eraikitzeko deia egitea

Oraingoz, Gitlabeko atalari dagokionez, proiektu bakoitzeko iturri-kodean aldaketa bat egiten denean Jenkins aplikaziora *webhook* bidez dei bat bidaltzen dela jakitearekin nahikoa da. Aurreragoko Jenkinsen atalean azalduko da zer egin behar duen Jenkinsek dei horrekin.

9.2. Maven

Atal honetan, software proiektuen eraikuntza konfiguratzeko tresna garapen-ingurunean nola erabiliko den eta zein egokitzapen egin behar zaizkion aztertuko da.

9.2.1. Mavenen funtzionamendua

Garapen-ingurunean garatuko diren software proiektuak eraikitzeko Maven [24] tresna erabiliko da. Baina garapen-ingurunean nola erabiliko den zehaztu aurretik, beharrezkoa da Mavenek nola funtzionatzen duen eta zer egiteko gai den zehazki jakitea. Horregatik, azpiatal honetan Mavenen funtzionatzeko modua aztertuko da.

Software proiektu bat Maven tresnaren bidez eraikitzeko, proiektuak estruktura [42] jakin bat izan behar du. Xehetasunetan gehiegi sakondu gabe, fitxategi bakoitza dagokion direktorioan ezarri behar da Mavenek proiektua konpilatzerakoan eta proiektuaren fitxategi-bitarra sortzerakoan kontutan har dezan. Horrez gain, aipatzekoa da Maven bidez eraikitzen diren proiektu guztiek pom.xml izeneko fitxategi bat izan behar dutela erroan. pom.xml [43] (ingelesezko *Project Object Model*-etik dator) fitxategian software proiektuak behar dituen liburutegiak eta proiektua eraikitzeko konfigurazioa idazten da, XML formatua erabiliz.

9.2.1.1. Mavenen bizi-zikloa

pom.xml izeneko fitxategian sakondu aurretik, Mavenek proiektu bat eraikitzerakoan duen bizi-zikloa [27] ulertu behar da. Esan bezala, Mavenek proiektu bat eraikitzeko bizi-ziklo bat dauka, eta bizi-ziklo hori hainbat fasetan banatzen da. Fase horiek ondorengoak dira, exekutatzen diren ordenean:

1. *validate*: proiektuaren pom.xml fitxategia egokia dela eta beharrezko informazio guztia dagoela egiaztatzen da fase honetan.

2. *compile*: proiektuaren iturri-kodea konpilatzen da.

3. *test*: testak exekutatzen dira.

4. *package*: konpilatutako kodea hartu eta definitutako formatuan paketatzen da fitxategi-bitarra sortuz. Adibidez, jar fitxategi bat sortzea.

5. *verify*: sortutako fitxategi-bitarra zuzena dela egiaztatzen da, adibidez integrazio-testak exekutatuz.

6. *install*: biltegi lokalean instalatzen da paketea (Maveneko biltegiak aurrerago azaldu dira).

7. *deploy*: Maveneko urruneko biltegian instalatzen da paketea.

Mavenek exekutatzen dituen prozesu guztiak plugin [44] bidez egiten ditu. Hau da, iturri-kodea konpilatzeko *Maven Compiler Plugin* izeneko plugin bat dauka adibidez, testak exekutatzeko *Maven Surefire Plugin*, etab. Horregatik, Mavenek bizi-zikloko fase bakoitzean lehenetsitako plugin batzuk ditu konfiguratuta, eta horri esker Maven proiektu bat eraikitzeko gai da.

Proiektu bati prozesu bat Maven bidez exekutatu nahi bazaio, bi aukera daude. Lehenengoa, komando bidez esplizituki prozesu hori exekutatzen duen plugina adieraztea eta exekutatzea da. Kasu horretan, plugin hori bakarrik exekutatuko da. Beste aukera, komando bidez plugina exekutatu beharrean, fase bat exekutatzea da. Kasu horretan, ordenean bizi-zikloko fase guztiak exekutatuko ditu adierazitako faseraino, fase bakoitzean konfiguratuta dituen plugin guztiak exekutatuz: bai lehenetsitakoak eta baita pom.xml-an adierazitakoak ere.

9.2.1.2. Maven biltegiak

Bizi-zikloko *install* eta *deploy* faseetan Maveneko biltegiak aipatu dira. Maveneko biltegietan liburutegi eta aplikazioen fitxategi-bitarrak biltegiratzen dira, adibidez, jar, war eta ear fixategiak. Hala ere, normalean liburutegiak biltegiratzeko erabiltzen dira. Honi esker, proiektu bat konpilatzeko eta exekutatzeko liburutegi bat behar denean, liburutegi hori Maveneko biltegietatik kontsumitu daiteke. Horretarako, proiektuaren pom.xml fitxategian proiektuak behar dituen liburutegiak adieraztearekin nahikoa da. Horrela, Maven automatikoki liburutegiak biltegietan bilatzeaz eta deskargatzeaz arduratzen da, ondoren proiektua exekutatzeko, konpilatzeko edo fitxategi-bitarra sortzeko.

Aurretik aipatu den bezala, bi biltegi mota daude: biltegi lokalak eta urruneko biltegiak. Urruneko biltegiak pribatuak edo publikoak izan daitezke. Publikoetan *open source* liburutegiak egon ohi dira, eta edozeinek erabil ditzake bertan dauden liburutegiak. Biltegi pribatuak berriz, sarbidea duten programatzaileek erabil ditzakete soilik, eta bertan pribatuak diren proiektuen liburutegiak egon ohi dira. Adibidez, proiektu honetako bezeroaren proiektuak pribatuak direnez, ezin dira biltegi publikoetara igo, beraz, biltegi pribatu batetara igo beharko lirateke.

Biltegi lokalak cache memoria batek bezala funtzionatzen du. Programatzaile bakoitzaren ordenagailuan dagoen biltegiak dira, Maven erabiltzerakoan automatikoki sortzen dena. Biltegi honen funtzioa cache memoria bat bezala funtzionatzea da. Hau da, proiektu bat eraikitzeko liburutegi bat behar denean (eta liburutegi hori pom.xml-an adierazten bada), Mavenek lehenengo aldiz liburutegi hori Maveneko instalazioan konfiguratuta duen urruneko biltegitik jaitsiko du biltegi lokalera, eta ondoren biltegi lokaletik kontsumituko du. Behin biltegi lokalean deskargatuta, hortik aurrera Mavenek ez du urruneko biltegira joan beharko liburutegi horren bila.

Biltegien azalpenekin amaitzeko, *install* fasea exekutatzen denean, eraikitako proiektuaren fitxategi-bitarra biltegi lokalean biltegiratzen da. *deploy* fasea exekutatzean berriz, biltegi lokalean instalatzeaz gain (*install* fasea ere exekutatzen baita bizi-zikloari esker), urruneko biltegian ere instalatzen da, betiere beharrezko baimen guztiak baditu.

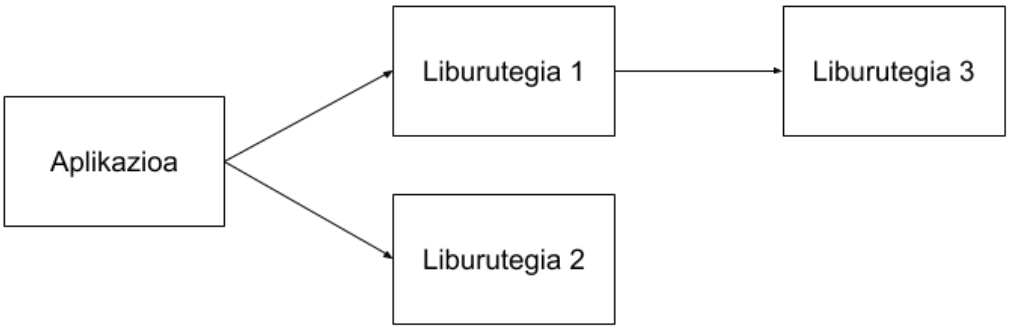
Jarraian, xehetasun gehiagorekin azalduko da Mavenek nola ebazten dituen software proiektu batek dituen menpekotasunak (menpekotasunak software batek konpilatzeko eta exekutatzeko behar dituen liburutegiei deitzen zaie).

9.2.1.3. Mavenen menpekotasunen ebazpena

Mavenek liburutegi bakoitza definitzeko hainbat datu behar ditu: liburutegiaren izena (*artifactId* deitzen zaio), taldearen izena (*groupId* deitzen zaio), bertsioa (*version*) eta menpekotasunak.

Informazio guzti horrekin, software proiektu batek behar dituen menpekotasunen kudeaketa sinplifikatzen eta automatizatzen du Mavenek. Hori ikusteko ondorengo adibidea erabiliko da.

Adibide gisa, suposatu aplikazio batek bi liburutegien menpekotasuna duela. Eta liburutegi horietako batek, hirugarren liburutegi baten beharra duela, ondorengo irudian ikus daitekeen bezala:



Irudia 10: Proiektuen arteko loturaren adibidea

Maven gabe, konpilatzaileari hiru liburutegi horien fitxategi-bitarra pasa behar zaio. Gainera, liburutegien kantitatea igotzen den heinean, liburutegi guztiak konpilatzaileari pasatzea geroz eta konplexuagoa egiten da.

Maveni esker berriz, aplikazioaren pom.xml fitxategian bi liburutegien erreferentzia (*groupId, artifactId* eta *version*) definitzea nahikoa izango litzateke. Izan ere, liburutegi bat identifikatuta, Mavenek liburutegi bakoitzaren menpekotasunen informazioa dauka (liburutegi bakoitzaren pom.xml-an baitago), ondorioz badaki aplikazioa konpilatzeko eta exekutatzeko hirugarren liburutegia behar duela. Beraz, hirugarren liburutegi hori Maveneko biltegietatik automatikoki jaisteko eta erabiltzeko gai da.

Kontzeptuak argi uzte arren, menpekotasun bat definitzeko *groupId, artifactId*, *version* eta liburutegiaren menpekotasunak behar diren bitartean (hau dena, liburutegi bakoitzaren pom.xml-an adierazten da), menpekotasun bat identifikatzeko *groupId, artifactId* eta *version* bakarrik behar dira. Hau da, proiektu batek behar dituen menpekotasunak definitzeko, proiektu horren pom.xml-an menpekotasunen *groupId, artifactId* eta *version* jarri behar da, eta informazio horrekin, definitutako menpekotasun horien menpekotasunak zeintzuk diren kalkulatzeko gai da Maven.

9.2.1.4. pom.xml fitxategien herentzia

Oso ohikoa da software proiektu batzuek konfigurazioaren zati bat komuna izatea. Adibidez, nahiz eta menpekotasun ezberdinak izan, eraikitzerakoan plugin berdinak exekutatzea. Kasu honetan, pom.xml

bat sortzeko aukera dago, zeinetan komuna den konfigurazioa jartzen den. Ondoren, gainontzeko proiektuek pom.xml hori heredatu dezakete, eta horri esker, software proiektu bakoitzaren pom.xml-an konfigurazio bera errepikatzea saihesten da.

Hau baliteke oso erabilgarria ez izatea proiektu gutxi garatu behar direnean. Baina proiektu honetako bezeroak garatzen dituen kantitateen kasuan, arkitekturako arauak bete ahal izateko egin behar den konfigurazioa toki batean egitea ahalbidetzen du. Proiektu guztiak modu berean eraikitzea ere ziurtatu daiteke, proiektuek heredatuko duten pom.xml fitxategia programatzaileen esku uzten ez bada. Horretarako, proiektuek heredatu beharreko pom.xml hori heredatzen dutela ziurtatu beharko da, baina hori aurrerago aztertuko da nola egingo den.

9.2.2. Maven garapen-ingurunean nola inplementatu

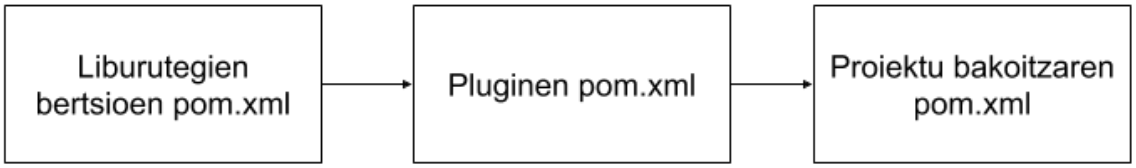
Maven bidez proiektu honetako arazo asko konpondu eta helburu asko beteko dira.

9.2.2.1. Heredatu beharreko pom.xml fitxategien hierarkia

Aurretik aipatu den bezala, pom.xml batek beste pom.xml bateko konfigurazioa heredatu dezake. Mavenen ezaugarri hau oso egokia da proiektu guztien eraikuntza toki bakarrean konfiguratzeko, horrela, proiektu guztientzako konfigurazio bera eginez.

Proiektu honetarako, ezaugarri hau erabiliko da. Alde batetik, proiektu guztiek eraikuntzako konfigurazio bera eduki dezaten, eta bestetik, programatzaileei eraikuntzako konfigurazioa aldatzea galaraziko zaie. Izan ere, programatzaileei ez zaie heredatuko diren pom.xml-ak aldatzeko baimenik emango.

Bezeroaren arkitekturako arauak kontutan hartuz, hiru mailako pom.xml hierarkia bat proposatzen da:



Irudia 11: pom.xml fitxategen herentzia

Lehenengo pom.xml-a (beste pom.xml fitxategirik heredatzen ez duena) zerbitzarian instalatuta dauden liburutegien bertsioak definitzeko erabiliko da. Hau da, bertan bezeroaren arkitekturak zehazten dituen liburutegiak definituko dira. Horrela, pom.xml hau heredatzen duten proiektu guztiek, beraien pom.xml-an, behar dituzten liburutegiak definitu beharko dituzte baino bertsioa zehaztu gabe, liburutegi bakoitzaren bertsioa “liburutegien bertsioen pom.xml”-tik lortuko baitute. Honi esker, proiektu guztiek liburutegien bertsio berak erabiltzen dituztela ziurtatzen da, eta arkitekturak zehazten duen bezala liburutegiak zerbitzari mailan jartzen direnez (gogoratu ez doazela aplikazio bakoitzaren barnean), liburutegi eta aplikazioen arteko bateragarritasuna bermatzen da.

Bigarren pom.xml-ak lehenengoa heredatzen du, horrela, bigarren pom.xml-a heredatzen duten proiektuek lehenengoa ere heredatuko dute. Bigarren pom.xml honetan, proiektuan eraikitzeko konfigurazioa jarriko da, hau da, gehienbat Mavenek bizi-zikloa exekutatzerakoan exekutatu behar dituen pluginak jarriko dira. Heredatzen duen pom.xml-a ez bezala, pom.xml hau beti bera izango da (heredatzen duen pom.xml-an liburutegien bertsioak aldatuko dira, zerbitzarietan instalatzen diren liburutegien arabera).

Azkeneko pom.xml fitxategia, proiektu bakoitzaren pom.xml-a izango da. Hauek, “pluginen pom.xml” fitxategia heredatuz, lehenengo eta bigarren pom.xml-etan jarritako konfigurazio osoa heredatuko dute. Beraz, pom.xml hauetan ondorengo informazioa bakarrik jarri beharko da:

- Proiektuaren *groupId*
- Proiektuaren *artifactId*
- Proiektuaren *version*
- jar edo war proiektu bat den
- Heredatu beharreko pom.xml-a zein den
- Proiektuak behar dituen liburutegiak (bertsiorik gabe)

Gainontzeko konfiguraziorik egiten ez zaie utzi behar programatzaileei, proiektu guztiak modu berean eraiki daitezen eta denek liburutegien bertsio berak erabil ditzaten. Hau ziurtatzeko, pom.xml-ak ondorengo baldintzak betetzen dituela ziurtatu behar da:

- pom.xml bat heredatzen duela, eta heredatu behar duena dela.
- Pluginik ez dela konfiguratzen, eraikuntza prozesua ez aldatzeko.
- Behar dituzten liburutegiak definitzerakoan, bertsiorik zehazten ez dela ziurtatzea. Horrela, heredatutako pom.xml-tik lortzeko liburutegien bertsioa.

Baldintza hauek ziurtatzeko, kontutan edukiz pom.xml-an idatzitakoa aztertu behar dela, egokiena Maveneko pluginen bidez egitea da. Baina egiaztapen horiek egiten dituen pluginik existitzen ez denez, proiektu honetan egiaztapen horiek egiten dituen plugina garatuko da. Plugin honen diseinua aurrerago egingo da.

Hierarkia hau ikusita, askok lehenengo bi pom.xml-ak zergatik ez diren pom.xml bakarrean jartzen galde dezakete. Honen arrazoia, lehenengo pom.xml-a aldakorra delako eta bigarrena berriz egonkorra delako da. Horren ondorioz, lehenengo pom.xml-an aldaketa bat egiten den bakoitzean *release* bertsio berri bat aterako da; bigarrengo pom.xml-ak berriz beti *snapshot* bertsio bera edukiko du. Honi esker, proiektu bakoitzeko pom.xml-ek beti bertsio bera duen pom.xml fitxategia heredatuko dute, bertsioz aldatuko ez dena, beraz inoiz aldatu beharko ez dena. Baina arkitekturak definitutako azken liburutegien bertsioak erabili ditzaten, nahitaezkoa da bigarren pom.xml-ak lehenengo pom.xml-aren azken bertsioa erabiltzea. Horretarako, sinkronizazio hori egiteko, Maveneko *Versions Maven Plugin* plugina erabiliko da.

9.2.2.2. Dagoeneko existitzen diren eta erabiliko diren pluginak

Atal honetan, Maven bidez proiektuak eraikitzerakoan, Mavenen bizi-zikloan zehar arkitekturako arauak betetzeko exekutatu behar diren pluginak zehaztuko dira. Baita ere, helburuak betetzeko plugin bakoitzaren kasuan konfigurazio bereziren bat behar izatekotan, zein konfigurazio izango duten definituko da. Plugin hauek “pluginen pom.xml”-an definituko dira.

Hasteko, *compile* fasean exekutatzen den *Maven Compiler Plugin* [45] pluginari esker konpilatzeko erabiliko den **konpiladorea definituko da, konpiladoreak erabili beharreko JDK bertsioa zehaztuz**, Java 7 hain zuzen.

Mavenen eraikuntzako bizi-zikloko *test* fasean testak exekutatzen dira, *Maven Surefire Plugin* pluginaren bidez. Testen exekuzioa aktibatuta dago lehenetsita, eta plugin hori ez da desaktibatuko. Beraz, **testak exekutatzea ziurtatuko** da.

Aurretik aipatu den bezala, proiektu bakoitzaren pom.xml fitxategian, proiektu bakoitza eraikitzerakoan jar fitxategia edo war fitxategia sortu behar den definitzen da. Ondoren, jar fitxategia sortu behar dela definitu bada, *Maven JAR Plugin* pluginaren bitartez **jar fitxategi-bitarra sortuko da**. War fitxategi-bitarra sortu behar dela definitu bada aldiz, *Maven WAR Plugin* pluginaren bitartez **sortuko da war fitxategia**. War proiektuen kasuan, *Maven WAR Plugin* pluginaren konfigurazioan definitu behar da liburutegiak war fitxategiaren barnean ez sartzeko, arkitekturak definitzen duen moduan. Horretarako war fitxategi barneko WEB-INF/libs/ karpetaren barnean .jar luzapenarekin amaitzen den fitxategirik ez sartzeko konfiguratuz.

Hala ere, arkitekturan definituta dagoenaren arabera, zerbitzarian ez dira war fitxategiak instalatuko, ear fitxategiak baizik. Horretarako, *Maven EAR Plugin* [46] plugina existitzen da, baina hori erabili ahal izateko lehendabizi software proiektua egokitu egin behar da. Egokitzapen hori nola egingo den beranduago azalduko da, 9.2.2.4. atalean hain zuzen, ondorioz, ear fitxategi-bitarrak nola sortuko direnaren azalpena ere beranduago ikusiko da, 9.3.2.2.1. atalean.

Maven Resources Plugin pluginaren bitartez, iturri-kodearen **kodifikazioa UTF-8 izan behar dela** zehaztuko da.

Mavenek menpekotasunak nola ebazten dituen azaldu denean (9.2.1.3. atalean), pom.xml-an liburutegi bat jartzean, Mavenek automatikoki liburutegi horren menpekotasunak ere ekartzen dituela ikusi da. Horri esker, proiektuaren menpekotasunak kudeatzea errazten da, baina proiektu horrek behar dituen liburutegi guztiak zeintzuk diren jakitetik abstraitzen du. Horregatik, zaila da jakitea zein liburutegi instalatu behar diren zerbitzarian liburutegi edo aplikazio horrek funtzionatzeko. Informazio hori guztia lortzeko plugin bat erabiliko da.

