УДК 004.4

Шкоба А. Є.

Національний університет "Львівська політехніка",

кафедра електронних обчислювальних машин

**Застосування програмних засобів безперервної інтеграції у програмних проектах з використанням моделі розгалуження GitFlow**

*© Шкоба А.Є., 2019*

**Розглянуто проблему застосування програмних засобів безперервної інтеграції у програмних проектах з використанням моделі розгалуження GitFlow. Проаналізовано способи вирішення проблеми. Запропоновано структурну схему та алгоритм роботи системи безперервної інтеграції.**

**Ключові слова: безперервна інтеграція, модель розгалуження, GitFlow.**

**The problem of application of continuous integration (CI) software in projects using GitFlow branching model is considered. The ways of solving the problem are analyzed. The structural scheme and algorithm of continuous integration system workflow were created.**

**Keywords: continuous integration (CI), branching model, GitFlow.**

**Вступ.** Зі зростанням складності високотехнологічних платформ важливим фактором стає безперебійність роботи критично важливих ІТ-систем. На етапі розробки можливість безперервного тестування і доставки коду дозволяє збільшити швидкість і якість готових рішень. Безперервне (кілька разів на день) злиття робочих копій програмного коду в загальну основну гілку і тестування результатів оформилося в «концепцію безперервної інтеграції і доставки програмного забезпечення» (англ. Continuous Integration & Continuous Delivery або скорочено CI / CD). Вперше її запропонував Граді Буч в 1991 році [1].

Принцип дії CI / CD схожий на конвеєр: методика виконує інтеграційну функцію, включаючи різні типи автоматичних тестів на кожному етапі, з наступною доставкою та розгортанням завершеного коду в готовий продукт для кінцевого користувача [2]. CI / CD-платформи, на базі яких реалізується концепція, підтримують виконання регулярної автоматизованого складання проекту для оперативного виявлення дефектів і рішення інтеграційних проблем. При стандартному підході (каскадна розробка ПО або Waterfall-методика), де розробники незалежно трудяться над різними частинами системи, стадія інтеграції є заключною, і при виявленні помилок може непередбачувано затримати закінчення робіт. Перехід до безперервної інтеграції дозволяє знизити трудомісткість роботи і зробити її більш передбачуваною за рахунок раннього і безперервного виявлення і усунення помилок і протиріч [3].

**Стан проблеми.** Концепція розгалуження та об'єднання в таких інструментах управління вихідним кодом, як GIT, дуже важлива для створення великих і складних програмних проектів. Розгалуження потрібне для досягнення паралельної чи паралельної розробки коду [4]. Гілки використовуються для виділення окремих потоків розробки. Наприклад, виправлення може бути розроблене на гілці, відгалуженній з гілки випуску, де як новий функціонал може бути розроблений на гілці, яка відгалужилась з гілки розробки. Перевага використання гілок у цьому випадку полягає в тому, що розробка на відгалужених гілках продовжується незалежно від інших гілок [5]. Занадто багато гілок може зробити розвиток дуже складним або занадто мало може перешкоджати більш швидкому розвитку. Правильна стратегія розгалуження полегшує доставку коду, особливо для команд, що працюють по методології Scrum, яка передбачає середовище для багатьох розробників зі скороченням часу доставки, не переступаючи код один одного.

GitFlow визначає сувору модель розгалуження, розроблену навколо випуску програмного проекту. Це забезпечує надійну основу для управління більшими проектами [6]. Gitflow ідеально підходить для проектів, які мають запланований цикл випуску. Цей робочий процес не додає нових понять або команд, що перевищують необхідні для робочого процесу "Відділення функції". Натомість він присвоює дуже специфічні ролі різним галузям і визначає, як і коли вони мають взаємодіяти. Окрім гілок функцій, він використовує окремі гілки для підготовки, ведення та запису релізів.

Замість однієї основної гілки цей робочий процес використовує дві гілки для запису історії проекту. Гілка «master» зберігає офіційну історію випусків, а «develop» гілка служить гілкою інтеграції для нових функцій.

Кожна нова функція повинна містити свою власну гілку, яку можна перенести до центрального сховища для резервного копіювання / співпраці. Але, замість того, щоб відгалужуватися від «master», функції гілок розвиваються як їх батьківська гілка. Коли функція завершена, вона об'єднується назад у «develop». Функції ніколи не повинні взаємодіяти безпосередньо з «master».

Зауважте, що функції гілок у поєднанні з галуззю розробки є для всіх намірів і цілей робочим потоком функціональної гілки. Але робочий процес Gitflow на цьому не зупиняється.

