千葉工業大学　社会システム科学部

プロジェクトマネジメント学科

平成26年度　卒業論文

バージョン管理システムを活用する

ソフトウェア開発の開発フロー

Development flow of software development to take advantage of the version control system

プロジェクトマネジメントコース

矢吹研究室

1142032　小野寺航己／Kouki ONODERA

|  |  |
| --- | --- |
| 指導教員印 | 学科受付印 |
|  |  |

目次

[1．序論 1](#_Toc410302566)

[1.1．研究背景 1](#_Toc410302567)

[1.2．研究目的 5](#_Toc410302568)

[1.3．研究方法 5](#_Toc410302569)

[1.4．プロジェクトマネジメントとの関係性 5](#_Toc410302570)

[1.5．参考文献 6](#_Toc410302571)

[2．バージョン管理システムとGitHub 7](#_Toc410302572)

[2.1．バージョン管理システムとは 7](#_Toc410302573)

[2.2．個別型のバージョン管理システム 8](#_Toc410302574)

[2.3．集中型のバージョン管理システム 9](#_Toc410302575)

[2.4．分散型のバージョン管理システム 10](#_Toc410302576)

[2.5．Gitとは 11](#_Toc410302577)

[2.6．GitHubについて 12](#_Toc410302578)

[2.6．GitHub Enterpriseについて 13](#_Toc410302579)

[2.7．参考文献 14](#_Toc410302580)

[3．GithHubを用いた開発フロー 15](#_Toc410302581)

[3.1．GitHub用語 15](#_Toc410302582)

[3.1.1．リポジトリ 15](#_Toc410302583)

[3.1.2．ディレクトリ 15](#_Toc410302584)

[3.1.3．リモートリポジトリ 15](#_Toc410302585)

[3.1.4．commit（コミット） 15](#_Toc410302586)

[3.1.5．clone（クローン） 15](#_Toc410302587)

[3.1.6．Origin 15](#_Toc410302588)

[3.1.7．Push（プッシュ） 15](#_Toc410302589)

[3.1.8．ブランチ 15](#_Toc410302590)

[3.1.9．Pull（プル） 15](#_Toc410302591)

[3.1.10．Pull Request（プルリクエスト） 16](#_Toc410302592)

[3.1.11．Revert（リブート） 16](#_Toc410302593)

[3.1.12．タグ 16](#_Toc410302594)

[3.1.13．Merge（マージ） 16](#_Toc410302595)

[3.1.14．Fork（フォーク） 16](#_Toc410302596)

[3.1.15．Issue（イシュー） 16](#_Toc410302597)

[3.1.16．デプロイ 16](#_Toc410302598)

[3.1.17．リリース 16](#_Toc410302599)

[3.2．GitHubを用いた開発フロー 17](#_Toc410302600)

[3.2.1．GitHubフロー 17](#_Toc410302601)

[3.2.2．Gitフロー 19](#_Toc410302602)

[3.2.3．はてなブログフロー 21](#_Toc410302603)

[3.2.4．日本CAWフロー 23](#_Toc410302604)

[3.2.5．ラクスルフロー 25](#_Toc410302605)

[3.2.6．キャスレーフロー 27](#_Toc410302606)

[3.2.7．Amingフロー 29](#_Toc410302607)

[3.2.8．LINEフロー 31](#_Toc410302608)

[3.2.9．サイボウズフロー 33](#_Toc410302609)

[3.2.10．フィヨルドフロー 35](#_Toc410302610)

[3.2.11．イストフロー 37](#_Toc410302611)

[3.2.12．矢吹研フロー① 39](#_Toc410302612)

[3.2.13．矢吹研フロー② 41](#_Toc410302613)

[3.3．参考文献 43](#_Toc410302614)

[4．各開発フローで想定されるリスク 44](#_Toc410302615)

[4.1．開発フローを使うことで発生しうるリスク 44](#_Toc410302616)

[4.2．各開発フローで発生しうるリスク 46](#_Toc410302617)

[4.2.1．GitHubフロー 46](#_Toc410302618)

[4.2.2．Gitフロー 46](#_Toc410302619)

[4.2.3．はてなブログフロー 47](#_Toc410302620)

[4.2.4．日本CAWフロー 48](#_Toc410302621)

[4.2.5．ラクスルフロー 49](#_Toc410302622)

[4.2.6．キャスレーフロー 50](#_Toc410302623)

[4.2.7．Amingフロー 51](#_Toc410302624)

[4.2.8．LINEフロー 52](#_Toc410302625)

[4.2.9．サイボウズフロー 53](#_Toc410302626)

[4.2.10．フィヨルドフロー 53](#_Toc410302627)

[4.2.11．イストフロー 54](#_Toc410302628)

[4.2.12．矢吹研フロー① 55](#_Toc410302629)

[4.2.13．矢吹研フロー② 56](#_Toc410302630)

[4.3．参考文献 58](#_Toc410302631)

[5．結果・考察 59](#_Toc410302632)

[5.1．分析結果 59](#_Toc410302633)

[5.2．考察 60](#_Toc410302634)

[5.2.1．分析結果を踏まえて 60](#_Toc410302635)

[5.2.2．クラスターごとの特徴に関する考察 61](#_Toc410302636)

[5.2.3．まとめ 62](#_Toc410302637)

[5.2.4．今後の課題 62](#_Toc410302638)

**表目次**

[表 1 GitHubで開発が進められているソフトウェア 12](#_Toc410297838)

[表 2　分析結果番号対応表 60](#_Toc410297839)

[表 3　プロジェクトと開発フロー 62](#_Toc410297840)

**図目次**

[図 1集中型の仕組み 2](#_Toc410297702)

[図 2分散型の仕組み 2](#_Toc410297703)

[図 3集中型と分散型のトレンド 3](#_Toc410297704)

[図 4有名ソフトウェア開発会社が使っている，バージョン管理システムの利用率 4](#_Toc410297705)

[図 5バージョン管理の仕組み 7](#_Toc410297706)

[図 6個別バージョン管理システム（個人開発） 8](#_Toc410297707)

[図 7個別バージョン管理システム（少数でのチーム開発） 8](#_Toc410297708)

[図 8集中型バージョン管理システムのイメージ 9](#_Toc410297709)

[図 9分散型バージョン管理システムのイメージ 10](#_Toc410297710)

[図 10 GitHubフロー図 17](#_Toc410297711)

[図 11 GitHubフローユースケース図 18](#_Toc410297712)

[図 12 GitHubフローシーケンス図 18](#_Toc410297713)

[図 13 Gitフロー図 19](#_Toc410297714)

[図 14 Gitフローユースケース図 20](#_Toc410297715)

[図 15 Gitフローシーケンス図3.2.3．はてなブログフロー 20](#_Toc410297716)

[図 16 はてなブログフロー図 21](#_Toc410297717)

[図 17 はてなブログフローユースケース図 22](#_Toc410297718)

[図 18 はてなブログフローシーケンス図 22](#_Toc410297719)

[図 19 日本CAWフロー図 23](#_Toc410297720)

[図 20　日本CAWフローユースケース図 24](#_Toc410297721)

[図 21　日本CAWフローシーケンス図 24](#_Toc410297722)

[図 22　ラクスルフロー図 25](#_Toc410297723)

[図 23　ラクスルフローユースケース図 26](#_Toc410297724)

[図 24　ラクスルフローシーケンス図 26](#_Toc410297725)

[図 25　キャスレーフロー図 27](#_Toc410297726)

[図 26　キャスレーフローユースケース図 27](#_Toc410297727)

[図 27　キャスレーフローシーケンス図 28](#_Toc410297728)

[図 28　Amingフロー図 29](#_Toc410297729)

[図 29　Amingフローユースケース図 30](#_Toc410297730)

[図 30　Amingフローシーケンス図 30](#_Toc410297731)

[図 31　LINEフロー図 31](#_Toc410297732)

[図 32　LINEフローユースケース図 32](#_Toc410297733)

[図 33　LINEフローシーケンス図 32](#_Toc410297734)

[図 34　サイボウズフロー図 33](#_Toc410297735)

[図 35　サイボウズフローユースケース図 34](#_Toc410297736)

[図 36　サイボウズフローシーケンス図 34](#_Toc410297737)

[図 37　フィヨルドフロー図 35](#_Toc410297738)

[図 38　フィヨルドフローユースケース図 36](#_Toc410297739)

[図 40　イストフロー図 37](#_Toc410297741)

[図 41　イストフローユースケース図 38](#_Toc410297742)

[図 42　イストフローシーケンス図 38](#_Toc410297743)

[図 43　矢吹研フロー①図 39](#_Toc410297744)

[図 44　矢吹研フロー①ユースケース図 40](#_Toc410297745)

[図 45　矢吹研フロー①シーケンス図 40](#_Toc410297746)

[図 46　矢吹研フロー②図 41](#_Toc410297747)

[図 47　矢吹研フロー②ユースケース図 42](#_Toc410297748)

[図 48　矢吹研フロー②シーケンス図 42](#_Toc410297749)

[図 49　クラスター分析結果 59](#_Toc410297750)

# 1．序論

この項では，当研究の背景，研究目的，マネジメントとの関係について記述する．

# 1.1．研究背景

　ソフトウェア開発プロジェクトにおいて，「誰が」「いつ」「何を変更したか」というような情報を記録することで，複数の人間が過去のファイルや，変更点の確認，ファイルの状態を復元することなどを可能とする，バージョン管理システムが利用され始めた．バージョン管理システムは，バージョン履歴を保存することができるデータベース（リポジトリ）にファイルを登録，登録されたファイルをローカルに取り出す，取り出して編集を行ったファイルをリポジトリに登録しなおす，ファイルが登録し直されると変更内容やバージョンが記録される，という仕組みで管理している．

バージョン管理システムには，主に『集中型』『分散型』の2種類がある．『集中型』は，リポジトリから直接ローカルにファイルを取り出し，編集を加えたファイルを直接リポジトリに反映させる，という仕組みである．『分散型』は，メインで扱うリポジトリとは別に，ローカルにもリポジトリを作成し2重に管理をする仕組みである．以下の図1.1は集中型，図1.2は分散型のバージョン管理システムの仕組みを簡略化して示したものである．