Alde batetik, *Maven Dependency Plugin* plugina erabiliko da. Plugin honek, proiektu baten iturri-kodean zein liburutegi erabili diren aztertzen du eta liburutegi horiek proiektuaren pom.xml-an definitzea behartzen du. Gainera, liburutegiren bat definitu bada pom.xml-an eta iturri-kodean ez bada erabili, pom.xml-tik kentzea behartzen du. Honi esker, proiektu bakoitzaren pom.xml-ak proiektu hori konpilatzeko behar diren liburutegien zerrenda eskainiko du.

Bestetik, plugin beraren konfigurazio ezberdina erabiliz, proiektu batek behar dituen liburutegi guztien zerrenda lortu eta fitxategi batean idatziko dira. Zerrenda honetan, bai konpilatzeko behar diren liburutegiak, baita proiektu hori exekutatzeko behar direnak ere egongo dira. Horrela, liburutegi edo aplikazio bat zerbitzarian instalatzeko, zerbitzarian instalatuta egon behar diren liburutegi guztiak zeintzuk diren jakin ahalko da. Zerrenda hori fitxategi-bitarraren barnean sartuko da.

Amaitzeko, liburutegi baten fitxategi-bitarra sortzen denean, gainontzeko programatzaileek liburutegi hori eskuragarri izan dezaten Maveneko urruneko biltegi batera igotzea geratzen da. Maveneko urruneko biltegi publikoak existitzen dira, baina garapen-ingurune honetako software proiektuak konfidentzialak direnez, garapen-ingurunean biltegi pribatu bat instalatuko da. Fitxategi-bitarrak biltegi pribatu horretara igotzeko *Maven Deploy Plugin* [47] plugina erabiliko da.

Dagoeneko existitzen diren Maveneko pluginez gain, beste plugin batzuk diseinatu eta garatu behar dira proiektu honetako helburuak bete ahal izateko. Hasteko, ondorengo Maven pluginak garatuko dira, baina Jenkinsen atalean ikusiko den bezala, Maveneko beste plugin batzuk ere garatu beharko dira:

- Proiektu bakoitzaren pom.xml fitxategia aurretik aipatutako baldintzak betetzen dituela ziurtatzen duen plugina
- Ear fitxategiak sortzeko plugina
- Software proiektuen bertsioak egitea automatizatzeko plugina

9.2.2.3. pom.xml-a egiaztatzen duen plugina

Lehenago aipatu den bezala, plugin honek hainbat eginkizun ditu:

- pom.xml bat heredatzen duela eta heredatu behar duena dela ziurtatzea.
- Proiektuak behar dituen liburutegiak definitzerakoan, pom.xml-an bertsiorik zehazten ez dela ziurtatzea.
- pom.xml-an eraikuntza prozesua aldatzen duen pluginik ez dela konfiguratzen ziurtatzea.

Lehenengo eginkizunari dagokionez, heredatzen den pom.xml-a *<parent>* izeneko elementuaren barnean definitzen da. Ondorioz, ***parent* elementuaren barnean definituta dagoen *groupId, artifactId* eta *version* zuzena dela egiaztatu**ko da.

Bigarren eginkizunari dagokionez, menpekotasunen bertsioa definitzen ez dela ziurtatzeko bi gauza egin behar dira. Hasteko, software proiektu baten menpekotasun guztiak *<dependencies>* izeneko elementuaren barnean definitzen dira. *<dependencies>* elementuaren barnean nahi adina *<dependency>* elementu definitu daitezke, eta *<dependency>* elementu bakoitzaren barnean menpekotasunaren *groupId, artifactId* eta *version* definitzen dira. Plugin honek, ***dependencies* barruan dauden *dependency* bakoitzean *<version>* izeneko elementurik ez dagoela ziurtatu**ko du.

Bigarren eginkizuna ziurtatzeko ez da nahikoa honekin, menpekotasunen bertsioa *<dependencyManagement>* izeneko elementuetan ere definitu baitaiteke. Elementu hau hain zuzen, heredatuko den liburutegien bertsioen pom.xml-an erabiliko da liburutegien bertsioak definitzeko. Izan ere, elementu horretako liburutegien bertsioak definitzeko balio du, eta bertan jarritako liburutegiak ez ditu Maven-ek proiektuan erabiltzeko jaitsiko. Horregatik, plugin honek software proiektu bakoitzaren pom.xml fitxategian ***<dependencyManagement>* elementurik ez dagoela ziurtatu**ko du.

Amaitzeko, hirugarren eginkizunari dagokionez, software proiektu bakoitzaren eraikuntzako konfigurazioa programatzaile bakoitzak aldatzen ez duela ziurtatu behar da. Konfigurazio guzti hau pom.xml barruko *<build>* izeneko elementuaren barnean egiten da. Horregatik, plugin honek software proiektu bakoitzaren pom.xml-an ***<build>* izeneko elementurik ez dagoela ziurtatu**ko du.

9.2.2.4. Ear fitxategiak sortzeko plugina

Aurretik aipatu den bezala, ear fitxategi bakoitzaren barnean war fitxategi bat egongo da. Beraz, ear fitxategia sortu aurretik aplikazioaren war fitxategi-bitarra sortu behar da, horretarako, *Maven War Plugin* plugina erabiliz.

Behin war fitxategia sortuta, ondorengo pausoa *deployment descriptor* [48] izeneko fitxategi bat sortzea da. Fitxategi horretan, sortuko den ear fitxategiaren deskribapena gordetzen da, ear fitxategia instalatuko duen zerbitzariak ear barruan dagoen edukia zein den jakin dezan. *Deployment descriptor* fitxategia sortzeko *Maven EAR Plugin* plugina erabiliko da, pluginak duen *generate-application-xml* aukera erabiliz. Baina *deployment descripctor* fitxategia sortzeko, pom.xml-a egokitu egin behar da aurretik, pluginak behar duen informazioa pom.xml-tik hartzen baitu. Egokitzapen hori egiten duen pluginik ez dago, horregatik, pom.xml hori egokitzen duen Maveneko plugin bat garatuko da proiektu honetan. Behin egokitzapen hori eginda (garatuko den pluginaren bitartez), dagoeneko existitzen den *Maven EAR Plugin* plugina erabiliz WebSphere zerbitzarietan exekutatu daitezkeen ear fitxategi-bitarrak sortu ahal dira war proiektuetatik abiatuta.

pom.xml-a ear proiektu baterako egokitzen duen pluginak egin behar duena ondorengoa da. Hasteko, pom.xml fitxategitik *groupId, artifactId, version* eta *contextRoot*a irakurri behar du. *ContextRoot,* web-aplikazioen kasuan pom.xml-an idazten den elementu bat da, zeinek aplikazioa zerbitzarian instalatzen denean, aplikazioak izango duen URLa definitzen duen. Ondoren, war proiektuaren pom.xml originala ondorengo edukiarengatik aldatu behar da:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId></groupId>
  <artifactId></artifactId>
  <version></version>
  <packaging>ear</packaging>

  <properties>
    <proiektua.groupId></proiektua.groupId>
    <proiektua.artifactIdWar></proiektua.artifactIdWar>
    <proiektua.version></proiektua.version>
    <proiektua.contextRoot></proiektua.contextRoot>
  </properties>

  <dependencies>
    <dependency>
      <groupId>${proiektua.groupId}</groupId>
      <artifactId>${proiektua.artifactIdWar}</artifactId>
      <version>${proiektua.version}</version>
      <type>war</type>
      <scope>system</scope>
      <systemPath></systemPath>
    </dependency>
  </dependencies>

  <build>
    <finalName>${project.artifactId}</finalName>
    <plugins>
      <plugin>
        <artifactId>maven-ear-plugin</artifactId>
        <version>2.10.1</version>
        <configuration>
          <fileNameMapping>no-version</fileNameMapping>
          <modules>
            <webModule>
              <groupId>${proiektua.groupId}</groupId>
              <artifactId>${proiektua.artifactIdWar}</artifactId>
              <contextRoot>${proiektua.contextRoot}</contextRoot>
            </webModule>
          </modules>
        </configuration>
```

```
        </plugin>
      </plugins>
    </build>

</project>
```

Eta amaitzeko, pom.xml horretan pom.xml originaletik irakurritako datuak gorriz dauden elementuen barruan idatzi. *artifactId* atalean ez da pom.xml originaletik irakurritako *artifactId* idatziko, baizik eta arkitekturak definitzen duenaren arabera, proiektu baten izenak jar, war edo ear proiektua den adierazten du. Beraz, izena egokitu egin beharko da, ear proiektua dela adieraz dezan. Ondorioz, proiektua.artifactIdWar elementuan *artifactId* originala idatziko da. Adibidez:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>eus.ehu.tfm</groupId>
  <artifactId>AplikazioaEAR</artifactId>
  <version>1.0.0-RELEASE</version>
  <packaging>ear</packaging>

  <properties>
    <proiektua.groupId>eus.ehu.tfm</proiektua.groupId>
    <proiektua.artifactIdWar>AplikazioaWEB</proiektua.artifactIdWar>
    <proiektua.version>1.0.0-RELEASE</proiektua.version>
    <proiektua.contextRoot>/AplikazioaWEB</proiektua.contextRoot>
  </properties>

...
```

Diseinatu den plugin honek, war proiektu baten pom.xml-a ear motako proiektu batetara egokituko du. Behin hori eginda, ear fitxategi-bitarra sortzeko ondorengo puasoa *deployment descriptor* fitxategia sortzea da *Maven EAR Plugin* plugina erabiliz. Amaitzeko, pom.xml-a egokituta eta *deployment descriptor* sortuta dagoela, ear fitxategi-bitarra sortzea geratzen da. Horretarako, beste behin ere, *Maven EAR Plugin* plugina erabiliz.

9.2.2.5. Software proiektuen bertsioak egitea automatizatzeko plugina

Plugin honen helburua, software proiektu bakoitza eraikitzen den bakoitzean bertsioa ezberdina izatea da. Honi esker, zerbitzarian instalatuko den bertsio bakoitza identifikatuta geratzen da.

Plugin honen bidez, software proiektuen bertsioa aldatzen dela ziurtatuko da, horretarako, proiektuaren pom.xml-an jarri behar den bertsioa programatzaileak eskuz aldatu ez badu, pluginak automatikoki aldatuko du. Horretarako, Maveneko bi plugin garatuko dira. Izan ere, bertsioak egiteko prozesua bi pausotan banatuko da.

Soluzioan proposatu den bezala, software proiektu bat garatzerakoan *snapshot* bertsioak erabiltzen dira. Bertsio bat *snapshot* denean, bertsio hori garapenean dagoela esan nahi du, hau da, aldaketak jasaten ari dela eta ondorioz, softwareak gaur gauza bat egin dezake eta bihar beste bat.

Egonkorrak diren bertsioak *release* izena dute. Bertsio bat *release* denean, softwarearen iturri-kodea ez da inoiz aldatuko bertsio zenbakia aldatzen ez den bitartean.

Horregatik, zerbitzarian instalatuko diren liburutegi eta aplikazio guztiak *release* bertsioak izan behar dira, ondo identifikatuta egon daitezen. Beraz, programatzaileek beti *snapshot* bertsioekin egingo

dute lan. Baina, proiektu bat eraikitzerakoan *snapshot* bertsiotik *release* bertsiora aldatu beharko da, ondoren zerbitzarian instalatu ahalko den bertsio bat izan dadin.

Mavenek eraikitzen duen software proiektu bakoitza identifikatzeko, Mavenek hainbat datu behar dituela aipatu da: *artifactId, groupId, version* eta menpekotasunak. Beraz, proiektu bakoitzaren bertsioa pom.xml-ko *version* elementuan jarritakoa izango da.

Bertsioak egiteko prozesuko lehen pluginak, proiektu bateko pom.xml fitxategi batean dagoen bertsioa *release* dela ziurtatuko du. Horretarako, *snapshot* bertsioa bada pom.xml-an jarrita dagoena, bertsioko "SNAPSHOT" hitzaren ordez "RELEASE" hitza idatziko du. Maven hitz horiek detektatzeko gai da, hau da, "SNAPSHOT" idatzita badago bertsioan *snapshot* bertsio bat dela badaki, eta berdin *release* bertsioekin.

Adibide gisa, 1.0.0-SNAPSHOT bertsioa badago idatzita, 1.0.0-RELEASE bertsioa idatziko du. Aldiz, pom.xml fitxategian *release* bertsio bat badago definituta, dagoen bezala utziko du. Ondoren, *release* bertsio hori existitzen den egiaztatuko du, horretarako Giteko etiketak erabiliko dira.

Giteko etiketak [49], software bertsioak egiteko Git sisteman proiektuaren bertsio zehatz bat identifikatzeko elementu bat da.

Beraz, pluginak *release* bertsio horren etiketa Gitlabeko biltegi zentralean existitzen den egiaztatuko du. Etiketa existitzen bada, bertsio hori dagoeneko existitzen dela esan nahiko du. Beraz, beste bertsio bat sortu beharko da, eta ondorioz, pluginaren exekuzioa amaituko da errore bat erakutsiz. Etiketa ez bada existitzen, etiketa sortuko du pluginak eta ondoren etiketa eta *release* bertsioaren aldaketa Giteko biltegi lokalean gordeko du. Hau da, oraingoz Gitlabeko biltegira ez da ezer igoko.

Bigarren plugina, software proiektu bakoitza eraiki ondoren exekutatuko da. Honen helburu nagusia, Git zerbitzarian programatzaileentzako *snapshot* bertsio berria garatu dezaten prest uztea da. Hau da, programatzaileek proiektu baten *release* bertsioa sortu ondoren, proiektuaren pom.xml-an *snapshot* bertsio berri bat jartzeko lanetik libratu nahi dira.

Adibidez, 1.0.0-RELEASE bertsioa eraiki bada, pluginak pom.xml-an 1.1.0-SNAPSHOT bertsioa idatziko du.

pom.xml-an *snapshot* bertsioa definitu ondoren, aldaketa hori Giteko zerbitzari lokalean gordetzeaz arduratuko da plugina.

Adibidean, bertsioaren *minor* zenbakia aldatu da. Hau horrela izango da. Izan ere, soluzioan azaldu den bezala *patch* zenbakia RELEASE batean akats bat aurkitzen denerako gordeko da, bertsio horren aldaketa programatzailearen eskutan geratuz. Hau da, zerbitzarian instalatuta dagoen aplikazio baten bertsioa 1.0.0-RELEASE bada, akats hori konpontzeko egin den aldaketarekin 1.0.1-RELEASE bertsioa aterako litzateke.

Major zenbakia aldatzea ere programatzailearen eskuetan geratzen da.

Plugin honek pom.xml-an egingo dituen aldaketa guztiak Giteko biltegi lokalean egingo dira. Izan ere, plugin hau Jenkinsen exekutatzeko dago diseinatua, eta Jenkinsek badauka plugin bat [50] Giteko biltegi lokalean egindako aldaketa guztiak urruneko biltegira (kasu honetan Gitlabera) igotzeaz arduratzen dena.

9.2.2.6. Mavenen urruneko biltegia: Artifactory

Garapen-ingurune honetan sortuko diren proiektuen bertsio guztiak biltegiratu egin behar dira. Horrela, sortutako bertsio guztiak eskuragarri egongo dira uneoro. Honek hainbat onura ditu:

- Liburutegiak gainontzeko liburutegi eta aplikazioak garatzeko erabili ahal dira, proiektu bakoitzaren pom.xml-an definituz.

- Zerbitzarian instalatuta dauden liburutegi eta aplikazioek funtzionatzea uzten badute, funtzionatzen zuten egoera batetara itzultzeko aukera egongo da. Hau da, backup bezala funtziona dezake.

Baina aurretik aipatu den bezala, proiektu honen bezeroak garatzen dituen proiektuak pribatuak dira. Horregatik, sortuko diren fitxategi-bitarrak ezin dira Maveneko urruneko biltegi publiko batetara igo. Beraz, garapen-ingurune honetan Maveneko urruneko biltegi propio bat instalatuko da, Artifactory [51] izeneko biltegia hain zuzen. Artifactoryn Maven bidez eraikitako fitxategi-bitarrak biltegiratu daitezke.

Gainera, beste enpresa batzuei erositako liburutegiak (lizentzia behar dutenak) ez daude biltegi publikoetan eta ezin dira bertara igo. Horregatik, liburutegi pribatu horiek ere Artifactoryra igoko dira, garapen-ingurune honetan garatzen duten programatzaileek Maven bidez proiektuak eraikitzerakoan eskuragarri eduki ditzaten.

9.3. Jenkins

Garapen-inguruneko elementu garrantzitsu bat etengabeko integrazioa inplementatzen duen aplikazioa da. Izan ere, elementu hau arduratuko da garapen-ingurunea automatizatzeaz; horretarako proiektuak eraikiz, zerbitzarietan instalatuz eta beste zeregin batzuk eginez. Alternatiben analisian egindako azterketaren arabera, Jenkins aplikazioa erabiliko da etengabeko integrazioa inplementatzeko proiektu honetan.

9.3.1. Sarrera

Jenkins aplikazioa *job* izeneko elementuetan oinarritzen da. *Job* batean Jenkinsek exekutatzea nahi dena programatzen da. Adibidez, biltegi batetik software baten iturri-kodea jaistea eta eraikitzea. Horretarako, *job*etan scriptak idatzi daitezke edota Jenkinseko pluginak erabili. Izan ere, Jenkinsek hainbat plugin ekartzen ditu lehenetsita *job*en programazioa errazten dutenak, eta nahi izanez gero, plugin gehiago instalatu daitezke edota plugin berriak sortu. Adibidez, proiektu honetan diseinatutako garapen-ingurunean garatuko diren proiektuak eraikitzeko erabiliko diren Jenkinseko pluginetako batzuk ondorengoak dira:

- Gitlab motako sistema batean, proiektu baten biltegian gertaera baten zain egon eta gertaera hori entzutean *job*a exekutatzen duen plugina. Plugin hau ez dator lehenetsita Jenkinsen, beraz, aparte instalatu behar zaio. Pluginak “Gitlab Plugin” [52] izena du.

- Softwareen bertsioak egiteko sistemetatik (Git, SVN eta antzeko biltegietatik) proiektuen iturri-kodea jaisten duen plugina. Plugin honek, Gitlabetik software proiektuen iturri-kodea jaisteko erabili daiteke.

- “Git Plugin” [50] izeneko plugina. Plugin honek *job*aren exekuzioan zehar iturri-kodean egindako aldaketak berriz Gitlabera igotzeko balio du.

9.3.2. Jenkins garapen-ingurunean nola inplementatu

Atal honetan, proiektu honetako zenbait helburu bete ahal izateko, Jenkins garapen-ingurunera egokitu da. Horretarako behar diren pluginak, scriptak, *job*ak, etab. diseinatu eta garapen-inguruneko gainontzeko elementuekin nola komunikatuko den definitu da.

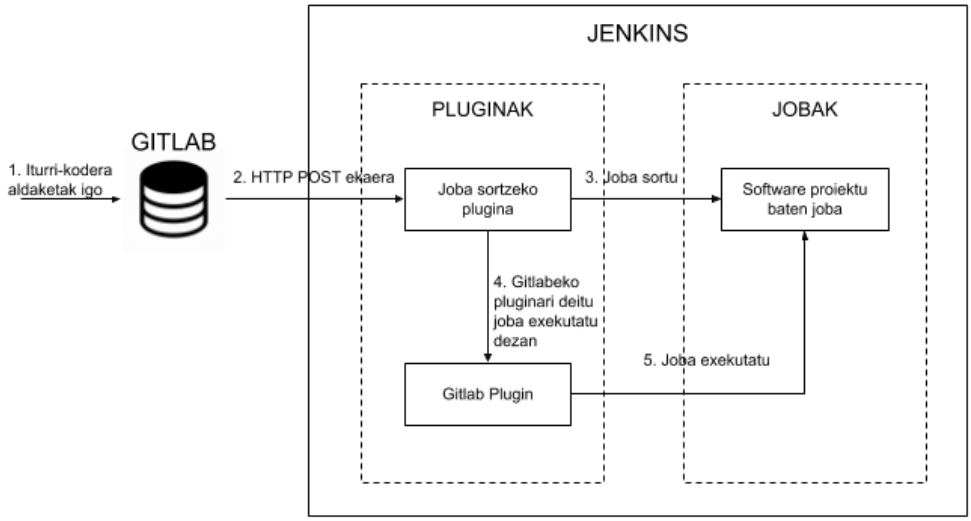
9.3.2.1. Gitlabek webhook bidez bidalitako HTTP POST eskaera prozesatzea

Jenkinsek badauka edozein *job*etan konfiguratu daitekeena. Plugin honi esker, konfiguratu zaion *job* bakoitzak URL bat edukiko du, zeinetara deitzen den bakoitzean *job* hori exekutaraziko duen. Berez, garapen-ingurune honetako *job*ak exekutatzeko nahikoa izango litzateke plugin hori erabiltzea. Baina horrek arazo bat dauka. Izan ere, proiektu bakoitza exekutatu ahal izateko, beharrezkoa da *job* hori sortuta egotea. Kontutan hartuz bezeroak garatzen dituen proiektuen kantitatea handia dela, proiektu bakoitzaren *job*a sortzeak lan handiegia eskatzen du. Horregatik, Jenkinsek *webhook*a jasotakoan, *job*a exekutatu aurretik, proiektuaren *job*a sortuta dagoela ziurtatu behar da, eta sortuta ez badago, *job*a sortu.