На жаль, сучасні програмні реалізації для неперервної інтеграції не мають підтримки підходу GitFlow.

**Постановка задачі.** Розробити програмний засіб безперервної інтеграції у програмних проектах з використанням моделі розгалуження GitFlow. Розробити структурну схему та описати алгоритм роботи системи безперервної інтеграції.

**Розв’язання задачі.** Для розв’язку поставленої задачі було вирішено взяти за основу програмну реалізацію платформи безперервної інтеграції Jenkins та доповнити її для підтримки проектів з використання GitFlow. Вибір впав на Jenkins, адже - це безкоштовна платформа безперервної інтеграції з відкритим кодом, написана на Java. Jenkins підтримує засоби управління версіями, включаючи AccuRev, CVS, Subversion, Git, Mercurial, Perforce, TD / OMS, ClearCase та RTC, а також може виконувати проекти Apache Ant, Apache Maven та sbt, а також довільні скрипти оболонки та пакетні команди Windows [7]. Для кожної з гілок: master, developer, feature, hotfix автоматично створюється задача на Jenkins (рис.1). Під час злиття однієї вітки в іншу буде автоматично створюватись завдання для побудови та тестування результату злиття гілок.

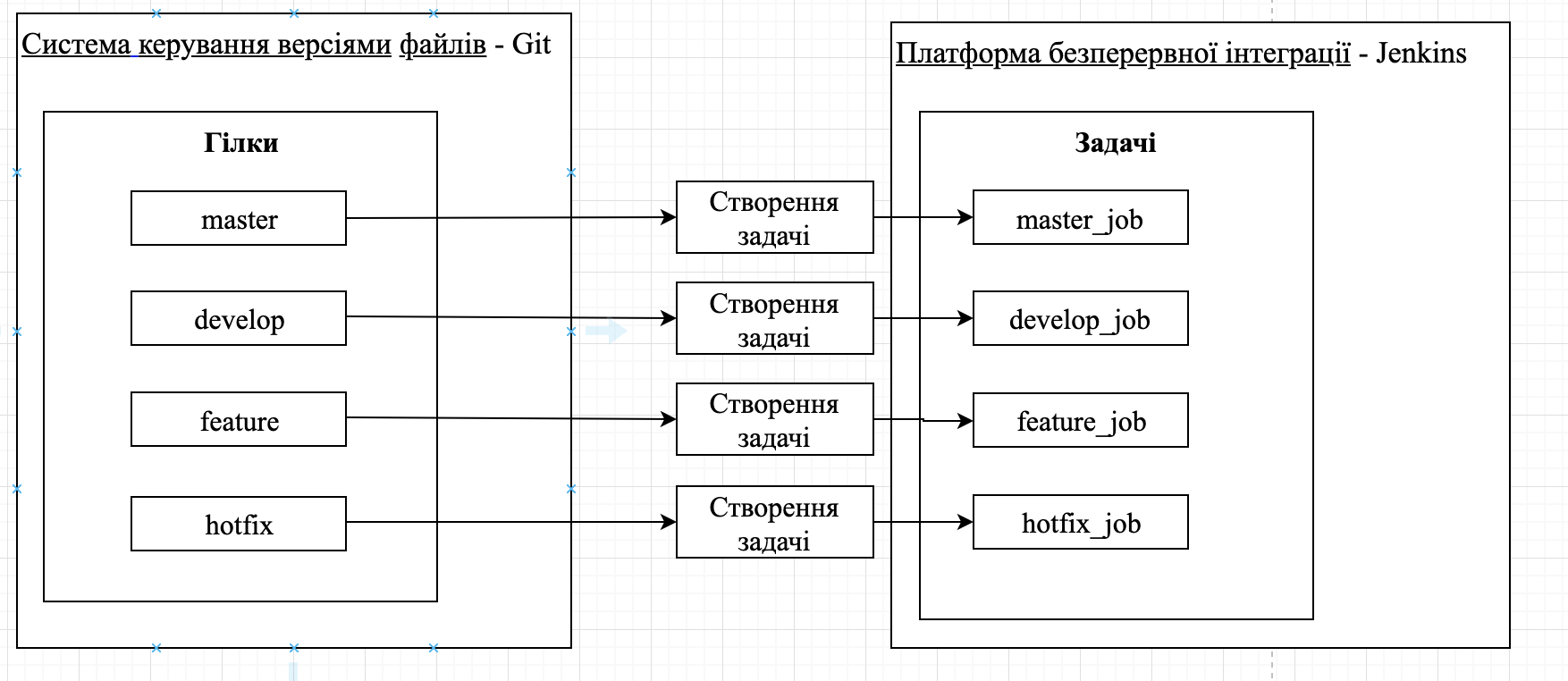


Рис. 1. Структурна схема роботи програмної системи безперервної інтеграції з підтримкою GitFlow.

