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

表目次

図目次

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．

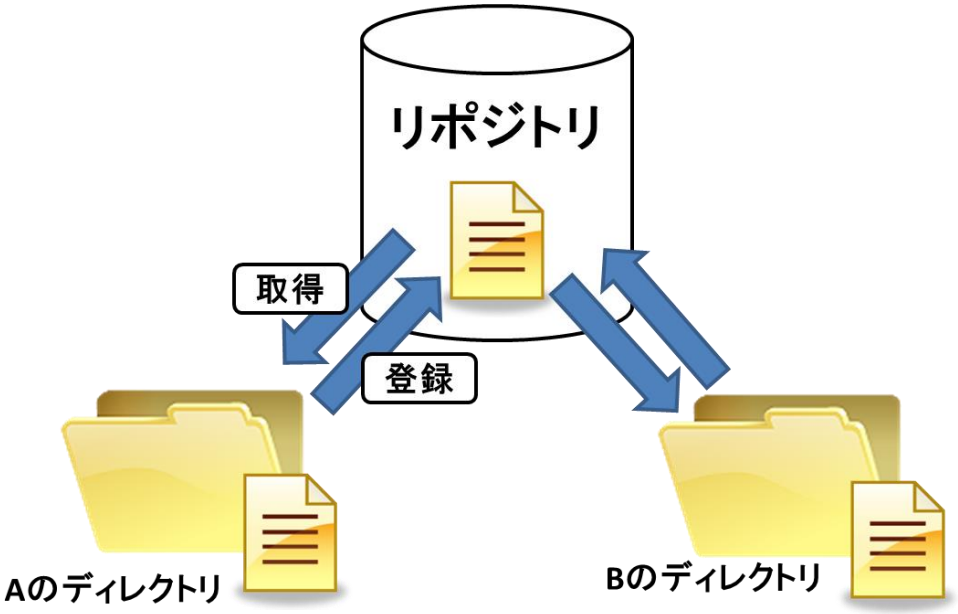

図 1 集中型の仕組み

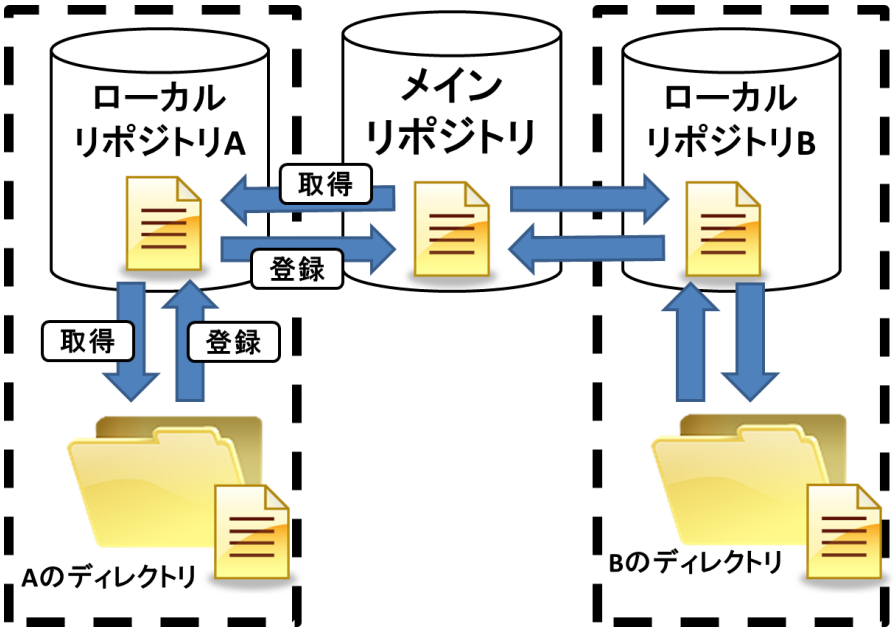

図 2 分散型の仕組み

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].