Prozesu hori automatizatzeko, Jenkinseko plugin bat garatu da. Horrek, *job* guztiak konfigurazio bera izatea eskatzen du, bakoitzak Gitlabetik jaitsiko duen proiektuaren iturri-kodea izan ezik. Aurrerago ikusiko dugun bezala, hau horrela izango da, hau da, *job* guztiek konfigurazio bera izango dute, salbuespen gutxi batzuk salbu.

Jenkinsek jasotzen dituen *webhook*ak prozesatzen dituen pluginarekin jarraituz, esan den bezala, Gitlabetik bidaltzen diren HTTP POST eskaeren zain egongo da plugina. Beraz, Gitlabeko biltegi guztiek URL berara egingo dute eskaera, eta ondoren plugina HTTP POST eskaeran JSON formatuan bidaltzen den informazioaz [53] baliatuko da zein proiekturi dagokion jakiteko eta eskaera horrekin zer egin behar duen jakiteko.

Gitlabeko HTTP eskaera bat pluginera iristean, jasotako informaziotik eskaera zein software proiekturena den irakurriko du pluginak. Informazio horrekin zein *job* exekutatu behar den jakingo du. Baina exekutatu aurretik *job* hori existitzen den begiratu beharko du, eta existitzen ez bada, sortu. Azaldutako prozesuaren fluxua ondorengo irudian azter daiteke:



Irudia 12: Jenkinsek proiektu bat eraikitzeko deia jasotzen duenean exekutatu behareko fluxua

Irudian azter daitekeen bezala, HTTP eskaera pluginak jasoko du. Eskaera horretako informazioaz [53] baliatuz, pluginak eskaera zein proiektuk egin duen eta proiektu horren Gitlabeko biltegiaren URLa

zein den jakingo du. Proiektu bakoitzaren *job*ak eta Gitlabeko biltegiek izen bera edukiko dutenez, Jenkinsek pluginak garatzeko eskaintzen duen APIa [54] erabilko da ea HTTP eskaera egin duen proiektuaren *job*a existitzen den jakiteko.

*Job*a existitzen ez bada, *job* berri bat sortu beharko da. Horretarako, Jenkinsen "Txantiloi" izeneko *job* bat sortuta egongo da. Beraz, proiektu bakoitzaren *job*a sortzerakoan "Txantiloi" izeneko *job*aren kopia bat sortuko da proiektuaren izena ezarriz. "Txantiloi" *job*ak izango duen konfigurazioa aurrerago aztertuko da, baño *job*aren kopia egin ondoren, Jenkinseko APIa erabiliz, *job* berriaren Giteko URLa aldatu behar da (HTTP eskaeratik lortutakoa ezarriz). Horrekin *job*a sortuta egongo da.

Pluginaren eginkizuna bukatutzat eman aurretik, pluginak HTTP eskaera bidali duen proiektuaren *job*a exekutatu beharko du. Horretarako, *job*ak konfiguratuta eduki behar du HTTP eskaeren bidez *job*a exekutatu ahal dela ("Txantiloi" izeneko *job*ean konfiguratuta egongo da, beraz, bertatik kopiatzen diren *job* guztiak baita ere). Beraz, *job*a exekutatzeko, pluginak HTTP POST eskaera bat egingo du URL horretara eta *job*a "Gitlab Pluginaren" [52] bidez exekutatuko da.

9.3.2.2. Joben definizioa

Orain arte azaldutakoarekin, Jenkinsen *job* bat exekutatu aurretik jarraituko diren pausoak ikusi dira. Atal honetan, "Txantiloi" izeneko *job*aren konfigurazioa azalduko da, garapen-ingurune honetan garatuko diren proiektu guztien eraikuntza nola egingo den azalduz.

Hasteko, *job* bakoitzak Jenkinsek lehenetsita dakarren plugin batekin [50] dagokion proiektuaren iturri-kodea jaitsiko du; ondoren, script bat exekutatuko du, besteak beste, proiektuaren bertsioa egiteaz (gogoratu *release* bertsioak sortuko direla, eta *release* bertsioa egiteko prozesua automatizatu egin dela Maveneko plugin berri baten bidez), proiektua eraikitzeaz, etab. arduratzen dena; eta amaitzeko, proiektuaren iturri-kodean egindako zenbait aldaketa Gitlabeko biltegi zentralera igoko dira, Jenkinsek dakarren plugin batekin [50].

Aurretik aipatu den bezala, *job* guztiek konfigurazio bera izango dute (salbuespen gutxi batzuk izan ezik, aurrerago ikusiko direnak), eta proiektu bakoitzak bere *job*a izango du, bakoitzak Gitlabetik jaitsi behar duen biltegiaren URLa ezberdina izango baitu.

Beraz, *job* bat exekutatzeko fluxua ondorengoa izango da:

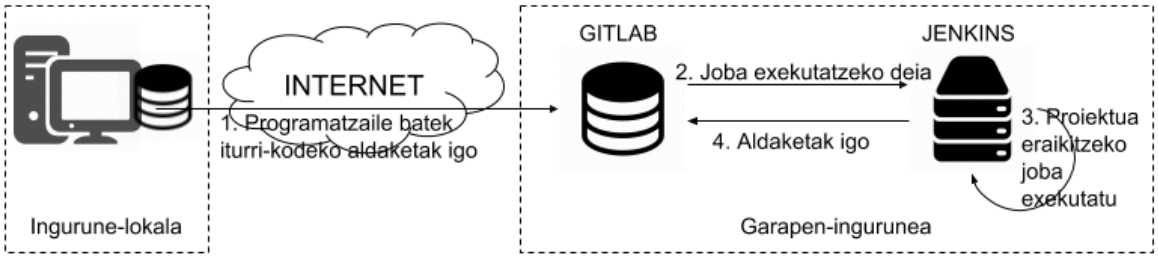
1. *Job* bakoitza Gitlabeko software proiektu batean aldaketak noiz igotzen diren zain egongo da. Lehenengo pausoan, programatzaile batek software proiektu baten egindako aldaketak Gitlabera igoko ditu.

2. Bigarren pausoan, Gitlabek software proiektu batean aldaketak izan dituela abisatzen dio Jenkinseri, Jenkinsek software proiektu horren *job*a exekutatu dezan.

3. Jenkinsek *job* hori exekutatzen du. *Job*aren exekuzioan zehar, proiektuaren iturri-kodea jaitsi eta proiektua eraikitzen da.

4. Softwarearen eraikuntzan eginiko aldaketak software proiektu horren Gitlabeko biltegira igotzen dira.

Ondorengo irudian azter daiteke fluxu hori:



Irudia 13: Job baten exekuzioko pausoak

Azaldutakoarekin hainbat puntu argitu gabe geratzen dira:

- *Job*ak exekutatzen duen scriptak zer egiten du software proiektua eraikitzeko, proiektuaren bertsioak egiteko, etab.?

- Zein aldaketak egiten dira software proiektu baten eraikuntzan zehar ondoren aldaketa horiek Gitlabera igo behar izateko?

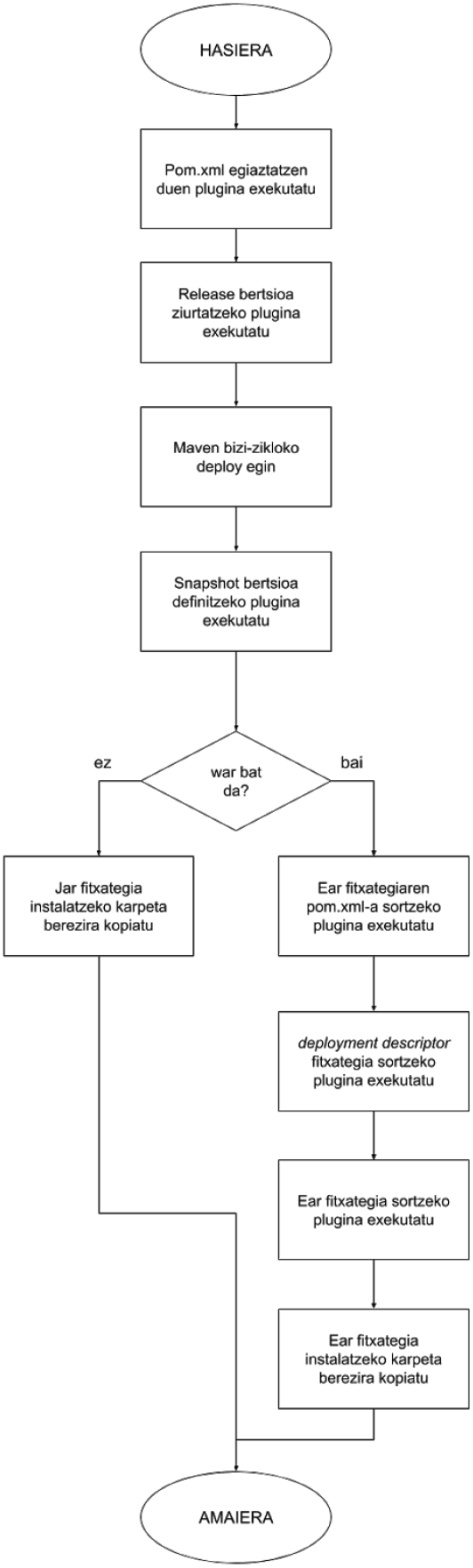
Ondorengo puntuen bidez galdera horiek erantzungo dira.

9.3.2.2.1. *Job*ak exekutatzen duen scriptaren definizioa

Atal honetan, *job* batek eraiki behar duen proiektuaren iturri-kodea jaitsi ondoren, exekutatuko duen scriptak zer egin behar duen definitu da.

Iturri-kodea jaitsi ondoren, software proiektuaren eraikuntza egin beharko da. Horretarako, scriptak gehienbat Mavenen komandoak erabiliko ditu software proiektuak eraikitzeko, proiektu honetarako garatu diren Maveneko pluginetako batzuk exekutatuz. Maveneko plugin hauen bidez, proiektu baten fitxategi-bitarra sortu aurretik, arkitekturako arauak betetzen direla ziurtatuko da.

Scriptaren fluxu-diagrama ondorengoa da:



Irudia 14: Jobek exekutatzen duten scriptaren fluxu-diagrama

Jarraian, aurreko scriptaren fluxu-diagrama azalduko da. Hasteko, lehenengo atazan software proiektuak duen pom.xml-a egokia dela ziurtatuko da. Horretarako, garapen-ingurune honetarako sortu den pom.xml-a egiaztatzen duen Maveneko plugina exekutatzen da. Pluginaren diseinuaren atalean azaldu den bezala, plugin honek hiru egiaztapen egingo ditu:

- Proiektuaren pom.xml-ak beste pom.xml bat heredatzen duela eta heredatu behar duena dela ziurtatzea.

- pom.xml-an proiektuak behar dituen liburutegiak definitzerakoan, pom.xml-an bertsiorik zehazten ez dela ziurtatzea.

- pom.xml-an proiektuaren eraikuntza prozesua aldatzen duen pluginik ez dela konfiguratzen ziurtatzea.

pom.xml fitxategia egiaztatu ondoren, software proiektuaren bertsioa *release* bat dela egiaztatzen duen plugina exekutatuko da. Plugin honen diseinuan azaldu den bezala, laburbilduz, pom.xml-an *snapshot* bertsio bat badago jarrita, *release* bertsioa definituko du pluginak. Gainera, *release* bertsio hori aurretik existitzen ez dela ziurtatuko du pluginak, eta egindako aldaketak, Jenkins makinaren Giteko biltegi lokalean gordeko ditu.

Ondoren, irudiko “Maven bizi-zikloko deploy egin” ataza dator. Ataza honetan, Mavenen bizi-ziklo osoa exekutatuko da proiektua eraikitzeko. Mavenen bizi-zikloa exekutatzean, software proiektu bakoitzaren pom.xml-an konfiguratutako guztia (heredatutako pom.xml-en konfiguratutakoa barne) exekutatuko da. Horrek ondorengoa suposatzen du:

1. Iturri-kodea konpilatzea: *Maven Compiler Plugin* pluginaren bidez bizi-zikloko *compile* fasean.

2. Testak exekutatzea: *Maven Surefire Plugin* pluginaren bidez *test* fasean.

3. Proiektua exekutatzeko behar diren liburutegien zerrenda lortzea: *Maven Dependency Plugin* pluginaren bidez *test* fasean.

4. Jar edo war fitxategi-bitarra sortzea: *Maven JAR Plugin* edo *Maven WAR Plugin* pluginaren bidez *package* fasean.

5. Konpilatzeko erabili diren liburutegi guztiak pom.xml-an definituta daudela ziurtatzea: *Maven Dependency Plugin* pluginarekin *verify* fasean.

6. Fitxategi-bitarra Maveneko biltegi lokalean biltegiratzea: *Maven Install Plugin* pluginaren bidez *install* fasean.

7. Fitxategi-bitarra Maveneko urruneko biltegian biltegiratzea, hau da, Artifactoryn biltegiratzea: *Maven Deploy Plugin* pluginaren bidez *deploy* fasean.

Behin proiektuaren jar edo war fitxategi-bitarra sortuta eta fitxategi-bitar hori Artifactoryra igota gainontzeko programatzaileentzat eskuragarri egon dadin, hurrengo pausoa programatzaileek proiektua garatzen jarrai dezaten pom.xml-an *snapshot* bertsio berri bat definitzea izango da. Horretarako proiektu honetarako diseinatu den *snapshot* bertsioa definitzeko Maveneko plugina erabiliko da. Plugin honek iturri-kodean egingo dituen aldaketak, aurretik azaldu den bezala, aldaketak Giteko biltegi lokalean gordeko ditu, hau da, ez dira Gitlabeko biltegira igoko. Izan ere, aldaketak

Gitlabera igotzeko Jenkinseko plugin bat [50] erabiliko da *job*etako azken pausoan (scripta exekutatu ondoren).

Behin fitxategi-bitarra sortuta dagoela eta Artifactoryra igota, jar motako fitxategia edo war motako fitxategia sortu den jakin behar da. Horretarako *job*aren izena erabiliko da, izan ere, arkitekturak definitzen duenaren arabera, liburutegien izena LIB letrekin amaitzen da eta hauek beti jar fitxategiak izango dira. Aplikazioen kasuan, izena WEB letrekin amaitzen da eta hauek war fitxategiak izango dira.

Beraz, jar motako fitxategia sortu bada, zuzenean jar fitxategi-bitar hori garapen-inguruneko direktorio batera kopiatuko da. Direktorio hori zerbitzarian instalatzeko prest dauden fitxategi-bitarrak gordetzeko erabiliko da. Beranduago WebSphere zerbitzarian liburutegi edo aplikazio bat instalatu nahi denean bertatik hartuko dira fitxategi-bitarrak.

War motako fitxategi-bitarra bada aldiz, ear motako fitxategi bat sortu behar da. Horretarako, lehendabizi, proiektu honetan garatu den ear fitxategiaren pom.xml-a sortzeko plugina erabiliko da. Ondoren, *deployment descriptor* fitxategia sortzeko plugina exekutatuko da, eta amaitzeko ear fitxategia sortzeko plugina exekutatuko da.

Behin ear fitxategi-bitarra sortu dela, jar fitxategiaren kasuan egin den bezala, direktorio berdinera kopiatuko da fitxategi-bitarra. Horrela, nahi denean WebSphere zerbitzarian instalatzeko prest egongo da.

Honekin, scriptaren exekuzioa amaituko da. Scriptaren exekuzioa amaitu ondoren, aldaketak Gitlabeko biltegi zentralera igotzea geratuko litzateke, egindako aldaketak programatzaileentzat eskuragarri egon daitezen.

9.3.2.2.2. Gitlabera igo beharreko aldaketak

*Job*ak egin beharreko funtzioetako bat proiektuaren bertsioak egitea da, horretarako sortu diren Maveneko pluginak exekutatuz. Plugin hauek egiten dituzten aldaketak Giteko biltegi lokalean gordetzen dira, baño ez dituzte aldaketa hauek biltegi zentralera igotzen. Hau da, aldaketa horiek Gitlabera igo ez direnez, ez daude eskuragarri programatzaileentzat oraindik.

*Job*aren exekuzioan zehar egin diren aldaketak Gitlabeko biltegi zentralera igotzeko Jenkinseko *Git Plugin* plugina [50] konfiguratuko da Jenkinseko *job*etan.

9.3.2.3. Zerbitzarian instalatuta dauden liburutegien informazioa eskaintzea

Software proiektu bakoitzaren *job*ek egin beharrekoa azaldu da dagoeneko. Horrekin etengabeko integrazioa lortu da garapen-ingurune honetan. Eta horrekin batera, proiektu honetako helburu asko bete eta arazo asko konpondu dira. Baina garapen-ingurunea oraindik ez da zerbitzarian instalatuta dauden liburutegiak zeintzuk diren eta zein bertsio diren esateko gai. Eta hori, proiektu honetan diseinatutako garapen-ingurunearen helburuetako bat da.

Proiektu honetan Maven nola erabiliko den aztertu denean, proiektu guztietako pom.xml-ek liburutegien bertsioak definituta dituen pom.xml bat heredatuko dutela azaldu da (heredatzen diren bi pom.xml-etako lehenengoa). Atal honetan, pom.xml hori zerbitzarian instalatuta dauden liburutegiekin nola sinkronizatuko den diseinatu da.

Sinkronizazio hori egiteko Maveneko beste plugin bat garatuko da. Plugin hau heredatuko den pom.xml-arekin exekutatuko da, eta exekuzio hori egiteko Jenkinseko *job* berezi bat sortuko da. Plugin honen exekuzioa ondorengo fluxu-diagraman azter daiteke:

Irudia 15: Liburutegien bertsioak definitzen dituen pom.xml fitxategia zerbitzarian instalatuta dauden liburutegiekin sinkronizatzeko exekutatu behar den pluginaren fluxu-diagrama

Fluxu-diagrama aztertuz, lehendabizi, zerbitzarian instalatuta dauden liburutegien direktorioko jar-ak hartuko dituela ikus daiteke. Ondoren jar bakoitzaren *groupId, artifactId* bertsioa irakurriko da. Informazio hau irakurtzeko jar barruan dagoen *META-INF/maven/{groupId}/{artifactId}/pom.properties* fitxategitik irakurriko da. *{groupId}* eta *{artifactId}* izeneko karpeten ordez software proiektuaren *groupId* eta *artifactId* izena duten karpetak izango dira,

baina kasu bakoitzean, maila berean egongo den karpeta bakarrak izango dira, beraz ez da arazorik egongo pom.properties izeneko fitxategia bilatzeko.

Behin fitxategia bilatuta, pom.properties fitxategian egongo den informazio bakarra *groupId, artifactId* eta *version* izango da. Ondorengo taulan dago pom.properties baten adibidea:

```
#Generated by Maven
#Wed May 31 08:46:16 CEST 2017
version=1.267.14-RELEASE
groupId=net.izfe.g240
artifactId=WIZIzenpeComunesLIB
```

pom.properties fitxategia Maven bidez eraikitako fitxategietan bakarrik egoten da. Beraz, plugin honen bidez garapen-ingurune honetan garatutako liburutegien sinkronizazioa egingo da. Gainontzeko liburutegien sinkronizazioa eskuz egingo da, honen arrazoiak bi dira: batetik, liburutegi horiek Maven bidez sortu direla ezin ziurtatu ahal izatea, ez baititu bezeroak sortzen; eta bestetik, gainontzeko liburutegiak bezeroak ez dituenez garatzen, egonkorrak izango dira, eta beraz, ez dira etengabe aldatuko. Garapen-ingurune honetan garatuko diren liburutegiak aldiz, etengabe aldatzen egongo dira, zerbitzarian liburutegi baten bertsioa maiz aldatuz.

Horregatik, garapen-ingurune honetan garatutako proiektu guztien *groupId* balioaren hasierako zatia berdina edukiko dute, adibidez, *eus.tfm*. Beraz, *groupId* balioa *eus.tfm*-rekin hasten ez diren liburutegi guztiak baztertu egingo dira sinkronizazioan. Eta pom.properties fitxategia ez dutenak ere bai.

Behin menpekotasun bakoitzaren *groupId, artifactId* eta *version* lortuta, menpekotasun hauek banan-bana ea pom.xml-an definituta dauden begiratuko da, *groupId* eta *artifactId* berdina den begiratuz. Menpekotasuna definituta badago, ea bertsioa berdina den konparatuko da, ezberdina bada aldatuz.