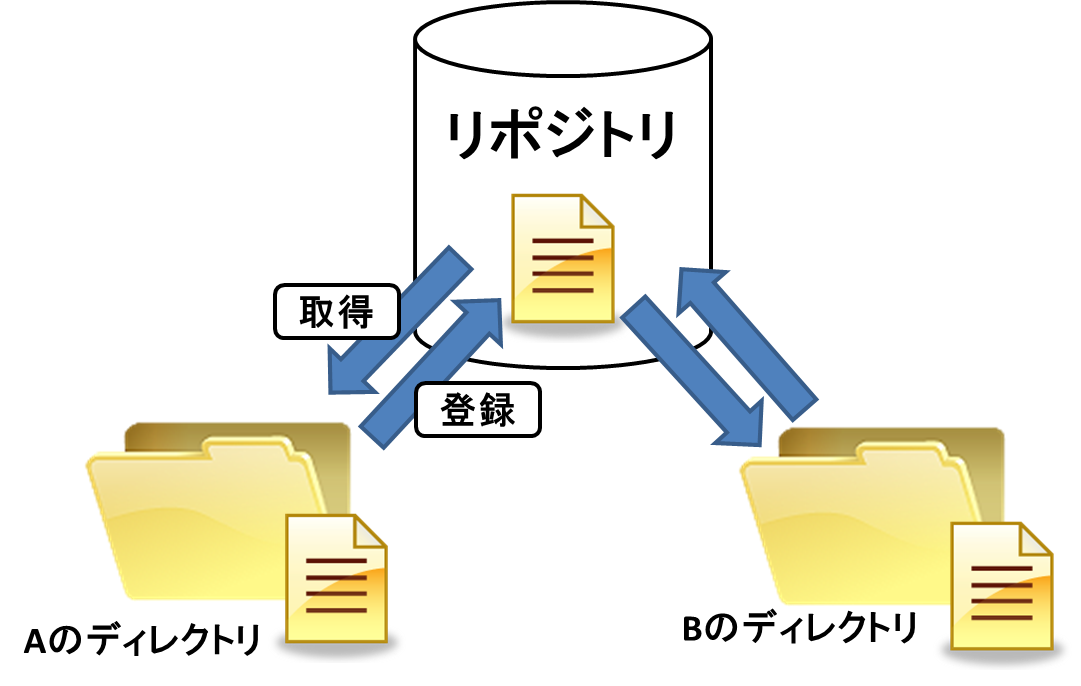


図 1集中型の仕組み

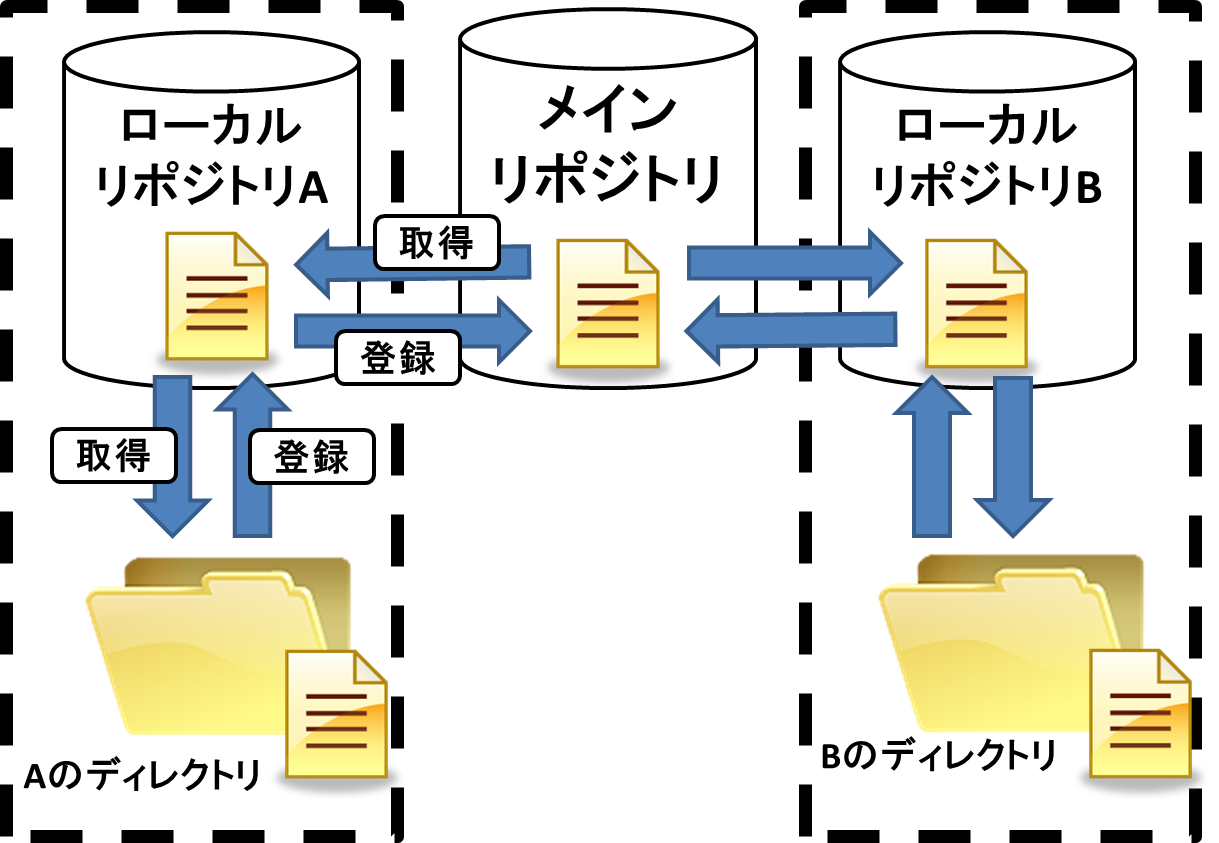


図 2分散型の仕組み

　集中型バージョン管理システムの「CVS」「Subversion」，分散型管理システムの「Bazaar」「Git」「Mercurial」などの種類がある中．その中でも2010年には， Subversionが、ソフトウェアの共同開発には欠かせないツールであるバージョン管理システムの60％以上を占めていた，と言われていた．しかし，同じく2010年ごろから，「Git」というバージョン管理システムの，利用率が増加している[1]．

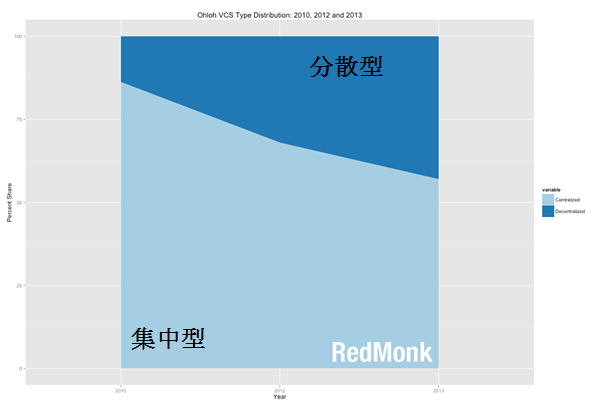


図 3集中型と分散型のトレンド

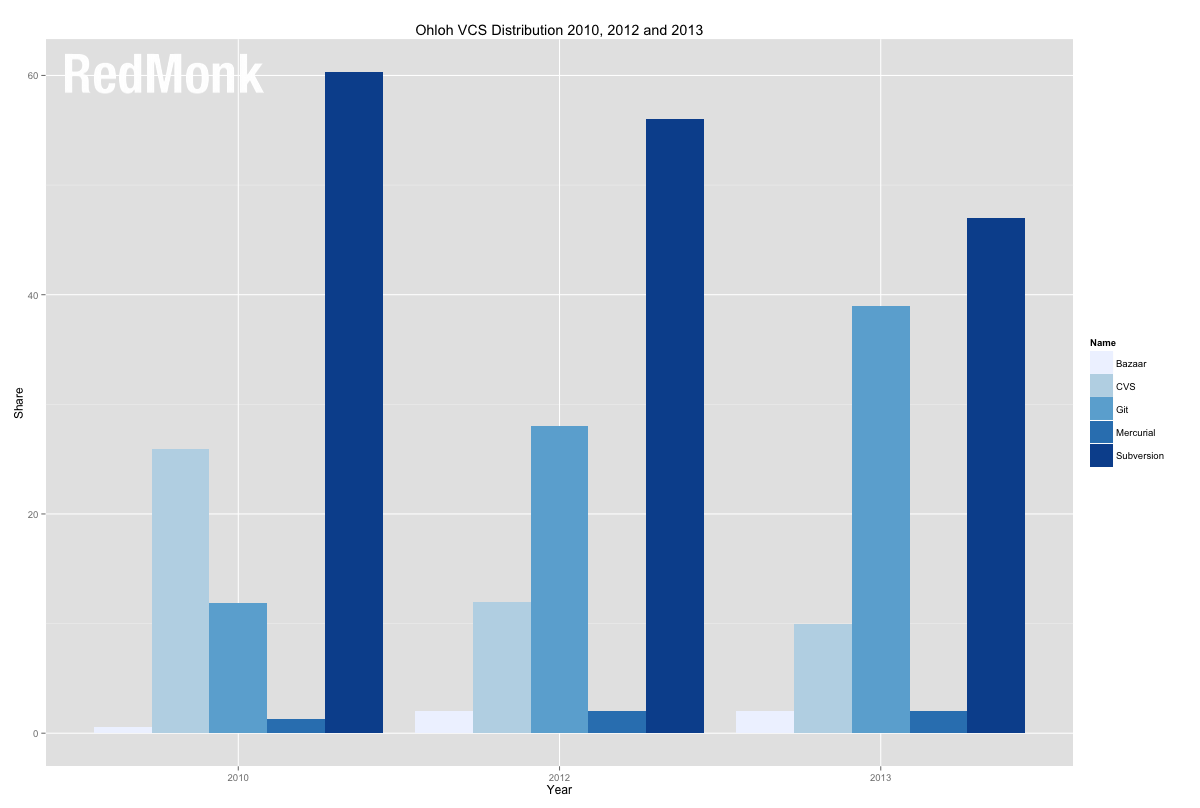


図 4有名ソフトウェア開発会社が使っている，バージョン管理システムの利用率

　「Git」が台頭した理由としては，変更を記録するリポジトリが単一な集中型の「Subversion」よりも，リポジトリを複数用いることで，メインのリポジトリに影響を与えることなく，同じファイルに対して複数の変更を，履歴を保持しつつ同時におこなうことができるよう，履歴の流れを分岐して記録できるブランチという機能があり，開発メンバが並行して開発作業を行える，分散型の「Git」の方が，開発スピードが速くなるからである．

　このGitを用いたウェブシステムとしてGitHubがある．GitHubはGitの機能を提供するウェブサービスであり，世界中の人々が自分の作品を保存、公開することもでき，ソフトウェア開発プロジェクトのための共有サービスでもある．

　Gitの台頭や，GitHubによって，Gitが使いやすくなり，ソフトウェア開発のツールとしてGitHubが用いられることが多くなっている．そうしてGitHubを使ったソフトウェア開発の流れ（開発フロー）も，利用者によって多岐にわたるようになった．

　そこで，多岐にわたったGitHubを用いた開発フローを，プロジェクトの規模や状況に応じて，円滑にプロジェクトを進められるものを適切に選択できるような基準の作成を当研究で行う．

# 1.2．研究目的

　ソフトウェア開発におけるプロジェクトは，プロジェクトの参加人数，メンバの力量などの理由から，多種多様になっている．Githubを利用したプロジェクトにおいても同様である．その様々なプロジェクトに対して円滑に遂行できるように，適切な開発フローを選択する基準を明確にする．

1.3．研究方法

1. GitHubを用いる開発フローを網羅的に調査．
2. 調査した開発フローを，フローごとに発生しうるリスクの観点から分類・整理する．
3. プロジェクトの性質に応じて適切な開発フローを選択できるようなガイドを作成する．

# 1.4．プロジェクトマネジメントとの関係性

本研究は，GitHubを用いたソフトウェア開発がどのような開発フロー（開発チーム内でのルールや手順）で行われているかを調べる研究である．調べた開発フローをプロジェクトの規模やスキルによって適切に選択できるということは，GitHubを用いたソフトウェア開発のマネジメントにも大きく関わる．本研究の成果はそれに貢献することが期待される．

# 1.5．参考文献

[1] Matt Asay, Git 対 Subversion：長引く争い, livedoor’NEWS,(参照2014-01-24),

<http://news.livedoor.com/article/detail/8463552/>

# 2．バージョン管理システムとGitHub

　この章ではバージョン管理システム，GitHubについて記述する．

# 2.1．バージョン管理システムとは

　バージョン管理システムとは変更履歴を管理するシステムのことである，

バージョン管理システムは，ソフトウェア開発の時に用いられることが多い．ソフトウェア開発でバージョン管理システムが多く使われる理由としては，開発中のソフトウェアにバグなどが発生した時，過去のバージョンを参照してバグに対応をしやすくなる．過去のバージョンが保存されていることで，開発中のソフトウェアに試験的に機能を追加してみるといった大きな変更に対するリスクが低くなる．といった理由から，ソフトウェア開発が素早く行えるところが大きいだろう．

　バージョン管理システムの基本的な仕組みとしては，①リポジトリと呼ばれる，変更履歴を管理するデータベースを作成する．②リポジトリにファイルを登録し，登録したファイルをローカルな環境に取り出す．③ローカル環境で取り出したファイルに変更を加えリポジトリに登録し直す．④変更した内容を履歴として保存され，変更前のファイルも必要に応じて取り出し可能になる．という仕組みである．

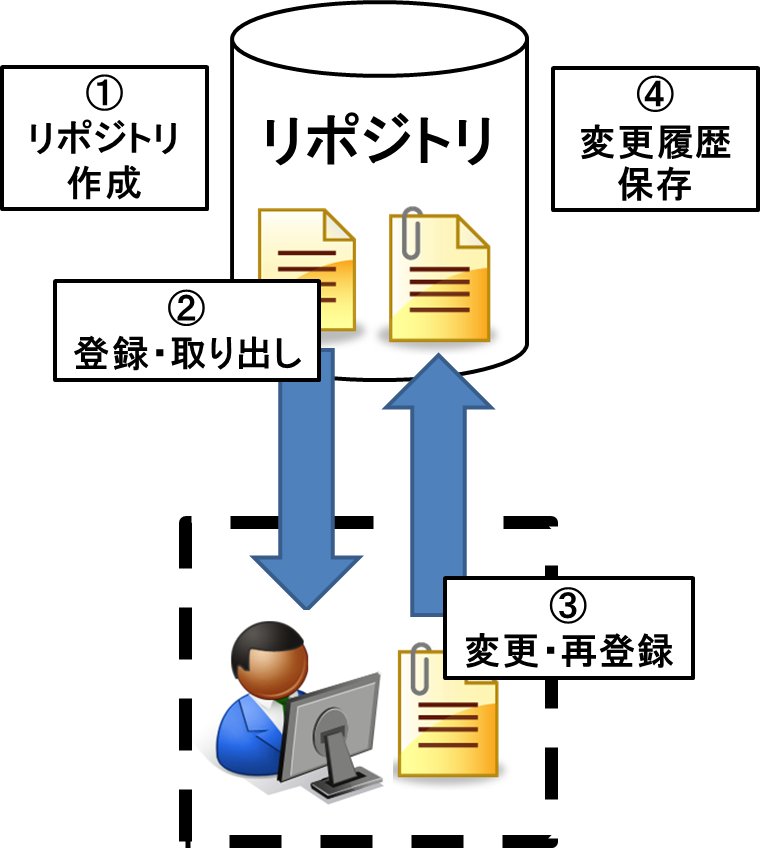


図 5バージョン管理の仕組み

　バージョン管理システムには個別型と集中型，分散型の大きく分けて3種類がある，2010年以前は集中型のバージョン管理システムである「CVS」「Subversion」などが主流だったが，それ以降は分散型のバージョン管理システムである「GitHub」「Mercurial」が主流になっている．

# 2.2．個別型のバージョン管理システム

　初期のオープンソースソフトウェアとしてのバージョン管理システムといえばRCSが挙げられる．RCSはファイル単位での管理しかできないため，バージョン管理については限定的なことしかできなかったのである．個人での利用であれば問題ないかもしれないが，複数人が利用するプロジェクトでは何らかの運用上の工夫が必要になることが多くなるため，集中型バージョン管理システムであるCVSなどへと発展していくことになったのである[1]．