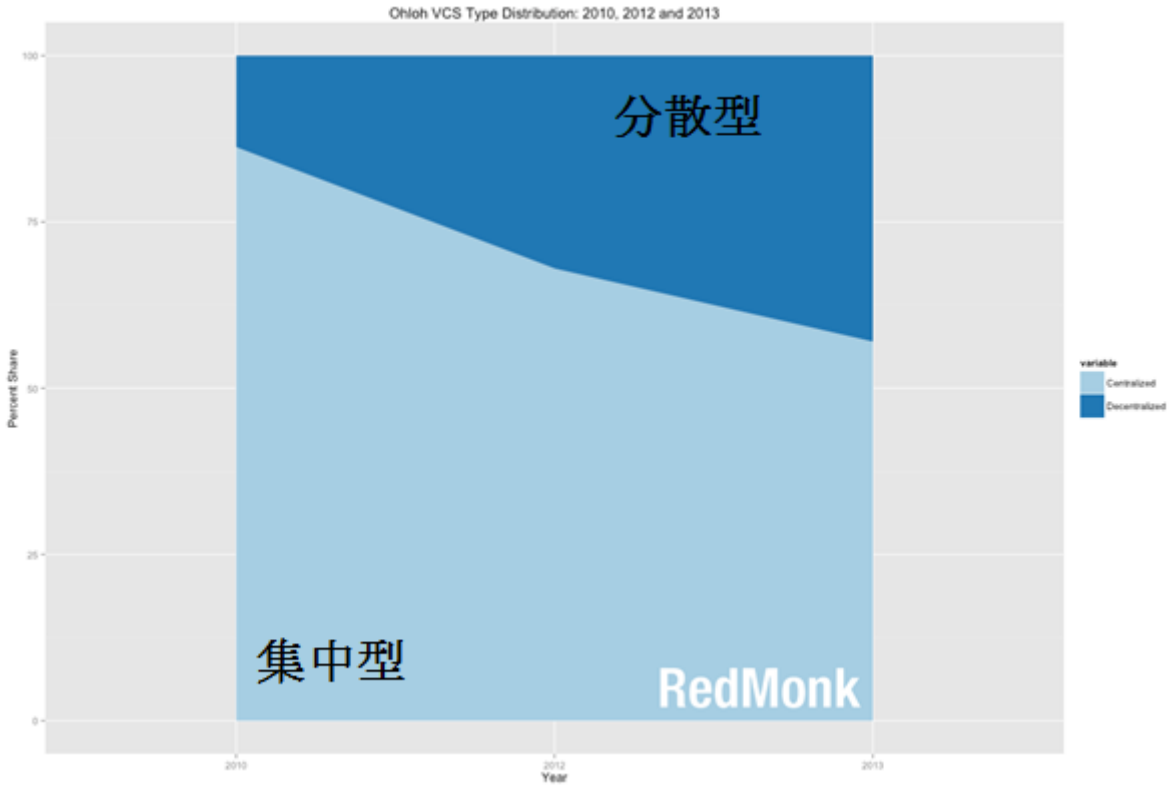

図 3 集中型と分散型のトレンド

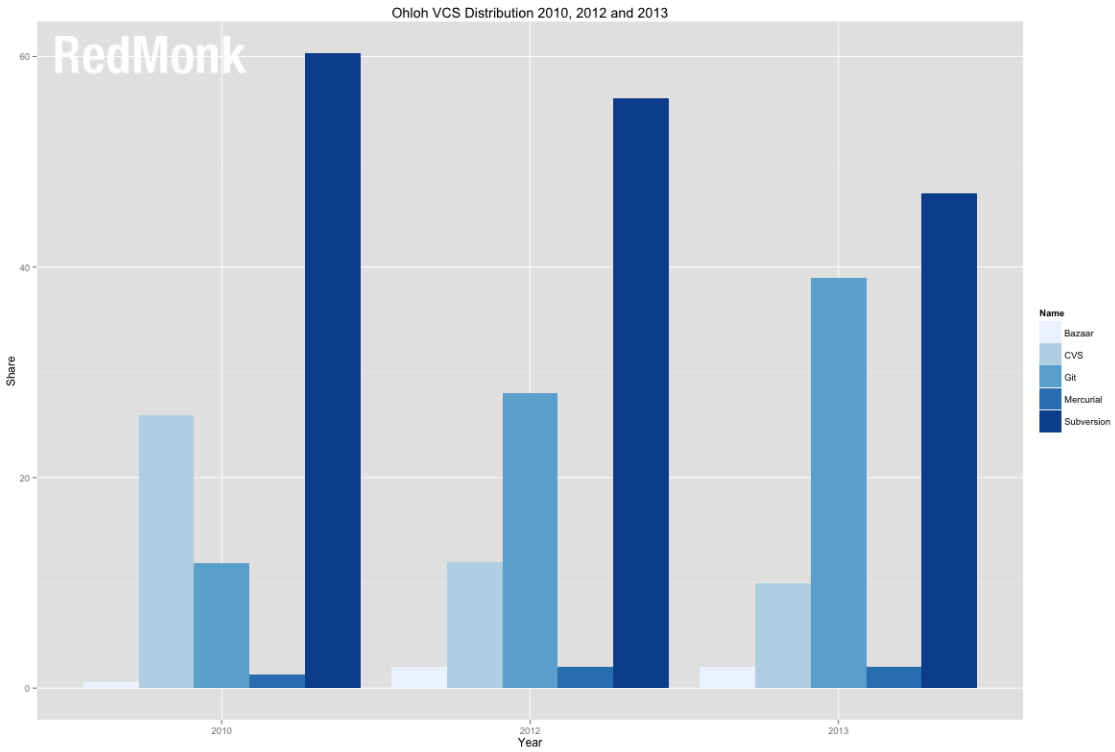

図 4 有名ソフトウェア開発会社が使っている，バージョン管理システムの利用率

「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．

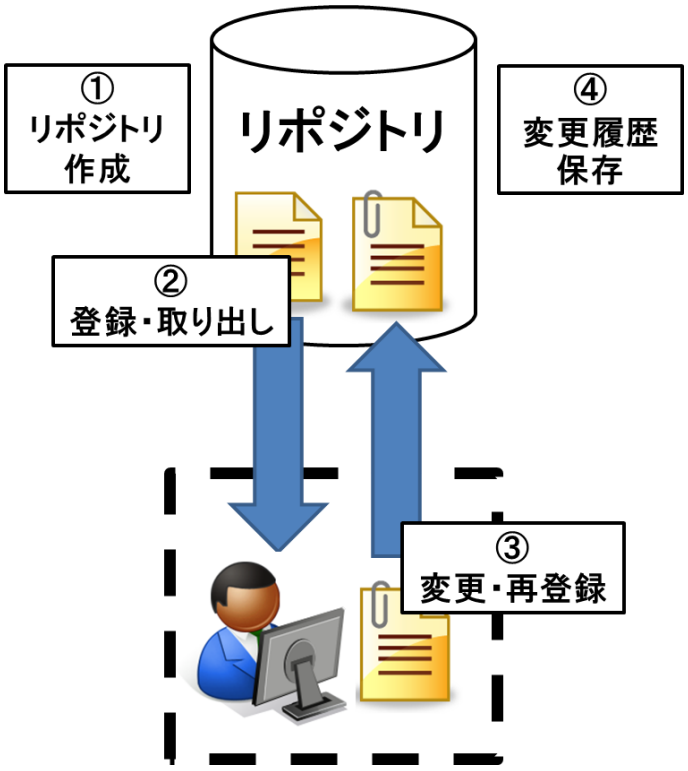

図 5 バージョン管理の仕組み

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）

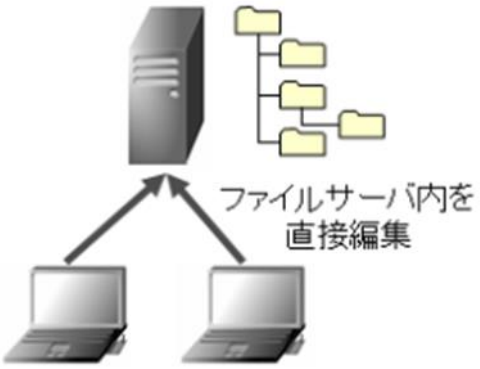

図 7 個別バージョン管理システム（少数でのチーム開発）

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．

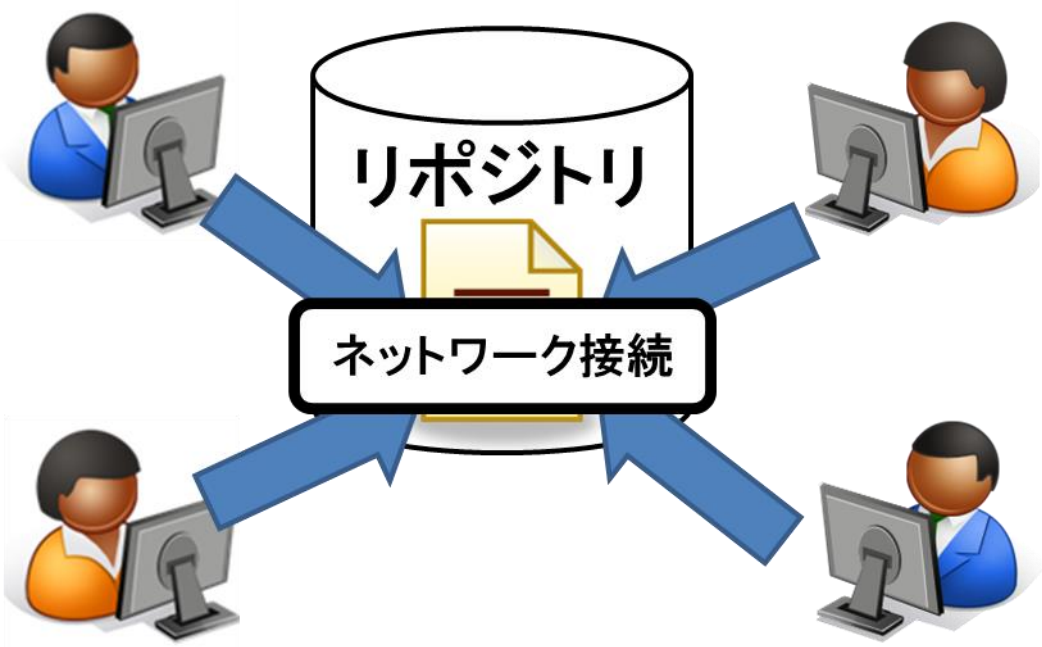

図 8 集中型バージョン管理システムのイメージ

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．

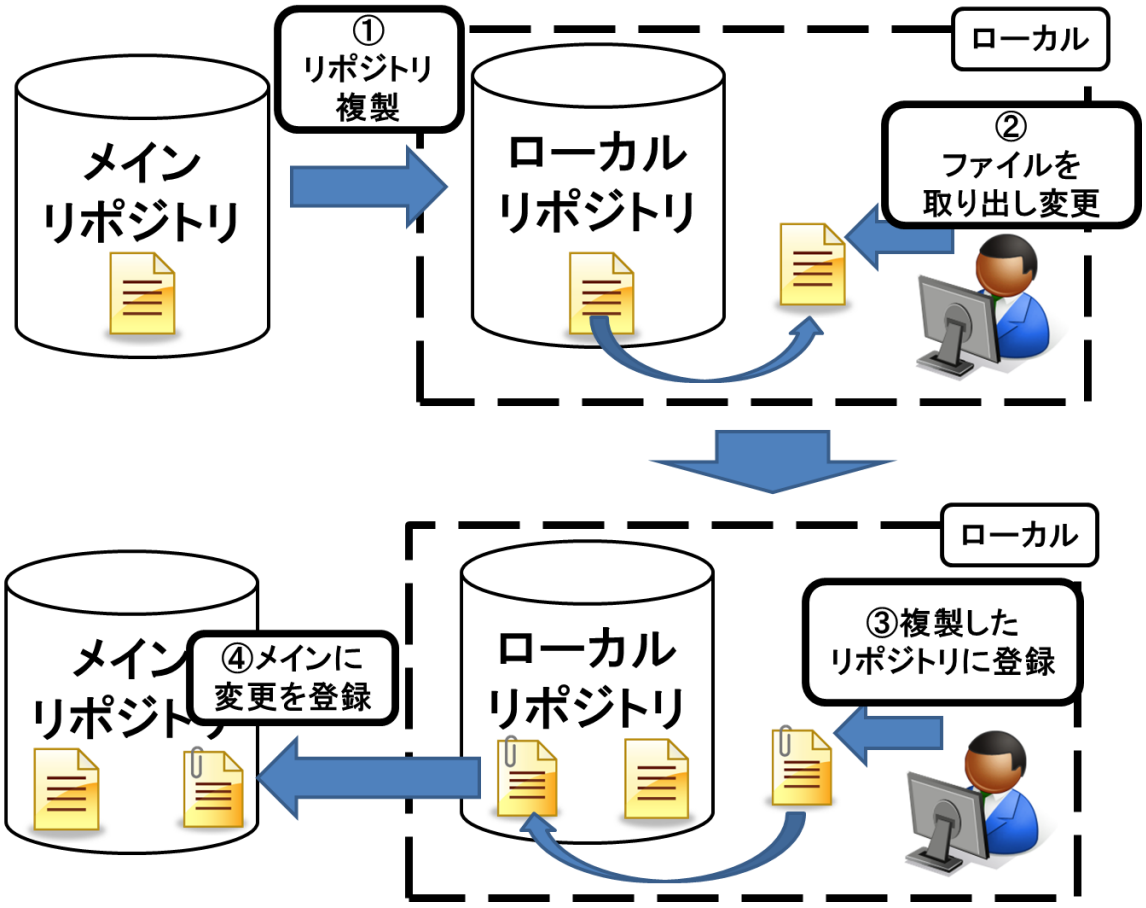

図 9 分散型バージョン管理システムのイメージ

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.

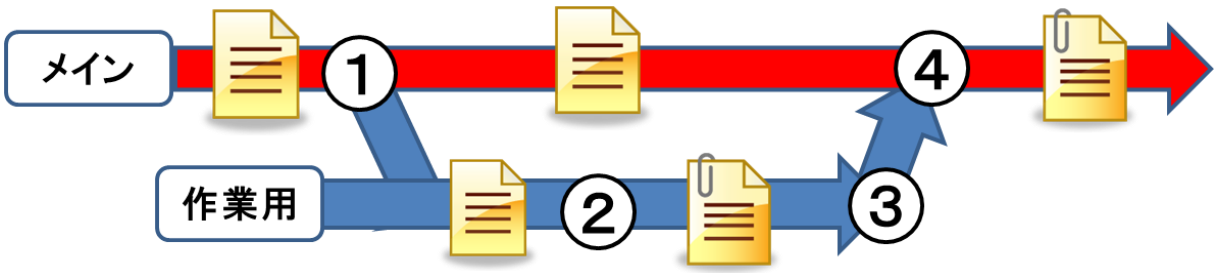

図 10 GitHub フロー図

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.