Menpekotasun bakoitza pom.xml-an definituta ez badago, menpekotasun hori osorik definitu egin beharko da.

Liburutegien bertsio guztiak pom.xml-an eguneratu ondoren, pom.xml-aren aldaketa berriak identifikatzeko bertsio berri bat sortu behar da. Horretarako, pom.xml-aren *version* elementuan *minor* zenbakia igoko da, *release* bertsio berri bat ateraz.

Aldaketa guztiak egin ondoren, Mavenen bizi-zikloa exekutatuko da *deploy* eginez. Horrela, pom.xml fitxategiaren *release* bertsio berria garapen-inguruneko Artifactory biltegira igoko da, programatzaile guztien eskura egon dadin. Honi esker, alde batetik, garapen-ingurunean garatuko diren software proiektuak zerbitzarian instalatuta dauden liburutegiekin konpilatuko direla ziurtatzen da. Bestetik, liburutegien bertsioak definitzen dituen pom.xml-ak zerbitzarian instalatuta dauden liburutegien informazioa eskainiko du.

Plugin hau exekutatzen duen *job*arekin amaitzeko, aldaketa horiek Gitlabeko biltegi zentralera ere igoko dira, horretarako Jenkinseko plugin ezaguna [50] erabiliz.

Behin liburutegien bertsioak definituta dituen pom.xml-a sinkronizatuta, pom.xml hori heredatzen duen pluginen pom.xml-a (proiektu honetan diseinatua izaten ari den garapen-ingurunearen bidez garatutako proiektu guztiek duten pom.xml aita) eguneratzea falta da.

Horretarako, aurreko *job*ak, amaitutakoan, beste *job* bat exekutaraziko du. Bigarren *job* honek egin beharko duen lehenengo gauza, Gitlabetik pluginen pom.xml-a jaistea da. Ondoren, bere pom.xml aitaren bertsio berria eguneratu beharko du. Horretarako, Maveneko plugin bat [55] existitzen da, Artifactoryn pom.xml baten aitaren azken bertsioa zein den lortu eta pom.xml-an eguneratzen duena. Amaitzeko, kasu honetan ere, aldaketak Gitlabera eta Artifactoryra igo beharko dira.

Aipatzekoa da, pluginen pom.xml-aren (proiektu guztiek heredatzen dutena) bertsioa ez dela aldatu (bere aitaren kasuan ez bezala). Izan ere, aurretik aipatu den bezala, pom.xml honek beti *snapshot* bertsio bera izango du. Honi esker, pom.xml-a eguneratzen den bakoitzean pom.xml hau aita bezala duten proiektu guztiak eguneratu behar izatea saihesten da. Aldiz, liburutegien bertsioak dituen pom.xml-an beharrezkoa da aldaketa bakoitzeko bertsio berri bat ateratzea. Horrela, fitxategi-bitar bakoitza zein liburutegirekin konpilatu den jakitea ahalbidetuko da, 9.3.2.5. atalean ikusiko den bezala.

9.3.2.4. Fitxategi-bitarrak zerbitzarian instalatzea

Oraindik ez da liburutegi eta aplikazioak WebSphere zerbitzarian modu erraz batean instalatzeko modurik eskaini. Horregatik, atal honetan, fitxategi eta aplikazioak instalatzeko modu erraz bat eskainiko da, etengabeko entregatzea lortuz.

Orain arte azaldutakoarekin, proiektu bakoitzaren *job*ak proiektu horren fitxategi-bitarra sortu eta fitxategi-bitarra instalatzeko prest uzten du karpeta berezi batean. Atal honetan, karpeta horretan utzitako fitxategi-bitarrak zerbitzarian nola instalatuko diren azalduko da.

Horretarako *job* berezi bat diseinatu da, proiektu guztien *job*aren eredua jarraituko ez duen *job* bat alegia. *Job* honek bi parametro [56] jasoko ditu exekutatzen hasi aurretik. Parametro hauek erabiltzaile batek sartuko ditu eta ondorengoak dira:

- Instalatu nahi diren liburutegien zerrenda.
- Instalatu nahi diren aplikazioen zerrenda.

Arkitekturan definituta dagoenaren arabera, liburutegiak zerbitzari-mailan jartzen dira, war fitxategi bakoitzaren barnean egon beharrean. Hau da, aplikazioek liburutegiak partekatzen dituzte. Liburutegi hauek zerbitzariko direktorio batean daude, beraz, *job*eko parametroan zehaztu diren liburutegiak direktorio horretan kopiatuko ditu *job*ak.

Bestetik, instalatu behar diren aplikazioak beste direktorio batean egoten dira. Beraz, ear fitxategiak direktorio horretara kopiatuko ditu *job*ak.

Ondoren, *job*ak WAS zerbitzaria berrabiarazi beharko du liburutegi berriak kargatu eta aplikazioak abiarazi ditzan. Liburutegirik instalatu ez den kasuan, ez dago zerbitzaria berrabiarazteko beharrik, izan ere, WAS zerbitzaria aplikazioak kargatzeko gai da zerbitzaria berrabiarazi gabe.

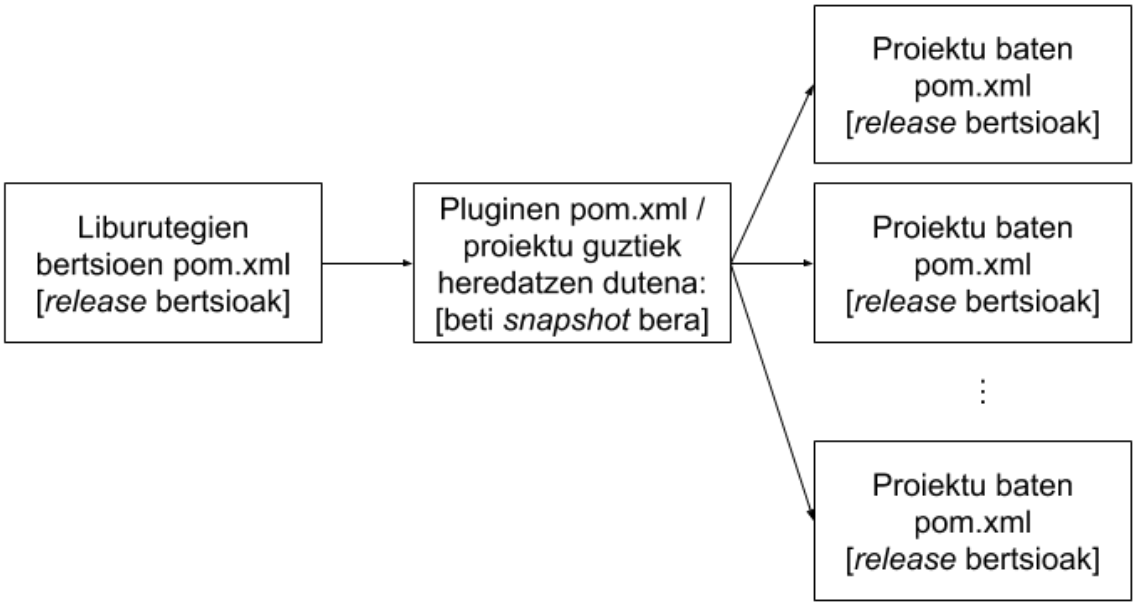
Amaitzeko, zerbitzariko liburutegiak liburutegien bertsioak dituen pom.xml-an sinkronizatzen dituen plugina (diseinatu berri dena) exekutatzen duen Jenkinseko *job*a exekutatuko da. Honi esker, zerbitzarian instalatu berri diren liburutegien bertsioak definituko dira proiektu guztiek heredatzen duten pom.xml-an. Beraz, liburutegien bertsioak dituen pom.xml-ak uneoro zerbitzarian dauden liburutegiak zeintzuk diren eskainiko ditu, eta proiektuak eraikitzerakoan, zerbitzarian instalatuta dauden liburutegiekin egiten dela ziurtatzen da.

9.3.2.5. Fitxategi-bitar bakoitza zein liburutegiren bertsioekin konpilatu den jakitea

9.3.2.2. atalean garapen-ingurune honetako proiektu guztiek exekutatuko duten *job*en diseinua egin da. Diseinu horrekin, proiektuak eraikitzerakoan bezeroaren arkitekturako arauak betetzen direla ziurtatzen da. Baina 9.2.2.1. atalean, proiektu guztietako pom.xml-ek beti pom.xml bera heredatuko dutela esan da, hau da, heredatuko den pom.xml horren bertsioa ez dela inoiz aldatuko.

Diseinu horrekin ezin daiteke jakin proiektua eraiki den unean liburutegien bertsioak definitzen dituen pom.xml-aren zein bertsio erabili den. Arazo hau konpontzea, proiektuaren helburuetako bat da, beraz, atal honetan soluzio bat proposatu da hori konpontzeko.

Hasteko, gogoratu behar da proiektu guztiek heredatzen duten pom.xml-ak (proiektu guztiek heredatzen dute pluginen pom.xml-a, eta pluginen pom.xml-ak liburutegien bertsioak definitzen dituen pom.xml-a heredatzen duenez, proiektu guztiek heredatzen dute azken pom.xml hori ere) beti *snapshot* motako bertsio bera izango duela pom.xml-an definituta.

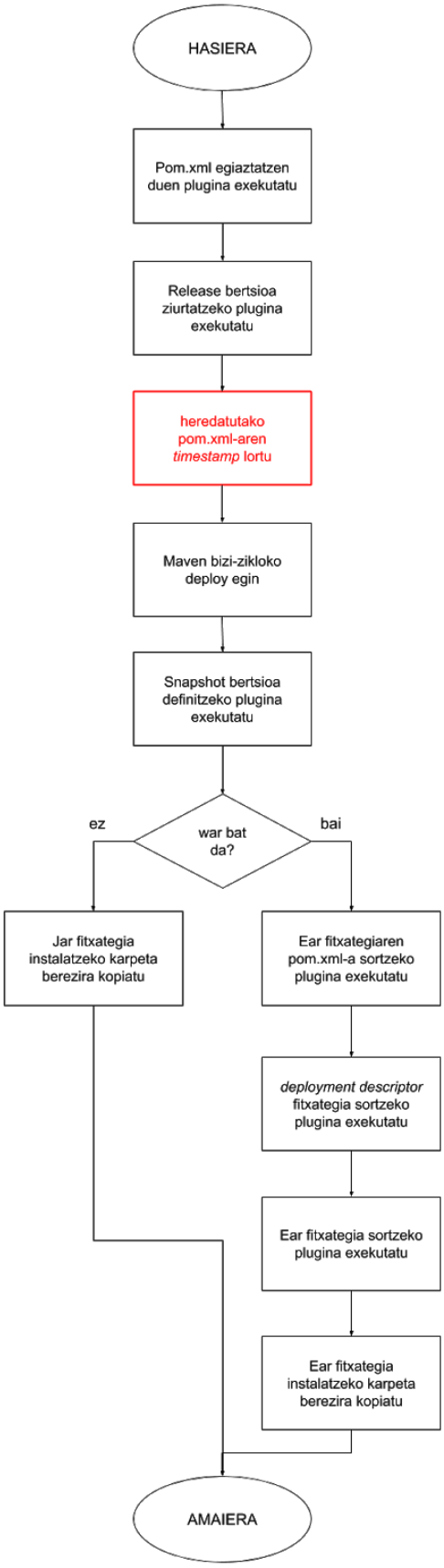


Irudia 16: pom.xml-en hierarkia, pom.xml bakoitzak duen bertsio mota erakutsiz

Baina jakin beharra dago, Artifactoryra *snapshot* bertsio bat igotzen denean, Artifactoryk *snapshot* bertsioan *timestamp* bat jartzen duela, *snapshot* bertsio beren artean ezberdindu ahal izateko. Hau da, pom.xml-aren bertsioa 1.0.0-SNAPSHOT bada, Artifactoryra bertsio hori igotzen den bakoitzean *timestamp* bat jartzen dio, adibidez, 1.0.0-20180306.081815-196. Honi esker, nahiz eta proiektu guztien pom.xml-ek heredatzen duten pom.xml-aren bertsioa beti bera izan, une bakoitzean erabiltzen den pom.xml-aren bertsioa Artifactoryn identifikatuta dago.

Gogoratu behar den bigarren gauza, beti proiektu guztiek heredatzen duten pom.xml horrek liburutegien bertsioak definitzen dituen pom.xml-a heredatzen duela. Kasu honetan, azken pom.xml horren bertsioa aldatzen denez, Artifactoryn identifikatuta dago *timestamp*aren beharrik gabe.

Beraz, proiektu bakoitzaren bertsio bakoitza zein liburutegirekin konpilatu den jakiteko, proiektu bakoitzaren fitxategi-bitarra sortzerakoan, fitxategi-bitarraren barnean heredatutako pom.xml-aren bertsioa gorde beharko da *timestamp* eta guzti. Horretarako *Versions Maven Plugin* [55] plugina erabiliko da. Plugin hau erabiltzeko, *job*aren definizioan aldaketa txiki batzuk egin beharko dira.



Irudia 17: Jobek exekutatu behar duten scriptaren fluxu-digrama, aurreko fluxu-diagramari pauso bat gehituz proiektu bakoitza zein liburutegirekin konpilatu den jakiteko

Fluxu-diagrama erreparatuz, proiektua eraikitzeko Maveneko biz-zikloa exekutatu aurretik, “heredatutako pom.xml-aren *timestamp* lortu” izeneko bloke bat gehitu dela azter daiteke. Bloke honetan, proiektu guztien pom.xml-ek heredatzen duten pom.xml-aren *timestamp*a duen fitxategi bat sortuko da, ondorengo blokean proiektuaren fitxategi-bitarra sortzen denean, barnean sartzeko.

Horretarako, lehendabizi proiektuaren pom.xml-aren kopia bat egingo da.

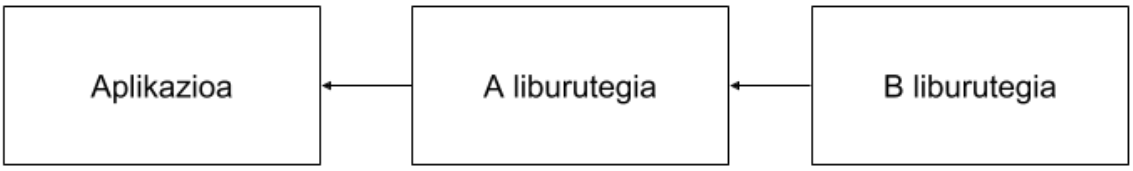
Ondoren, *Versions Maven Plugin* plugina erabiliko da. Honen bidez, proiektuaren pom.xml fitxategian, heredatzen duen pom.xml-aren bertsioan *timestamp*a duen bertsioa definitzen da. Adibidez, heredatzen den bertsioa 1.0.0-SNAPSHOT bada eta Artifactoryn dagoen 1.0.0-SNAPSHOTaren bertsiorik berriena 1.0.0-20180306.081815-196 bada, pom.xml-an 1.0.0-20180306.081815-196 jarriko du pluginak.

Behin *timestamp*a duen pom.xml fitxategia lortuta, fitxategi honi izena aldatuko zaio. Horrela, lehenengo pausoan egindako pom.xml-aren kopia erabiliz, hasierako pom.xml-a berreskuratuko da (*timestamp*a ez duena).

Ondorioz, proiektuan pom.xml fitxategia edukitzeaz gain, *timestamp*a finkatuta duen pom.xml horren kopia ere egongo da. Honekin, fluxu-diagramako bloke berriaren funtzioa amaitutzat egongo da. Baina fitxategi-bitarra sortzerakoan *timestamp*a duen fitxategia ere sartu dadin, beharrezkoa izango da pluginak definitzen dituen pom.xml-an (proiektu guztiek heredatzen dutena, zeinaren *timestamp*a finkatu den bloke honetan) *Maven War Plugin* eta *Maven Jar Plugin* pluginak konfiguratzea.

Amaitzeko, proiektuaren pom.xml-an *timestamp*a finkatuta ez uztearen arrazoia azalduko da. *Timestamp*a finkatuta uzten bada, proiektua balio horrekin eraikiko da, ondorioz, Artifactoryra ere *timestamp*arekin igoz. Artifactoryra *timestamp*arekin igotzen bada, gainontzeko proiektuek liburutegi hori erabiltzen dutenean, bertsioak definituta izango ditu. Eta *parent*aren bertsioa *snapshot* bertsioa izatearen arrazoia, proiektua eraikitzerakoan beti liburutegien bertsioak definitzen dituen pom.xml-ak une horretan definitzen dituen bertsioak erabiltzea da. Arazoa ondo ulertzeko adibide bat erabiliko da.

Demagun aplikazio batek liburutegi bat (A) erabiltzen duela, eta liburutegi horrek beste liburutegi bat (B) erabiltzen duela, ondorengo irudian ikusten den bezala:



Irudia 18: Aplikazio bat eta bi liburutegien arteko erlazioaren adibide bat

Lehendabizi, B liburutegiaren 1.0.0-RELEASE bertsioa sortu eta zerbitzarian instalatzen da. Beraz, liburutegien bertsioak definitzen dituen pom.xml-ak B liburutegiaren 1.0.0-RELEASE bertsioa izango du definituta. Ondorioz, A liburutegia eraikitzen denean B liburutegiaren 1.0.0-RELEASE bertsioa erabiliko da konpilatzeko. Honekin, A liburutegiaren 1.0.0-RELEASE bertsioa sortuko da, eta pom.xml-a *timestamp*arekin igo bada, beti liburutegiaren 1.0.0-RELEASE bertsioa izango du definituta, izan ere, beti liburutegiak deifnitzen dituen pom.xml-aren bertsio bera hartuko baitu.

Ondoren, B liburutegiaren 1.1.0-RELEASE bertsioa ateratzen da, eta hau ere zerbitzarian instalatzen da A liburutegiarekin batera. Ondorioz, liburutegiak definitzen dituen proiektuak ondorengoa izango du definituta:

- A liburutegia: 1.0.0-RELEASE
- B liburutegia: 1.1.0-RELEASE

Baina Artifactoryn igota dagoen A liburutegiaren pom.xml-ak B liburutegiaren 1.0.0-RELEASE bertsio dauka definituta. Ondorioz, aplikazioa eraikitzen denean, B liburutegiaren bertsioa liburutegiak definitzen dituen pom.xml-aren azken bertsiotik lortu beharrean (1.1.0-RELEASE izango litzatekeena), A liburutegiak esaten dionetik lortuko du. Hau da, A liburutegiak *timestamp*a definituta duenez, liburutegiak definitzen dituen pom.xml-aren azken bertsioa erabili beharrean, A liburutegiaren sortu zenean erabili zen liburutegien bertsioak definituta dituen pom.xml-aren bertsioa erabiliko du, hau da, B liburutegiaren 1.0.0-RELEASE bertsioa.

Horregatik, liburutegien bertsioak definitzen dituen pom.xml-ak definitzen dituen bertsioak erabiltzen direla ziurtatzeko, Artifactoryra igotzen den liburutegiek pom.xml-an *timestamp*a finkatu gabe izatea garrantzitsua da.

9.3.2.6. Zerbitzarian oraindik instalatu gabe dauden liburutegiekin konpilatzea

Orain arteko diseinuarekin ia arazo eta helburu guztiak bete eta konpondu dira. Baina arazo bat dago.

Arazoaren adibide gisa, suposatu programatzaile bat LiburutegiaLIB izeneko liburutegi bat garatzen ari dela eta AplikazioaWEB izeneko aplikazio bat. Zerbitzarian LiburutegiaLIB izeneko liburutegiaren 1.0.0-RELEASE bertsioa dago instalatuta, beraz, herentziako pom.xml fitxategian 1.0.0-RELEASE bertsioa dago definituta. Ondorioz, liburutegi hori erabiltzen duten liburutegi eta aplikazio guztiak 1.0.0-RELEASE bertsioarekin konpilatuko dira. Programatzaileak LiburutegiaLIB liburutegiaren 1.1.0 bertsioa garatzen ari da, eta AplikazioaWEB aplikazioaren bertsio berriak liburutegiaren bertsio berria behar du.

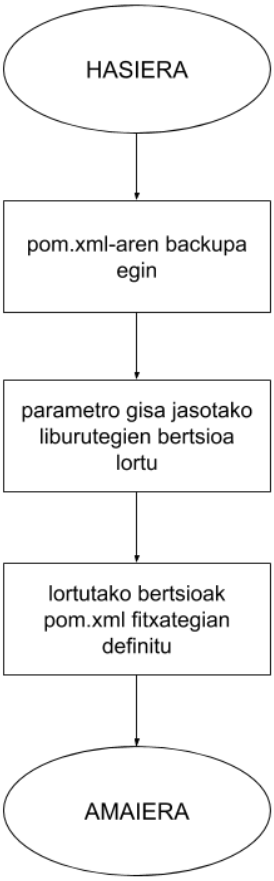
Orain arteko diseinuarekin, Jenkinsen AplikazioaWEB aplikazioa liburutegiaren bertsio berriarekin konpilatu dadin, lehendabizi liburutegiaren *job*a exekutatu behar du, liburutegiaren 1.1.0-RELEASE bertsioa ateraz. Ondoren, liburutegia zerbitzarian instalatu arte itxaron beharko du. Izan ere, programatzaile guztiek ez dute baimenik liburutegi eta aplikazioa zerbitzarian instalatzeko. Beraz, baimena duen norbaitek instalatu arte itxaron beharko du. Liburutegia zerbitzarian instalatzen denean, herentziako pom.xml-an definituko da bertsio berria automatikoki. Hori gertatzen denean, gai izango da AplikazioaWEB aplikazioaren bertsio berria liburutegiaren bertsio berriarekin konpilatzeko.