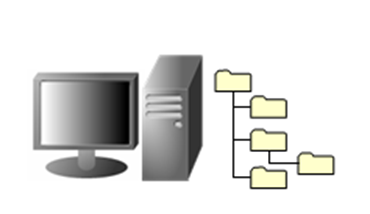


図 6個別バージョン管理システム（個人開発）

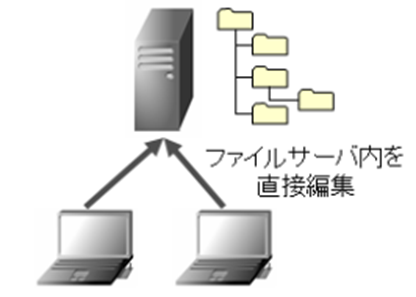


図 7個別バージョン管理システム（少数でのチーム開発）

# 2.3．集中型のバージョン管理システム

　集中型のバージョン管理システムの特徴は，開発するソフトウェアに対してリポジトリが1つしか作成されないことである．利点としては作成されるリポジトリが単一のため複雑な操作を必要としない点である．欠点は作成されたリポジトリにアクセスしファイルを取り出すためにはリポジトリにネットワークによる接続が必要なことである．

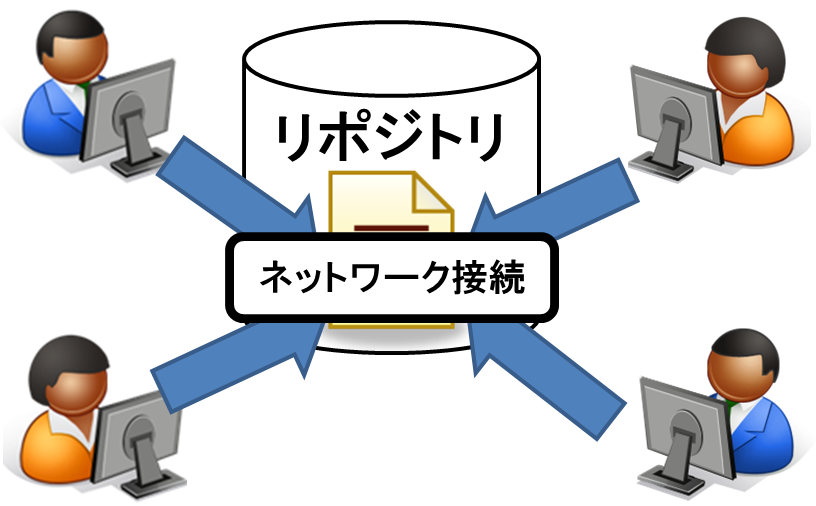


図 8集中型バージョン管理システムのイメージ

# 2.4．分散型のバージョン管理システム

　分散型のバージョン管理システムの特徴は，開発するソフトウェアに対するリポジトリの他にも開発に参加するメンバごとに複数のリポジトリをローカルな環境に作成できることである．基本的な仕組みとしては，①メインで使うリポジトリごと，ローカルな環境に複製する．②複製したリポジトリからファイルを取り出し変更を加える．③複製したリポジトリに変更を加えたファイルを登録する．同時にファイルの変更履歴が保存される．④さらに複製したリポジトリから，変更を加えたファイルをメインで使うリポジトリに登録する．同時にファイルの変更履歴が保存される．という仕組みである．

　利点としては，ローカルな環境にリポジトリを複製するため，ネット環境がなくてもリポジトリに登録されたファイルに対して編集が行える．複数人で同時に複数のリポジトリを使用できるため，並行して作業が行えるということである．欠点としては，リポジトリが複数作成され手間が増えるので使いこなすのが難しいということにある．



図 9分散型バージョン管理システムのイメージ

# 2.5．Gitとは

Gitとはバージョン管理システムに分類されるソフトウェアであり，GitHubに用いられているバージョン管理システムである．

　Gitの原形は2005年頃に，Linuxの創始者であるLinus Torvalds氏によって，バージョン管理システムを新しいものに置き換えることを目的として開発したシステムである．

Gitの原形時システムは一部のハッカーからぐらいにしか利用価値のものがないとされてきた，しかしLinusと濱野純の2人を中心に開発が進められ，開発から3か月経ったごろにはGitを熟知していない一般ユーザにも十分利用できるようなシステムになり，今では多くの人々によってGitに改良を加えられているシステムである．

　Gitの特徴としては，Linuxでの開発で培われた高性能な分散型バージョン管理システムであることである．代表的な機能としてリポジトリ同士での変更内容の共有，共有時に競合が起こった場合には警告をしてくれる，というのが挙げられるだろう．

# 2.6．GitHubについて

　GitHubとは，Gitで作成されたリポジトリを置く場所をネット上に無償で提供することでソフトウェア開発プロジェクトを手助けするために，2008年4月に設立した「github social coding」によって提供されている共有ウェブサービスである．

　GitHubの主な機能として，GitHubに置くリポジトリを無料で何個でも作成でき，有料で限られた人にしか公開できないような非公開リポジトリを作成できる「Gitリポジトリ」，

いつでもだれでも文章の書き換え保存できる「Wiki」，他人にリポジトリ内の変更内容を取り込むように要求する「Pull Request」などがある．

　他にも有用な機能があるためか，世界中の開発者の中でもGitHubの使用者も多く，以下の表に記したソフトウェアもGitHubで開発が進められている．

また，本論文で出てくるGitとGitHubは別物である．GitとはGitリポジトリというデータの貯蔵庫にソースコードなどを入れて利用する．このGitリポジトリを置く場所をインターネット上に提供しているのがGitHubである．つまり，GitHubで公開されているソフトウェアのコードはすべてGitでされている．Gitについて理解しておくことはGitHubを使う上で重要である．

表 1 GitHubで開発が進められているソフトウェア

|  |  |
| --- | --- |
| Ruby on Rail | Rubyにて使われる代表的なWEBフレームワーク |
| node | JavaScriptにて人気のあるプラットフォーム |
| jQuery | JavaScriptライブラリ |
| Symfony2 | PHPにてつくられたフルスタックWEBフレームワーク |
| Bootstrap | ツイッターのようなインターフェースをつくれるコンポーネント集 |

# 2.6．GitHub Enterpriseについて

　GitHubをクローズドな環境で使うために、GitHub社が提供しているソフトウェアである．機能やインタフェースもGitHubとほぼ同じなため，プライベートながらも，ユーザから見ると違和感なく使うことができるのが利点である．

基本的な設定はWebベースの管理画面で行い．細かい設定をする際には，SSHでログインできるadminユーザがあるため，提供されているコマンドユーティリティを

使って運用管理を行うことができる．

　エンジニアの中から「Gitを標準リポジトリにしてほしい」という声が高まり，2013年にGitを正式導入することが決定された． GitリポジトリにはGitHub Enterprise以外に様々な選択肢があるが． 選定基準として，セキュリティ，提供されている機能，運用のしやすさ，コストなどがあるが，現場からはGitHubのようなソーシャルコーディングの環境が求める声も多かったため，ソーシャルコーディングのしやすさも重要なポイントになった． そして総合的な判断の結果，GitHub Enterpriseを採用することとなった．

　このGitHub Enterpriseを浸透させるために，Git初心者を対象にした勉強会が行われるなどの，エンジニアたちによる普及活動も，新規・既存プロジェクトで利用するバージョン管理システムがGitHubに移行する要因にもなった[7]．

2.7．参考文献

[1] 工藤亮, プロジェクトホスティングサービスのためのEVM自動描画システムの開発, 千葉工業大学, 2014, 卒業論文

[2] 濱野純, 入門Git, 株式会社 秀和システム, 第1版第3刷, 2010-01-15

[3] Shinpei Maruyama, Git とはなんぞや, github.com, 2013-05-08

<https://github.com/Shinpeim/introduction-to-git/blob/master/01_what_is_git.md>

[4] 大塚弘記, GitHub実践入門 Pull Requestによる開発の変革, 技術評論社, 2014.

[5] 松下雅和,船ヶ山慶,平木聡,土橋林太郎,三上丈晴, 開発効率をUPする Git逆引き入門, ,株式会社 シーアンドアール研究所, 2014-04-15

[6] 平屋真吾, ガチで5分で分かる分散型バージョン管理システムGit (3/6), @IT, 2013-07-05, <http://www.atmarkit.co.jp/ait/articles/1307/05/news028_3.html>

[7] 松下雅和, サイバーエージェントのGitHub活用 ～ 導入から運用体制、開発フロー、勉強会による現場への普及活動まで, CodeZine, 2014-09-09

<http://codezine.jp/article/detail/8092>

# 3．GithHubを用いた開発フロー

この項では調査したGitHubを使った開発フロー，その説明に必要となるGitHubの専門用語の説明について記す．

# 3.1．GitHub用語

　以下に調査した開発フローを説明する際に用いる用語を記述する[1]．

# 3.1.1．リポジトリ

ファイルやディレクトリの状態を記録する場所である．保存された状態は，内容の変更履歴として格納され，変更履歴を管理したいディレクトリをリポジトリの管理下に置くことで，そのディレクトリ内のファイルやディレクトリの変更履歴を記録することができるのである．

# 3.1.2．ディレクトリ

　フォルダのことである．ファイルを分類・整理するための保管場所である．

# 3.1.3．リモートリポジトリ

　手元に置いてあるローカルなリポジトリ以外の，ネット上に置かれたリポジトリのことである．

# 3.1.4．commit（コミット）

ファイルやディレクトリの追加・変更を，リポジトリに記録するために行う操作のことである．

# 3.1.5．clone（クローン）

ネット上にあるリモートリポジトリをローカルにコピーすることである．

# 3.1.6．Origin

clone元のリモートリポジトリのことである．

# 3.1.7．Push（プッシュ）

リモートリポジトリに自分の変更履歴がアップロードされ，リモートリポジトリ内の変更履歴がローカルリポジトリの変更履歴と同じ状態になるようにすることである．

# 3.1.8．ブランチ

履歴の流れを分岐して記録していくためのものである．分岐したブランチは他のブランチの影響を受けないため，同じリポジトリ中で複数の変更を同時に進めていくことができるのである．

# 3.1.9．Pull（プル）

リモートリポジトリから最新の変更履歴をダウンロードしてきて，自分のローカルリポジトリにその内容を取り込むことである．

# 3.1.10．Pull Request（プルリクエスト）

相手に対して自分のブランチをpullしてもらうように要求する機能のことである．

# 3.1.11．Revert（リブート）

ステージングエリアに追加した変更を戻すことである．

# 3.1.12．タグ

コミットを参照しやすくするために，わかりやすい名前を付けるもののことである．

# 3.1.13．Merge（マージ）

当該ブランチに対して別のブランチの差分を取り込むことである．

# 3.1.14．Fork（フォーク）

GitHubのサービスで，相手のリポジトリを自分のリポジトリとしてコピー・保持できる機能のことである．

# 3.1.15．Issue（イシュー）

一つのタスクを一つのIssueに割り当てて，トラッキングや管理を行えるようにするための機能である．

# 3.1.16．デプロイ

ソフトウェアの分野で、開発したソフトウェアを利用できるように実際の運用環境に展開すること．

# 3.1.17．リリース

　プロセスを次の段階に進めることを認めること．

# 3.2．GitHubを用いた開発フロー

　以下に調査した開発フローを示す．

# 3.2.1．GitHubフロー

　GitHubフローはGitHub社が実践しているシンプルなワークフローである．

常にメインにするブランチはデプロイできる状況にするという前提でこの開発フローは行われる，これは数時間ごとにデプロイを行うことによって大きなバグが複数入り込むということを防ぐためである．

開発フローの流れとしては，①メインとなるブランチから新しい作業用のブランチを作成する，この時ブランチの名前は何の作業をしているのかが他の開発者がわかるように記述するのが望ましい．②作業用に作成したブランチにcommitする，作成された作業用ブランチは，何の作業をするかが明確になっているはずなので1つのcommitの量は小さくすることを意識したほうが良い．③Pull Requestを使って他の開発者に作業用ブランチをメインブランチにMergeを求める，この時Mergeを求めるだけでなく他の開発者に指摘をもらいたいときにもPull Requestは有用である．④他の開発者からの許可が下りれば作業用ブランチをメインブランチにMergeしデプロイする．という流れで行われる．

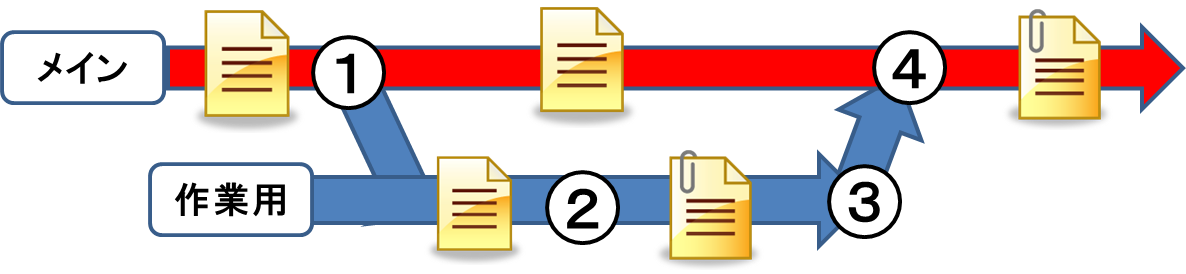


図 10 GitHubフロー図

　この開発フローはデプロイを中心にしたとてもシンプルなワークフローであり，シンプルが故に小さなチームでも大きなチームでも機能するフローである，実際にGitHub社でこの開発フローを利用したとき，20人程度のプロジェクトまでなら大きな問題が発生したことがないといわれている．

　逆にデプロイを中心にしているが故の弊害として，すぐに作業中ブランチをメインブランチにMergeすることを狙ったフローのため，各々の開発者がすんなりとレビューを通るだけのものを作れるレベルにまで能力アップする必要があるという欠点もある．



図 11 GitHubフローユースケース図



図 12 GitHubフローシーケンス図

# 3.2.2．Gitフロー

　オランダのプログラマVincent Driessen氏のA successful Git branching modelというブランチ戦略をベースにGitHubを組み合わせたワークフローがGitフローである．