Коли програмний проект набув достатньо функцій для випуску (або наближається заздалегідь визначена дата випуску), відщеплюється гілка випуску розробки [8, 9]. Створення цієї гілки розпочинає наступний цикл випуску, тому після цієї точки жодних нових функцій не можна додавати - у цій гілці мають бути лише виправлення помилок, створення документації та інші завдання, орієнтовані на випуск. Після того, як вона буде готова до доставки, відділення випуску об'єднується в головний і позначається номером версії.

Використання спеціальної гілки для підготовки випусків дає можливість одній команді відполірувати поточний випуск, а інша команда продовжує працювати над функціями для наступного випуску [10]. Здійснення випуску гілок - ще одна операція розгалуження. Як і особливі гілки, випускні гілки базуються на гілці, що розвивається. Нова гілка випуску може бути створена за допомогою наступних методів. (рис. 2)



Рис. 2. Модель розгалуження GitFlow

Для програмного засобу безперервної інтеграції у програмних проектах з використанням моделі розгалуження GitFlow розроблено алгоритм (рис. 3).

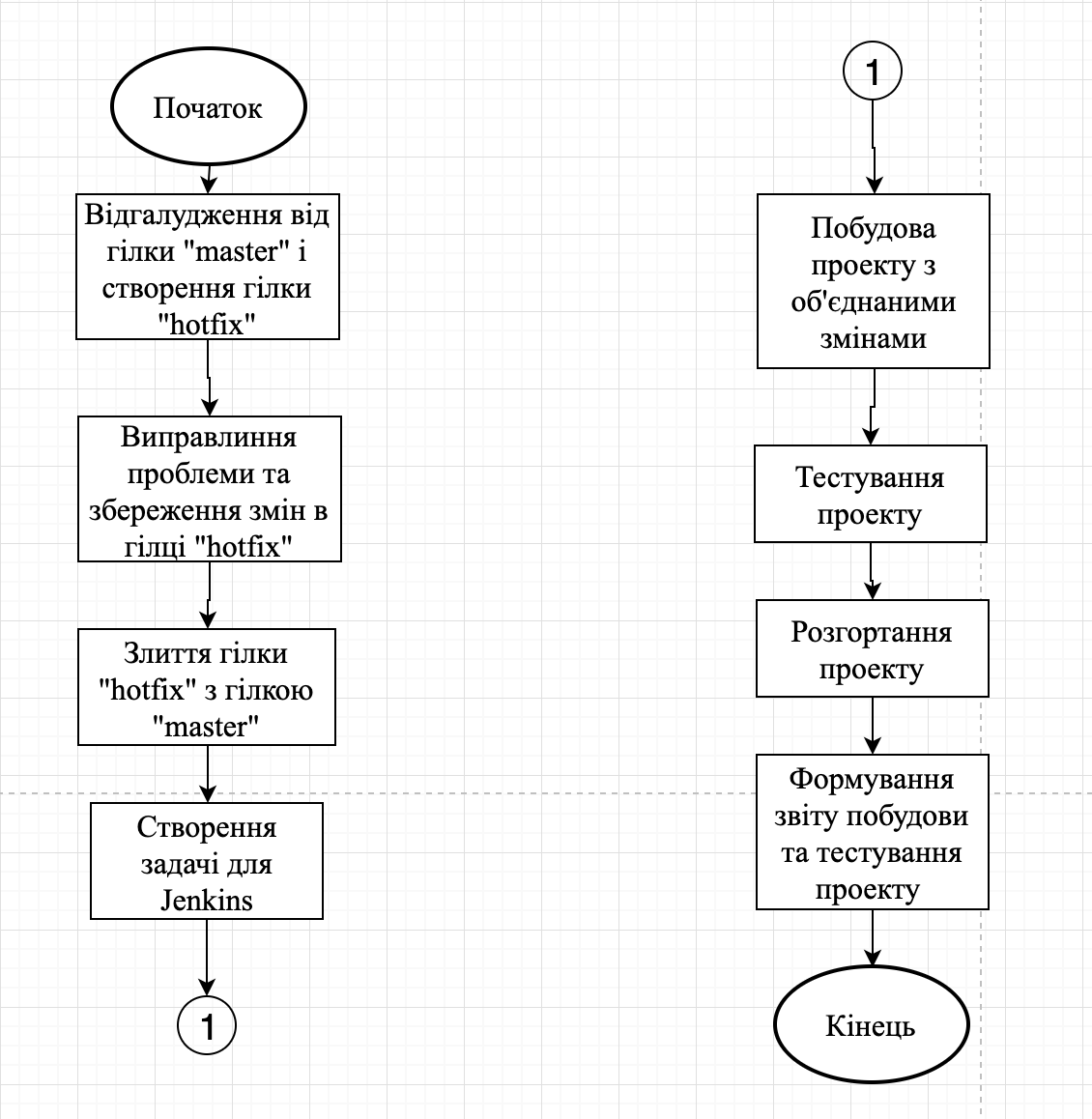


Рис. 3. Алгоритм роботи системи дистанційного керування автономним мобільним апаратом.