Arazo honek, garapen-inguruneko aplikazioen eta liburutegien garapena moteltzen du. Arazo honi konponbidea emateko Maveneko plugin bat garatuko da eta software proiektuak eraikitzeko *job*en konfigurazioa aldatuko da.

Hasteko, *job*ak parametrizatuak [56] izango dira. Hau da, exekuzioa hasi aurretik parametro bat sartu ahalko du erabiltzaileak. Parametro honetan, software proiektua konpilatzeko herentziako pom.xml-an definituta dagoenaren bertsio ezberdinak erabili nahi diren liburutegien zerrenda idatziko da. Hau da, aurreko adibidearekin jarraituz, herentziako pom.xml-an LiburutegiaLIBen bertsioa 1.0.0-RELEASE bada eta AplikazioaWEB LiburutegiaLIBen bertsio berriarekin konpilatu nahi bada, Jenkinseko *job*ean parametro gisa LiburutegiaLIB jarriko da.

Ondoren, *job*ak parametro gisa hartzen duen liburutegiaren bertsioa aldatu behar da pom.xml fitxategian. Horretarako, pom.xml fitxategia aldatzen duen Maveneko plugin bat sortuko da.

Plugin honek egingo duena ondorengo fluxu-diagraman azter daiteke:



Irudia 19: Proiektu bat zerbitzarian instalatu gabe dagoen liburutegi batek bertsio batekin konpilatu ahal izateko, Maveneko plugin batek jarraitu beharreko pausoak

Hasteko, pom.xml fitxategiaren *backup*a egingo da. Izan ere, garapen-ingurune honetan garatuko diren software proiektuek pom.xml-an ezin dute bertsioa idatzita eduki. Horregatik, software proiektu baten *job*aren exekuzioko azken pausoan aldaketak igotzen direnen Gitlabera, plugin honek egindako aldaketak ez dira igo beharko. Bestela, software proiektua garatzen ari den programatzaileak eskuz borratu beharko luke bertsioa. Horretarako, *job*ean fitxategi-bitarra sortu ondoren pom.xml-ren *backupa* berriztatuko da, aurrerago ikusiko den bezala.

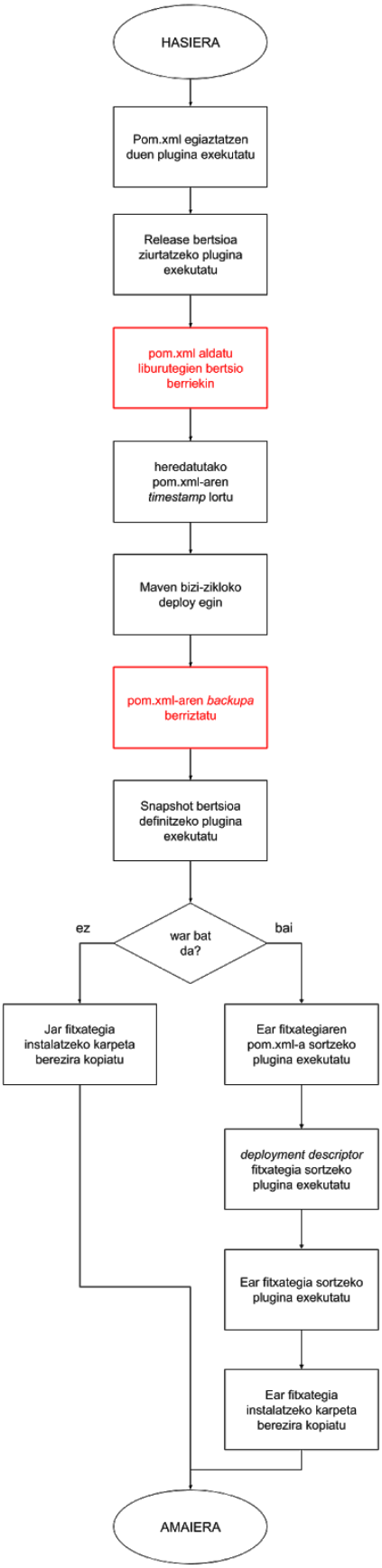
*Backup*a sortu ondoren, pom.xml-an definitu behar diren liburutegien bertsioak zein diren lortu behar ditu pluginak. Horretarako, pluginak sarrerako parametro gisa pasa zaizkion liburutegien lista hartuko du, eta banan bana, liburutegi bakoitzaren jar fitxategia bilatuko du zerbitzarian instalatzeko prest dauden fitxategi-bitarrak uzten diren direktorioan. Jar bakoitzaren bertsioa lortzeko, sinkronizazioko pluginak egiten duen bezala egingo du.

Amaitzeko, lortu diren liburutegien bertsioak pom.xml fitxategiko menpekotasunetan idatzi behar dira. Behin bertsioa fitxategian idatzita, software proiektua Maven bidez eraikitzen denean bertsio hori Artifactorytik hartuko du, izan ere, nahiz eta liburutegi hauek zerbitzarian oraindik instalatu gabe egon, fitxategi-bitarrak sortzerakoan Artifactoryra igotzen dira.

Plugin honek funtzionatzeko, nahitaezkoa izango da bertsio berriaren *job*a exekutatzea aurretik. Hau da, aurreko adibidearekin jarraituz, lehendabizi LiburutegiaLIBen 1.1.0-RELEASE bertsioa atera beharko litzateke bere *job*a exekutatuz. *Job* horrek 1.1.0-RELEASE bertsioa utziko luke bai zerbitzarian instalatzeko prest dauden fitxategien direktorioan, eta baita Artifactoryn ere. Ondoren, AplikazioWEBen *job*a exekutatuko litzateke parametro gisa LiburutegiaLIB sartuta. Horrela, azkena garatu den pluginari esker, AplikazioaWEB konpilatzeko LiburutegiaLIBen 1.1.0-RELEASE bertsioa

erabiltzea lortuko da. Amaitzeko, zerbitzarian LiburutegiaLIB eta AplikazioaWEBen bertsio berriak instalatzea geratuko litzateke, bien instalazioa aldi berean egin daitekeelarik.

Plugin honen erruz, software proiektu bakoitzaren *job*ean exekutatzen den scriptaren fluxu diagrama aldatuko litzateke:



Irudia 20: Jobek exekutatu beharreko scriptaren fluxu-diagrama, zerbitzarian instalatu gabe dauden liburutegiekin konpilatu ahal izateko pauso berriak sartuz

Aurreko scritparekiko bi ezberdintasun daude. Batetik, RELEASE bertsioa ziurtatu ondoren pom.xml-ko menpekotasunen bertsioa aldatzeko plugina exekutatzen dela. Bestetik, proiektua eraiki eta fitxategi-bitarra sortu ondoren, bigarren plugin batek, lehenengo pluginak sortutako pom.xml fitxategiaren *backup*a berriztatu egingo du.

9.3.2.7. Proiektu ugari sekuentzialki exekutatzeko job*a*

Programatzaileak batzuetan liburutegi edo aplikazio bakarra garatzen aritu beharrean, askotan hainbat proiekturekin aldi berean lanean aritzen dira. Horrelako kasuetan ez da arraroa liburutegi batean egiten diren aldaketek gainontzeko liburutegi eta aplikazioetan eragitea.

Proiektu guztietako aldaketak prestatu ondoren, proiektu horiek Jenkinsen eraikitzeko ordua iristen da. Kasu hauetan, kontutan eduki behar da bai proiektuak eraikitzeko ordena, baita proiektuen arteko menpekotasuna ere. Adibidez, liburutegi baten eta liburutegi hori erabiltzen duen aplikazio baten kasuan, lehendabizi liburutegia eraiki beharko da eta ondoren aplikazioa, liburutegi horren azken bertsio hori erabiliz. Izan ere, aplikazioaren iturri-kodeak liburutegi horren menpekotasuna du (alderantziz ez), beraz, aplikazioak liburutegi horren azken bertsioarekin konpilatu beharko da, horretarako beharrezkoa izanik liburutegiaren fitxategi-bitarra sortuta eta Artifactoryn biltegiratuta egotea.

Atal honetan diseinatu den *job* berriarekin, parametro bezala pasatzen zaizkion proiektuak eraikiko ditu. Horretarako, proiektu bakoitzaren *job*a exekutatzeaz arduratuko da. Horretarako, 9.3.2.5. atalean aipatutako funtzionaltasunaz baliatuko da. Hau da, *job*aren parametroan pasatako proiektuen *job*ak, idatzi zaizkion ordenean exekutatuko ditu. Baina proiektu bakoitzaren *job*a exekutatzerakoan, *job* nagusiak aurretik exekutatu dituen proiektuak pasako dizkie parametro gisa *job* bakoitzari.

Adibidez, demagun 3 proiektu eraiki behar direla: A liburutegia, B liburutegia eta C aplikazioa. Ak ez du gainontzekoen menpekotasunik, Bk Arena dauka eta Ck Arena eta Brena. *Job* nagusiari parametro gisa A, B eta C pasako zaizkio, orden horretan. Beraz, *job* nagusiak A exekutatuko du parametro bezala beste proiekturik pasa gabe. A eraiki ondoren, *job* nagusiak B exekutatuko du, baina kasu honetan, parametro bezala A pasaz. B eraiki ondoren, *job* nagusiak C exekutatuko du, eta kasu honetan, parametro bezala A eta B pasako dizkio.

Proiektu bakoitzaren *job*a exekutatzerakoan gainontzeko proiektuak pasaz parametro bezala 9.3.2.5. ataleko funtzionaltasunaz baliatuz, proiektu horiek konpilatzerakoan WAS zerbitzarian instalatua dauden liburutegien bertsioak erabili beharrean, sortu berri diren proiektuen bertsioak erabiltzen direla ziurtatzen da. Beraz, adibidearekin jarraituz, C aplikazioak A eta B liburutegien bertsio berrian bakarrik dauden funtzionalitate berriak behar baditu, arazorik ez dela egongo ziurtatzen da.

Job honi esker, elkarren arteko menpekotasunak dituzten *job*ak eraikitzea errazten da eta automatizatzen da.

LANERAKO ERABILITAKO

METODOLOGIA

II. ZATIA

10.Atazen deskribapena

Atal honetan proiektua gauzatu ahal izateko bete behar diren zereginak deskribatuko dira. Horretarako, zeregin hauek lan-paketeetan banatu dira.

Baina lehendabizi, proiektu hau gauzatzeko beharrezkoa den lan-taldea nolakoa izan behar den aztertuko da.

10.1. Lan-taldea

Lan-taldea ondorengo kideek osatzen dute:

- **Project Director:** Telekomunikazioetako ingeniari senior bat izango da, mota honetako proiektuak gainbegiratzen aditua dena. Honen eginkizuna proiektuaren garapena gainbegiratzea izango da.

- **Project Manager:** Informatikako ingeniari senior bat izango da, mota honetako proiektuak egiten aditua dena. Bere eginkizun nagusia, garatzaileari diseinuan laguntzea izango da.

- **Garatzailea:** Software proiektuak diseinatzeko eta garatzeko ezagutzak dituen telekomunikazio ingeniari junior bat izango da. Garatzailearen eginkizunen artean egoeraren eta konpondu nahi den arazoaren analisia egitea izango da, horretarako bezeroarekin hitz eginez. Arazoa modu eraginkorrenean konpondu ahal izateko diseinua egiteaz ere arduratuko da, eta ondoren diseinu hori inplementatzeaz arduratuko da. Horrez gain, proiektuaren dokumentazioa egiteaz ere arduratuko da.

10.2. Lan-pakete eta zereginen banaketa

Aurretik aipatu den bezala, atal honetan proiektua gauzatu ahal izateko bete behar diren lan eta zereginak definitu dira. Honekin batera, lan eta zeregin bakoitzean zein kideek jardungo duten eta kasu bakoitzean zenbat ordu eskaini beharko zaizkion zehaztuko da.

LP.1 Lanaren definizioa

Lehen lan-paketean proiektuaren helburu eta eskakizunak definituko dira. Ondoren, merkatuan dauden alternatiben azterketa bat eginez, aurretik aipatu diren helburuak ahalik eta ondoen betetzeko. Horregatik, lan-pakete hau bi zereginetan banatu da: “Proiektuen eskakizunak definitu” eta “Alternatibak aztertu”.

Z1.1 Proiektuaren eskakizunak definitu

Lehenengo zeregina proiektuaren helburuak eta eskakizunak definitzean datza.

Z1.2 Alternatibak aztertu

Atal honetan, proiektuaren helburuak eta eskakizunak bete ahal izateko zein alternatiba dauden aztertuko da. Honi esker, ondorengo zereginetan garapen-ingurunearen diseinua egiten hasterakoan, nondik hasi behar den bideratuko da.

LP.2 Garapen-ingurunearen diseinua

Lan-pakete honen helburu nagusia garapen-ingurunearen diseinua egitea da. Horretarako, hainbat zereginetan banatu da lan-paketea.

Z2.1 Software bertsioak egiteko eta iturri-kodea biltegiratzeko sistemarekin bete daitezkeen helburuak identifikatu

Software proiektuen iturri-kodea biltegiratu eta software bertsioak egiteko balio duten sistema garapen-inguruneko gainerako sistemekin nola komunikatuko den aztertuko da. Gainera, programatzaileek sistema hauekin zer-nolako harremana izango duten definituko da.

Z2.2 Software proiektuak eraikitzeko tresnarekin bete daitezkeen helburuak identifikatu

Bigarren zeregin honetan, software proiektuak eraikitzeko tresna, proiektu honetako helburuak betetzeko nola erabiliko den definituko da. Horretarako, lehendabizi, zein helburu bete daitezkeen identifikatu beharko da. Ondoren, helburu horiek betetzeko existitzen diren pluginak aukeratuz, eta existitzen ez direnak diseinatuz.

Z2.3 Etengabeko integrazioarekin bete daitezkeen helburuak identifikatu

Azken bi zereginekin egin den bezala, oraingo honetan, etengabeko integrazioak garapen-ingurunean izango dituen funtzioak definituko dira. Horretarako, lehendabizi, bete daitezkeen helburuak identifikatuz eta proiektu honetarako egokituz, beharrezkoak diren pluginak diseinatuz.

Z2.4 Tresna eta sistema guztiak garapen-ingurunean integratuz diseinua egin

Zeregin honetan, aurreko zereginetan egokitu diren sistemak garapen-ingurunean integratuko dira. Horretarako, garapen-ingurunearen diseinua egingo da.

LP.3 Garapen-ingurunearen garapena

Hirugarren lan-pakete honen helburua, bigarren lan-paketean diseinatutako garapen-ingurunea muntatzea eta ondoren probatzea da.

Z3.1 Garapen-inguruneko elementu bakoitzaren pluginak garatu

Bigarren lan-paketean diseinatu diren sistema bakoitzaren pluginak inplementatuko dira zeregin honetan.

Z3.2 Diseinatutako garapen-ingurunea inplementatu

Bigarren lan-paketean diseinatutako garapen-ingurunea muntatuko da, Z3.1 zereginean garatutako pluginak integratuz.

LP.4 Errendimendu probak eta kalitatearen azterketa

Lan-pakete honen helburu nagusia, garapen-inguruneak ondo funtzionatzen duela eta errendimendu-eskakizunak betetzen dituela ziurtatzea da.

Z4.1 Errendimendu probak

Zeregin honetan, garapen-ingurunea hardware ezberdinetan probatuko da, konfigurazio ezberdinekin instalatuz.

Z4.2 Kalitatearen azterketa

Errendimendu probetan lortutako informazioarekin, proiektu honen azken diseinu bat lortuko da, eta garapen-ingurunea muntatzeko bezeroaren zerbitzariek behar dituzten hardware eskakizunak atera ahalko dira.

LP.5 Proiektuaren kudeaketa

Bosgarren lan-paketea azkenekoa da, eta proiektu honen emaitzari jarraipena eta nola eman daitekeen eta nola hobetu daitekeen aztertzera bideratuta dago. Horretarako, lehendabizi, proiektu honen helburuak bete direla ziurtatuz. Bi zeregin definitu dira: “Proiektuaren jarraipena” eta “Dokumentazioa”.

Z5.1 Proiektuaren jarraipena

Zeregin honetan, proiektuari jarraipena eman egingo zaio, lehenengo zereginean definitutako helburuak eta eskakizunak bete direla ziurtatzeko. Ondoren, proiektu honen ondorioak aterako dira. Amaitzeko, proiektu honi jarraipena nola eman daitekeen aztertuko da.

Z5.2 Dokumentazioa

Azken zeregin hau, proiektu honi buruzko dokumentazioa egitean datza. Honi esker, proiektu honekin lotutako etorkizunean egiten diren proiektu berriei lana erraztea lortu nahi da.

11.Gantt-en diagrama

Garapen-ingurune baten diseinua eta inplementazioa

26-Jun-2018

2

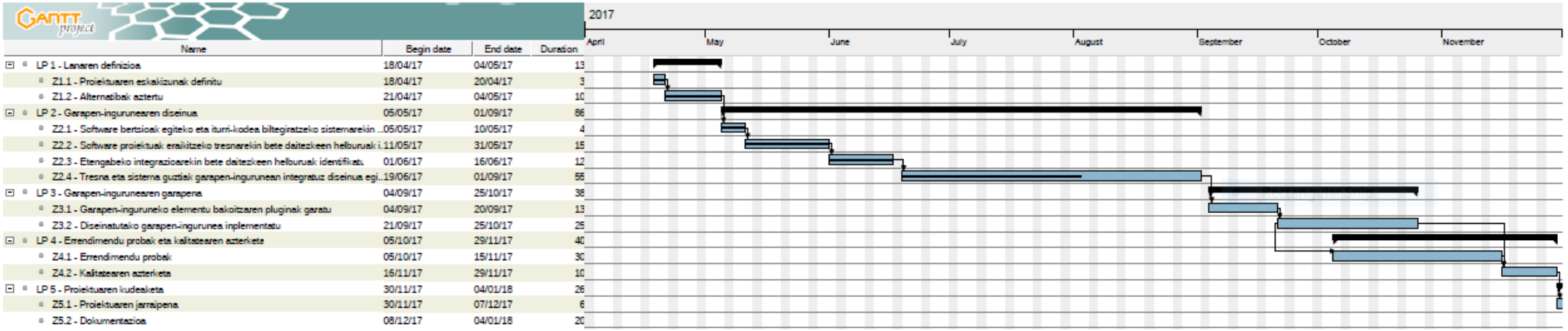
Tasks

Name	Begin date	End date	Duration
LP 1 - Lanaren definizioa	18/04/17	04/05/17	13
Z1.1 - Proiektuaren eskakizunak definitu	18/04/17	20/04/17	3
Z1.2 - Alternatibak aztertu	21/04/17	04/05/17	10
LP 2 - Garapen-ingurunearen diseinua	05/05/17	01/09/17	86
Z2.1 - Software bertsioak egiteko eta iturri-kodea biltegiratzeko sistemarekin bete daitezkeen helburuak identifikatu	05/05/17	10/05/17	4
Z2.2 - Software proiektuak eraikitzeko tresnarekin bete daitezkeen helburuak identifikatu	11/05/17	31/05/17	15
Z2.3 - Etengabeko integrazioarekin bete daitezkeen helburuak identifikatu	01/06/17	16/06/17	12
Z2.4 - Tresna eta sistema guztiak garapen-ingurunean integratuz diseinua egin	19/06/17	01/09/17	55
LP 3 - Garapen-ingurunearen garapena	04/09/17	25/10/17	38
Z3.1 - Garapen-inguruneko elementu bakoitzaren pluginak garatu	04/09/17	20/09/17	13
Z3.2 - Diseinatutako garapen-ingurunea inplementatu	21/09/17	25/10/17	25
LP 4 - Errendimendu probak eta kalitatearen azterketa	05/10/17	29/11/17	40
Z4.1 - Errendimendu probak	05/10/17	15/11/17	30
Z4.2 - Kalitatearen azterketa	16/11/17	29/11/17	10
LP 5 - Proiektuaren kudeaketa	30/11/17	04/01/18	26
Z5.1 - Proiektuaren jarraipena	30/11/17	07/12/17	6
Z5.2 - Dokumentazioa	08/12/17	04/01/18	20

Garapen-ingurune baten diseinua eta inplementazioa

26-Jun-2018

Gantt Chart



Name	Begin date	End date	Duration
LP 1 - Lanaren definizioa	18/04/17	04/05/17	13
Z1.1 - Proiektuaren ezaugarri nagusiak definitu	18/04/17	20/04/17	3
Z1.2 - Alternatibak aztertu	21/04/17	04/05/17	10
LP 2 - Garapen-ingurunearen diseinua	05/05/17	01/09/17	86
Z2.1 - Software bertsioak egiteko eta iturri-kodea biltegiratzeko sistemarekin ...	05/05/17	10/05/17	4
Z2.2 - Software proiektuak eraikitzeko tresnarekin bete daitezkeen helburuak i...	11/05/17	31/05/17	15
Z2.3 - Etengabeko integrazioarekin bete daitezkeen helburuak identifikatu	01/06/17	16/06/17	12
Z2.4 - Tresna eta sistema guztiak garapen-ingurunean integratuz diseinua egi...	19/06/17	01/09/17	55
LP 3 - Garapen-ingurunearen garapena	04/09/17	25/10/17	38
Z3.1 - Garapen-inguruneko elementu bakoitzaren pluginak garatu	04/09/17	20/09/17	13
Z3.2 - Diseinatutako garapen-ingurunea inplementatu	21/09/17	25/10/17	25
LP 4 - Errendimendu probak eta kalitatearen azterketa	05/10/17	29/11/17	40
Z4.1 - Errendimendu probak	05/10/17	15/11/17	30
Z4.2 - Kalitatearen azterketa	16/11/17	29/11/17	10
LP 5 - Proiektuaren kudeaketa	30/11/17	04/01/18	26
Z5.1 - Proiektuaren jarraipena	30/11/17	07/12/17	6
Z5.2 - Dokumentazioa	08/12/17	04/01/18	20

12.Probak eta emaitzak

Atal honetan diseinatutako garapen-ingurunearen errendimendua aztertuko da. Azterketa hau egiteko, Jenkins aplikazioan zentratuko gara, aplikazio hau baita garapen-ingurunearen nukleoa: liburutegi eta aplikazioak eraikitzeaz eta gainontzeko tresnak (Artifactory, Gitlab eta WAS zerbitzaria) erabiltzeaz arduratzen da.