　このフローはブランチを複数作成し，それぞれのブランチがソフトウェアの状態を表す，リリースを中心にしたソフトウェア開発に向いている．

　手順としては①メインブランチから開発版（develop）ブランチを作成する．②開発版ブランチから作業用（feature）ブランチを作成し，機能を実装・修正する．③作業用ブランチでの修正が終了すると開発版ブランチにMergeされる．④上記の②と③を繰り返し，リリースできるレベルにまで機能を作り上げる．⑤リリース用（release）ブランチを作りリリースのための作業をする．⑥リリース作業が終わるとメインブランチに取り込まれバージョンタグを打つ．⑦リリースしたものにバグがあれば，修正（hotfixes）ブランチを作り，修正を行う．という手順で行われる．

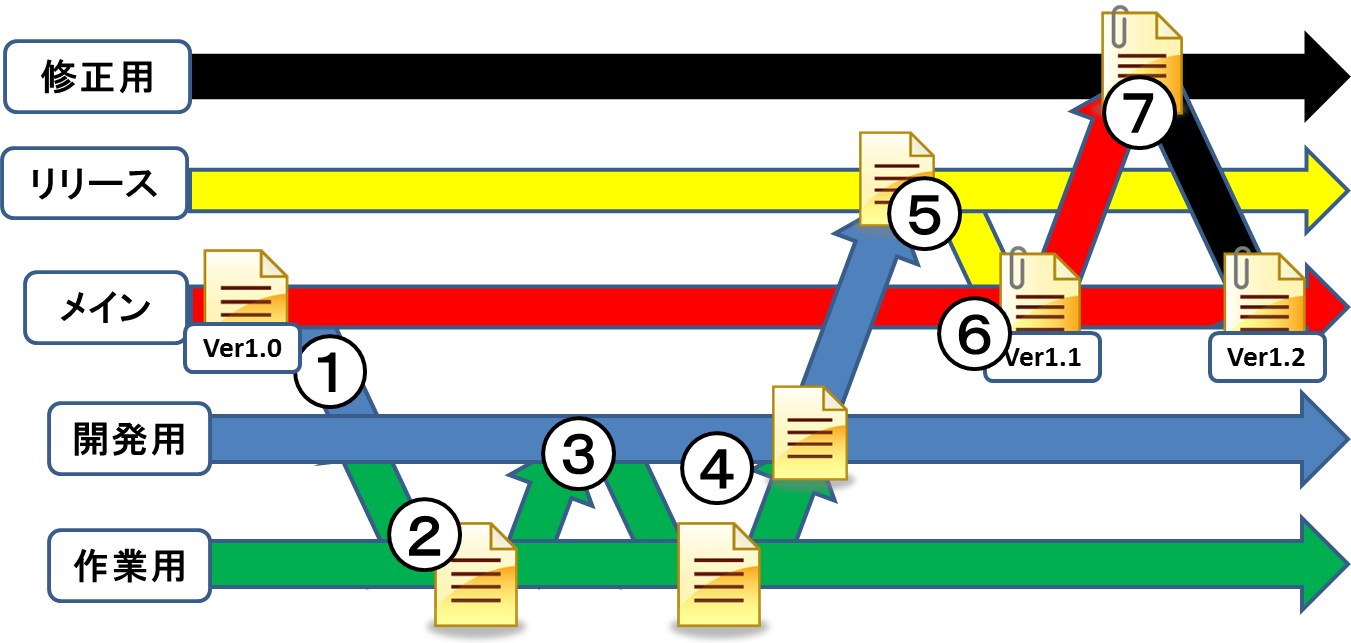


図 13 Gitフロー図

　上記の図のように複雑な手順を踏むため，覚えるブランチの状態が多い，プログラマは現在行っている作業がどのブランチに影響を与えるかを理解しなければならない，Merge先が複数になる場合もあるのでそこで人的ミスが発生する，などの問題が発生しやすいのでワークフロー全体をメンバ全員がしっかり学習してから導入する必要がある．

　このフローの利点としては，フロー自体がアジャイル開発モデルのように，ソフトウェア開発者であれば理解しやすい流れが多くなっているということである．



図 14 Gitフローユースケース図



図 15 Gitフローシーケンス図3.2.3．はてなブログフロー

# 3.2.3．はてなブログフロー

　この開発フローは，株式会社はてなに属するはてなブログチームが行っている開発フローである．

　はてなブログチームは，エンジニア5人，デザイナー2人程度で構成され，はてなブログを作っている．

開発の流れとしては①issueを登録，指定．②issueに対応する作業用ブランチを作成．③開発，レビュー，開発用ブランチにMerge．④Mergeがたまったらリリース．というものである．

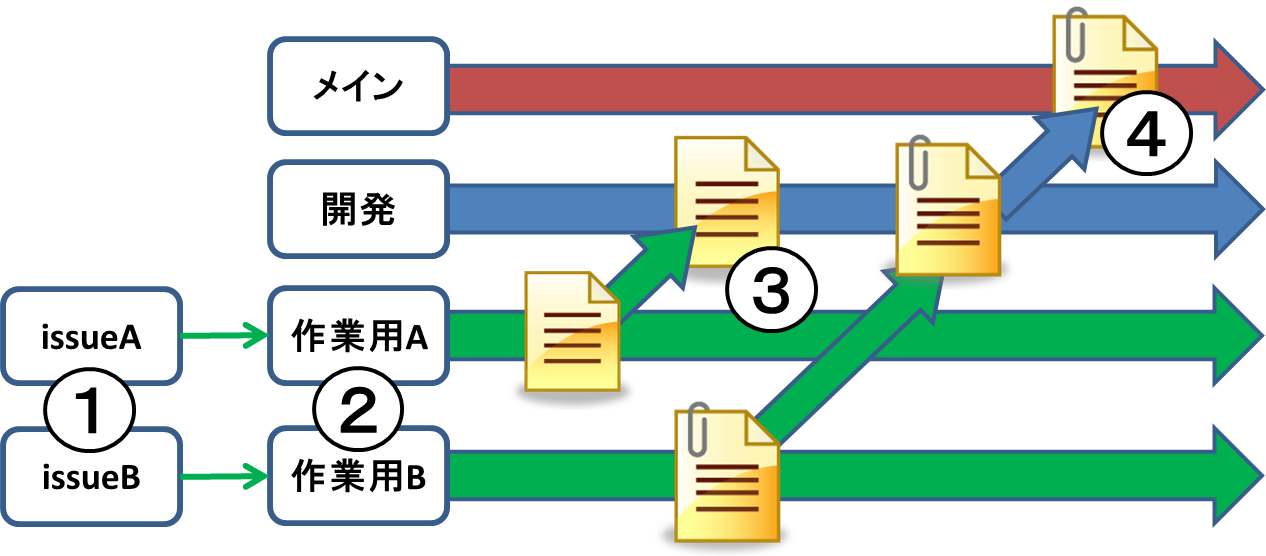


図 16 はてなブログフロー図

　issueの重要度や，だれが担当するかを定義するために，カンバン方式を採用している．これにより開発者はGitHubのissueを見ることで，マネージャはカンバンを見ることで，タスクの重要度を把握できて，開発者に効率の良い管理ができる仕組みになっている．

　はてなブログフローは，Pull Requestでレビューを行う際，Pull Requestに重要度を示すラベルを付ける，毎日14時からレビュータイムを設けるなどの工夫もされている[3][4]．



図 17 はてなブログフローユースケース図



図 18 はてなブログフローシーケンス図

# 3.2.4．日本CAWフロー

日本CAWでの開発フローはGitHub フローをベースにしてpull requestを活用したスタイルを採用している．

　基本的にGitHubフローと同じであるが，Pull Requestの時に[WIP]を付けることで，Mergeをするために送るPull Requestではなく，議論をするためだけのPull Requestとして利用できる[5]．

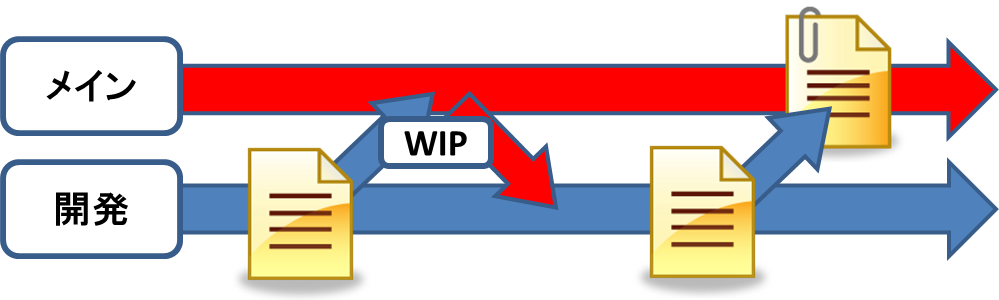


図 19 日本CAWフロー図



図 20日本CAWフローユースケース図



図 21日本CAWフローシーケンス図

# 3.2.5．ラクスルフロー

　ラクスルフローとは，印刷のポータルサイトを運営している日本の企業，ラクスル株式会社で利用している開発フローである．

　GitHub，オープンソースのプロジェクト管理ツールであるRedmine，コミュニケーションツールであるskypeを使って，ラクスルは開発を行っている．

開発フローの流れとしては①メインブランチから機能ごとにブランチを作成する．②機能ごとに，開発が終了し次第Pull Requestを送り，レビュー．③問題がなければメインブランチにMergeし，Mergeした機能ブランチを破棄という流れで行われている[6]．

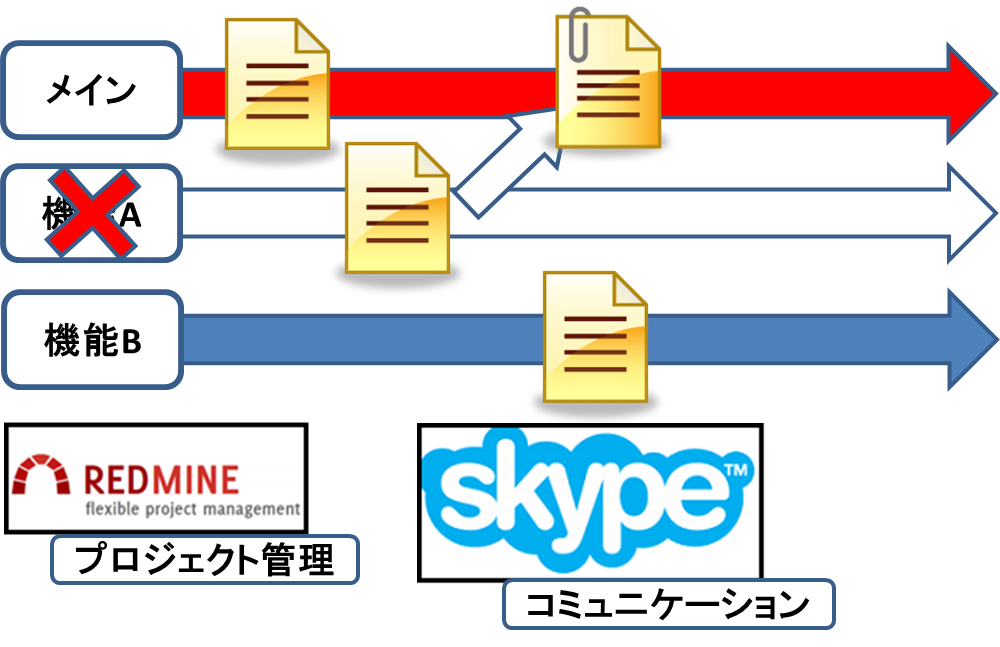


図 22ラクスルフロー図



図 23ラクスルフローユースケース図



図 24ラクスルフローシーケンス図

# 3.2.6．キャスレーフロー

　キャスレーフローはキャスレーコンサルティング株式会社内で利用している開発フローである．

　キャスレーフローは，基本的にGitフローを踏襲するが，製品版ブランチ上で開発を行わず，製品版ブランチから作成した作業用ブランチで開発する，開発が終わった作業用ブランチが製品版ブランチにMergeしたいときには，責任者にレビューのためPull Requestを送る，というGitフローの簡略版である．