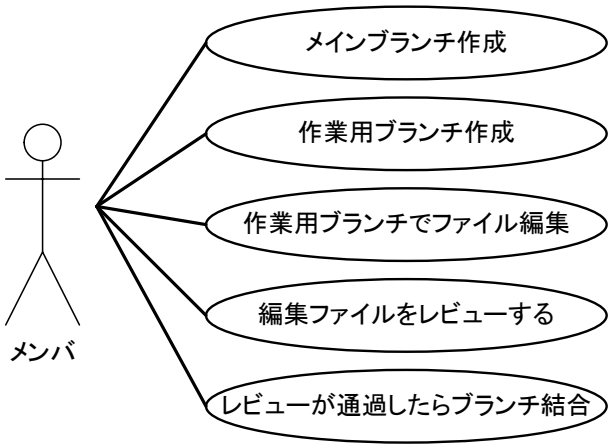

図 11 GitHub フローユースケース図

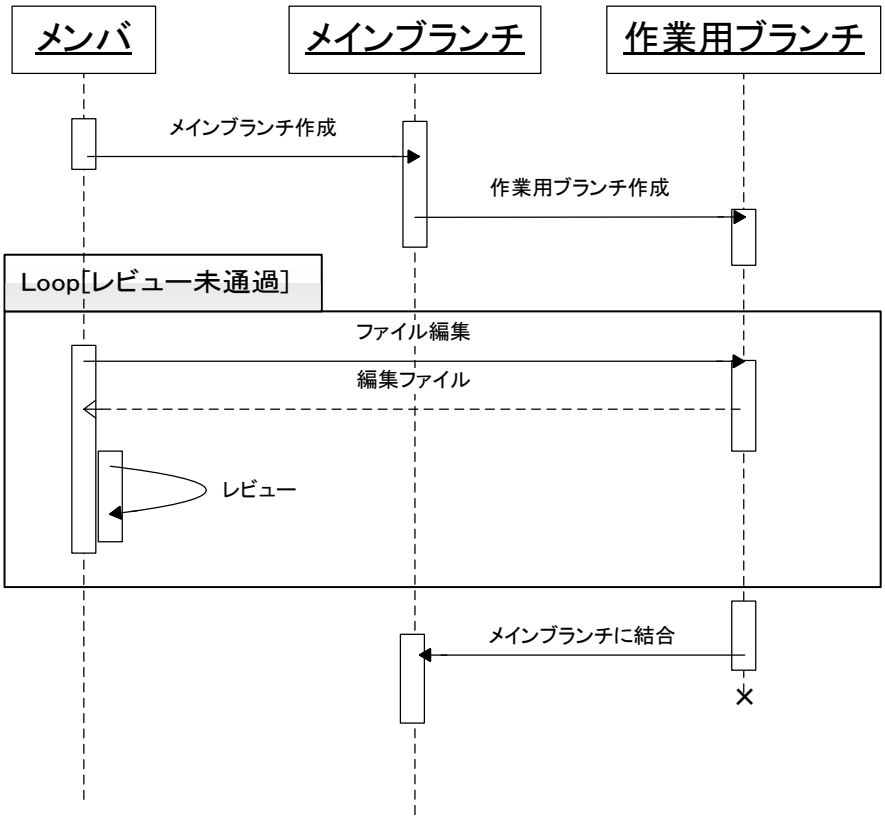

図 12 GitHub フローシーケンス図

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．

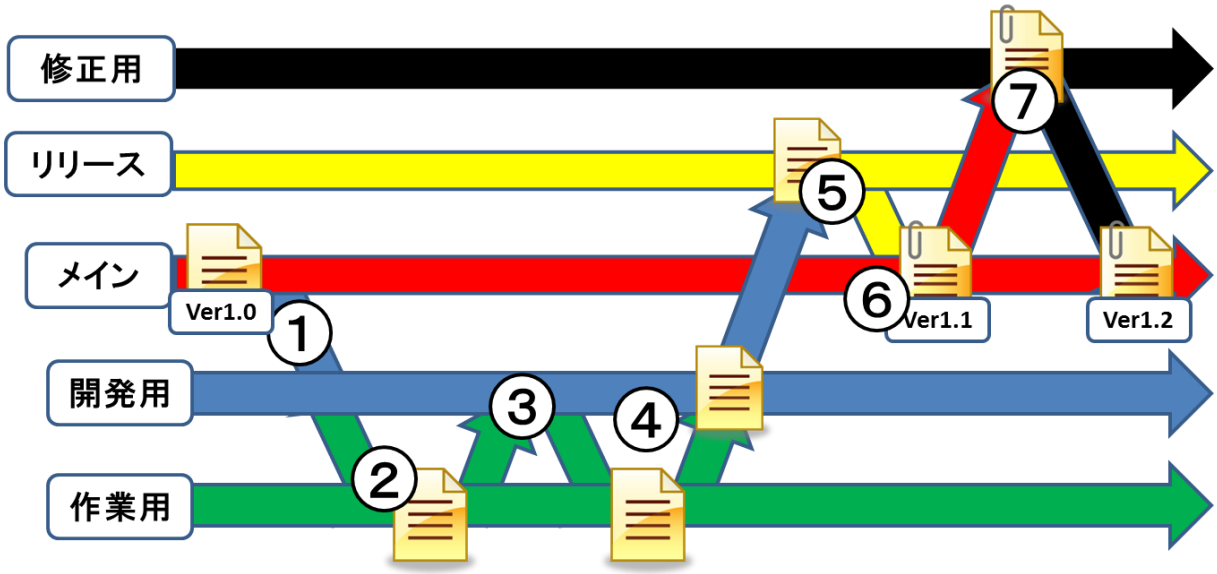

図 13 Git フロー図

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．

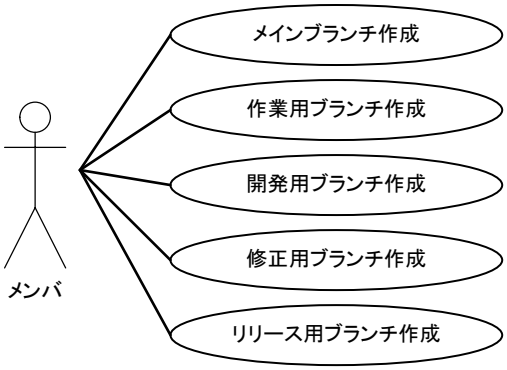

図 14 Git フローユースケース図

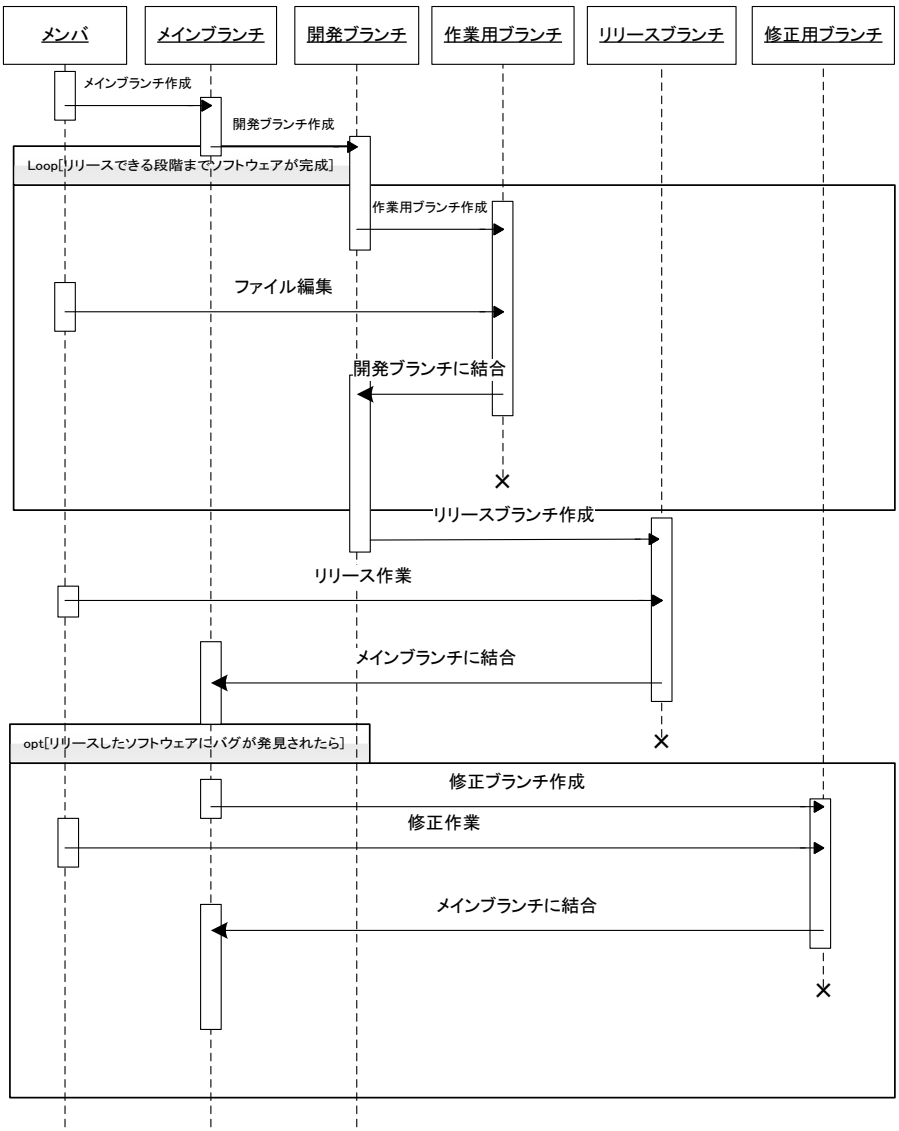

図 15 Git フローシーケンス図 3.2.3. はてなブログフロー

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．

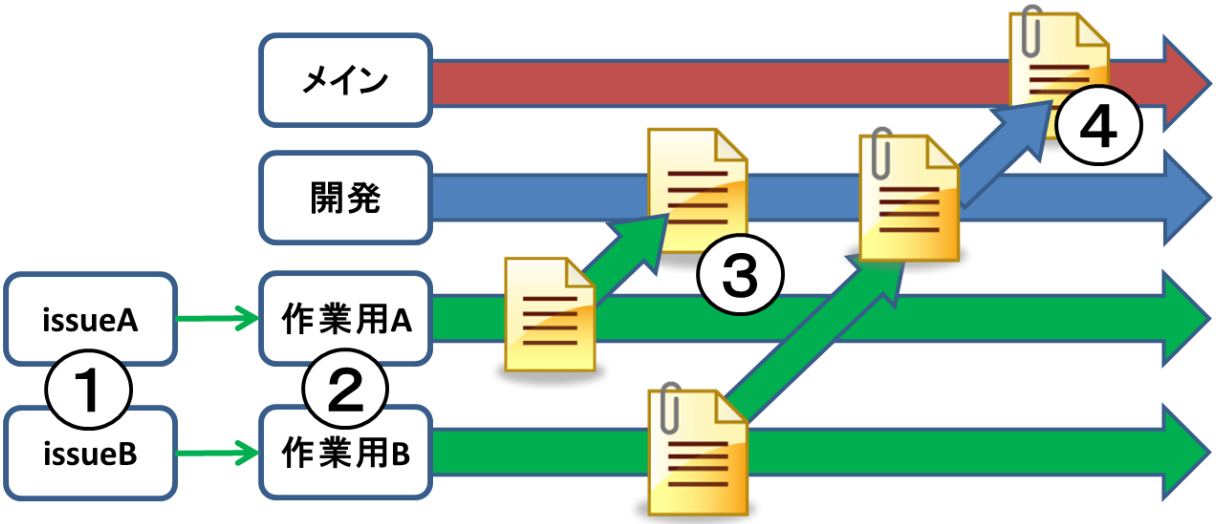

図 16 はてなブログフロー図

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].

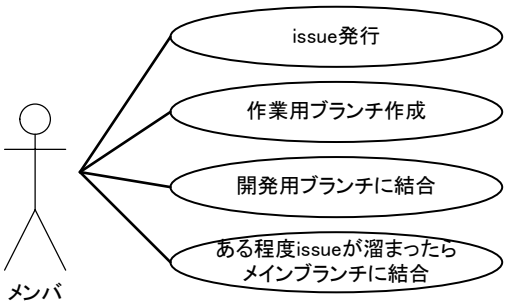

図 **17** はてなブログフローユースケース図

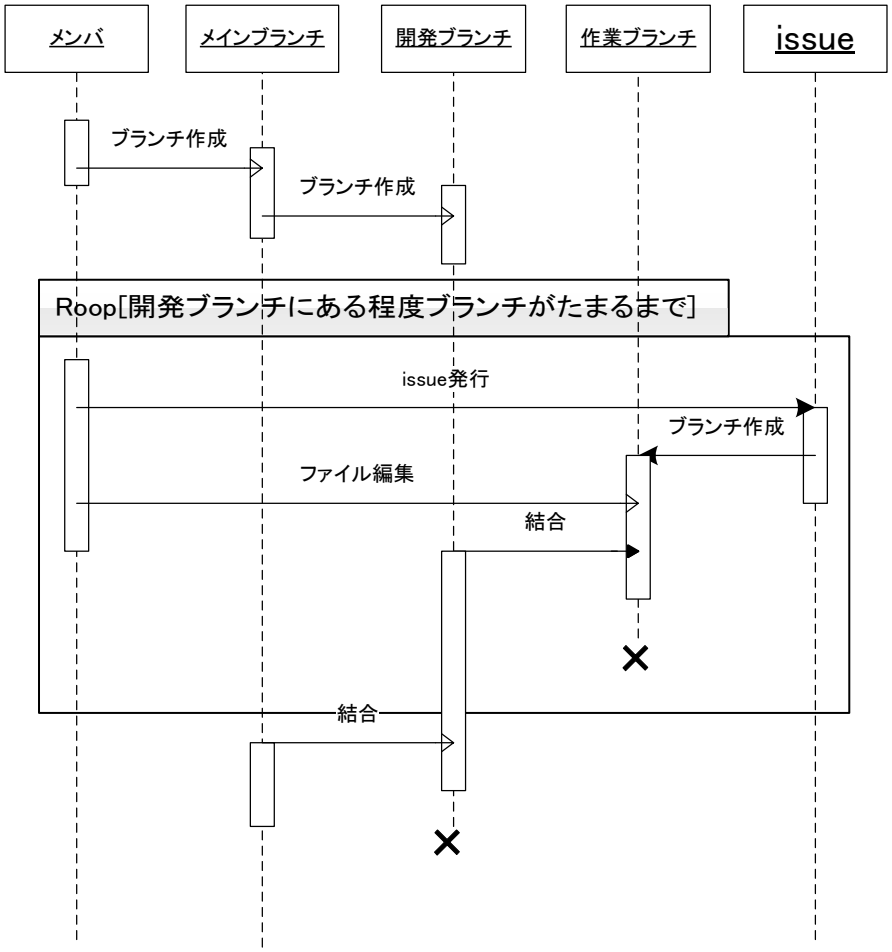

図 **18** はてなブログフローシーケンス図

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].

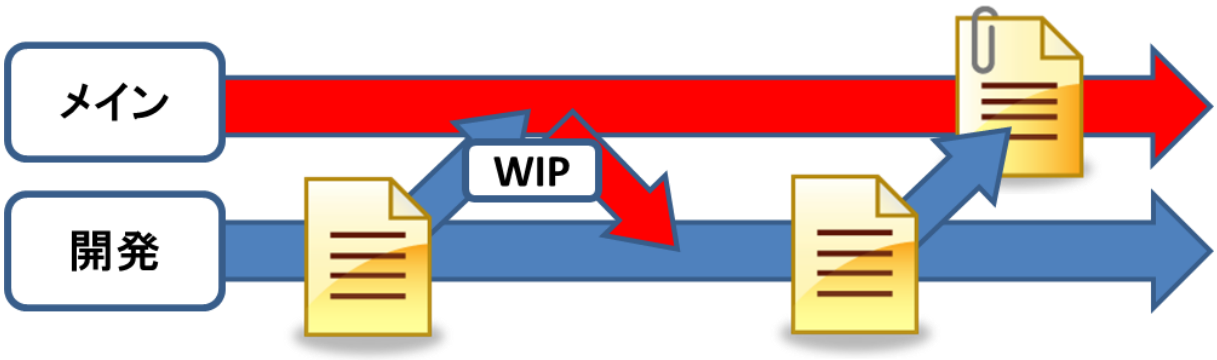

図 19 日本 CAW フロー図

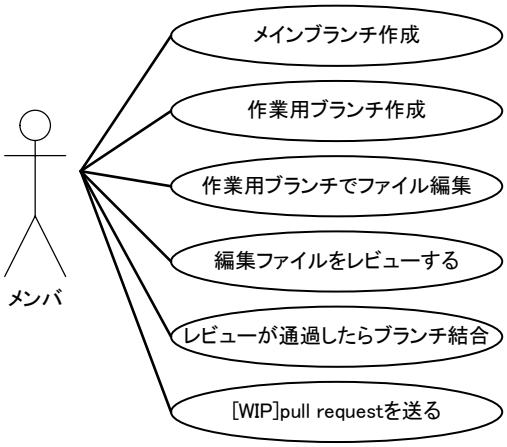

図 20 日本 CAW フローユースケース図

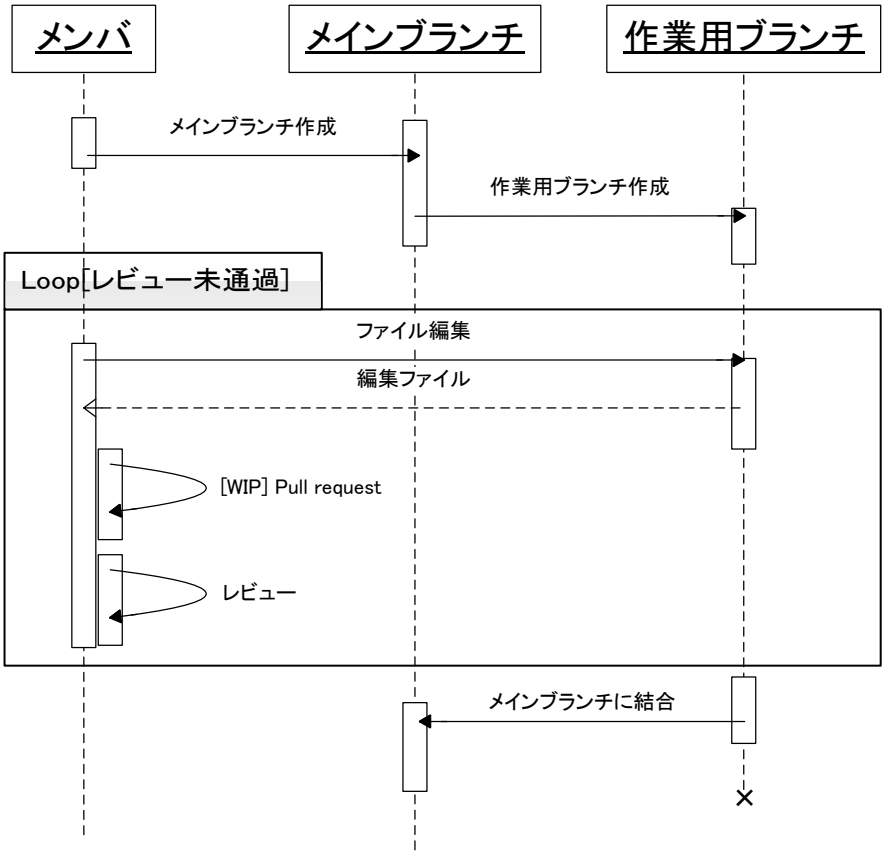

図 21 日本 CAW フローシーケンス図

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].