Erabiltzaileek egiten duten zerbitzu baten eskaera zerbitzu horren gaitasuna baino handiagoa denean ilara sortzen da. Hau da hain zuzen Jenkinseko *job*ekin gertatu daitekeena. Horregatik, garapen-ingurunearen errendimendua aztertzeko, helburuetan aipatu den bezala, Jenkinsek aplikazio eta liburutegi bakoitzaren eraikuntza exekutatzeko behar duen **batezbesteko denbora** eta *job* bat hasi aurretik **ilaran batezbeste itxaroten egon daitekeen denbora** kalkulatuko da. Kalkulu hauek egiteko **ilaren teoria** erabili da

Kalkulatuko diren bi aldagai horiek erlazionatuta daude, proiektu bat eraikitzeko geroz eta denbora gutxiago behar badu Jenkinsek, orduan eta denbora gutxiago egongo dira *job*ak ilaran exekutatzeko zain.

Horregatik, lehendabizi Jenkinsen konfigurazio ezberdinak eta ekipoen *hardware* ezberdinak probatuko dira, kasu bakoitzarekin *job*ak hainbat aldiz exekutatuz, bakoitza exekutatzen iraundako denbora kalkulatuz. Kasu bakoitzean, lortutako denborekin, *job*en eraikitze denboren batezbestekoa kalkulatuko da.

Ondoren, lortutako batezbesteko denbora horiekin, beste hainbat parametro lortuko dira:

- *Job* bat **ilaran batezbeste zain egongo den denbora**. Hau da, *job* bat exekutatzeko agindua eman denetik exekutatzen hasten den arteko denbora. Izan ere, Jenkinsek aldi berean exekutatu ditzakeen *job* kopurua mugatua da, nahiz eta konfiguragarria izan.

- **Proiektu baten *job*a exekutatzeko agindua eman denetik exekuzioa amaitzen den arteko batezbesteko denbora**. Hau da, *job*a ilaran zain egon den denbora gehi exekutatzeko behar duen batezbesteko denbora.

- **Proiektu baten *job*a exekutatzeko agindua eman denetik exekuzioa amaitzen den arteko batezbesteko denbora 150 segundo baino handiagoa izateko probabilitatea**. Izan ere, denbora hori 150 baino handiagoa bada, denbora handiegia dela eta onartezina dela kontsideratzen du bezeroak. Beraz, helburuetan aipatu den bezala, probabilitate hori % 20 baino txikiagoa izan behar da.

- Proiektu baten *job*a exekutatzeko agindua eman denetik, zuzenean exekutatu beharrean, beste *job* batzuk exekutatzeaz amaitu arte zain egoteko probabilitatea.

12.1. Errendimendua aldatzeko aukerak

Atal honetan, garapen-ingurunearen errendimendua nola aldatu daitekeen aztertu da. Horretarako, konfiguratu daitezkeen aldagaiak eta aldatu daitekeen *hardware*a identifikatu da. Hau da, ingurunea muntatuko den makinaren *hardware* ezaugarrietan, etengabeko integrazioko Jenkins sistemak eskaintzen duen konfigurazio ezberdinetan eta Javaren makina birtualeko (JVM - *Java Virtual Machine*) konfigurazioan erreparatu da.

*Hardware*ari dagokionez, ingurunea muntatuko den prozesadoreetan, prozesadore kopuruetan eta RAM memorian erreparatuko da.

Jenkinsen konfigurazioari dagokionez, alde batetik Jenkinsek aldi berean exekutatu ditzakeen *job* kopurua aztertuko da. Bestetik, Jenkinsen arkitektura aztertuko da. Izan ere, Jenkinseko *job*ak exekutatzeak kostu konputazional handia du. Horregatik aldi berean exekutatu daitezkeen *job* kopurua mugatua da. Kopuru hori mugatua denez, jobak blokeatuta geratu daitezke exekutatzeko zain, beraz, denbora hori ahalik eta laburrena izan dadin lortuko da atal honetan.

Jenkinsen arkitekturari dagokionez, *job* guztiak Jenkinsen instantzia bakarrak exekutatu beharrean, *master-slave* [57] moduko arkitektura erabiltzeko aukera dago. Horrela, baliabideak ekipo bakarra hobetzen inbertitu beharrean, Jenkins exekutatzen den makinen kopurua handitu daiteke, *job*ak makina ezberdinetan zehar exekutatuz. Hau da, bertikalki eskalatu beharrean makina baten *hardware*a hobetuz, horizontalki eskalatzeko aukera dago. Horizontalki eskalatzearen abantaila nagusia ekonomikoki merkeagoa dela da.

JVMren konfigurazioari dagokionez, *Heap* izeneko memoria konfiguratu daiteke. JVMa Java aplikazioak exekutatzen dituen plataforma edo *software*a da. JVMak Java aplikazioak exekutatzeko, Sistema Eragileari (SE) memoria eskatu eta kentzen dio. SEari kendutako memoria JVMak hainbat zatitan banatzen du, horietako bat *Heap* deitzen delarik. *Heap* memorian aplikazio baten exekuzioan instantziatutako Java objektuak gordetzen dira, eta JVM arduratzen da beharraren arabera memoria horren tamaina handitzeaz edo txikitzeaz exekuzioan zehar. Baina memoriaren handitze eta txikitze hori kontrolatzen ez bada, bi arazo gerta daitezke:

1. Java programa batek, kasu honetan Jenkinsek (Jenkins Java aplikazio bat baita, JVMan exekutatzen dena), *Heap* memoria asko behar izatea eta SEak memoriarik libre ez duenez, gai ez izatea eskatzen duen memoria guztia emateko. Kasu honetan SEak memoriarik ez duela erantzungo dio JVMari eta exekuzioa eten egingo da.

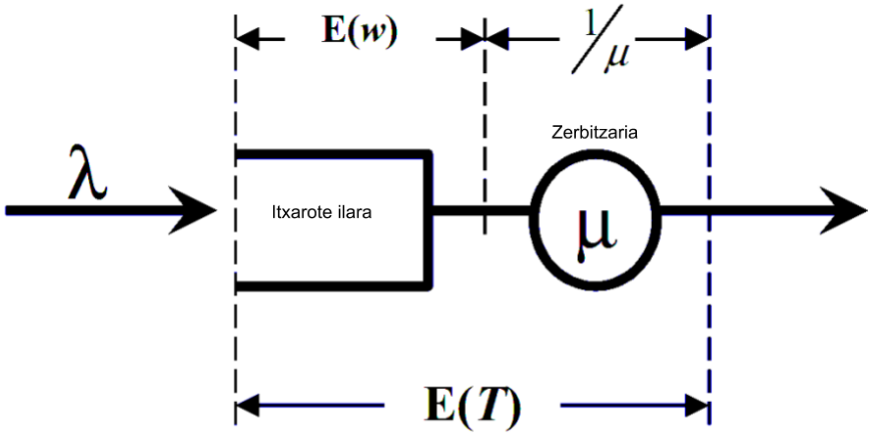
2. Bestetik, JVMak *Heap* txikiegia konfiguratuta izatea. Kasu honetan, programa bat exekutatzeko baino memoria gutxiago edukiko du JVMak eta ondorioz ez da aplikazioa exekutatzeko gai izango, SEari ez baitio gehiago eskatuko.

Horregatik, JVMko *Heap*aren hasierako tamaina eta gehienez har dezakeen tamainaren balioa konfiguratzea garrantzitsua da. Balio hauek oso loturik daude makinaren RAM memoriarekin. Izan ere, geroz eta RAM memoria handiagoa eduki makinak, orduan eta handiagoa izango da SEak JVMari eman dakiokeen memoria kantitatea.

12.2. Ilaren teoria

Ilaren teoriak Jenkinsen errendimenduaren analisi kuantitatiboa egitea ahalbidetzen du. Izan ere, ilaren teoriak analisi kuantitatiboa burutzeko, tresna probabilistikoak eskaintzen ditu.

Ilara bat ondorengo irudian ikus daitekeen bezala modelatzen du ilaren teoriak:



Irudia 21: Ilara baten atalak eta parametroak

Irudi horretan hainbat atal eta parametro agertzen dira. Bi atal izanik, itxarote ilara bat eta zerbitzari bat. Proiektu honen kasuan, zerbitzari bat baino gehiago egon daitezke, eta Jenkinseko joben exekutatzaileak izango dira. Exekutatzaile guztientzat ilara bakar bat egongo da, exekutatu beharreko *job*ak exekutatzeko zain egongo diren ilara hain zuzen.

Zerbitzariaren gaitasuna (zerbitzu tasa) μ aldagaiarekin zehazten da, proiektu honen kasuan *job*/s unitatez neurtua. Hau da, zerbitzu tasak Jenkinsen segundoko exekutatzen den *job* kopurua adierazten du. Sistemara iristen diren *job*ak exekutatzeko aginduen etorrera λ aldagaiaz adierazten da, eta zerbitzu tasaren kasuan bezala, *job*/s unitatez neurtu da.

Beraz, *job* baten aginduak sisteman (Jenkinsen) igarotzen duen batezbesteko denbora osoa E(T) aldagaiaz adieraziko da. Hau E(T) balioak, *job* baten aginduak ilara itxaroten duen denbora, gehi *job*a exekutatzeko behar izan den denbora izango da. *Job* baten aginduak itxarote ilaran igarotzen duen denbora E(w) aldagaiaz adierazten da, eta zerbitzatzen (*job*a exekutatzen) igarotzen duen denbora 1/μ moduan adierazten da.

12.2.1. Ilararen identifikazioa

Ilara bat identifikatzeko hainbat parametro ezagutu behar dira:

- Etorreren arteko denboraren banaketa.
- Zerbitzu denboraren banaketa.
- Sisteman paraleloan dauden zerbitzari kopurua. Hau da, proiektu honetan, *job*en exekutatzaile kopurua.
- Ilararen gaitasuna, hau da, sisteman (Jenkinsen) aldi berean egon daitezkeen *job* kopurua: bai ilaran daudenak kontutan hartuz, baita exekutatzen ari direnak ere.
- Ilara mota: FIFO (*First In First Out*), LIFO (*Last In First Out*), etab.

Etorrerei dagokienez (*job*ak exekutatzeko aginduak), gizakiak abiatutako jazoerak dira. Hori dela eta, Poisson-en banaketa jarraitzen duten prozesuak dira. Honek, etorreren denborak beraien artean independenteak eta esponentzialki banatutako denborak direla esan nahi du.

Zerbitzu denboren banaketari dagokionez, banaketa esponentziala ere jarraitzen du. *Job*en exekutatzaile kopuruari dagokionez, kopurua konfiguragarria da eta proba ezberdinak egingo dira errendimendua aztertzeko balio ezberdinekin.

Ilarari buruz, Jenkinseko ilara infinitutzat hartuko da, FIFO motakoa dena. FIFO motako ilaretan, ilaran sartutako ordena berean ateratzen dira ilaratik. Hau da, proiektu honen kasuan, ilarara lehendabizi iritsi den *job*a exekutatzeko agindua izango da ilaratik irtengo den lehena.

Ilara identifikatzeko ilararen parametroak ere zehaztu behar dira. Etorrera tasari dagokionez, proiektu honen bezeroak zehaztu duenez, ez da inoiz 0.01 *job*/s baino handiagoa izango. Beraz, garapen-ingurunearen errendimendu kalkuluak egiteko 0.01 *job*/s etorrera tasa erabiliko da.

Zerbitzu tasari dagokionez, *job*en exekutatzaile kopuruaren eta ilararen egoeraren araberakoa da. Hau da, demagun *job*en bi exekutatzaile daudela konfiguratuta Jenkinsen. Jenkinsera *job* bakarra exekutatzeko deia bakarrik iristen bada, zerbitzari bakarrak egingo du lan. Aipatzekoa da zerbitzu tasak banaketa esponentziala jarraitzen duela.

Behin ilara identifikatzeko beharrezko informazio guztia lortuta, Kendallen notazioa erabili daiteke ilara modu erraz batean identifikatzeko. Ilaren teorian ilarak identifikatzeko erabiltzen da Kendallen notazioa. Proiektu honetan, Kendallen notazioa modu sinplifikatuan erabiliko da, 3 balio erabiliz. Honela: **A/B/m**.

- **A**: etorreren arteko denboraren banaketa. Aipatu den bezala, etorrerek Poisson-en banaketa jarraitzen dute, hau da, esponentziala da. Banaketa esponentziala dela adierazteko “M” erabiltzen da.

- **B**: zerbitzu denboren banaketa. Aurretik aipatu den bezala, banaketa esponentziala jarraitzen dute, beraz, “M” bidez adieraziko da.

- **m**: sisteman paraleloan dauden zerbitzari kopurua. Proiektu honetan, *job*en exekutatzaile kopurua izango da, beraz, konfiguragarria da.

Beraz, errendimendu azterketa egiteko, ondorengo ilara ezberdinak probatuko dira *hardware* ezberdinekin:

- **M/M/1**
- **M/M/2**
- **M/M/3**
- **M/M/4**

Kendallen notazioak, besterik adierazi ezean ilara infinitua dela eta FIFO motakoa dela suposatzen du.

12.2.2. Ilaren errendimendua aztertzeko formulak

Zerbitzu tasa (μ).

Etorrera tasa (λ).

*Job*en exekutatzaile kopurua (m).

Ilararen erabilera edo trafiko-intentsitatea (ρ): $\rho \to 0$ gerturatu heinean, itxarote denborak handitzen dira eta oso altuak dira. Gainera, ilaran dagoen pakete kopurua oso azkar handitzen da. $\rho = \frac{\lambda}{m \cdot \mu}$

$E(n)$: ilaran batezbeste dauden *job* kopurua, exekutatzen ari direnak kontutan hartuz.

$$E(n) = \sum_{n=0}^{\infty} n \cdot p_n$$

Ekuazioa 1: Ilaran batezbeste dauden job kopurua

$E(T)$: *job*ek ilaran batezbeste igarotako denbora, exekutatzeko behar izandako denbora kontutan hartuz.

$$E(T) = \frac{E(n)}{\lambda}$$

Ekuazioa 2: Jobek ilaran batezbeste igarotako denbora

$E(w)$: *job*ek ilaran batezbeste igarotako denbora, exekutatzeko behar izandako denbora kontutan hartu gabe.

$$E(w) = E(T) - \frac{1}{\mu}$$

Ekuazioa 3: Jobek ilaran batezbeste igarotako denbora, exekutatzeko behar izandako denbora kontutan hartu gabe

p_n: Ilaran n *job* exekutatzeko agindu egoteko probabilitatea, exekutatzen ari direnak kontutan hartuz. $n \geq 0$ izango da.

p_0: Ilaran *job*-ik exekutatzeko agindurik ez egoteko probabilitatea.

$$p_n = \frac{\prod_{i=0}^{n-1} \lambda_i}{\prod_{i=1}^{n} \mu_i} \cdot p_0$$

Ekuazioa 4: Ilaran n job exekutatzeko agindu egoteko probabilitatea

Jarraian, ilara mota bakoitzera egokitzen diren formulak garatu dira.

Ilara mota	$\boldsymbol{\rho}$	$\boldsymbol{p_n}$	$\boldsymbol{p_0}$	$\boldsymbol{E(n)}$
M/M/1	$\rho = \frac{\lambda}{\mu}$	$p_n = p_0 \cdot p^n$	$p_0 = 1 - \rho$	$E(n) = \frac{\rho}{1-\rho}$
M/M/2	$\rho = \frac{\lambda}{2 \cdot \mu}$	$p_n = 2 \cdot p_0 \cdot p^n$	$p_0 = \frac{1-\rho}{1+\rho}$	$E(n) = \frac{2\rho}{1-\rho^2}$
M/M/3	$\rho = \frac{\lambda}{3 \cdot \mu}$	$p_n = \frac{9}{2} \cdot p_0 \cdot p^n$	$p_0 = \frac{2-2\rho}{2+7\rho}$	$E(n) = \frac{9\rho}{2+5\rho-7\rho^2}$
M/M/4	$\rho = \frac{\lambda}{4 \cdot \mu}$	$p_n = \frac{32}{3} \cdot p_0 \cdot p^n$	$p_0 = \frac{3-\rho}{3+29\rho}$	$E(n) = \frac{96\rho-32\rho^2}{6+75\rho-168\rho^2+87\rho^3}$

Taula 10: Ilara mota bakoitzera egokitzen diren formulak

12.3. Kalkuluak

Atal honetan, proba ezberdinak egingo dira aipatutako konfigurazioen konbinazio ezberdinak eginez. Horretarako, makina bakarrean muntatu dira Jenkins, Artifactory eta Gitlab.

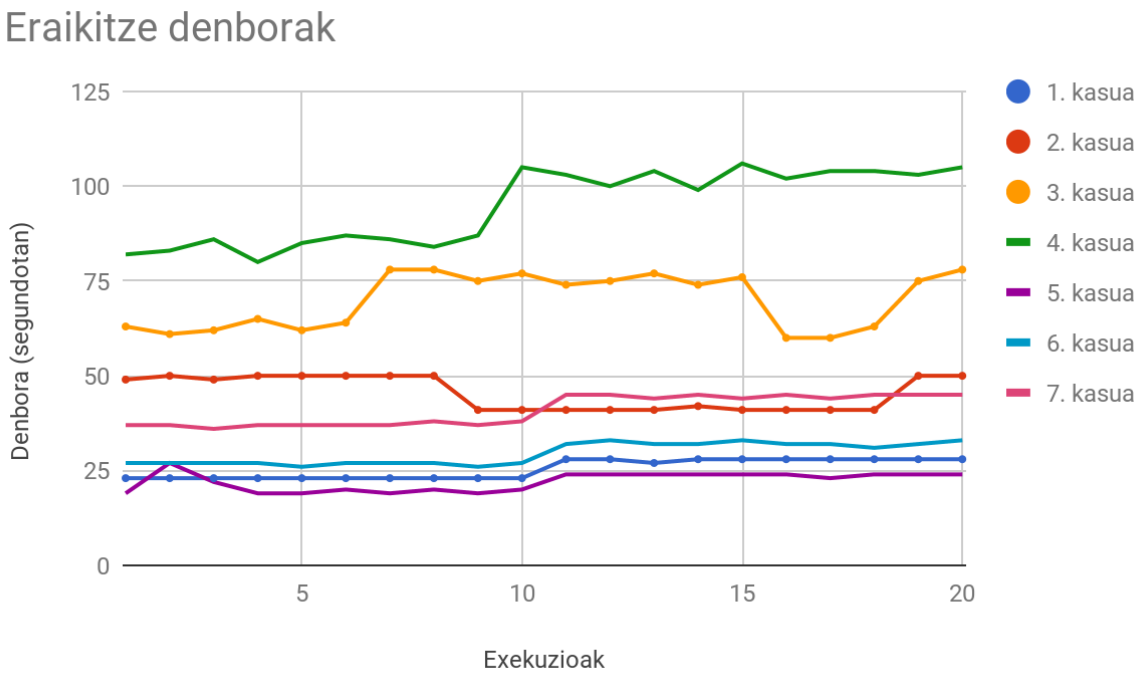
Proba hauen helburua, garapen-ingurunearen errendimendu-eskakizunak betetzen direla erakustea da, horretarako proba ezberdinak eginez.