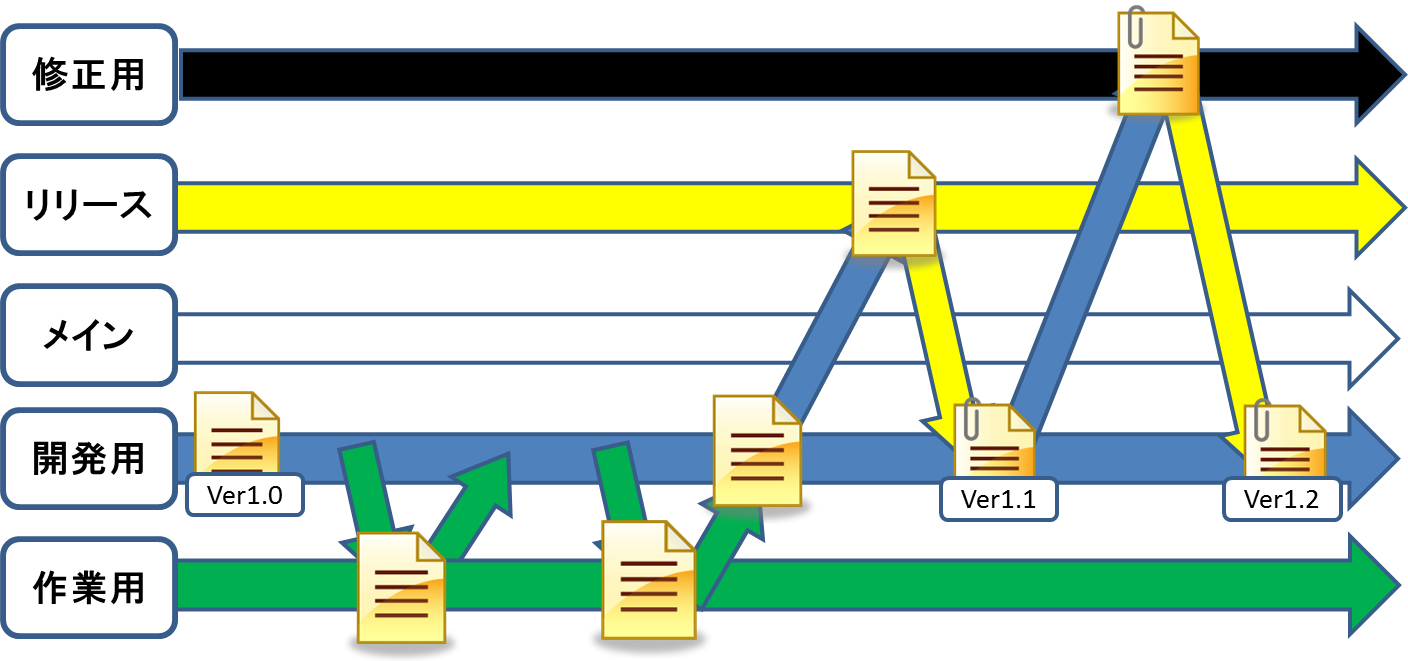


図 25キャスレーフロー図

　Gitフローを踏襲した理由としては，社内開発に携わっているエンジニアは15人，一部のメンバを除き，大半のメンバがgitを利用した開発経験がない，中にはスキルレベル的に他のエンジニアのフォローが必要なメンバもいる，１日に何回もリリースしたりはしない，という理由が挙げられる．



図 26キャスレーフローユースケース図



図 27キャスレーフローシーケンス図

# 3.2.7．Amingフロー

　Amingフローは，オンラインゲーム会社Amingでよく用いられる開発フローである．

手順としては，①メインリポジトリを各メンバがForkする．②メンバのリポジトリに作業用ブランチを作成．③メンバのブランチからメインリポジトリにPull Requestを送りレビュー．④メインブランチにMergeする．といったものである[7]．

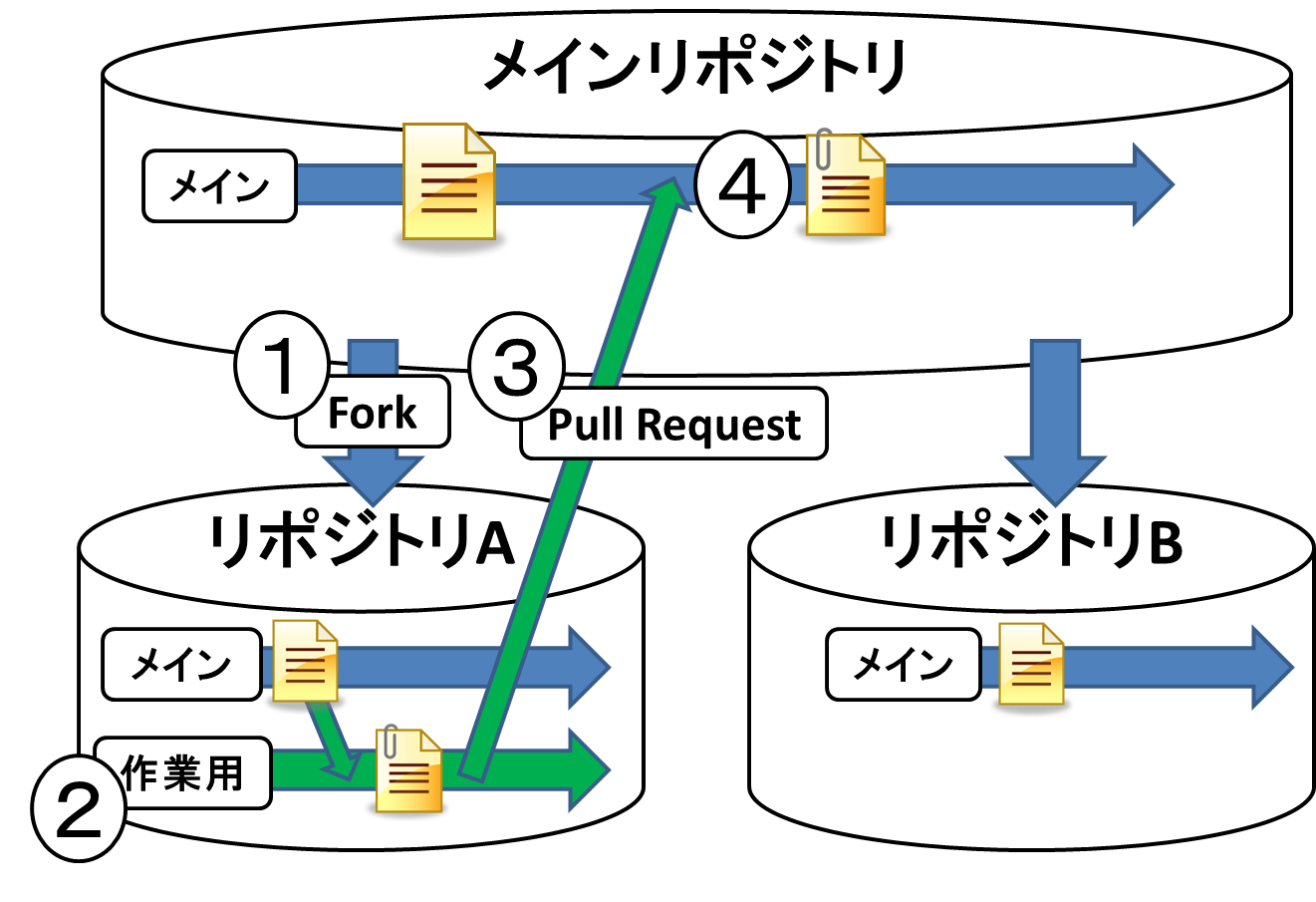


図 28Amingフロー図



図 29Amingフローユースケース図



図 30Amingフローシーケンス図

# 3.2.8．LINEフロー

　LINEフローはGitHub Enterpriseを用いて行われる，LINEのiOSアプリ開発の際に使用されるフローである．

　LINEフローはGitHubフローによく似た開発フローであり，常にデプロイ可能なメインブランチを使うのではなく，常にデプロイ可能なメインブランチを開発バージョンごとに作成する．過去のバージョンのブランチがリリースされれば，そこから次のバージョンのブランチを作成する，を繰り返す開発フローである

　それぞれがデプロイ可能なブランチであり，現在開発中の最も下位のバージョンの修正がリリースされたら、そのブランチを上位のブランチにマージして下位のブランチを消していくという流れで行われる[8]．

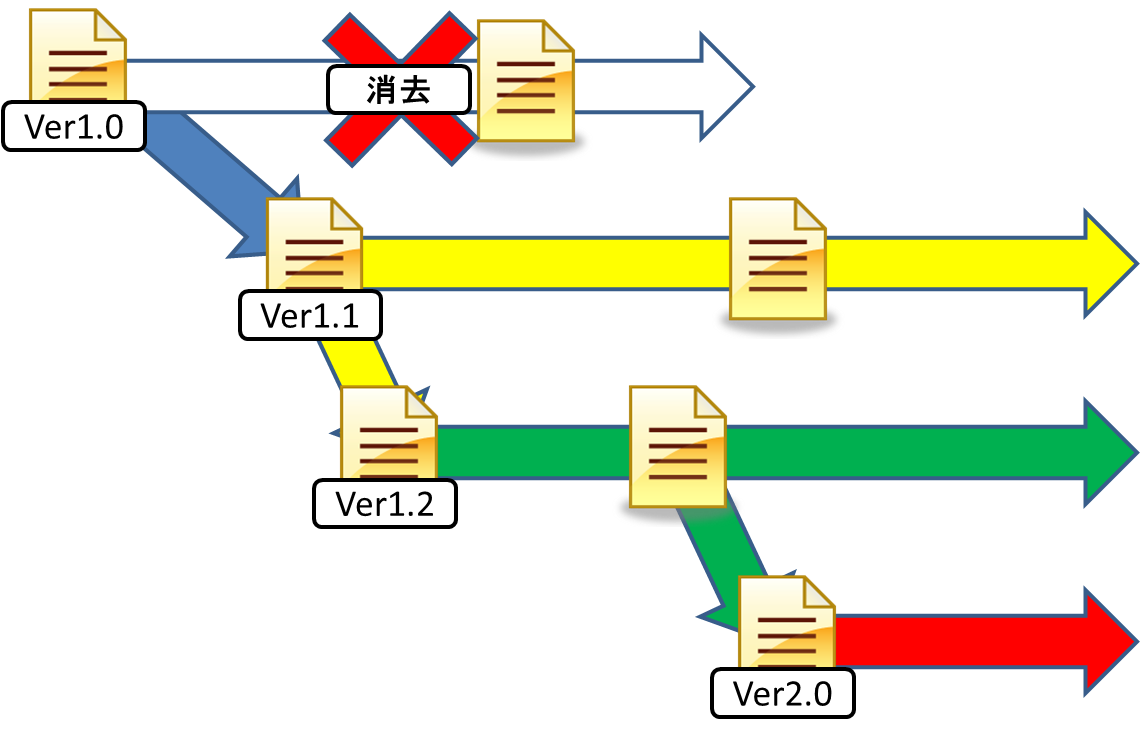


図 31LINEフロー図



図 32LINEフローユースケース図



図 33LINEフローシーケンス図

# 3.2.9．サイボウズフロー

　ソフトウェア開発会社であるサイボウズ株式会社が利用している開発フローである．

　複数のリポジトリを扱う開発フローであり，開発メンバ各自のリポジトリのトピックブランチで設計・実装を行い，開発リポジトリにMergeし検出された不具合を修正する，不具合を修正したら安定リポジトリ（安定した環境）にMergeし，最終試験を行うという流れで行われる[9]．

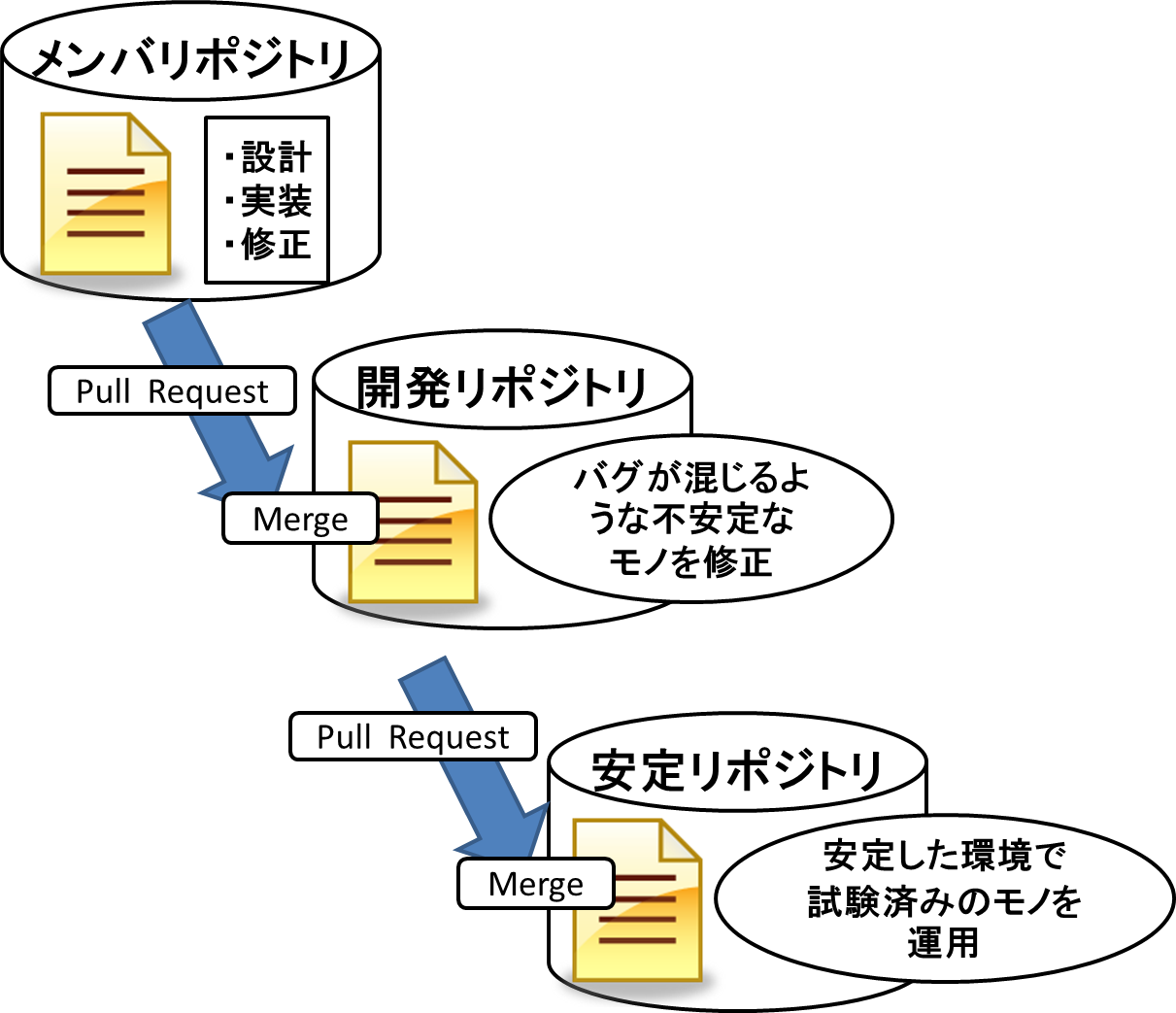


図 34サイボウズフロー図



図 35サイボウズフローユースケース図



図 36サイボウズフローシーケンス図

# 3.2.10．フィヨルドフロー

　フィヨルドフローは，「怖い話がスマホで読める・投稿できる怖話」のWebサイト，アプリの開発とWebアプリ開発・デザインを請け負っている，合同会社フィヨルドが利用している開発フローである．