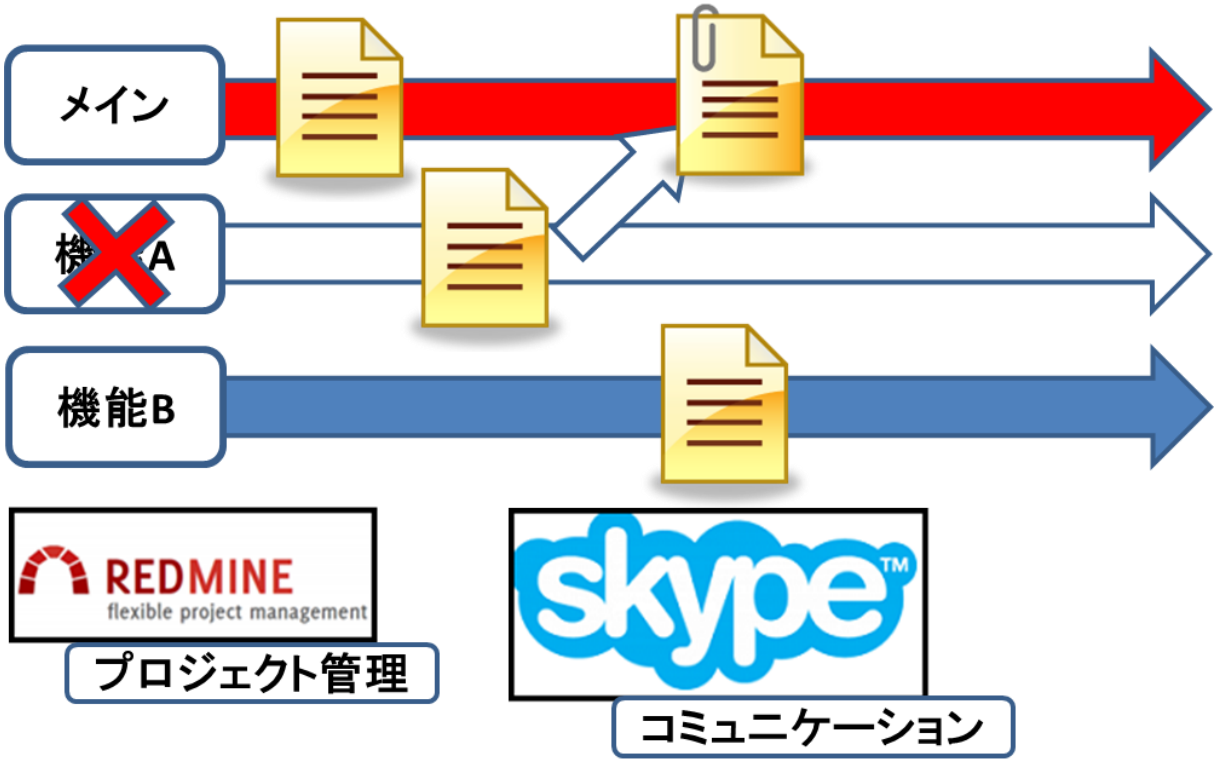

図 22 ラクスルフロー図

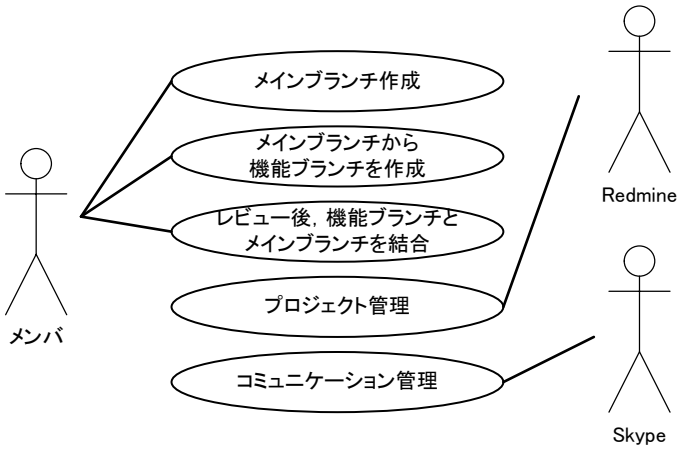

図 **23** ラクスルフローユースケース図

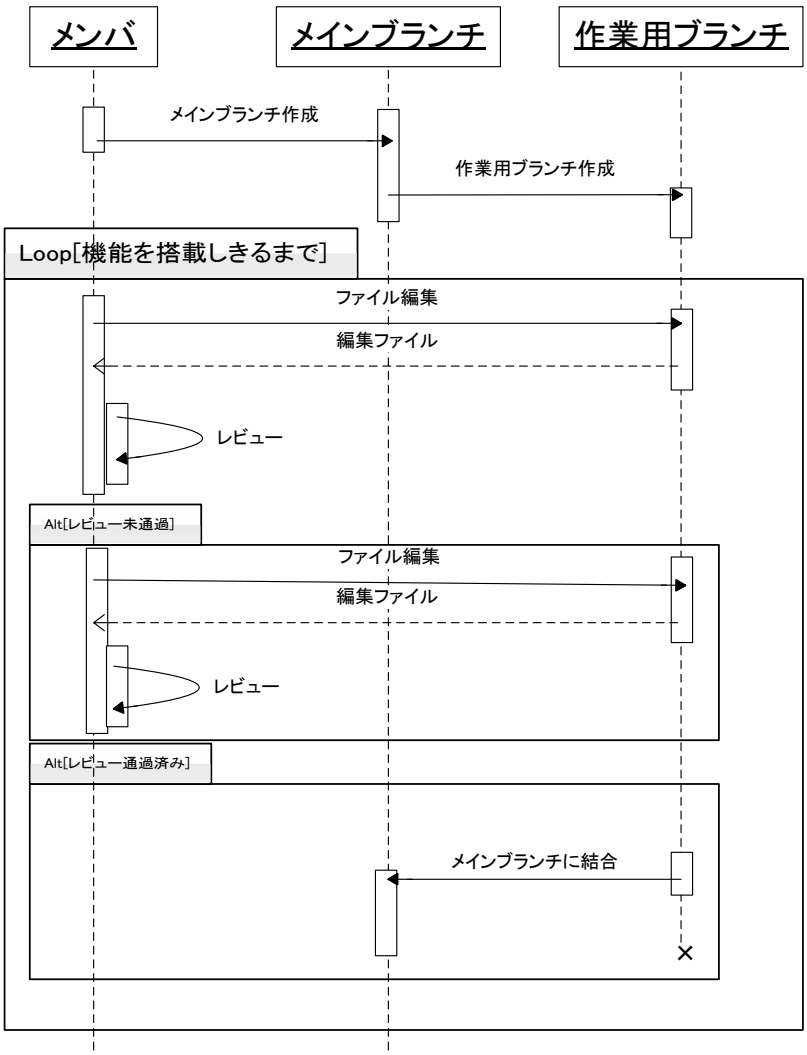

図 **24** ラクスルフローシーケンス図

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．

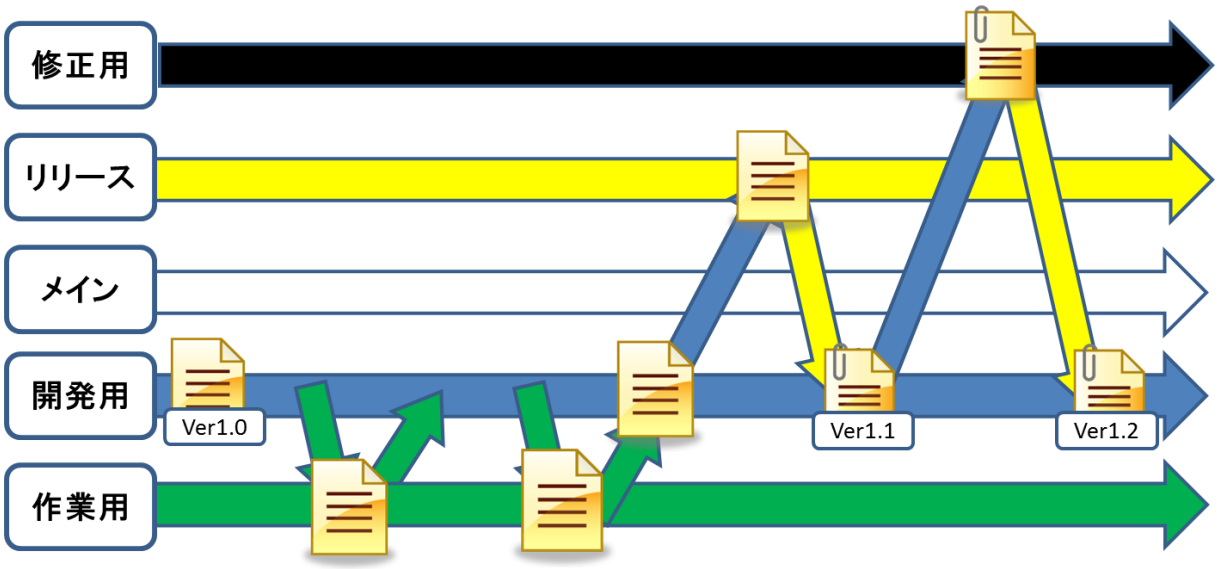

図 25 キャスレーフロー図

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．

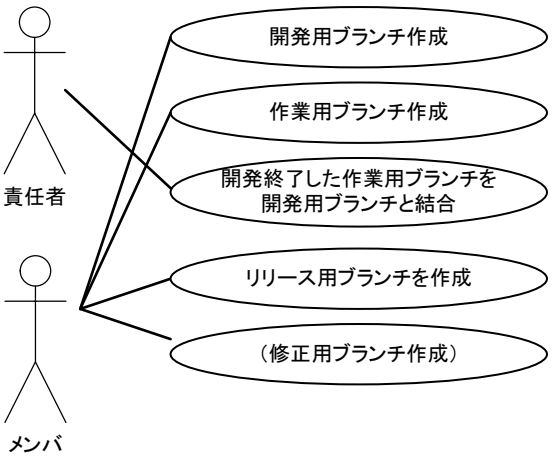

