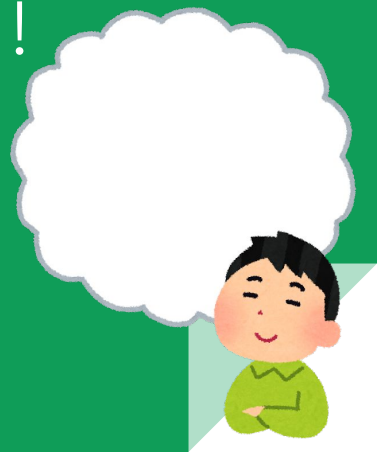
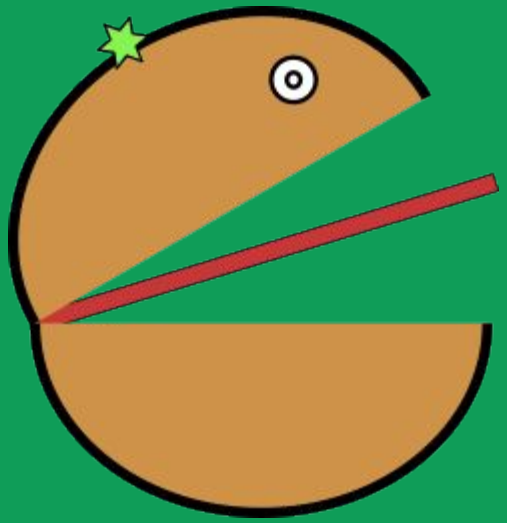
チーム開発で最低限意識しておくことを抜粋！

- 「ヤバい」と思ったら、すぐ報告！
- 開発上でのマニュアルを作ろう！そして従おう！

小林 栄太



自己紹介



名前: **小林 栄太 (通称: こばC)**

- S高 5期生 (1年生)

マネジメント系が得意で、チームでの創作活動が好きです

報告をしよう!

～「ヤバい」と思ったら、すぐ報告！～

団体の活動全体で最低限の基本となる考え方

Q. ちょっとまって！？
それは当たり前じゃない？

と思いましたね？
実は、当たり前ではないのです。

チームの場合、全員が同じ思想や技量,熱量を持っているとは限りません。
人によっては、「そもそも必要性を感じない」や「やり方が分からない」など
あるかもしれません。

隠すのが最悪の選択だ...

先ほどのスライドをふまえた上で再度考えてみましょう。🤔

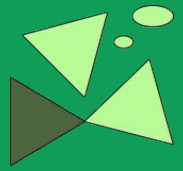
例えば、自分に権限がないはずのデータが見えてしまっているのを発見しました。「どう報告していいかわからないし...責任がとれない」と思いました。しかしよく考えてみましょう。

あなたの作業は、問題を解決することじゃない。ただ『ヤバいかも』と声を上げること。

そう考えると気が楽になりますね。

開発上でのマニュアルを作ろう！そして従おう！

団体の活動全体で最低限の基本となる考え方



～ マニュアルを作ろう！～

マニュアルを作らないと…

開発ノウハウが特定の人に依存し共有されない
「ブラックボックス化（属人化）」している状態は大変危険です。

- ノウハウの自然消滅
  - 担当者が離脱した際、その知識やスキルが失われ、開発が停滞する。
- 新規メンバーの参入障壁
  - 頼るべき情報がないため、新規メンバーは開発環境設定や進め方を自律的に習得できず、早期戦力化が困難になり、他メンバーの対応コスト増加を招く

