

Gitを導入するときにやったこと

目次

はじめに	1
スタート時の状況	1
ブランチ戦略	1
数ヶ月ごとにリリースがある。	1
チームメンバーが、それぞれ機能追加を担当する。	1
Git-Flowを採用。	2
ツールの選定	2
人にやらせる前に自分で実際にやってみる	2
ドキュメントを作る	2
アンチパターンの周知	2
チーム内のフォロー	3
困ったらいつでもヘルプをすると宣言する。	3
宣言通り、可能な限り手助けする。	3
安心感を演出する。	3
まとめ	4

GitとGitHubをプロジェクトに導入する際にやったことをメモする。

はじめに

プロジェクトの初期から参画し、
ある程度の舵取りができる機会に恵まれた。

その時に得た知見を記述する。

スタート時の状況

プロジェクト参画時の状況としては以下の通り。

- Githubの採用は決定しているが、運用方法の詳細は決まっていない。
- Javaのチームで、人数は8人程度。
 - 開発環境はこれから構築する。
 - 設計担当が2名で、残りの6名がそれぞればらばらに製造を担当する。
 - メンバーはGitの経験はほとんどない。

ブランチ戦略

まず最初にブランチ戦略を決定する。

Gitでよく使われるブランチ戦略は数種類ある。

Git-Flow、Github-Flow、GitLab-Flowなど。

今回の導入には、Git-Flowを採用した。考慮した点は以下の通り。

- 数ヶ月ごとにリリースがある。
- チームメンバーが、それぞれ機能追加を担当する。

数ヶ月ごとにリリースがある。

数か月ごとにリリースがある。

数か月ごとにリリースがあり、リリースされたものはエンドユーザに公開されて実践に即した動作テストに提供される。

次のリリースまでの間は、次々と機能追加が実施される。

つまり、リリース済みのブランチと作業中のブランチは明確に分離して管理する必要があった。

チームメンバーが、それぞれ機能追加を担当する。

チームメンバーが一つのシステムに対して、それぞれ個別に機能追加を担当する。

多機能な画面が複数ある複雑なシステムを複数の人数で分担して作業することで、短期間に構築する必要があった。

それぞれの作業はできるだけ独立性を高くしたかった。

Git-Flowを採用。

二つの要件を同時に満たすのがGit-Flowだった。

とくに、一度リリースしたものをmasterブランチに保存しておいて、別ブランチで実装作業を進めるという手順が明確に記述されているという点を評価した。

神経使って管理するべきブランチが2本になることでオペレーションがややこしくなるよう見えるが、ホットフィックスブランチやリリースブランチが完了する際にmasterとdevelopの両方に行うマージを一動作で実行するので手順の煩雑さは影響しない。

ツールの選定

使用するツールはSourceTreeを導入した。

コマンドラインにこだわる必要はない。

複数の人員で作業すると、必ずコンフリクトを起こす。

それはチームメンバーの技術レベルや習熟度などとは直接の関係はなく、だれでも起こしうる事故と解釈すべき。

その場合、修復作業をコマンドラインでやることは、不可能ではないがそれなりの学習コストがかかる。同じ低い習熟度からスタートした場合、やはりGUIがあったほうがとっかかりを得やすい。

人にやらせる前に自分で実際にやってみる

本格的な作業用リポジトリを作る前に練習用のリポジトリを作って、自分で一通りの操作を経験しておく。

リポジトリの内容はなんでもよく、例えば、ドキュメント用のリポジトリを一つ作成してみる。そのリポジトリを複数のフォルダにチェックアウトし、それぞれを別の担当者に割り振ったという体で、フィーチャーブランチやリリースブランチ、ホットフィックスブランチの開始から完了までの動作をファイル修正を交えて実験する。

事前に操作をすることで、チームメンバーにレクチャーする際の要点を洗い出せる。

ドキュメントを作る

自分でやってみる過程において、作業工程をテキストに記録しておく。そしてそれをもとに公開用の手順書をテキストにする。

作成したテキストは、閲覧・修正が容易な形で配布する。

Githubの採用は当初から決定していたので、テキストはasciidoc形式で記述することにした。

Githubでasciidoc形式のテキストを使うことで、コミット・プッシュするだけでファイルの公開まで可能となる。

そのままGitの管理下としてバージョン管理されるので、詳細な変更履歴はあえて作る必要はない。

アンチパターンの周知

Gitを使う上でどの戦略を使うにしても、必ずアンチパターンがある。

これをチームメンバーに周知することが必要である。

- masterのメンテは必ずすること。
Git-Flow戦略に限らず、masterブランチを正しくメンテナンスすることは、Gitを運用する上での命綱となる。
- 作業ブランチ同士のマージはしない。
どの戦略も、master（またはdevelop）ブランチから作業用のブランチを作るが、作業用ブランチ同士をマージするのは危険である。

チーム内のフォロー

テキストを用意するだけでは不十分なので、実際の操作をチームメンバーの目の前で、何度でも実演して見せる必要がある。

- 困ったらいつでもヘルプをすると宣言する。
- 宣言通り、可能な限り手助けする。
- 安心感を演出する。

困ったらいつでもヘルプをすると宣言する。

操作に困ったらいつでも声をかけるようにアナウンスをする。

チームメンバーにGitの経験が少なければ、Gitの概念そのものから学習する必要がある。
ところがこれが、けっこう難易度が高い。個人レベルの学習だと、目に見えない工数がどうしても必要となる。

一度のレクチャーやひとつのドキュメント公開ではきっと対応し切れない。

事前の想定できる範囲では、チームメンバーが実際に困ることのすべてをリストアップすることはできない。

困った時点で、その困った事柄を例題として、デモンストレーションをやって見せるのが、最終的にはもっとも対応工数が少なくなる。

宣言通り、可能な限り手助けする。

実際にヘルプの要請がかかった場合はできる限り優先して対応する。

こまごまとした疑問にも丁寧に対応して行くと、そのうち、チームメンバーは、何かやらかす前に相談してくるようになる。

そうなったらしめたものである。

安心感を演出する。

Git-Flow戦略の場合は、機能追加では master ブランチは手を付けない。

また、作業用である develop ブランチも、実作業は feature ブランチを分岐して行うので直接編集することはない。

その上、ローカル開発環境にもリポジトリを作るので、プッシュしない限りは中央リポジトリには送信されない。

最悪の場合でもローカルのチェックアウトを削除すれば、全ての作業はなかったことになる。

つまり本格的に破局的な失敗をする前に何段階もセーフティーネットがある。

実演の際には、それらの使い方を説明する。

とにかくはじめて使う場合は不安がつきまとうので、ぶっ壊しても大丈夫という安心感を演出する。

まとめ

- ブランチの運用を最初に決めておく
- コマンドラインにこだわらない。ツールでハードルが下がるならそっちで行く。
- 自分でやってみて、手順書を作る。
- 軌道に乗るまでは、積極的に手助けする。