図 26 キャスレーフローユースケース図

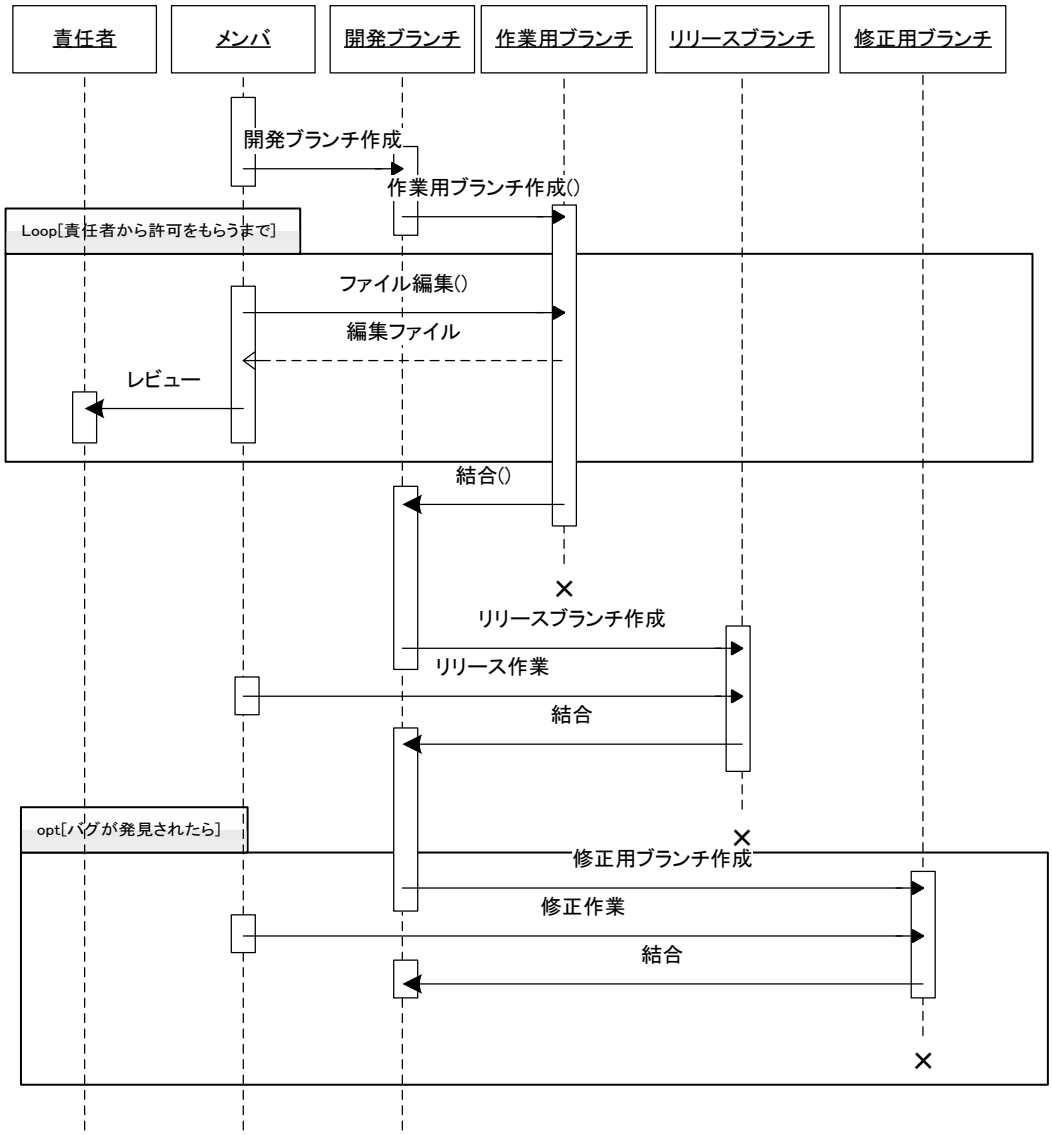

図 27 キャスレーフローシーケンス図

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].

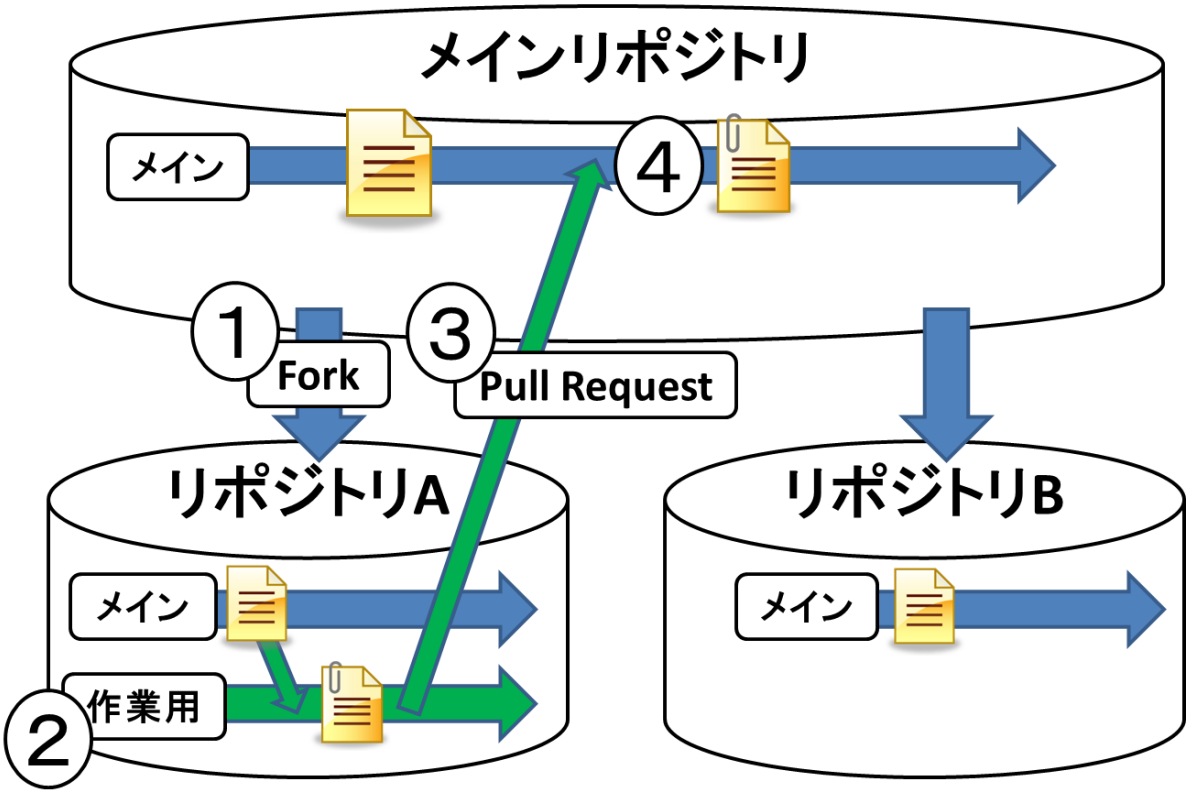

図 28Aming フロー図

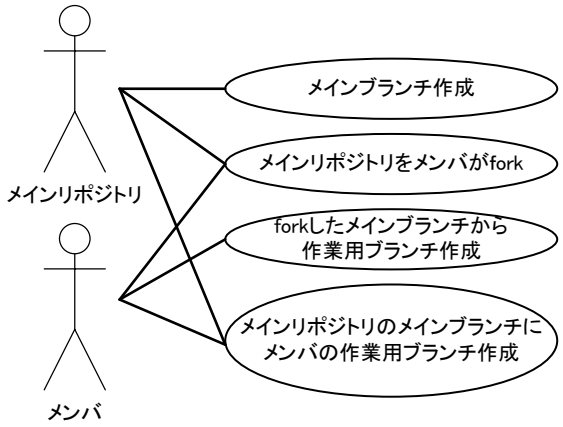

図 29Aming フローユースケース図

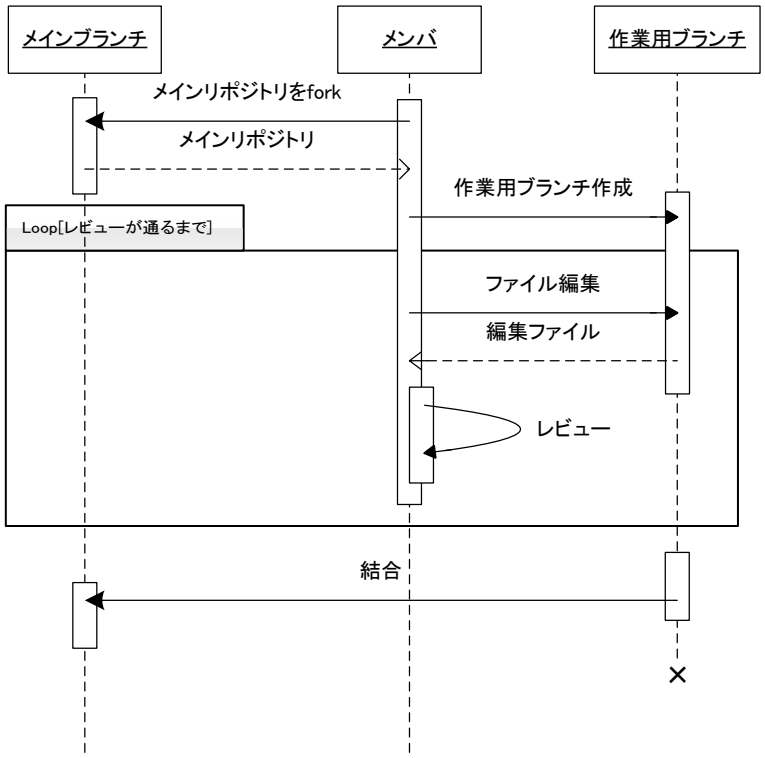

図 30Aming フローシーケンス図

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].