Ezaugarria	1. kasua	2. kasua	3. kasua	4. kasua	5. kasua	6. kasua	7. kasua
Jenkins *slave* kopurua	0	0	0	0	0	0	0
*Job*en exekutatzaile kopurua	1	2	3	4	1	2	3
Prozesadore kopurua	2	2	2	2	4	4	4
Prozesadorearen abiadura (GHz)	2.4	2.4	2.4	2.4	2.4	2.4	2.4
RAM memoria (GB)	8	8	8	8	16	16	16
Jenkinsen *heap* memoria (MB)	512	512	512	512	512	512	512
Mavenen *heap* memoria (MB)	512	512	512	512	512	512	512
Ilara mota	M/M/1	M/M/2	M/M/3	M/M/4	M/M/1	M/M/2	M/M/3

Taula 11: Probatutako kasu bakoitzaren ezaugarriak

12.3.1. Lortutako eraikitze denborak

Ondorengo grafikoan kasu bakoitzean *job*ek exekutatzeko behar izan duten denborak azter daitezke. Denbora hauek lortzeko, *job*en exekutatzaile guztiak aldi berean lanean jarri dira. Hau, garapen-ingurunea denbora okerrenak eman ditzakeen egoeran jarri da.



Grafikoa 2: Kasu bakoitzean lortutako exekuzioen eraikitze denborak

Lehenengo lau kasuen arteko aldaketa bakarra Jenkinsek aldi berean exekutatu ditzakeen *job* kopurua da: lehenengoak bat, bigarrenak bi, hirugarrenak hiru eta laugarrenak lau. Argi ikusten da, baliabide berdinak dituen makina batean (*hardware* bera) geroz eta *job* gehiago exekutatu aldi berean, *job* bakoitza exekutatzeko behar den denborak gora egiten duela. Hala ere, kontutan eduki behar da, nahiz eta *job* bakoitza exekutatzeko behar den denbora igo, aldi berean *job* gehiago exekutatzen direla. Beraz, garapen-ingurunearen errendimendua aztertzeko nahitaezkoa izango da ilaren teoria aplikatzea.

Bosgarren, seigarren eta zazpigarren kasuetako denborak lortzeko, zerbitzariaren hardware baliabideak hobetu dira. Erraz atzematen da denborak hobetu egiten direla.

12.3.2. Eraikitze denboren batezbestekoak eta zerbitzu tasa

Kasua	Batezbesteko denbora (segundotan)
1. kasua	25.45
2. kasua	45.45
3. kasua	69.85
4. kasua	94.75
5. kasua	22.15
6. kasua	29.50
7. kasua	40.90

Taula 12: Kasu bakoitzean kalkulatutako batezbesteko denborak

Eraikitzeko batezbesteko denborak lortuta, kasu bakoitzean *job*en exekutatzaile bakoitzak duen zerbitzu tasa kalkulatu daiteke. Horretarako eraikitze denboraren alderantzizkoa kalkulatuz:

Kasua	Zerbitzu tasa (*job*/s)	Etorrera tasa (*job*/s)
1. kasua	0.039	0.040
2. kasua	0.022	0.040
3. kasua	0.014	0.040
4. kasua	0.011	0.040
5. kasua	0.045	0.040
6. kasua	0.034	0.040
7. kasua	0.024	0.040

Taula 13: Kasu bakoitzean kalkulatutako zerbitzu tasa, eta erabiliko den etorrera tasa

12.3.3. *Softwareak* eraikitzeko batezbesteko denborak

Kasua	ρ	p_0	$E(n)$	$E(T)$	$E(w)$
1. kasua	1.018 > 1	-	-	-	-
2. kasua	0.909	0.048	10.465	261.629	216.179
3. kasua	0.931	0.016	14.328	358.209	288.359
4. kasua	0.948	0.005	18.949	473.730	378.980
5. kasua	0.886	0.114	7.772	194.298	172.148
6. kasua	0.590	0.258	1.810	45.252	15.752
7. kasua	0.545	0.156	1.856	46.390	5.490

Taula 14: Kasu bakoitzean kalkulatutako ilararen parametroak

12.3.4. Job bat exekutatzeko agindua eman eta itxarote ilarara pasatzeko probabilitatea

Job bat exekutatzeko agindua eman eta zuzenean exekutatu beharrean itxarote ilara batetara pasatzeko probabilitatea kalkulatu da atal honetan. Horretarako beharrezkoa da *job* exekutatzaileen kopurua kontutan hartzea. Izan ere, exekutatzaile bakarra badago, bigarren *job* bat sistemara iristean itxarote ilarara pasako da. Aldiz, 4 exekutatzaile badaude, 5. *job*a exekutatzeko agindutik aurrera pasako dira *job*ak itxarote ilarara.

Kalkulu hauek egiteko erabili den formula, bat ken itxarote ilararik ez dauden egoeren probabilitateen batura egingo da:

Ilara mota	Formula
M/M/1	$p = 1 - (p_0 + p_1)$
M/M/2	$p = 1 - (p_0 + p_1 + p_2)$
M/M/3	$p = 1 - (p_0 + p_1 + p_2 + p_3)$
M/M/4	$p = 1 - (p_0 + p_1 + p_2 + p_3 + p_4)$

Taula 15: Ilara motaren arabera, itxarote ilaran gutxienez job bat egoteko probabilitatea kalkulatzeko formulak

Behin formulak lortuta, kasu bakoitzean egoera bakoitza gertatzeko probabilitateak kalkulatu dira:

Kasua	p_0	p_1	p_2	p_3	p_4
1. kasua	-	-	-	-	-
2. kasua	0.0477	0.087	0.079		
3. kasua	0.016	0.068	0.063	0.059	
4. kasua	0.005	0.052	0.049	0.047	0.044
5. kasua	0.114	0.101			
6. kasua	0.258	0.304	0.180		
7. kasua	0.156	0.384	0.209	0.114	

Taula 16: Kasu bakoitzean, itxarote ilaran gutxienez job bat egoteko probabilitatearen kalkuluak

Amaitzeko, beharrezko datu guztiak lortuta, kasu bakoitzean *job* bat exekutatzeko agindua eman eta *job* zuzenean exekutatu beharrean itxarote ilarara pasatzeko probabilitatea kalkulatu da:

Kasua	Itxarote probabilitatea
1. kasua	-
2. kasua	0.787
3. kasua	0.795
4. kasua	0.802
5. kasua	0.785
6. kasua	0.258
7. kasua	0.137

Taula 17: Kasu bakoitzaren itxarote probabilitatea

12.3.5. Eraikitze denbora 150 segundo baino handiagoa izateko probabilitatea

Helburuetan aipatu den bezala, *job* bat exekutatzeko agindua eman denetik *job* hori exekutatzeaz amaitzen den arteko batezbesteko denbora 150 segundo baino handiagoa izateko probabilitatea ezin da % 20 baino handiagoa izan.

Kalkulu hau egiteko ondorengo formula erabili da:

$$p(E(T) > t) = e^{-\mu(1-\rho)t}$$

Ekuazioa 5: Eraikitze denbora t segundo baino handiago izateko probabilitatea kalkulatzeko formula

Beraz, formula hori aplikatuta ondorengo emaitzak lortu dira:

Kasua	$\boldsymbol{p(E(T) > 150\ s)}$
1. kasua	-
2. kasua	0.741
3. kasua	0.863
4. kasua	0.920
5. kasua	0.462
6. kasua	0.124
7. kasua	0.189

Taula 18: Kasu bakoitzean, eraikitze denbora 150 segundo baino handiago izateko probabilitatearen kalkuluak

12.4. Emaitzak

Errendimendu-eskakizun guztiak zein konfiguraziok betetzen duten modu erraz batean aztertzeko, emaitza guztiak taula bakarrean jarri dira:

Kasua	Eraikitzeko batezbesteko denbora (s): $E(T)$	Itxarote probabilitatea	Itxarote ilaran igaro beharreko batezbesteko denbora (s): $E(w)$	$p(E(T) > 150\ s)$
1. kasua	-	-	-	-
2. kasua	261.629	0.787	216.179	0.741
3. kasua	358.209	0.795	288.359	0.863
4. kasua	473.730	0.802	378.980	0.920
5. kasua	194.298	0.785	172.148	0.462
6. kasua	45.252	0.258	15.752	0.124
7. kasua	46.390	0.137	5.490	0.189

Taula 19: Kasu bakoitzean kalkulatutako ilararen parametroen emaizak

Betetzen ez diren helburuak gorriz jarri dira. Beraz, emaitzak erreparatuz argi ikusten da errendimendu-eskakizunak 7. kasuan bakarrik betetzen direla.

Beraz, ondorengo ezaugarriak dituen makina batean muntatu behar da garapen-ingurunea errendimendu-eskakizunak bete ahal izateko:

Ezaugarria	Balioa
Jenkins esklabo kopurua	0
*Job*en exekutatzaile kopurua	3
Prozesadore kopurua	4
Prozesadorearen abiadura (GHz)	2.4
RAM memoria (GB)	16
Jenkinsen *heap* memoria (MB)	512
Mavenen *heap* memoria (MB)	512
Ilara mota	M/M/3

Taula 20: Errendimendu-eskakizunak bete ahal izateko garapen-inguruneak izan behar dituen ezaugarriak

Amaitzeko, aipatzekoa da garapen-ingurunearen errendimendua hobetzeko aukeren artean Maven eta Jenkinsen *Heap* memoria aldatzea, Jenkinsen *master-slave* arkitektura erabiltzea (horizontalki eskalatzeko) eta makinaren prozesadorea hobetzea dagoela. Aukera horiek atal honen hasieran aipatu dira, baina ez da horiek erabiltzeko beharrik izan errendimendu-eskakizunak betetzeko.

ALDERDI EKONOMIKOAK

III. ZATIA

13.Aurrekontuaren deskribapena

Atal honetan exekutatutako proiektuaren aspektu ekonomikoak aztertuko dira. Horretarako, zati ezberdinetan banatuko da, ondoren gastu guztien laburpen bat eginez.

13.1. Barne-orduak

Ondorengo taulan, proiektuan parte hartutako lan-taldearen rolak eta bakoitzaren kostuak agertzen dira:

KARGUA	ORDUKO KOSTUA (€/h)
Project Director	50
Project Manager	50
Garatzailea	10

Taula 21: Lan-taldeko rol bakoitzak duen kostua orduko

Kargu horiek kontutan hartuz, aurreko atalean aztertu diren atazak eta zereginak nola banatu diren, eta ondorioz, zeregin bakoitzean barne-orduek izandako kostea kalkulatu da:

	Project Director		Project Manager		Garatzailea	
	Orduak	**Kostua**	**Orduak**	**Kostua**	**Orduak**	**Kostua**
Z1.1	0 h	0 €	8 h	400 €	24 h	240 €
Z1.2	0 h	0 €	1 h	50 €	80 h	800 €
Z2.1	0 h	0 €	3 h	150 €	40 h	400 €
Z2.2	0 h	0 €	5 h	250 €	160 h	1 600 €
Z2.3	0 h	0 €	5 h	250 €	144 h	1 440 €
Z2.4	0 h	0 €	16 h	800 €	640 h	6 400 €
Z2.5	0 h	0 €	0 h	0 €	320 h	3 200 €
Z3.1	15 h	750 €	15 h	750 €	48 h	480 €
Z3.2	35 h	1 750 €	35 h	1 750 €	160 h	1 600 €
BATURAK	50 h	2 500 €	88 h	4 400 €	1 616 h	16 160 €
TOTALA						23 060 €

Taula 22: Barne-orduen kostua

13.2. Amortizazioak

Proiektu hau gauzatzeko ez da material handirik behar. Garapen-ingurunea digitala izanik, ahal den heinean software librea erabiliko da. Horri esker, bi ekipo erostearekin nahikoa da proiektu hau aurrera ateratzeko amortizazioei dagokienez. Izan ere, aukera bat WebSphere aplikazioen zerbitzariaren lizentzia bat erostea litzateke, baina dagoeneko bezeroak bere zerbitzarietan duenarekin nahikoa dela aurreikusten da.

Ondorioz, ondorengo taula erabili da proiektu honen amortizazioa kalkulatzeko:

Materiala	Kopurua	Erabilera (h)	Hasierako prezioa (€)	Bizitza erabilgarria (h)	Proiektu honetako kostua (€)
Ordenagailuak	2	400	1000	1800	444,44
GUZTIRA					444,44

Taula 23: Amortizazioen kalkulua

13.3. Gastuak

Proiektu honen garapenerako, bulegoko materialez gain, garapen-ingurunearen probak egiteko zerbitzari batzuk behar dira. Izan ere, bezeroaren zerbitzarietan garapen-ingurunearen azken diseinua instalatuko da. Baina azken diseinua lortu aurretik egin behar diren probak eta errendimendu azterketak, beste zerbitzari batzuetan egin behar dira.

Proba horiek guztiak egiteko, Amazon Web Services [58] plataformak eskaintzen dituen zerbitzuak eta makina birtualak erabiliko dira. Beraz, Amazonek eskaintzen dituen zerbitzuen prezioak aztertuz, marjina bat kontutan hartuz, orduko 1 € gastua izango dela kalkulatzen da [59] Frankfurteko prezioak begiratuz Amazon EC2 zerbitzuan.

Beraz, gastuen ondorengo taula kalkulatu da:

Produktua	Zenbatekoa (€)
Bulegoko materiala	500
Amazon Web Services	10 000
GUZTIRA	10 500

Taula 24: Gastuen kalkulua

13.4. Gastu ez-zuzenak

Gastu ez-zuzenak proiektuan era zuzen batean eragina ez duten gastuak dira. Adibidez, garatzen den bitartean gastatutako argindarra, bulegoaren alokairua, etab. Gastu hauen aurrekontua egiteko, azpitotalaren % 10 erabiliko da, nahikoa izango delakoan.

13.5. Gastu guztien laburpena

Atal honetan aurretik aipatu diren gastu guztien laburpena egingo da, aurrekontu osoa kalkulatuz.

Horretarako, aurreko puntuetan banatuta aztertu diren zati guztiak batu dira. Azpitotala kalkulatzeko ez dira gastu ez-zuzenak kontutan hartzen. Izan ere, gastu ez-zuzenak kalkulatzeko azpitotalaren ehuneko bat erabiltzen da. Horregatik, lehendabizi azpitotala zuzena kalkulatu da, eta ondoren azpitotal osoa. Betiere BEZa kontutan hartu gabe:

Kontzeptua	Zenbatekoa (€)
Barne-orduak	23 060,00
Amortizazioak	444,44
Gastuak	10 500,00
AZPITOTAL ZUZENA	34 004,44
Gastu ez-zuzenak (% 10)	3 400,44
GUZTIRA	37 404,88

Taula 25: Gastu guztien laburpena

Beraz, proiektu hau gauzatu ahal izateko behar den zenbatekoa hogeita hamazazpi mila laurehun eta lau koma laurogeita zortzi euro (37.404,88 €) dira.

ONDORIOAK

IV. ZATIA

14. Ondorioak

Proiektu honetan garatu den garapen-inguruneari esker, proiektu honen bezeroak duen arkitekturara moldatzen diren proiektuak modu erraz batean garatzea eta mantentzea ahalbidetzen da. Garatutako garapen-ingurunea ahalik eta automatizatuena egin da, programatzaileak ahal den heinean liburutegi eta aplikazioak garatzeaz bakarrik arduratu daitezen. Hala ere, programatzaileek garapen-ingurune honetan programatzeko, beharrezkoa dute Git sistema erabiltzen ikastea.

Dokumentuan zehar proiektu honen bezeroak liburutegi eta aplikazio ugari garatzen dituela aipatu da, proiektu hauek garatzen programatzaile eta bezero askok parte hartzen dutela. Egin den errendimenduaren azterketari esker, garapen-ingurunea muntatuko den makinak izan behar dituen eskakizunak lortu dira. Horrela, proiektu eta programatzaile kantitate horrekin garapen-inguruneak arazorik ez izatea ziurtatu da.

Amaitzeko, aipatzekoa da proiektu honi jarraipena emateko aukera ezberdinak daudela. Alde batetik, *software* proiektuen iturri-kodea analizatzen duen aplikazio bat gehitu daiteke garapen-ingurunera. Aplikazio hauek iturri-kodearen kalitatea aztertzen dute, adibidez, errepikatutako kodea identifikatuz, programatzeko praktika txarrak detektatuz, testek iturri-kodearen zein ehuneko exekutatzen duten aztertuz, etab.

BIBLIOGRAFIA

[1] IBM, «Introduction to WebSphere,» 04 08 2018. [Online]. Available: https://www.ibm.com/developerworks/ssa/websphere/newto/?S_TACT=105AGY80&S_CMP=GRMEX&ca=dgr-es-wikipe01. [Atzitze-data: 08 08 2018].

[2] IBM, «WebSphere 8.5.5,» [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/SSEQTP_8.5.5/as_ditamaps/was855_welcome_base_dist_iseries.html. [Atzitze-data: 08 08 2018].

[3] Oracle, «JAR,» [Online]. Available: https://docs.oracle.com/javase/1.5.0/docs/guide/jar/jar.html. [Atzitze-data: 08 08 2018].

[4] Oracle, «EAR,» [Online]. Available: https://docs.oracle.com/javaee/5/tutorial/doc/bnaby.html#indexterm-47. [Atzitze-data: 08 08 2018].

[5] IBM, «Java versions for WebSphere,» [Online]. Available: https://www-01.ibm.com/support/docview.wss?uid=swg27005002. [Atzitze-data: 08 08 2018].

[6] Apache, «Subversion,» [Online]. Available: https://subversion.apache.org/. [Atzitze-data: 08 08 2018].

[7] D. a. Wheeler, «Revision-Control Systems,» [Online]. Available: https://www.dwheeler.com/essays/scm.html. [Atzitze-data: 08 08 2018].

[8] Object Technology International, «Eclipse,» [Online]. Available: https://web.archive.org/web/20130402233256/http://www.eclipse.org/whitepapers/eclipse-overview.pdf. [Atzitze-data: 08 08 2018].

[9] Technopedia, «Integrated Development Environment,» [Online]. Available: https://www.techopedia.com/definition/16376/development-environment. [Atzitze-data: 08 08 2018].

[10] Oracle, «Classpath,» [Online]. Available: https://docs.oracle.com/javase/tutorial/essential/environment/paths.html. [Atzitze-data: 08 08 2018].

[11] M. Rouse, «Development environment,» [Online]. Available: https://searchsoftwarequality.techtarget.com/definition/development-environment. [Atzitze-data: 08 08 2018].

[12] M. J. Rochkind, «The Source Code Control System,» [Online]. Available: https://basepath.com/aup/talks/SCCS-Slideshow.pdf. [Atzitze-data: 08 08 2018].

[13] GNU, «Revision Control System,» [Online]. Available: https://www.gnu.org/software/rcs/rcs.html. [Atzitze-data: 08 08 2018].

[14] K. F. a. M. Bar, «Concurrent Versions System,» [Online]. Available: http://cvsbook.red-bean.com/cvsbook.html. [Atzitze-data: 08 08 2018].

[15] N. Gift, «Introduction to distributed version control systems,» [Online]. Available: https://web.archive.org/web/20090602084310/http://www.ibm.com/developerworks/aix/library/au-dist_ver_control/. [Atzitze-data: 08 08 2018].

[16] GNU, «GNU Arch,» [Online]. Available: https://www.gnu.org/software/gnu-arch/. [Atzitze-data: 08 08 2018].

[17] Canonical, «Bazaar,» [Online]. Available: https://bazaar.canonical.com/en/. [Atzitze-data: 08 08 2018].

[18] GIT, «GIT,» [Online]. Available: https://git-scm.com/. [Atzitze-data: 08 08 2018].

[19] Mercurial, «Mercurial,» [Online]. Available: https://www.mercurial-scm.org/. [Atzitze-data: 08 08 2018].

[20] Amazon, «Continuous Integration,» [Online]. Available: https://aws.amazon.com/es/devops/continuous-integration/?nc1=h_ls. [Atzitze-data: 08 08 2018].

[21] Amazon, «Continuous Delivery,» [Online]. Available: https://aws.amazon.com/es/devops/continuous-delivery/. [Atzitze-data: 08 08 2018].