　流れとしては，①PivotalTracker（アジャイルプロジェクト管理ツール）に登録してあるストーリー（まとまった単位のタスク）をStartする．②ローカルのMacでtopicブランチを作り，プログラマはrails, rspec（テストフレームワーク）を使ってテストと実装を書く．③コードはGitHubにpushする．④GitHubにpushされるとLingr（チャットサービス）のchatに通知が行き，CIサーバー（テストを自動的に行うサーバー）がテストを実行し，statingサーバーにデプロイする．デプロイが失敗した場合はメンバ全員にメールが飛ぶ．また，CIサーバーでは午前0時に定期デプロイが行われるようになっている．⑤stagingへのデプロイが成功したら、ローカルからproductionにcapistrano（デプロイ自動化ソフトウェア）を使ってデプロイする．というものである[10]．

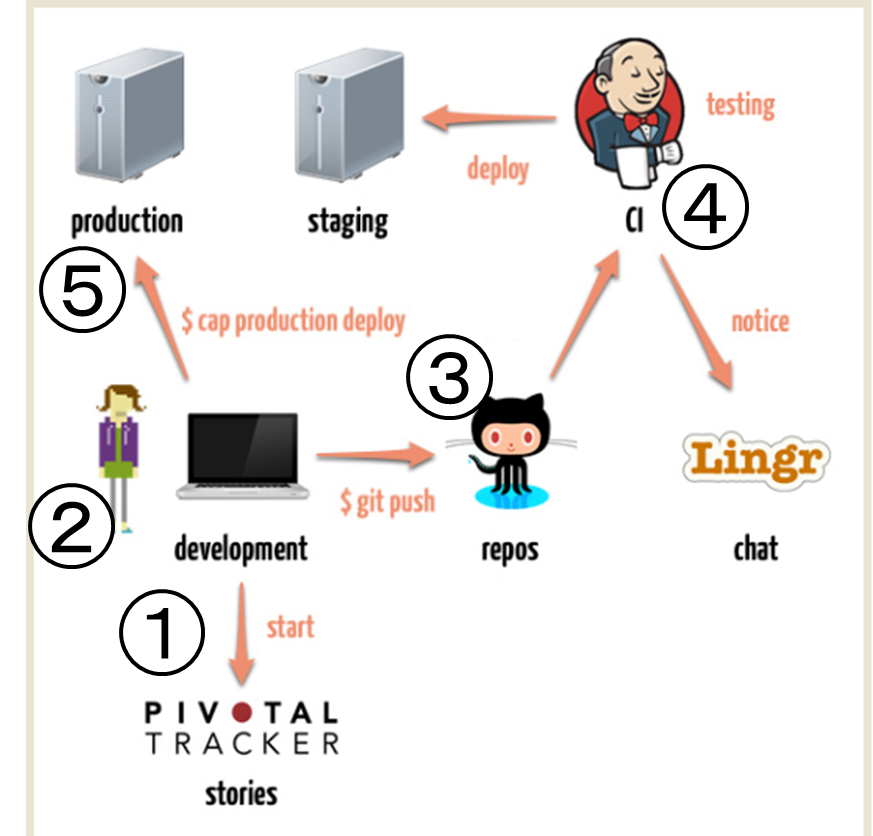


図 37フィヨルドフロー図



図 38フィヨルドフローユースケース図



図 39フィヨルドフローシーケンス図

# 3.2.11．イストフロー

　イストフローは，当時の株式会社イストが用いていたフローである．

　GitoriousというGitHubに似たサービスを用いた開発フローであり，メインリポジトリを中心に変更を加える．メインリポジトリに変更を加える際のPull Requestは管理者に送られ，問題なければメインリポジトリに取り込まれる．Jenkins（テストを自動的に行うソフトウェア）がメインリポジトリの変更を5分おきに監視し，変更があった場合テストを行う，テスト終了後に，ナイトリー（開発初期段階）環境にデプロイされる．管理者はステージング（本番環境）でcapistranoを用いてデプロイされる．という流れで行われる．

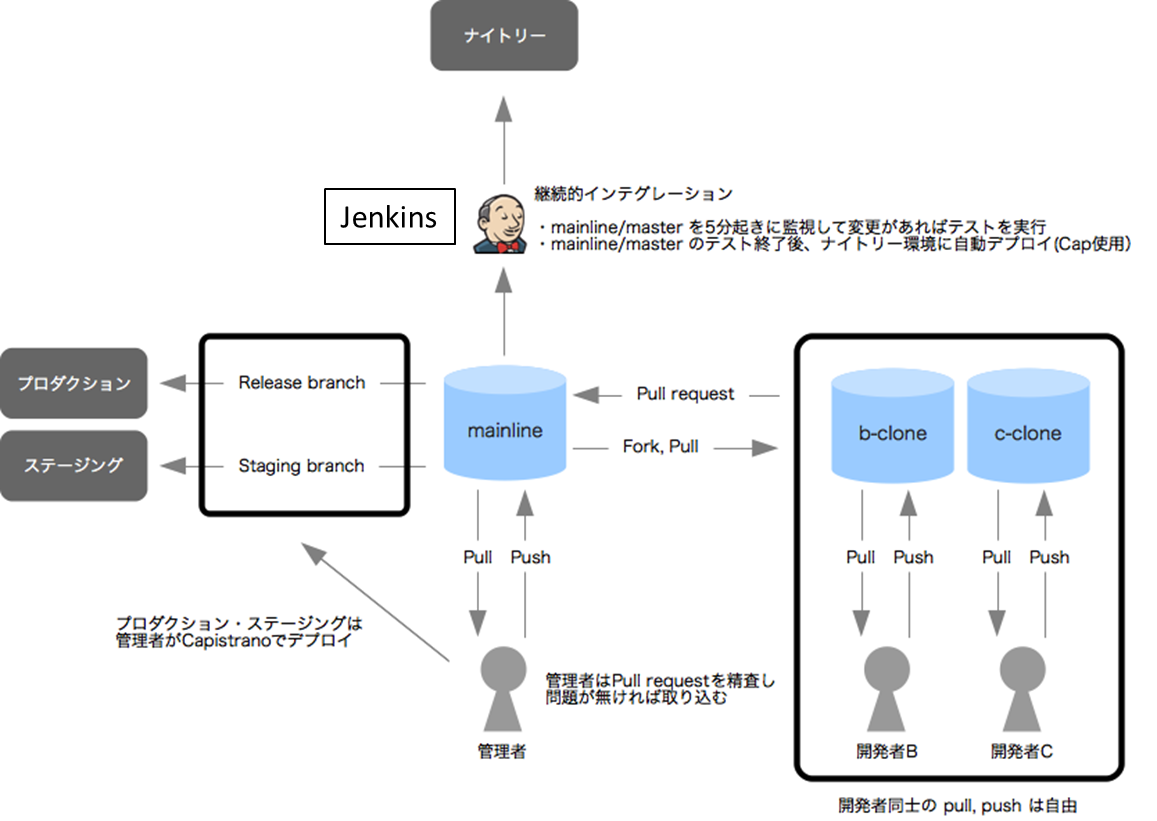


図 40イストフロー図



図 41イストフローユースケース図



図 42イストフローシーケンス図

# 3.2.12．矢吹研フロー①

　このフローは，当時の矢吹研究室内で課題研究のための成果物のやり取りに使われたフローである．

　仕組みとしては，初めにメイン（先生の）リポジトリを学生全員がcloneし，学生は直接成果物となるファイルをメインリポジトリにcommitするというものである．

　学生は当時GitHubに慣れておらず，トラブルも多かったが，GitHubの名も聞いたこともないようなメンバでも運用可能なフローであるともいえる．

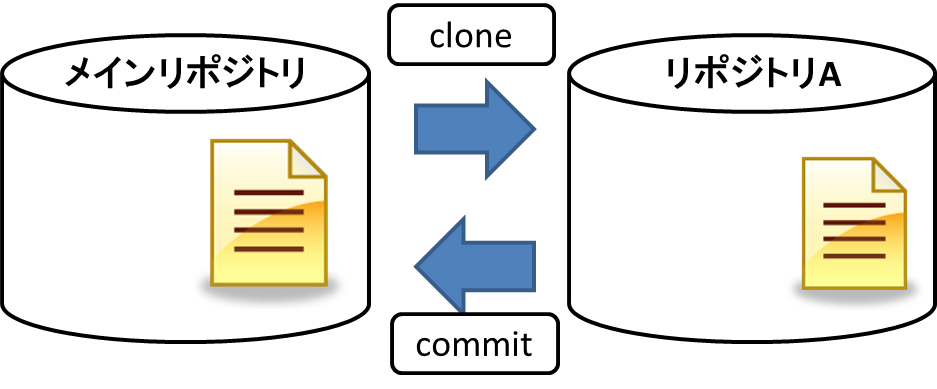


図 43矢吹研フロー①図



図 44矢吹研フロー①ユースケース図



図 45矢吹研フロー①シーケンス図

# 3.2.13．矢吹研フロー②

　このフローは，現在の矢吹研究室内で卒業研究のための成果物のやり取りに使われるフローである．

　仕組みとしては，メインリポジトリから，各々の学生がForkし，各々のリポジトリで作業を行い，成果物をメインリポジトリのMergeしてもらうためにPull Requestを送り，レビューが通ればメインリポジトリにMergeされる，という仕組みである．

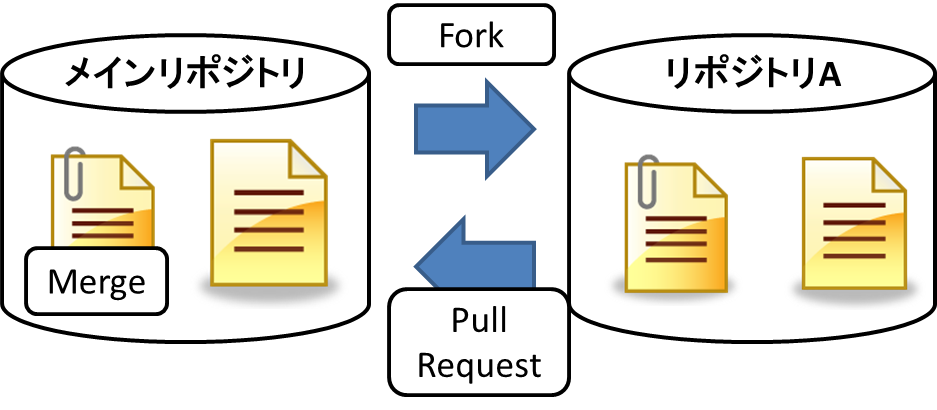


図 46矢吹研フロー②図



図 47矢吹研フロー②ユースケース図



図 48矢吹研フロー②シーケンス図

# 3.3．参考文献

[1] 清水竜吾, テストを基準にしたソフトウェア開発プロセスの調査, 千葉工業大学, 2014, 卒業論文

[2] 大塚弘記, GitHub実践入門 Pull Requestによる開発の変革, 技術評論社, 2014.