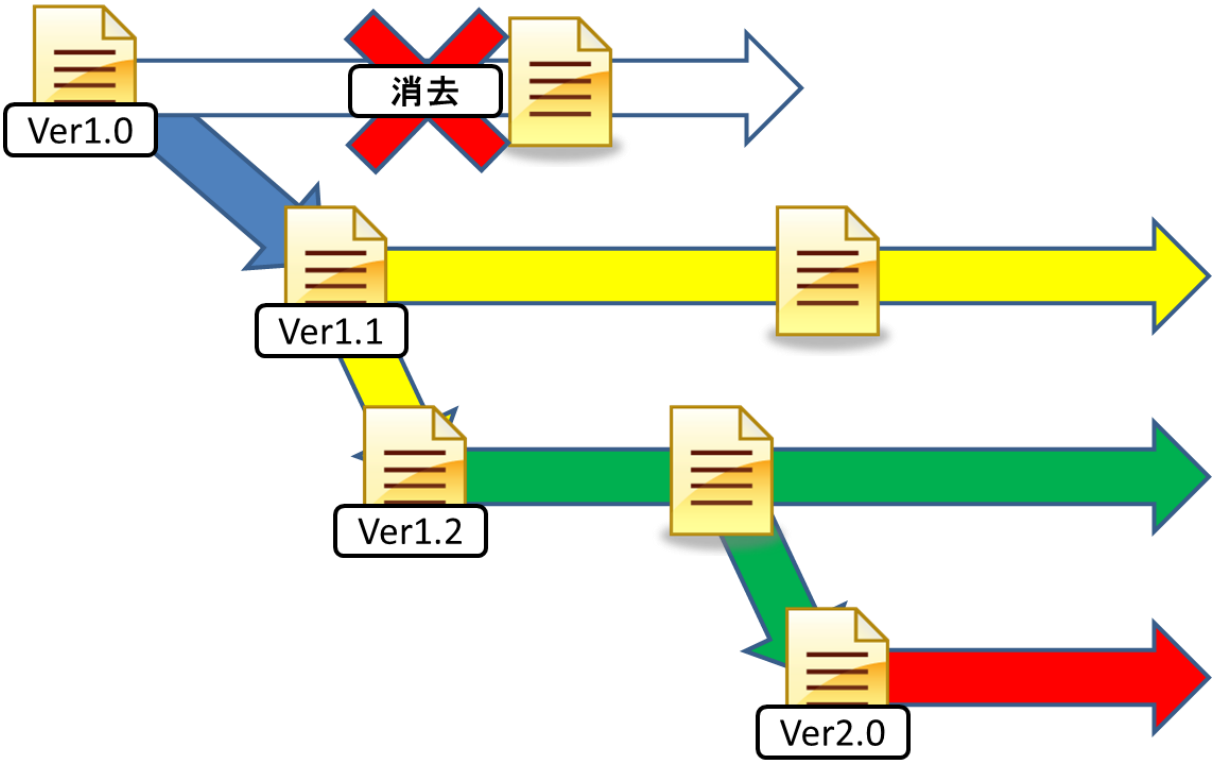

図 31LINE フロー図

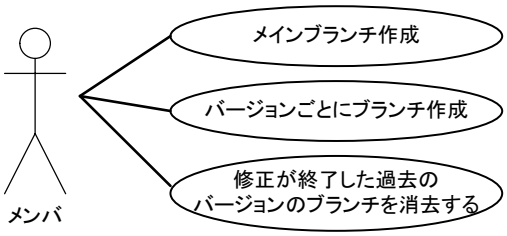

図 32LINE フローユースケース図

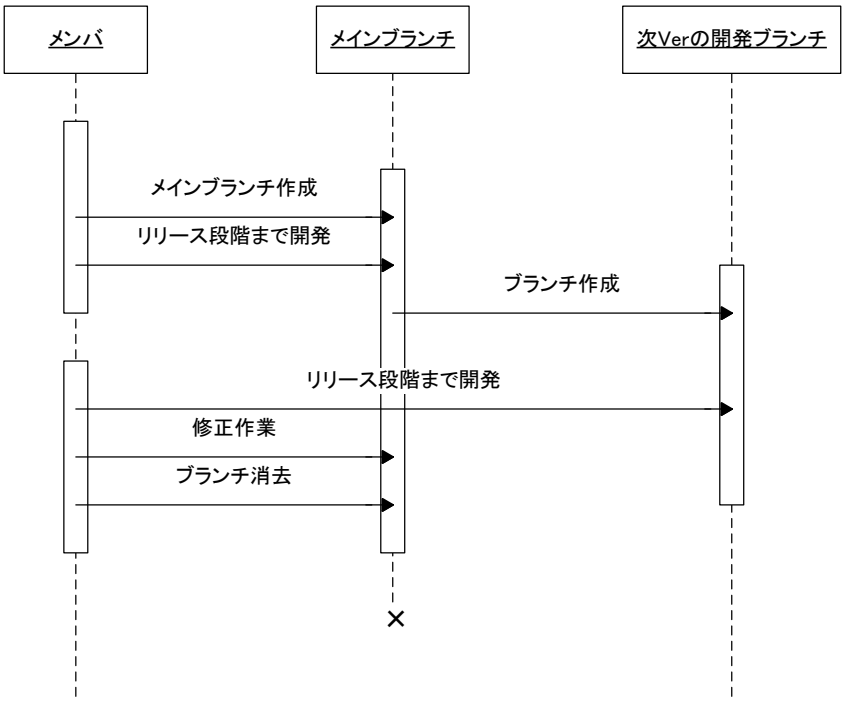

図 33LINE フローシーケンス図

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].

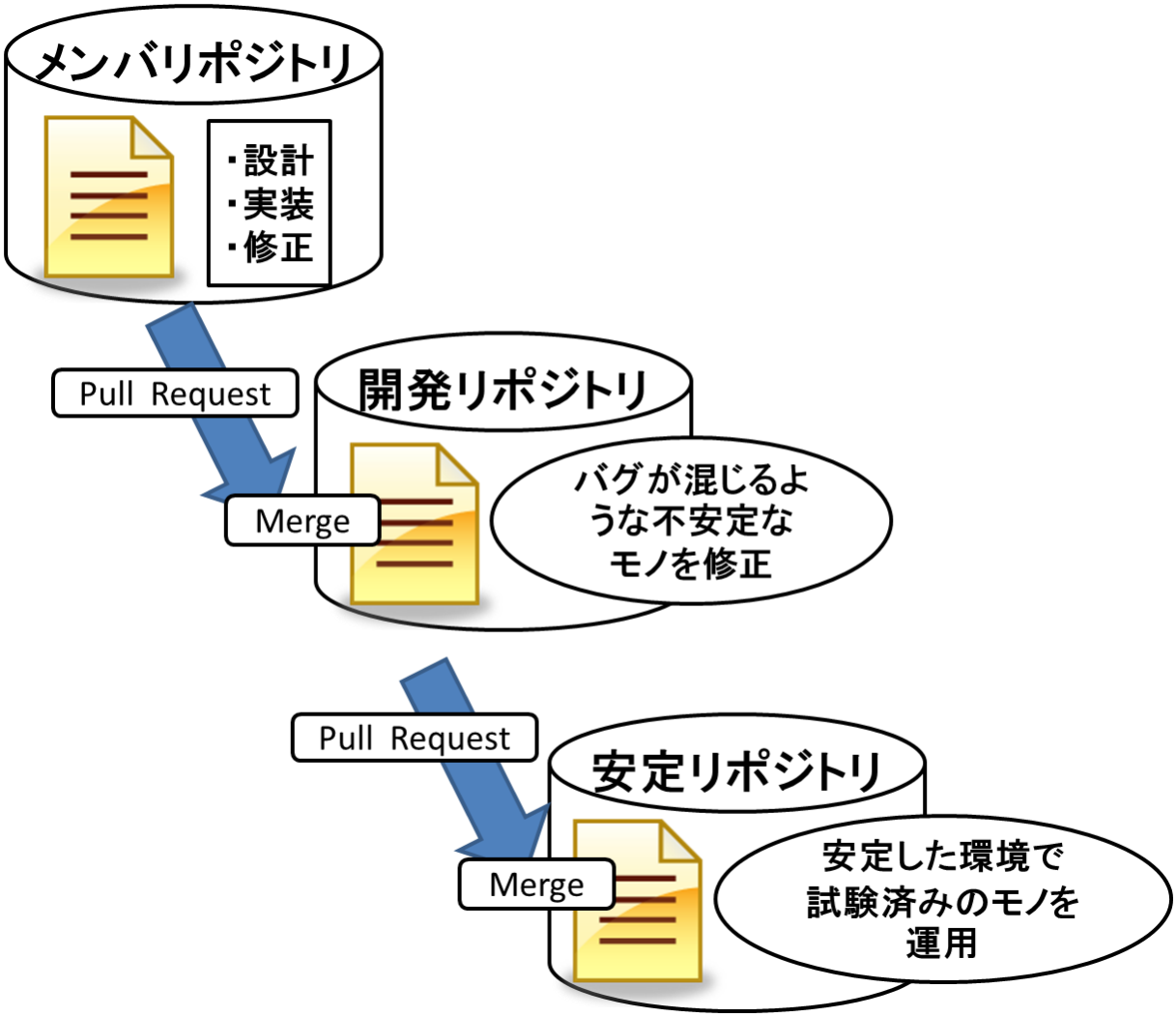

図 34 サイボウズフロー図

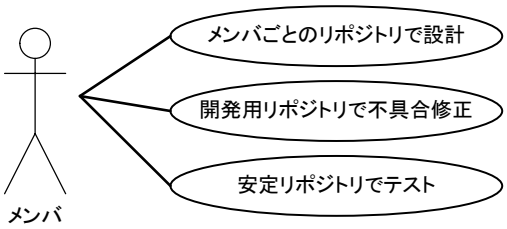

図 35 サイボウズフローユースケース図

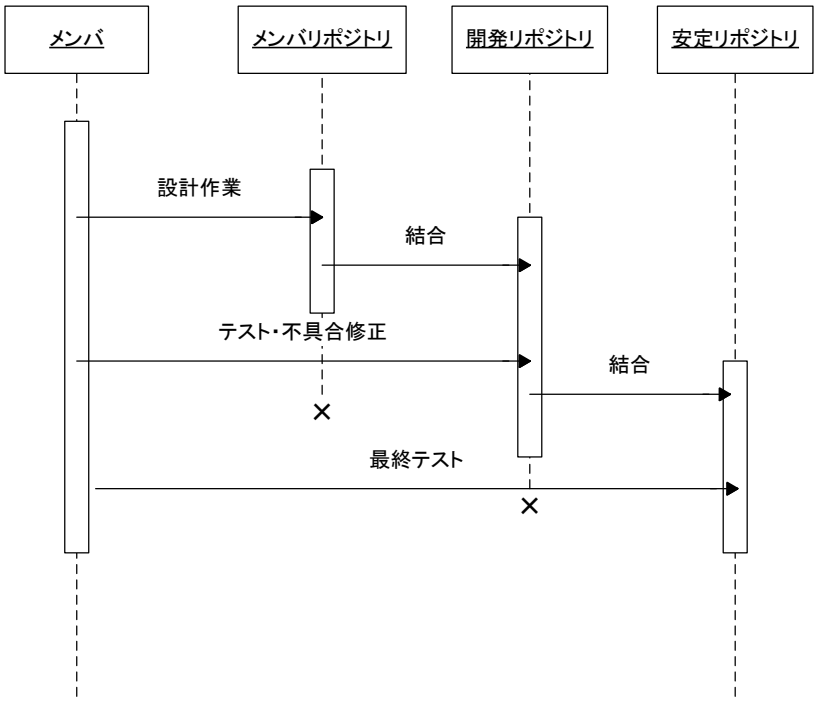

図 36 サイボウズフローシーケンス図

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].