[22] GNU, «Make,» [Online]. Available: https://www.gnu.org/software/make/manual/make.html. [Atzitze-data: 08 08 2018].

[23] Apache, «Ant + Ivy,» [Online]. Available: https://ant.apache.org/ivy/. [Atzitze-data: 08 08 2018].

[24] Apache, «Maven,» [Online]. Available: https://maven.apache.org/what-is-maven.html. [Atzitze-data: 08 08 2018].

[25] Gradle, «Gradle,» [Online]. Available: https://gradle.org/. [Atzitze-data: 08 08 2018].

[26] K. Eliason, «Difference between object-oriented programming and procedural programming languages,» [Online]. Available: https://neonbrand.com/website-design/procedural-programming-vs-object-oriented-programming-a-review/. [Atzitze-data: 08 08 2018].

[27] Apache, «Introduction to the Build Lifecycle,» [Online]. Available: https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html. [Atzitze-data: 08 08 2018].

[28] Groovy, «Groovy,» [Online]. Available: http://groovy-lang.org/. [Atzitze-data: 08 08 2018].

[29] Jenkins, «Jenkins,» 03 09 2018. [Online]. Available: https://jenkins.io/.

[30] Hudson, «Hudson,» 03 09 2018. [Online]. Available: http://hudson-ci.org/.

[31] Travis, «Travis CI,» 03 09 2018. [Online]. Available: https://travis-ci.com/.

[32] Semantic Versioning, «Semantic Versioning,» 03 09 2018. [Online]. Available: https://semver.org/.

[33] Subversion, «Versioning Models,» 03 09 2018. [Online]. Available: http://svnbook.red-bean.com/en/1.0/ch02s02.html.

[34] Gitlab, «Gitlab,» 03 09 2018. [Online]. Available: https://about.gitlab.com/.

[35] O. White, «Zeroturnaround,» 03 09 2018. [Online]. Available: https://zeroturnaround.com/rebellabs/java-build-tools-part-2-a-decision-makers-comparison-of-maven-gradle-and-ant-ivy/.

[36] Gitlab, «Gitlab CI,» 03 09 2018. [Online]. Available: https://about.gitlab.com/features/gitlab-ci-cd/.

[37] Jenkins, «Hudson future,» 03 09 2018. [Online]. Available: https://jenkins.io/blog/2011/01/11/hudsons-future/.

[38] Jenkins, «Jenkins download,» 03 09 2018. [Online]. Available: https://jenkins.io/download/.

[39] Travis, «Travis CI plans,» 03 09 2018. [Online]. Available: https://travis-ci.com/plans.

[40] Gitlab, «Gitlab installations,» 03 09 2018. [Online]. Available: https://about.gitlab.com/installation/.

[41] Gitlab, «Webhooks,» 03 09 2018. [Online]. Available: https://docs.gitlab.com/ce/user/project/integrations/webhooks.html.

[42] Apache, «Maven: Introduction to the Standard Directory Layout,» 03 09 2018. [Online]. Available: https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html.

[43] Apache, «Maven: POM reference,» 03 09 2018. [Online]. Available: https://maven.apache.org/pom.html.

[44] Apache, «Maven: Available plugins,» 03 09 2018. [Online]. Available: https://maven.apache.org/plugins/index.html.

[45] Apache, «Maven Compiler Plugin,» 03 09 2018. [Online]. Available: https://maven.apache.org/plugins/maven-compiler-plugin/.

[46] Apache, «Maven EAR Plugin,» 03 09 2018. [Online]. Available: https://maven.apache.org/plugins/maven-ear-plugin/.

[47] Apache, «Maven Deploy Plugin,» 03 09 2018. [Online]. Available: https://maven.apache.org/plugins/maven-deploy-plugin/.

[48] IBM, «Java EE application deployment descriptors,» 03 09 2018. [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/SSAW57_8.5.5/com.ibm.websphere.nd.multiplatform.doc/ae/trun_app_deploymtdesc.html.

[49] GIT, «Git Basics: Tagging,» 03 09 2018. [Online]. Available: https://git-scm.com/book/en/v2/Git-Basics-Tagging.

[50] Jenkins, «GIT Plugin,» 03 09 2018. [Online]. Available: https://wiki.jenkins.io/display/JENKINS/Git+Plugin.

[51] Jfrog, «Artifactory,» 03 09 2018. [Online]. Available: https://jfrog.com/artifactory/.

[52] Jenkins, «Gitlab Plugin,» 03 09 2018. [Online]. Available: https://github.com/jenkinsci/gitlab-plugin.

[53] Gitlab, «Webhooks: push events,» 03 09 2018. [Online]. Available: https://docs.gitlab.com/ce/user/project/integrations/webhooks.html#push-events.

[54] Jenkins, «Plugin tutorial,» 03 09 2018. [Online]. Available: https://wiki.jenkins.io/display/JENKINS/Plugin+tutorial.

[55] Mojohaus, «Versions Maven Plugin,» 03 09 2018. [Online]. Available: https://www.mojohaus.org/versions-maven-plugin/.

[56] Jenkins, «Parameterized build,» 03 09 2018. [Online]. Available: https://wiki.jenkins.io/display/JENKINS/Parameterized+Build.

[57] Jenkins, «Distributed builds,» 03 09 2018. [Online]. Available: https://wiki.jenkins-ci.org/display/JENKINS/Distributed+builds.

[58] Amazon, «Amazon Web Services,» 03 09 2018. [Online]. Available: https://aws.amazon.com/es/.

[59] Amazon, «Amazon Web Services pricing,» 03 09 2018. [Online]. Available: https://aws.amazon.com/es/ec2/pricing/on-demand/.

I. ERANSKINA

V. ZATIA

1. Sarrera

Dokumentu honetan, proiektuaren gauzatze egokirako kontratistak eta proiektuaren garatzaileak onartu beharko dituzten baldintzak biltzen dira. Horregatik dokumentu hau loteslea da, bien arteko kontratuan sartuz.

2. Baldintza teknikoak

2.1. Baliabide materialak

Atal honetan, proiektu hau garatzeko beharrezkoak diren baliabide materialak azalduko dira, bai *hardware* baliabideak, baita *software* baliabideak ere.

2.1.1. *Hardware* baliabideak

Hardware baliabideei dagokienez, erabilera handia eta beharrezkoa duten baliabideak aipatuko dira. Ondorioz, momentu puntualetan erabilitako ekipo pertsonalak ez dira aintzat hartuko.

Proiektuaren garapenerako, Amazon Web Services-eko EC2 instantzia ezberdinak erabiliko dira. Instantzia mota ezberdinekin garapen-ingurunearen errendimendua aztertuko da:

- **t2.micro:** Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz motako prozesadore bakarra eta 1 GB RAM memoria duen instantzia.
- **t2.small:** Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz motako prozesadore bakarra eta 2 GB RAM memoria dituen instantzia.
- **t2.medium:** Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz motako bi prozesadore eta 4 GB RAM memoria dituen instantzia.
- **t2.large:** Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz motako bi prozesadore eta 8 GB RAM memoria dituen instantzia.
- **t2.xlarge:** Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz motako lau prozesadore eta 16 GB RAM memoria dituen instantzia.

2.1.2. Software baliabideak

Azpiatal honetan, proiektua gauzatzeko beharrezkoak diren programazio eta garapen profesionaleko *software* tresnak aipatuko dira. Erabili diren *software* gehienak lizentzia askekoak izango dira, hala ere, lizentzia batzuk ere beharrezkoak dira:

- Garapenerako eta administraziorako sistema-eragilea: Ubuntu 16.04
- Ingurune ofimatikoa: Microsoft Office 2016 Professional
- Interfazearen garapen-ingurunea: Eclipse
- Garapen-ingurunea muntatu beharreko makinaren sistema-eragilea: CentOS 7 - x86_64
- Bertsioen kudeatzailea: Gitlab (GIT)

Horien artean, Microsoft Office 2016 Professional *software*a da ordaindu beharreko lizentzia bakarra, gainontzekoak askeak izanik.

2.2. Dokumentazioa

Atal honetan, proiektuaren jabeari entregatu behar zaizkion dokumentuak daude. Dokumentuak aipatzeaz gain, dokumentu bakoitza zertan datzan eta edukia laburtuko da.

2.2.1. Lehenengo dokumentua: Memoria

Dokumentu honetan, proiektua egiteko arrazoiak azaltzen dira, bideragarria den aztertzen da, helburuak zehazten dira, eta proiektua gauzatzeak dakarren onurak azaltzen dira. Gainera, proiektuaren testuingurua finkatzen da.

Horrez gain, proiektuan gauzatzeko alternatiba ezberdinak aztertzen dira, egokiena aukeratuz, eta horien araberako soluzio baten diseinua egiten da. Diseinu horrek proiektuaren helburu guztiak beteko dituela ziurtatzen da.

2.2.2. Bigarren dokumentua: Lanerako erabilitako metodologia

Dokumentu honetan, proiektua gauzatzeko eman beharreko pausoak aztertzen dira. Horretarako pausoak atazetan banatuz eta ataza horien planifikazio bat eginez. Ondoren, Gantt-en diagrama bat ere egingo da dokumentu honetan.

2.2.3. Hirugarren dokumentua: Alderdi ekonomikoak

Dokumentu honek proiektu hau aurrera ateratzeko behar den aurrekontua jasotzen du.

2.2.4. Laugarren dokumentua: Ondorioak

Laugarren dokumentu honetan proiektu honetatik atera diren ondorioak azaltzen dira.

2.2.5. Bosgarren dokumentua: I. eranskina

Dokumentu honek, proiektua zein baldintzatan entregatu behar den biltzen du. Baldintza hauetan kontratistak eta garatzaileak onartu beharreko kontratuzko baldintzak sartzen dira. Horretaz gain, gauzatu beharreko froga-plana ere sartzen da.

3. Eginbeharrak

Atal honetan, proiektu hau gauzatzeko egin behar diren lan eta zereginak zehaztuko dira. Zeregin horiek bost lan paketetan banatu dira. Ondoren, lan pakete bakoitzaren helburuak zehaztuko eta azalduko dira.

LP.1 Lanaren definizioa

Pakete honen helburu nagusia, proiektu honen helburuak eta eskakizunak zeintzuk diren zehaztea da. Proiektu honetan garapen-ingurune bat garatuko denez, garapen-ingurune hori diseinatzeko merkatuan dauden alternatiben azterketa bat ere egingo da lan-pakete honetan.

LP.2 Garapen-ingurunearen diseinua

Pakete honetan, garapen-ingurunearen diseinua egingo da. Garapen-inguruneak hainbat elementu izango dituenez, lan-pakete hau hainbat zereginetan banatu da. Honi esker, zeregin bakoitzean elementu bakoitzak bete ditzakeen helburuak bete ahal izateko garatu behar diren gehigarriak diseinatuko dira.

Diseinuekin amaitzeko, garapen-inguruneko elementu eta gehigarri guztiak integratzen dituen sistema baten diseinua egingo da.

LP.3 Garapen-ingurunearen garapena

Nahiz eta aurreko bi paketeek izan ingeniaritza ikuspuntutik lan zama gehien dutenak, lan gehien eskatzen duen paketea honako hau da, pakete honetan kodifikatu eta garatu beharko baita aurreko paketean diseinatutakoa. Beraz, lan-pakete honetan, garapen-inguruneko elementu bakoitzaren gehigarriak garatuko dira.

Lan zama handia dela eta, aurreko lan-paketean bezala, pakete hau ere hainbat zereginetan banatu da: zeregin bat garapen-ingurunearen elementu bakoitzeko, eta beste bat elementu guztiak sistema bakarrean integratzeko.

Lan pakete hau eta aurrekoa, garrantzitsuenetakotzat hartzen dira, sistema ezberdinen diseinua egiteak eta garatzeak ingeniaritzarekin lotutako lan zama handia duelako.

LP.4 Errendimendu probak eta kalitatearen azterketa

Lan-pakete honetan, errendimendu probak definitu, gauzatu eta ondoren, lortutako emaitzekin kalitatearen azterketa bat egingo da. Helburua garapen-inguruneak, errendimendu-eskakizunak zein proiektuaren gainerako helburu guztiak betetzen dituela ziurtatzea da.

Lan pakete honen amaieran, proiektu honetako produktu finala eta proben emaitzak lortzen dira, produktua kontratistari entregatu ahalko zaiolarik.

LP.5 Proiektuaren kudeaketa

Lan pakete honek, proiektu honen kudeaketaren lan zama biltzen du. Proiektuari buruzko dokumentazio eta informazio bilaketa pakete honetan sartzen da.

Proiektua gauzatzeko egindako entregagai guztiak lan-pakete honen barruan sartzen dira.

4. Jasotze baldintzak

Atal honetan, proiektua entregatzerakoan proiektua baliozkotzat hartzeko bete behar diren baldintzak azaltzen dira.

Proiektuak garapena egokia izan duela eta hitzartutako ezaugarri guztiak betetzen dituela ziurtatzeko egin behar diren proba batzuk definitu dira. Egindako probak, garatutako sistemak onartzeko ezinbestekoak dira. Gainera, egiaztapen gehiago egitearren egindako proba gehigarriak egiteko posibilitatea dago.

4.1. Egite eta entregatze epeak

Proiektu honen dokumentuaren bederatzi eta hamargarren ataletan azaldutakoaren arabera, proiektu hau 2018ko urtarrilaren 4rako entregatu behar da.

Kontratista eta garatzailearen artean adostutako lan-planeko edozein aldaketa bi aldeen artean eztabaidatu eta onartu beharko da, beharrezko aldaketak eginez dagokion dokumentuetan, eta aurrekontu berria eginez aldatua izango balitz.

4.2. Produktuaren ustiapen eskubideak

Proiektu hau gauzatzean lortutako software sistemen eskubide guztiak kontratistaren jabetzara pasako dira, proiektu honen autore bezala agertzeko eskubidea erreserbatuz lan taldeari.

4.3. Balioztatze probak

Dokumentu honetako hamargarren atalean, balioztatze plan bat definitzen da. Proiektu honetako garapen-inguruneak bertan planifikatutakoa gainditu beharko du, eta hori gainditzean proiektua entregatuko da.

5. Baldintza ekonomikoak

Atal honetan, proiektua garatzeko errespetatu behar diren baldintza ekonomikoak azaltzen dira. Horretarako, bi azpiataletan banatu da: proiektuaren koste totalaren espezifikazioak eta proiektua ordaintzeko modua.

5.1. Proiektuaren kostea

Proiektu hau gauzatzeko beharrezkoa den guztizko zenbatekoa **hogeita hamazazpi mila laurehun eta lau koma laurogeita zortzi euro (37.404,88 €)** da, zergak kontutan hartu gabe.

5.2. Ordaintze modua

Ordainketa hiru alditan banatuko da. Proiektua onartzerakoan, 10.000 €-tako lehenengo ordainketa bat egingo da, kontratista ados dagoela proiektuarekin ziurtatzeko. Ondoren, lehenengo ordainketa egin eta sei hilabeteko epean 16.000 €-tako bigarren ordainketa bat egingo da. Azkenik, proiektua amaiturik entregatzean 11.404,88 €-tako azken ordainketa egingo da.

6. Kontratuzko baldintzak

6.1. Jasotze-aktak

Produktua balioztatutakoan, bai kontratistak eta bai proiektuaren garatzaileak behin-behineko jasotze-akta sinatu beharko dute. Horrela, produktua behin-behinean emango zaio kontratistari.

Akta sortzen den unean, erreklamazio periodoa hasiko da, bi astetako luzera izango duena. Bi aste horiek pasa ondoren, kontratistak ez badu proiektuak dituen akatsez jakinarazten, behin-betiko jasotze-akta sinatuko da.

6.2. Bezeroaren erantzukizunak

Bezeroa erantzule izango da indarrean dagoen kontratatutako zerbitzuen, erabiltze baimen eta lizentzien, jabetza intelektualen, eta garapenaren konfidentzialtasunaren arautegi legal guztiak betetzeaz.

6.3. Proiektua garatzen duenaren erantzukizunak

Proiektu hau, bezeroaren beharrak asetzen dituen diseinu bat egitean datza. Baita diseinu hori garatzean, amaieran produktu bat eskaintzeko.

Proiektu honen diseinatzailea, diseinuaren erruz produktuan ager daitezkeen akatsen erantzule izango da. Hala ere, diseinua garatzerakoan egindako akatsen edota bigarren diseinu posible batean egindako akatsen ardura ez du hartuko diseinatzaileak.

Aitzitik, garatzailea, diseinuaren garapenaz eta garapen prozesuan zehar egindako aldaketa posibleen erantzule izango da.

6.4. Kontratuaren iraungitzea

Kontratua amaitzean edota betetzean kontratua iraungituko da. Edo kontratuan sartuta dauden bi parteetako batek erabakitzen duenean iraungituko da. Honakoak izango dira kontratua iraungitzeko kausak:

- Baldintza Pleguan dauden klausulak ez betetzea.
- Bi parteetako baten merkataritza-sozietateko pertsonalitate juridikoak desagertzea, ondarea beste entitate batek jasotzen ez duen bitartean.
- Bi parteen arteko erabakiagatik.
- Bi parteetako baten kiebra deklarazioagatik edota ordainketen etenaldiagatik.

6.5. Arazoen konpontzea

Kontratuaren interpretazio eta aldaketen aurrean sor daitezkeen liskarrak Auzitegi eta Epaimahaien bitartez konponduko dira. Ondorioz, bi parteen ardura izango da arbitroen eta arbitrajearen administrazioa izendatzea, eta bi parteak hauen akatsak onartzera konprometitzen dira.

7. Aspektu juridikoak

Ondorengo aspektu juridikoak aurreikusten dira proiektu honen kontratazio eta garapenean: ezinbesteko kasua, eta arbitrajea eta epaimahaia.

7.1. Ezinbesteko kasua

Kontratista ez da betebeharrak ez betetzearen erantzule izango lanen exekuzioa atzeratu egiten bada edo ezinezkoa egiten bada ezinbesteko kasuagatik.

Ezinbesteko kasuko kausatzat hartuko dira kontratistaren edo eroslearen kontroletik kanpo dauden gertaera edota zirkunstantziak. Baita aurreikusi ezin diren zirkunstantziak, edo aurreikusi ahal izanda ere saihestezinak direnak. Hau dena, Kode Zibilean dagoen jurisprudentzia eta doktrina legalaren araberakoa da.

Aipatutako ezinbesteko kasuko kausak kontutan hartuko dira hornikuntzari zuzenean eragiten dienean.

7.2. Arbitrajea eta epaimahaia

Bai kontratista eta baita eroslea, eskuartean dugun Baldintza Pleguan eta kontratuaren dokumentu-osagarrian ezarri diren baldintzak betetzeaz konprometitzen dira. Baldintzen aplikazioan, garapenean, betetzean, exekuzioan edota interpretazioan ager daitezkeen desadostasunak euren arteko akordioen eta negoziazioen bitartez konponduz.

Bien arteko edozein desadostasun ezingo balitzateke konpondu esandako modu horretan, kontratistak arbitrajearen menpe utziko ditu desadostasunak, Zuzenbide Pribatuko Arbitrajearen Legeak esandako formalizatzeko arauak errespetatuz.

Arbitrajea Donostian egingo da, eta eskrituran nabarmendu behar da 30 eguneko epea dutela arbitroek desadostasuna ebazteko.

8. Balioztatze plana

Atal honetan, proiektu honetan sortuko den garapen-inguruneak egoki funtzionatzen duela ziurtatzeko egingo zaizkion probak biltzen dira.

Garapen-inguruneak *software* proiektuak garatzeko sistema bat izanik, *software* proiektuak eraikitzeko duen errendimendua aztertuko da. Horretarako **Ilaren teoriaz** baliatuz.

8.1. Proben espezifikazioak

Hardware ezberdinarekin garapen-inguruneak *software* proiektuak eraikitzeko behar duen denbora kalkulatuko da. Ondoren, **Ilararen teoria** erabili ahalko da errendimendua aztertzeko. **Ilaren teoria** erabili ahal izateko, *software* proiektuak eraikitzeko segundoko 0.04 agindu izango direla garapen-ingurunean suposatuko da. Izan ere, bezeroaren arabera, tasa hori ez da inoiz gaindituko.

Beraz, *hardware* ezberdinekin lortutako datuentzat ondorengoa kalkulatu beharko da:

- *Software*ak eraikitzeko batezbesteko denbora ez dela 100 segundo baino handiagoa.

- *Software* bat exekutatzeko agindua eman eta exekutatu aurretik zain egoteko probabilitatea ez dela % 20 baino handiagoa.

- *Software* bat exekutatzeko agindua eman denetik exekutatzen hasi arteko batezbesteko denbora ez dela 30 segundo baino handiagoa.

- *Software*a eraikitzeko agindua eman denetik eraikitzeaz amaitzen den arteko denbora 150 segundo baino handiagoa izateko probabilitatea ez dela % 20 baino handiagoa.