[3] Junichi Niino, はてなブログチームの開発フローとGitHub（前編）。GitHub Kaigi 2014, Publickey, 2014-06-24, <http://www.publickey1.jp/blog/14/githubgithub_kaigi_2014.html>

[4] Junichi Niino, はてなブログチームの開発フローとGitHub（後編）。GitHub Kaigi 2014, Publickey, 2014-06-24,

<http://www.publickey1.jp/blog/14/githubgithub_kaigi_2014_1.html>

[5] Junichi Niino, GitHub FlowでPull Requestベースな開発フローの進め方, Qiita, 2014-08-03,

<http://qiita.com/harada4atsushi/items/527d5f98320d993b3072>

[6] yohshima, ラクスルの開発フローについて, ラクスル tech blog, 2014-08-04,

<http://tech.raksul.com/2014/08/04/devlopment_flow/>

[7] 藤村大介, , AimingのGitHubを使った開発フロー, Speaker Deck, 2012-06-23, <https://speakerdeck.com/fujimura/aiminggithub>

[8] hayaishi, LINE iOSアプリ開発についてのご紹介, LINE Engineers' Blog, 2014-04-21, <http://developers.linecorp.com/blog/?p=2921>

[9] 山本泰宇, Git & GitHub & kintone でウルトラハッピー！, slideshare, 2012-11-01, <http://www.slideshare.net/HirotakaYamamoto/git-github-kintone>

[10] komagata, 小さい会社のツールスタック・開発フロー, Design Computer FJORD,LLC 素敵なウェブアプリを作ります, 2012-06-12, <http://fjord.jp/love/1084.html>

[11] 赤塚大, Gitを使った開発・運用フローの紹介, dakatsuka's blog 開発な日々, 2011-05-24, <http://blog.dakatsuka.jp/2011/05/24/git-flow.html>

# 4．各開発フローで想定されるリスク

　この章ではGitHubを用いた開発フローを分類する要素となるリスクについて記述する．

# 4.1．開発フローを使うことで発生しうるリスク

　以下に想定されるリスクを記す．

・常にデプロイする

　フローには作成したソフトウェアを常に実際の運営方法で運営できるようにするものがあるが，逆に言えば常に実際の運営方法で運営できるレベルのソフトウェアを比較的短期間で作成しなければならないということである．すなわちそのレベルにまでソフトウェアを作りこむ必要があるため，プロジェクトに参加しているメンバにはそれだけのスキルが必要となる．スキルを持ち合わせていなかった場合，デプロイをし続けるのが売りな開発フローなのにデプロイがあまりされないという事態になってしまう．

・デプロイをあまりしない

　逆にデプロイをあまりしない場合は，短期間に一定のレベルまで作りこむ必要性は無いが，デプロイをする期間が長ければ長いほど，大きなバグが入りやすく，気づきにくい．そのため対処に手間取ることになる．

・フローが複雑

　フローそのものが複雑であったりすると，まずそのフローを理解するのに時間がかかる，そのフローの運用に手間取るといった問題が発生しやすくなる．

・複数のブランチを扱う

　フローによってはブランチを複数使うことになる，大抵のフローではメインで扱うブランチとは別に作業をするためのブランチを作成し，作業が終了し次第メインのブランチに結合するという流れをとるが，作業が完了していないのにメインのブランチに結合してしまう，メインとは別のブランチに結合してしまうなどの人的ミスがブランチの本数が多いほど発生しやすくなる．

・GitHub以外のツールを使う

　フローによってはコミュニケーション支援などのためにGitHub以外のツールを導入しているものがあるが，外部のサービスであるため，使用しているツールに不具合などが発生してしまうと，最悪フローそのものが停まってしまう可能性がある．

・単一のリポジトリを使用している

　リポジトリを１つしか使わない場合，その開発フローで作成した成果物や，使用したデータなどが１つのリポジトリに集約されているため，１つのリポジトリに対する負荷が大きくなり，そのリポジトリに何かあるとフロー自体が崩壊してしまう可能性がある．

・複数のリポジトリを使用している

　逆に複数のリポジトリを使う場合は１つのリポジトリにかかる負担こそ小さくなるものの，リポジトリを複数管理するということはそれだけバックアップの数や煩雑さが増えるということである．

・Pull Requestでのコミュニケーション

　GitHubに備わっているPull Requestという機能はフロー中にメンバや責任者のレビューのために使われるが，成果物の質によりレビューが通らないのではなく，責任者等が多忙などの理由でレビューができなかった場合にはフローが停滞してしまうという可能性がある．

・ブランチを破棄する

　フローによってはリポジトリの負荷を小さくするためなどの理由で，使い終わったブランチを破棄してしまうが，そのブランチ上で必要な作業の漏れが破棄した後に発覚した場合，もう一度同じブランチを作る，もしくは作業しているブランチでそのもれた作業を行うなどしなくてはならなくなる．結果ブランチごとの役割が混乱するなどの不都合が発生してしまう．

・フローが自動化されている

　定期的にテストを行いたい，フロー中のメンバの負担を減らしたいなどの理由からフローを自動化しているものがある，しかし構築には自動化に関する知識が必要となる，自動化構築時点で予想できなかった作業が発覚したときはその作業を手動で行う，または自動化するプログラムなど再構築しなければならないなど，自動化に関するスキルが構築時点で求められる．

# 4.2．各開発フローで発生しうるリスク

　以下に各開発フローで発生する可能性があるリスクについて記述する．

# 4.2.1．GitHubフロー

・常にデプロイする

　このフローはデプロイを中心にしたフローなので当然，常にデプロイすることのリスクは発生する．

・デプロイをあまりしない

　上記のようにデプロイを頻繁に行うフローなので該当しない．

・フローが複雑

　フローはむしろシンプルなものであり，20人程度のプロジェクトまでなら大きな問題が発生したことがないといわれている．よってこのリスクには該当しない

・複数のブランチを扱う

　メインブランチ，作業用ブランチと2本のブランチを使用するため，複数のブランチを使用することのリスクは発生する．

・GitHub以外のツールを使う

　このフローにはGitHub以外のツールを使用しないので，該当しない．

・単一のリポジトリを使用している

　基本的に単一のリポジトリで運用するフローなので，このリスクは発生する．

・複数のリポジトリを使用している

　上記のように単一のリポジトリを使用するので該当しない．

・Pull Requestでのコミュニケーション

　フロー中，作業用ブランチをメインブランチにMergeする許可をメンバにPull

Requestで求めるため，このリスクは発生する．

・ブランチを破棄する

　フロー中にブランチをMergeすることはあっても破棄はしないので該当しない．

・フローが自動化されている

　このフローは自動化されていないため該当しない．

# 4.2.2．Gitフロー

・常にデプロイする

　このフローは常にデプロイすることはしないので該当しない．

・デプロイをあまりしない

　このフローはデプロイをするまでが長いので，このリスクは発生する．

・フローが複雑

　使用するブランチが複数あり，フロー自体が覚えることが多いため，このフローは複雑であるといえる．よってこのリスクは発生する．

・複数のブランチを扱う

　メインブランチ，開発用ブランチ，作業用ブランチ，リリース用ブランチ，修正用ブランチと5種類ものブランチを扱うフローであり，フロー自体の複雑さもあってMergeミスもかなり発生しやすいと考えられる．よってこのリスクは発生する．

・GitHub以外のツールを使う

　このフローにはGitHub以外のツールを使わないので該当しない．

・単一のリポジトリを使用している

　このフローは単一のリポジトリで運用するので，このリスクは発生する．

・複数のリポジトリを使用している

　上記のようにこのフローで使われるリポジトリは1つなので該当しない．

・Pull Requestでのコミュニケーション

　このフローではPull Requestによるレビューが行われないので該当しない．

・ブランチを破棄する

　フロー中にブランチをMergeすることはあっても破棄はしないので該当しない．

・フローが自動化されている

　このフローは自動化されていないため該当しない．

# 4.2.3．はてなブログフロー

・常にデプロイする

　このフローは常にデプロイすることはしないので該当しない．

・デプロイをあまりしない

　上記のようにこのフローはあまりデプロイをしないため，このリスクは発生する．

・フローが複雑

　必要となる機能ごとにissueを発行とブランチの作成を行い，そのブランチの作業が完了したらMergeするといったシンプルな流れを基本としているので該当しない．

・複数のブランチを扱う

　メインブランチと開発用ブランチ，そして機能ごとに作業用ブランチが作成されるため，複数のブランチを扱うフローである．よってこのリスクは発生する．

・GitHub以外のツールを使う

　このフローにはGitHub以外のツールを使用しないので，該当しない．

・単一のリポジトリを使用している

　このフローは単一のリポジトリで運用するので，このリスクは発生する．

・複数のリポジトリを使用している

　上記のようにこのフローで使われるリポジトリは1つなので該当しない．

・Pull Requestでのコミュニケーション

　フロー中，作業用ブランチをメインブランチにMergeする許可をメンバにPull

Requestで求めるが，このフローでは毎日14時からレビュータイムを設けるなどをして，このリスクが起きる可能性を低くしている．

・ブランチを破棄する

　フロー中にブランチをMergeすることはあっても破棄はしないので該当しない．

・フローが自動化されている

　このフローは自動化されていないため該当しない．

# 4.2.4．日本CAWフロー

・常にデプロイする

　このフローはデプロイを中心にしたGitHubフローをベースにしているので，常にデプロイすることのリスクは発生する．

・デプロイをあまりしない

　上記のようにデプロイを頻繁に行うフローなので該当しない．

・フローが複雑

　シンプルなGitHubフローをベースにしているので，このフローもシンプルである．よってこのリスクは該当しない．

・複数のブランチを扱う

　メインブランチ，開発ブランチと2本のブランチを使用するため，複数のブランチを使用することのリスクは発生する．

・GitHub以外のツールを使う

　このフローにはGitHub以外のツールを使用しないので，該当しない．

・単一のリポジトリを使用している

　基本的に単一のリポジトリで運用するフローなので，このリスクは発生する．

・複数のリポジトリを使用している

　上記のように単一のリポジトリを使用するので該当しない．

・Pull Requestでのコミュニケーション

　フロー中，作業用ブランチをメインブランチにMergeする許可をメンバにPull

Requestで求めるため，このリスクは発生する．

・ブランチを破棄する

　フロー中にブランチをMergeすることはあっても破棄はしないので該当しない．

・フローが自動化されている

　このフローは自動化されていないため該当しない．

# 4.2.5．ラクスルフロー

・常にデプロイする

　このフローは常にデプロイすることはしないので該当しない．

・デプロイをあまりしない

　上記のようにこのフローはあまりデプロイをしないため，このリスクは発生する．

・フローが複雑

　メインブランチから必要な機能の分だけブランチを作成し，それぞれのブランチで開発が終われば，メインブランチにMergeされるというシンプルなフローのため該当しない．

・複数のブランチを扱う

　メインブランチと必要とされる機能の分だけ扱うブランチが増えるので，このリスクは発生する．

・GitHub以外のツールを使う

　プロジェクト管理ツールであるRedmine，コミュニケーションツールであるskypeを使っているので，このリスクは発生する．

・単一のリポジトリを使用している

　このフローは単一のリポジトリで運用するので，このリスクは発生する．

・複数のリポジトリを使用している

　上記のようにこのフローで使われるリポジトリは1つなので該当しない．

・Pull Requestでのコミュニケーション

　フロー中，機能ごとに作られたブランチをメインブランチにMergeする許可をメンバにPull Requestで求めるため，このリスクは発生する．

・ブランチを破棄する

　開発が終了したブランチはメインブランチにMergeされた後，破棄されるのでこのリスクは発生する．

・フローが自動化されている

　このフローは自動化されていないため該当しない．

# 4.2.6．キャスレーフロー

・常にデプロイする

　このフローは常にデプロイすることはしないので該当しない．

・デプロイをあまりしない

　上記のようにこのフローはあまりデプロイをしないため，このリスクは発生する．

・フローが複雑

　Gitフローをベースに作られたフローであり，使用するブランチを少なくして簡略化したフローであるが，Gitフローほどではないにしろ複雑なフローであるのは変わりない．よってこのリスクは発生する．