Даний алгоритм відслідковує коли відбувається злиття гілок. Після цього він створює задачу для побудови та тестування об’єднаних змін. Далі формується звіт який описує стан побудови та тестування.

Для реалізації алгоритму розроблено програмний модуль (набір етапів) (рис. 4).

|  |
| --- |
| stage('build') {  node {  checkout scm  def v = version()  currentBuild.displayName = "${env.BRANCH\_NAME}-${v}-${env.BUILD\_NUMBER}"  mvn "clean verify"}}  stage('build docker image') {node {mvn "clean package docker:build -DskipTests"}}  def branch\_type = get\_branch\_type "${env.BRANCH\_NAME}"  def branch\_deployment\_environment = get\_branch\_deployment\_environment branch\_type  if (branch\_deployment\_environment) {  stage('deploy') {  if (branch\_deployment\_environment == "prod") {  timeout(time: 1, unit: 'DAYS') {  input "Deploy to ${branch\_deployment\_environment} ?" }}  node {sh "echo Deploying to ${branch\_deployment\_environment}" }}  if (branch\_deployment\_environment != "prod") {  stage('integration tests') {  node {sh "echo Running integration tests in ${branch\_deployment\_environment}"  if (branch\_type == "dev") {  stage('start release') {  timeout(time: 1, unit: 'HOURS') {input "Do you want to start a release?" }  node {sshagent(['f1ad0f5d-df0d-441a-bea0-fd2c34801427']) {  mvn("jgitflow:release-start")  if (branch\_type == "release") {  stage('finish release') {  timeout(time: 1, unit: 'HOURS') {input "Is the release finished?" }  node {  sshagent(['f1ad0f5d-df0d-441a-bea0-fd2c34801427']) {  mvn("jgitflow:release-finish -Dmaven.javadoc.skip=true -DnoDeploy=true")  if (branch\_type == "hotfix") {  stage('finish hotfix') {  timeout(time: 1, unit: 'HOURS') { input "Is the hotfix finished?" }  node { sshagent(['f1ad0f5d-df0d-441a-bea0-fd2c34801427'])  {mvn("jgitflow:hotfix-finish -Dmaven.javadoc.skip=true -DnoDeploy=true") |

Рис. 4. Опис реалізації на мові JenkinsFile.

Основними методами розробленого інтерфейсу є:

* get\_branch\_type – повертає тип гілки, залежно від її префікса;
* stage('build') – етап, який визначає послідовність операцій для побудови проекта;
* stage("Run tests") – етап, для тестування проекту. Включає в себе як і функціональне тестування так і інтегрувальне.
* Stage("Deploy") – етап, для розгортання проекту у вірутальному контейнері Docker.

**Висновки**. В роботі розроблено систему програмний засіб безперервної інтеграції у програмних проектах з використанням моделі розгалуження GitFlow. Розроблено структурну схему та описано алгоритм роботи системи безперервної інтеграції.

**Література**

1. Booch, Grady (1991). Object Oriented Design: With Applications. Benjamin Cummings. p. 209. ISBN 9780805300918.

2. "Embracing Change with Extreme Programming". CiteSeerX 10.1.1.617.9195.

3. G. E. Kaiser, D. E. Perry and W. M. Schell, "Infuse: fusing integration test management with change management," [1989] Proceedings of the Thirteenth Annual International Computer Software & Applications Conference, Orlando, FL, USA, 1989, pp. 552-558. doi:10.1109/CMPSAC.1989.65147

4. "A Short History of Git". Pro Git (2nd ed.) 2015.

5. "BitKeeper and Linux: The end of the road?” Linus Torvalds, 2017.

6. "Git Source Code Mirror". 2017.

7. "What is Continuous Integration?". Amazon Web Services.

8. "Commit Graph". Github. 2016.

9. "Releases - git/git". 2019.

10. McAllister, Neil (2 May 2005). "Linus Torvalds' BitKeeper blunder". InfoWorld. IDG. Archived from the original on 26 August 2015. Retrieved 8 September 2015.