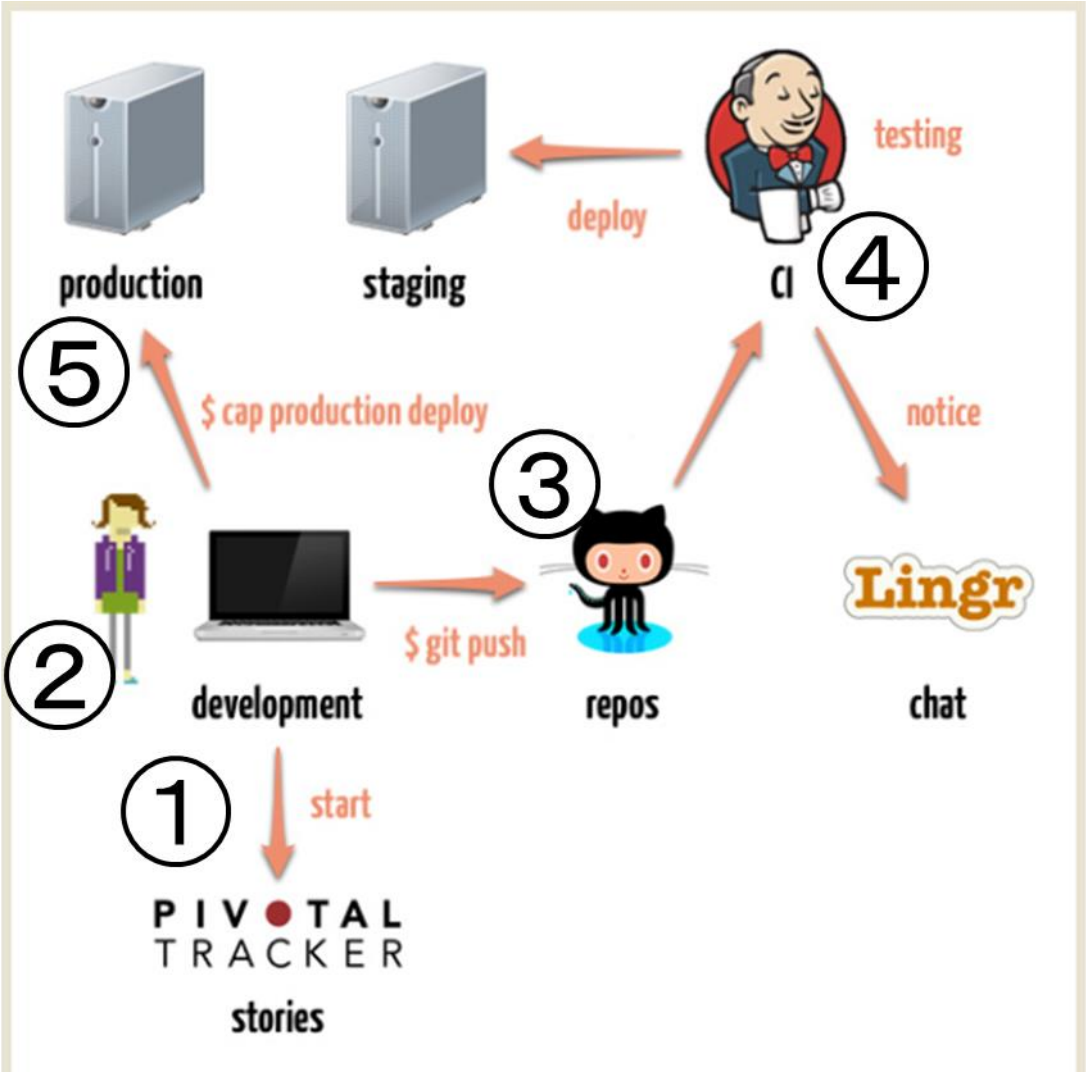

図 37 フィヨルドフロー図

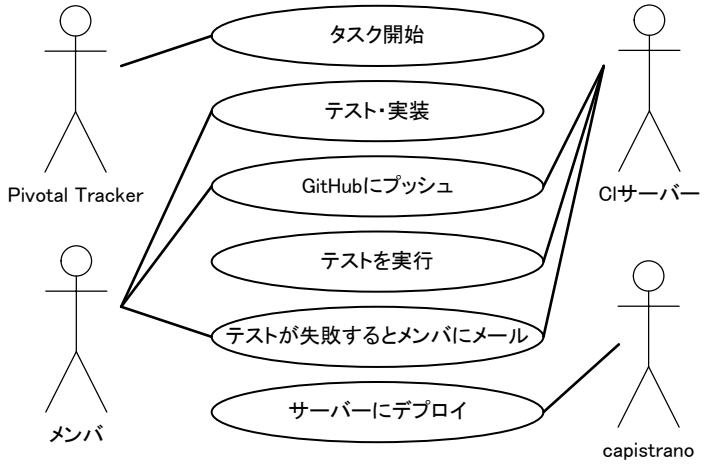

図 **38** フィヨルドフローユースケース図

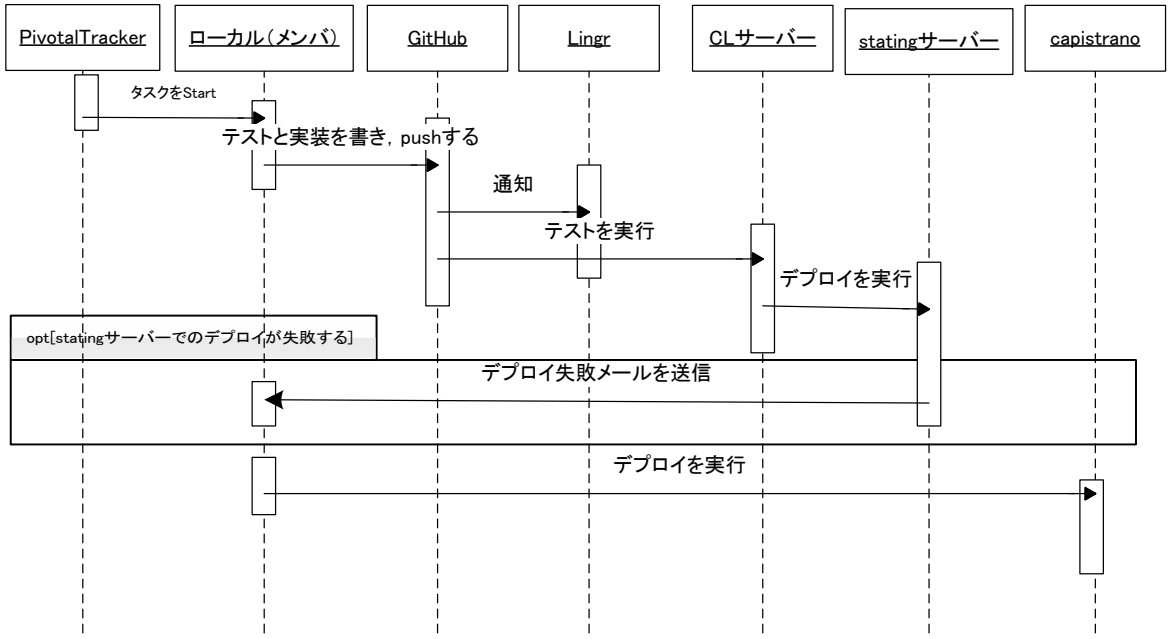

図 **39** フィヨルドフローシーケンス図

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．

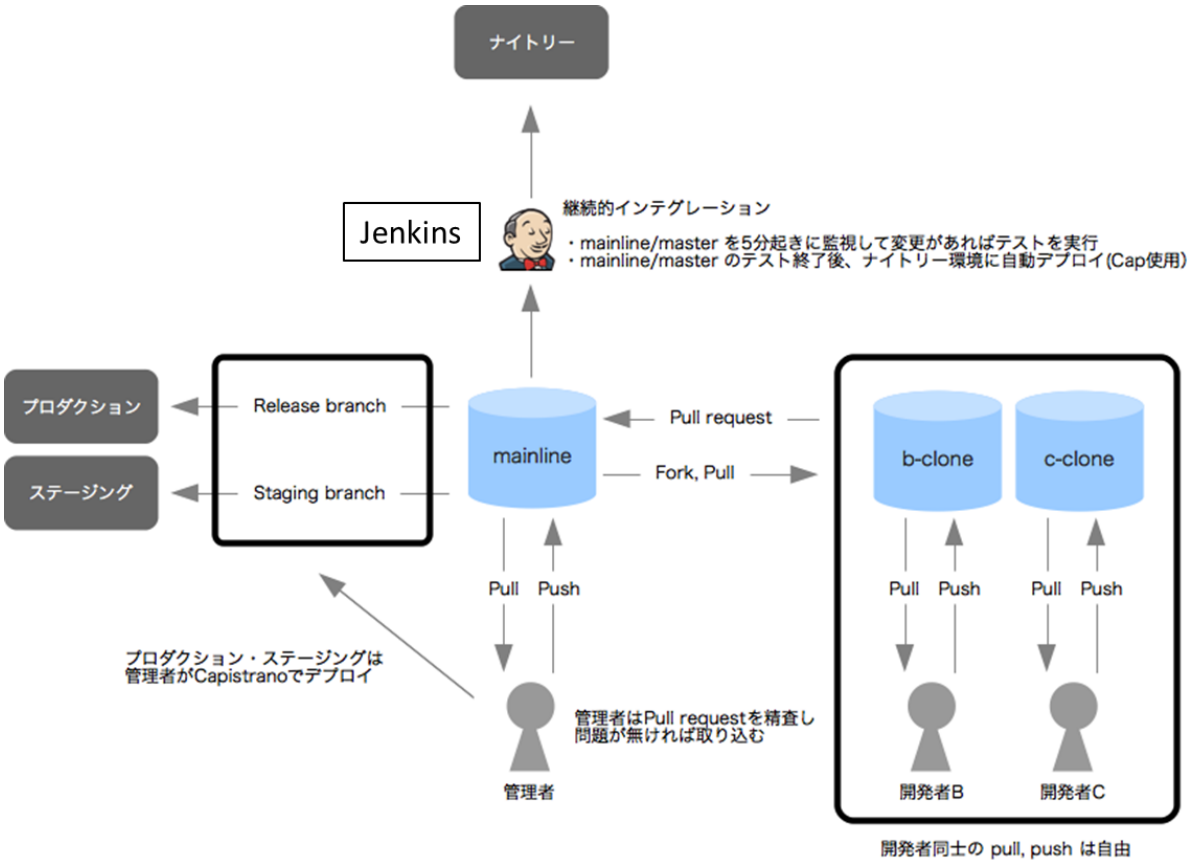

図 40 イストフロー図

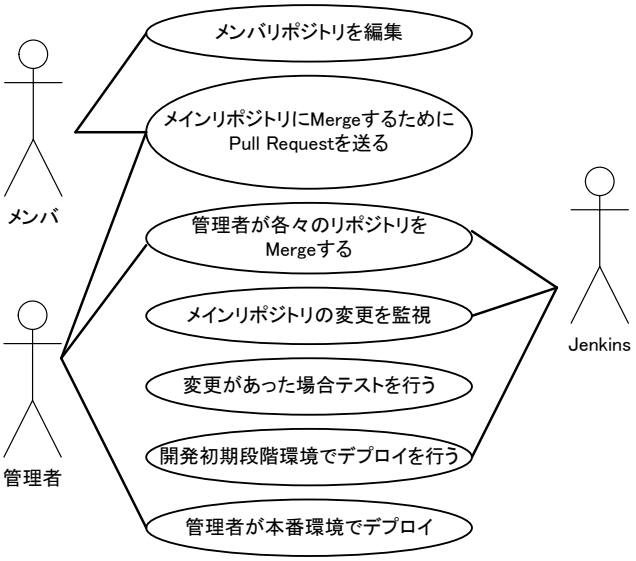

図 41 イストフローユースケース図

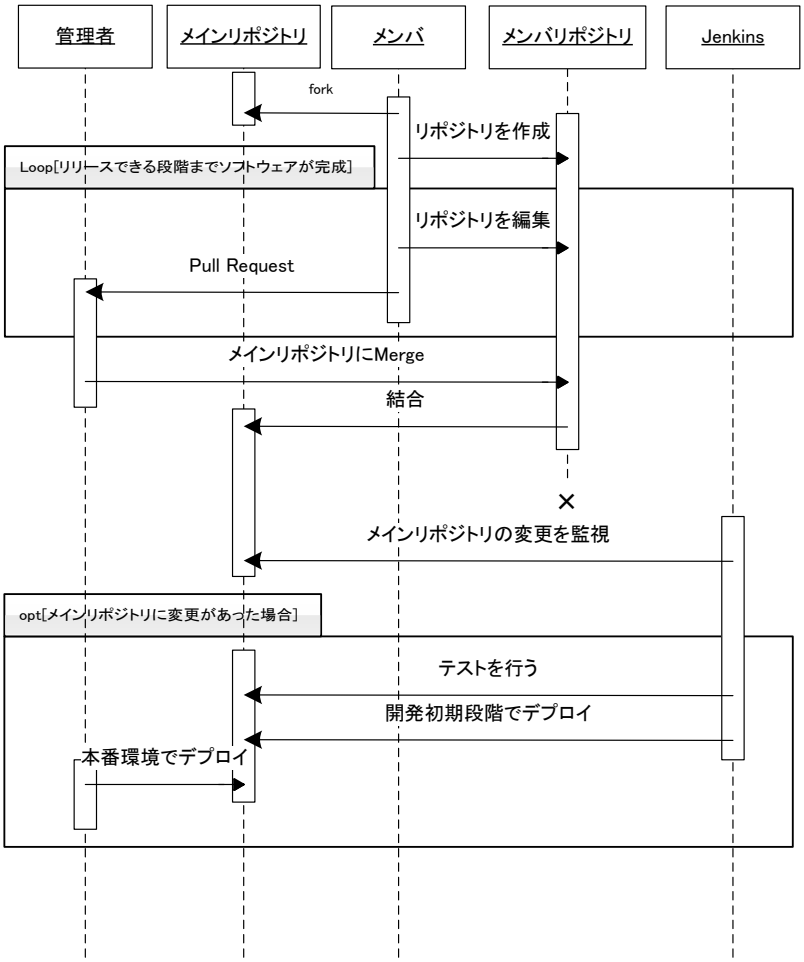

図 42 イストフローシーケンス図

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．

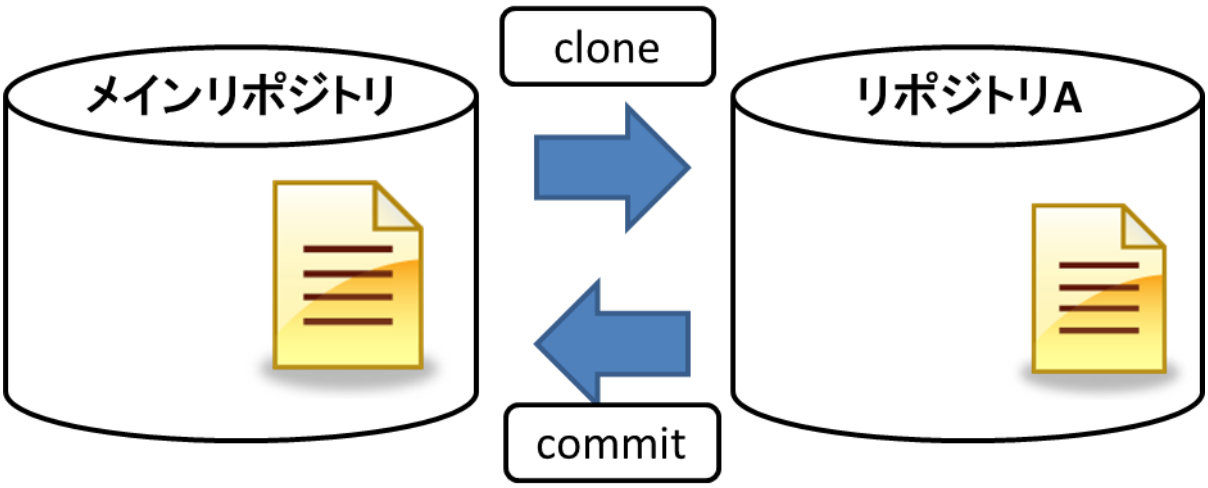

図 43 矢吹研フロー①図