・複数のブランチを扱う

　開発用ブランチ，作業用ブランチ，リリース用ブランチ，修正用ブランチと4本のブランチを扱うため，このリスクは発生する．

・GitHub以外のツールを使う

　このフローにはGitHub以外のツールを使用しないので，該当しない．

・単一のリポジトリを使用している

　このフローは単一のリポジトリで運用するので，このリスクは発生する．

・複数のリポジトリを使用している

　上記のようにこのフローで使われるリポジトリは1つなので該当しない．

・Pull Requestでのコミュニケーション

　開発が終わった作業用ブランチを製品版ブランチにMergeするときには，Pull Requestを送りレビューを行うので，このリスクは発生する．

・ブランチを破棄する

　フロー中にブランチをMergeすることはあっても破棄はしないので該当しない．

・フローが自動化されている

　このフローは自動化されていないため該当しない．

# 4.2.7．Amingフロー

・常にデプロイする

　このフローは常にデプロイすることはしないので該当しない．

・デプロイをあまりしない

　上記のようにこのフローはあまりデプロイをしないため，このリスクは発生する．

・フローが複雑

　リポジトリ同士を跨ぐとはいえ，仕組みはGitHubフローと大差ないため，フローが複雑ということではない，よって該当しない．

・複数のブランチを扱う

　リポジトリごとにメインブランチと作業用ブランチを作成するため複数のブランチを扱う，よってこのリスクは発生する．

・GitHub以外のツールを使う

　このフローにはGitHub以外のツールを使用しないので，該当しない．

・単一のリポジトリを使用している

　このフローではメンバごとにメインリポジトリをforkしたリポジトリも使用するため該当しない．

・複数のリポジトリを使用している

　上記のようにリポジトリをメンバごとに作成するので，このリスクは発生する．

・Pull Requestでのコミュニケーション

　メンバが作成したブランチをメインリポジトリにMergeするときにPull Requestを送りレビューを行うので，このリスクは発生する．

・ブランチを破棄する

　フロー中にブランチをMergeすることはあっても破棄はしないので該当しない．

・フローが自動化されている

　このフローは自動化されていないため該当しない．

# 4.2.8．LINEフロー

・常にデプロイする

　このフローは常にデプロイ可能なブランチをバージョンごとに作成して運用するので，

このリスクは発生する．

・デプロイをあまりしない

　上記のようにデプロイを頻繁に行うフローなので該当しない．

・フローが複雑

　デプロイ可能なブランチをバージョンごとに次々と作り出すフローなので，フローは複雑とはいえない，よって該当しない．

・複数のブランチを扱う

　バージョンの数だけ複数のブランチを扱うので，このリスクは発生する．

・GitHub以外のツールを使う

　このフローにはGitHub以外のツールを使用しないので，該当しない．

・単一のリポジトリを使用している

　このフローは単一のリポジトリで運用するので，このリスクは発生する．

・複数のリポジトリを使用している

　上記のようにこのフローで使われるリポジトリは1つなので該当しない．

・Pull Requestでのコミュニケーション

　このフローではPull Requestによるレビューが行われないので該当しない．

・ブランチを破棄する

　修正が終わったブランチを破棄するので，このリスクは発生する．

・フローが自動化されている

　このフローは自動化されていないため該当しない．

# 4.2.9．サイボウズフロー

・常にデプロイする

　このフローは常にデプロイすることはしないので該当しない．

・デプロイをあまりしない

　上記のようにこのフローはあまりデプロイをしないが段階を進めるたびにバグが発生しそうな不具合を修正するので，デプロイをあまりしないことのリスクの発生率は低い．

・フローが複雑

　段階ごとにリポジトリにMergeしていくフローなので，フロー辞退は複雑ではないので該当しない．

・複数のブランチを扱う

　メンバごとにブランチを用意するので複数のブランチを扱うフローだが，ブランチというよりもリポジトリごとに作業を行うので，このリスクの発生確率は低くなる．

・GitHub以外のツールを使う

　このフローにはGitHub以外のツールを使用しないので，該当しない．

・単一のリポジトリを使用している

　開発用リポジトリ，安定環境リポジトリ，メンバごとにリポジトリを扱うので該当しない．

・複数のリポジトリを使用している

　上記のようにリポジトリを複数に作成するので，このリスクは発生する．

・Pull Requestでのコミュニケーション

　次の段階のリポジトリにMergeするたびにPull Requestを送りレビューを行うので，このリスクは発生する．

・ブランチを破棄する

　フロー中にブランチをMergeすることはあっても破棄はしないので該当しない．

・フローが自動化されている

　このフローは自動化されていないため該当しない．

# 4.2.10．フィヨルドフロー

・常にデプロイする

　CIサーバーで午前0時に定期デプロイが行われるようになっているので，このリスクは発生する．

・デプロイをあまりしない

　上記のように頻繁にデプロイを行うので該当しない

・フローが複雑

　フロー自体は複雑だが，すべてをメンバがやるのではなくある程度は負荷が少なくいので，フローが複雑なことで発生するリスクは低くなる．

・複数のブランチを扱う

　このフローでは複数のブランチを扱わないので該当しない．

・GitHub以外のツールを使う

　PivotalTrackerやcapistranoといったツールを使うので，このリスクは発生する．

・単一のリポジトリを使用している

　このフローは単一のリポジトリで運用するので，このリスクは発生する．

・複数のリポジトリを使用している

　上記のようにこのフローで使われるリポジトリは1つなので該当しない．

・Pull Requestでのコミュニケーション

　このフローではPull Requestによるレビューが行われないので該当しない．

・ブランチを破棄する

　フロー中にブランチを破棄しないので該当しない．

・フローが自動化されている

　フローの半分以上が自動化されているので，このリスクは発生する．

# 4.2.11．イストフロー

・常にデプロイする

　5分おきにテストとデプロイが行われるためデプロイを頻繁に行うが，変更があった場合のみのため，このリスクには該当しない．

・デプロイをあまりしない

　上記のようにほぼ変更を行うたびにデプロイされるので，大きなバグが入り込むといったリスクは発生しない．

・フローが複雑

　フロー自体は複雑だが，すべてをメンバがやるのではなくある程度は負荷が少なくいので，フローが複雑なことで発生するリスクは低くなる．

・複数のブランチを扱う

　デプロイ用に複数のブランチを扱うが，それぞれのブランチにMergeするなどの変更を加えないため，このリスクの発生確率は低くなる．

・GitHub以外のツールを使う

　Jenkinsやcapistranoといったツールを使用するため，このリスクは発生する．

・単一のリポジトリを使用している

　このフローではメンバごとにメインリポジトリをforkしたリポジトリも使用するため該当しない．

・複数のリポジトリを使用している

　上記のようにリポジトリをメンバごとに作成するので，このリスクは発生する．

・Pull Requestでのコミュニケーション

　メンバが加えた変更をメインリポジトリにMergeするときにPull Requestを送りレビューを行うので，このリスクは発生する．

・ブランチを破棄する

　フロー中にブランチを破棄しないので該当しない．

・フローが自動化されている

　フローの半分以上が自動化されているので，このリスクは発生する．

# 4.2.12．矢吹研フロー①

・常にデプロイする

　このフローは常にデプロイすることはしないので該当しない．

・デプロイをあまりしない

　上記のようにこのフローはあまりデプロイをしないため，このリスクは発生する．

・フローが複雑

　基本的にリポジトリにcommitするだけのフローなので該当しない．

・複数のブランチを扱う

　ブランチでフローの管理を行わないため該当しない．

・GitHub以外のツールを使う

　このフローにはGitHub以外のツールを使用しないので，該当しない．

・単一のリポジトリを使用している

　このフローではメンバごとにメインリポジトリをforkしたリポジトリも使用するため該当しない．

・複数のリポジトリを使用している

　上記のようにリポジトリをメンバごとに作成するので，このリスクは発生する．

・Pull Requestでのコミュニケーション

　このフローではPull Requestによるレビューが行われないので該当しない．

・ブランチを破棄する

　ブランチを破棄しないので該当しない．

・フローが自動化されている

　このフローは自動化されていないため該当しない．

# 4.2.13．矢吹研フロー②

・常にデプロイする

　このフローは常にデプロイすることはしないので該当しない．

・デプロイをあまりしない

　上記のようにこのフローはあまりデプロイをしないため，このリスクは発生する．

・フローが複雑

　各々のリポジトリに変更を加え，変更をメインリポジトリにMergeするだけのフローなので該当しない．

・複数のブランチを扱う

　ブランチでフローの管理を行わないため該当しない．

・GitHub以外のツールを使う

　このフローにはGitHub以外のツールを使用しないので，該当しない．

・単一のリポジトリを使用している

　このフローではメンバごとにメインリポジトリをforkしたリポジトリも使用するため該当しない．

・複数のリポジトリを使用している

　上記のようにリポジトリをメンバごとに作成するので，このリスクは発生する．・単一のリポジトリを使用している

・Pull Requestでのコミュニケーション

　メンバが加えた変更をメインリポジトリにMergeするときにPull Requestを送りレビューを行うので，このリスクは発生する．

・ブランチを破棄する

　フロー中にブランチをMergeすることはあっても破棄はしないので該当しない．

・フローが自動化されている

　このフローは自動化されていないため該当しない．

# 4.3．参考文献

[1] Sato\_4tree, Git / GitFlowをチームに導入してよかったこと・よくなかったこと 2014, @satomikko94.b, 2014-07-16,

<http://satomikko94.hatenablog.com/entry/2014/07/16/232726>

[2] Gab-km, GitHub Flow(Japanese translation), GitHub Gist, 2011-08-31, <https://gist.github.com/Gab-km/3705015>

[3] 大塚弘記, GitHub実践入門 Pull Requestによる開発の変革, 技術評論社, 2014.

# 5．結果・考察

　この項では，分析結果，それについての考察を記述する．

# 5.1．分析結果

　調査したフローをリスクの観点から分類するために，階層的クラスター分析を用いて分類した結果，13種類の開発フローは5つのクラスターに分類することができた．下部の数字は開発フローを示したものである．

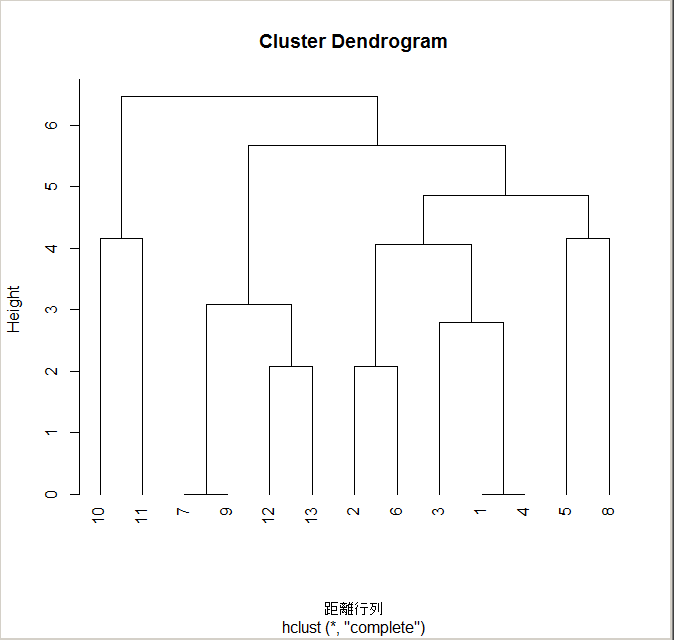


図 49クラスター分析結果

表 2分析結果番号対応表

|  |  |
| --- | --- |
| 番号 | 開発フロー名 |
| 1 | GitHubフロー |
| 2 | Gitフロー |
| 3 | はてなブログフロー |
| 4 | 日本CAWフロー |
| 5 | ラクスルフロー |
| 6 | キャスレーフロー |
| 7 | Amingフロー |
| 8 | LINEフロー |
| 9 | サイボウズフロー |
| 10 | フィヨルドフロー |
| 11 | イストフロー |
| 12 | 矢吹研フロー① |
| 13 | 矢吹研フロー② |

# 5.2．考察

# 5.2.1．分析結果を踏まえて

　便宜上，分析結果の左からクラスター1，クラスター2，クラスター3，クラスター4，クラスター5と呼ぶ．

・クラスター1（フィヨルドフロー，イストフロー）

　このクラスターに属する開発フローの共通している特徴は「開発フローに，自動化されている部分がある」ということである．

・クラスター2（Amingフロー，サイボウズフロー，矢吹研フロー①，矢吹研フロー②）

　このクラスターに属する開発フローの共通している特徴は「リポジトリを複数使用している開発フローである」ということである．

・クラスター3（Gitフロー，キャスレーフロー）

　このクラスターに属する開発フローの共通している特徴は「アジャイル開発のような流れが可能な開発フロー」ということである．

・クラスター4（GitHubフロー，はてなブログフロー，日本CAWフロー）

　このクラスターに属する開発フローの共通している特徴は「使用するブランチが少ないもの」ということである．

・クラスター5（ラクスルフロー，LINEフロー）

　このクラスターに属する開発フローの共通している特徴は「使用したブランチを破棄するもの」ということである．

　すなわち，この5つの特徴こそが，ソフトウェア開発における開発フローの選択する指標になるのではないかと考える．

# 5.2.2．クラスターごとの特徴に関する考察

・開発フローの自動化

開発フローの自動化によって何がメリットとなるかというと，フローの一部を自動化することによってプロジェクトに参加しているメンバの負担を少しでも減らすこと，また自動化することでフロー自体を繰り返すこと，それも膨大な回数を行うことでテストを数多く行い，バグや不具合をこまめに発見・修正できることだろう．このクラスターに属するフローは，開発するソフトウェアに，テストを何度も繰り返すほどの必要性や規模がある，またフローを自動化するためには自動化のための知識が必要となる．すなわち「スキルが高いメンバが集まる大規模なソフトウェア開発プロジェクト」に向いている開発フローだと考えられる．

・リポジトリの複数利用（メンバごとにリポジトリを利用）

　リポジトリの複数利用によって何がメリットになるかというと，リポジトリを複数に分けることで，リポジトリごとに用途を分けることができる，ブランチに直接変更を加えるよりも，メンバのリポジトリにまず変更を加えてから，メインのリポジトリに変更を加えたほうが誤操作を起こしにくい，メンバごとの変更履歴が確認しやすいということだろう．このクラスターに属するフローは，リポジトリごとに機能を待たせたい，メンバ毎にリポジトリを作成することでメインリポジトリに誤った更新をさせないようにすることだろう．すなわち「参加メンバが多いプロジェクト」に向いている開発フローだと考えられる．

・アジャイル開発のような流れが可能

　アジャイル開発のように開発できることの何がメリットかというと，アジャイル開発をよく理解したものならこのクラスターに属する開発フローを理解しやすいということだろう．よってこの開発フローは「アジャイル型のソフトウェア開発を行うプロジェクト」に向いているプロジェクトだと考えられる．

・使用するブランチが少ない

　使用するブランチが少ないことによって何がメリットになるかというと，管理するブランチが少ないので必然的にシンプルな開発フローになるので，GitHubにあまりかかわりが無く，GitHubのスキルがあまり無いメンバでも運用可能ということである．よってこの開発フローは「GitHubを今まで導入した経験が無いプロジェクト」に向いていると考えられる．

・使用済ブランチを破棄

　使用済ブランチを破棄することによって何がメリットになるかというと，破棄されるブランチは大抵，開発するべき機能やバージョンなど明確な目的を持ったブランチであることが多く，Merge後に破棄されても漏れが無ければ明確に何を更新したのかわかりやすく，変更管理がよりやりやすくなるということだろう．よってこの開発フローは「開発するソフトウェアに何を盛り込むのかが明確になっているプロジェクト」に向いていると考えられる．

# 5.2.3．まとめ

表 3プロジェクトと開発フロー

|  |  |  |
| --- | --- | --- |
| プロジェクトの特徴 | 開発フローの特徴 | 該当する開発フロー |
| ・メンバのスキルが高い  ・大規模なプロジェクト | ・フローが自動化されている | フィヨルドフロー, イストフロー |
| ・参加メンバが多い | ・複数のリポジトリを使用する | Amingフロー, サイボウズフロー, 矢吹研フロー①, 矢吹研フロー② |
| ・アジャイル型の  ソフトウェア開発 | ・アジャイル開発の  ような流れが可能 | Gitフロー, キャスレーフロー |
| ・GitHubを今まで導入した  経験が無いプロジェクト | ・使用するブランチが少ない | GitHubフロー, 日本CAWフロー, はてなブログフロー, |
| ・ソフトウェアに何を  盛り込むのかが明確 | ・使用済ブランチを破棄 | ラクスルフロー, LINEフロー |

# 5.2.4．今後の課題

　今回の研究では各開発フローに発生しうるリスクを観点に分類を行ったが，表5.2に挙げた以外のプロジェクトの特徴を持ったプロジェクトに対してどのような開発フローが向いているのか，また，リスク以外の観点からも調査することも今後の課題となるだろう．