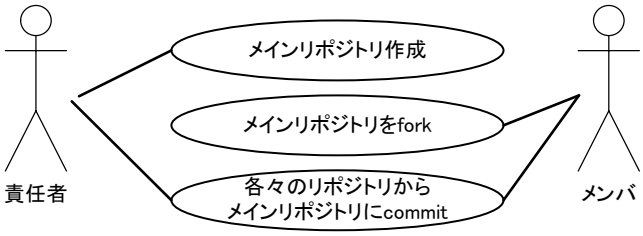

図 44 矢吹研フロー①ユースケース図

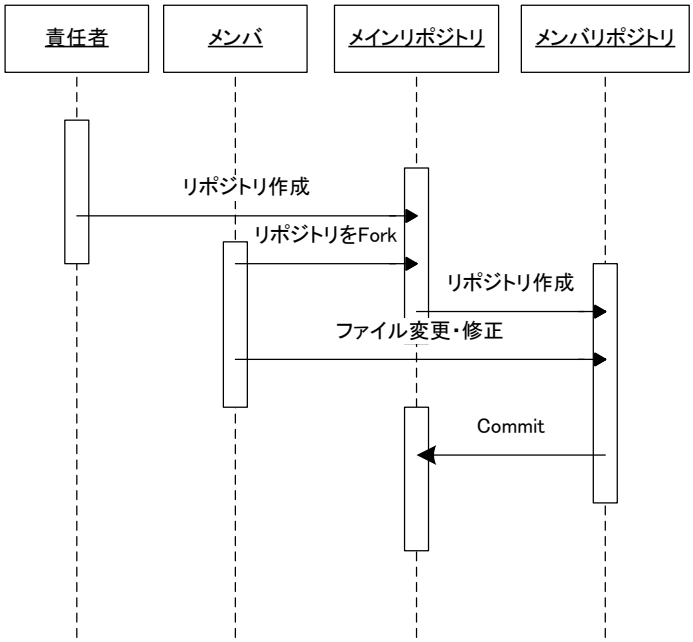

図 45 矢吹研フロー①シーケンス図

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．

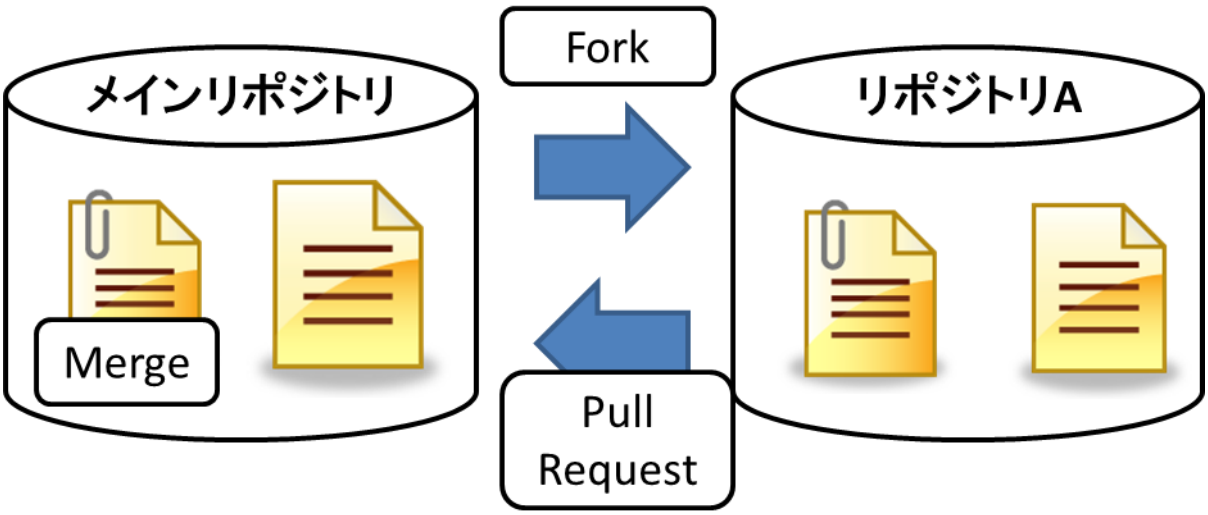

図 46 矢吹研フロー②図

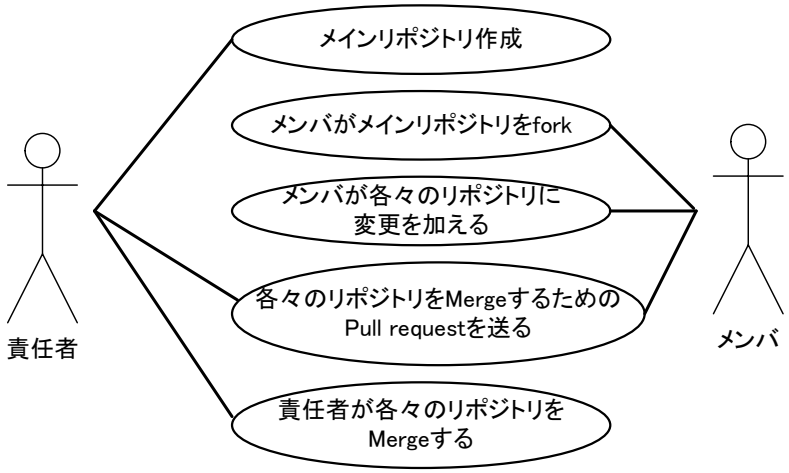

図 47 矢吹研フロー②ユースケース図

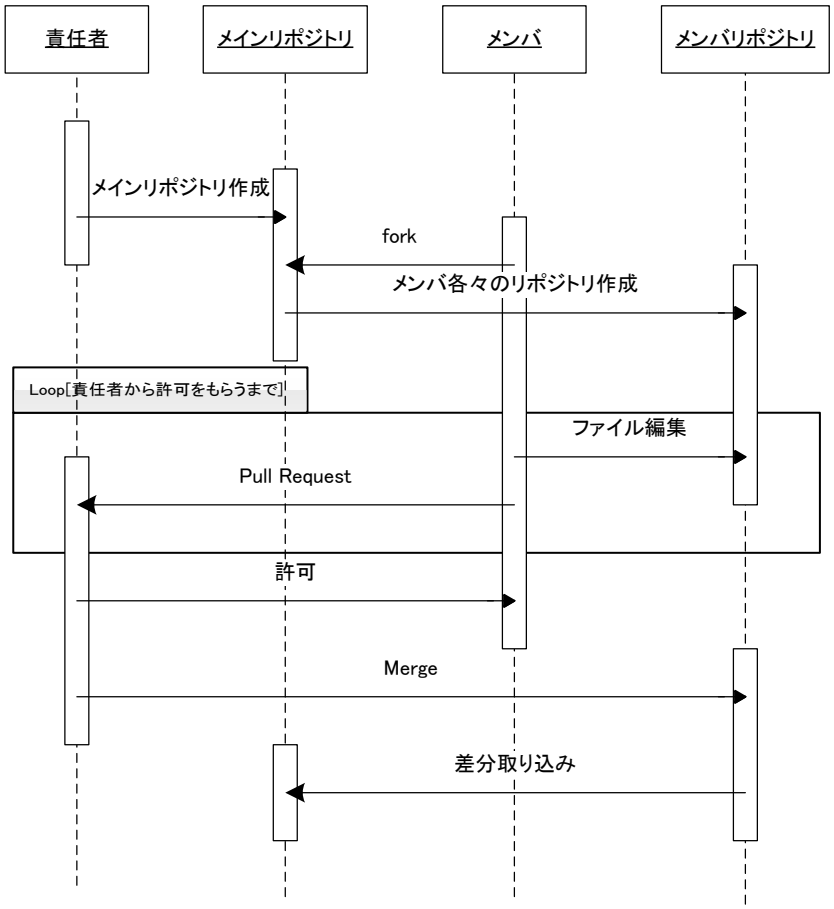

図 48 矢吹研フロー②シーケンス図

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．

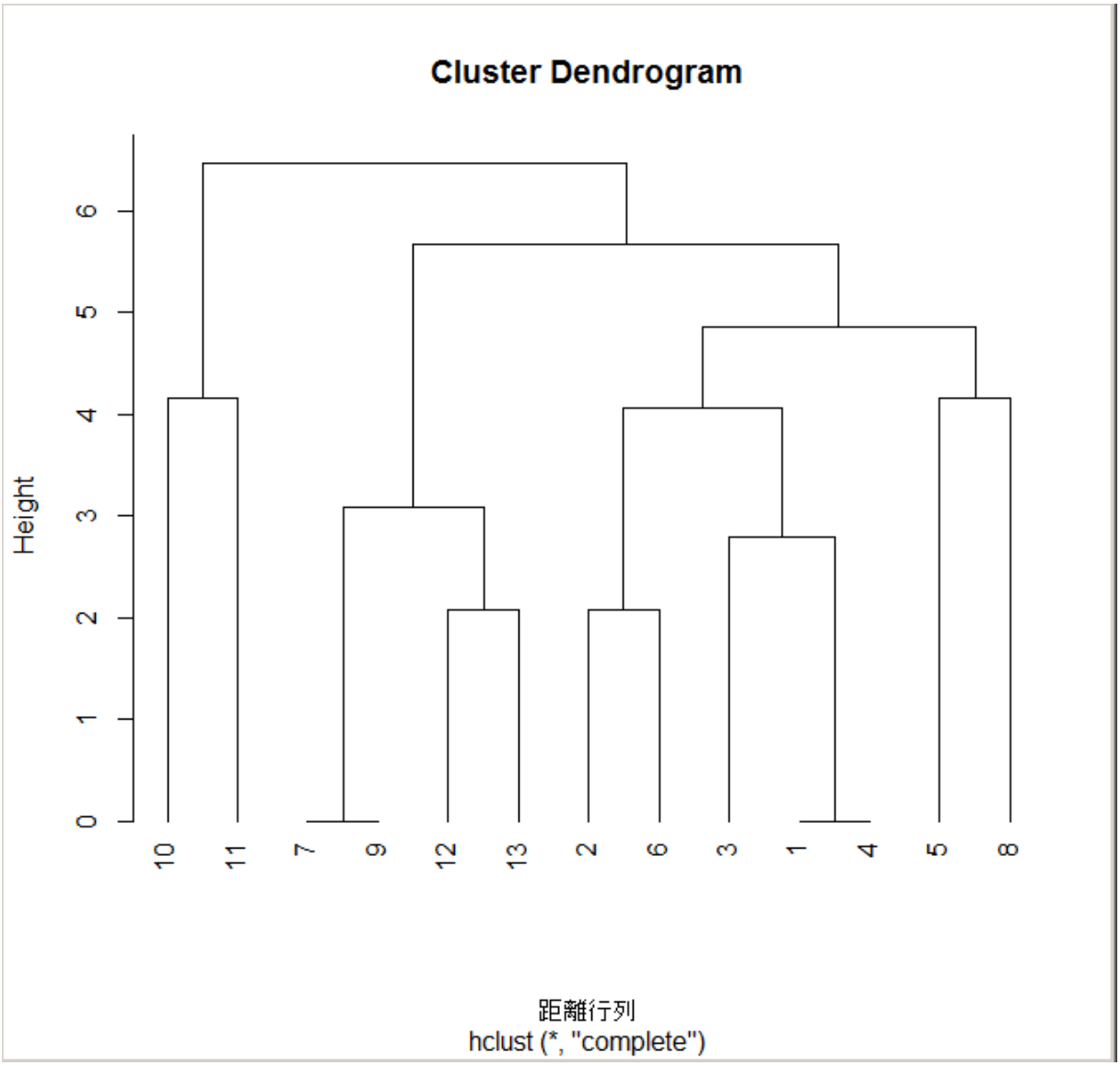

図 49 クラスター分析結果

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．