～ マニュアルを作ろう！～

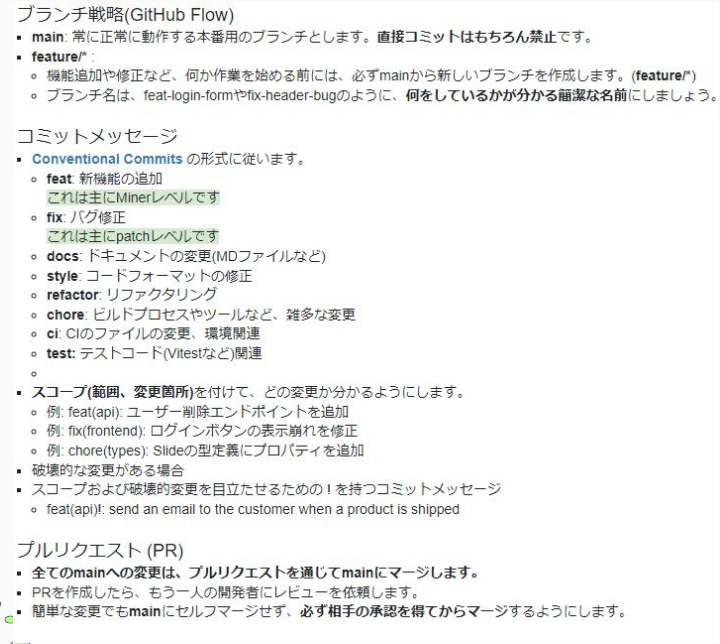
マニュアルを作ることで解決されること👍

開発ノウハウなどがまとまったマニュアルがあると、とても心強いです！

- 作業効率UP！そして一定の品質を確保
  - 頼るべき情報があるため、一貫した作業ができ一定の品質になります
- 担当者が離脱してもノウハウが残る
  - 人は去るものですが、ノウハウの継承をすればノウハウはなくなりません
    - そして、マニュアルがあれば、ノウハウを確実に継承できます。

～ マニュアルに従おう～

反対に従うほうの視点



ブランチ戦略(GitHub Flow)

- **main**: 常に正常に動作する本番用のブランチとします。**直接コミットはもちろん禁止**です。
- **feature/*** :
  - 機能追加や修正など、何か作業を始める前には、必ずmainから新しいブランチを作成します。(**feature/***)
  - ブランチ名は、feat-login-formやfix-header-bugのように、**何をしているかが分かる簡潔な名前**にしましょう。

コミットメッセージ

- Conventional Commits の形式に従います。
  - **feat**: 新機能の追加
    これは主にMinerレベルです
  - **fix**: バグ修正
    これは主にpatchレベルです
  - **docs**: ドキュメントの変更(MDファイルなど)
  - **style**: コードフォーマットの修正
  - **refactor**: リファクタリング
  - **chore**: ビルドプロセスやツールなど、雑多な変更
  - **ci**: CIのファイルの変更、環境関連
  - **test:** テストコード(Vitestなど)関連
  -
- **スコープ(範囲、変更箇所)**を付けて、どの変更か分かるようにします。
  - 例: feat(api): ユーザー削除エンドポイントを追加
  - 例: fix(frontend): ログインボタンの表示崩れを修正
  - 例: chore(types): Slideの型定義にプロパティを追加
- 破壊的な変更がある場合
- スコープおよび破壊的変更を目立たせるための ! を持つコミットメッセージ
  - feat(api)!: send an email to the customer when a product is shipped

プルリクエスト (PR)

- **全てのmainへの変更は、プルリクエストを通じてmainにマージします。**
- PRを作成したら、もう一人の開発者にレビューを依頼します。
- 簡単な変更でも**main**にセルフマージせず、**必ず相手の承認を得てからマージ**するようにします。

例として、実際に使用されているGitHubの運用ルールのマニュアルを用意しました。
これを参考に従うほうの視点を考えてみましょう。

→ Next

反対に従うほうの視点 (詳細)

ブランチ戦略(GitHub Flow)

- **main**: 常に正常に動作する本番用のブランチとします。**直接コミットはもちろん禁止**です。
- **feature/*** :
  - 機能追加や修正など、何か作業を始める前には、必ずmainから新しいブランチを作成します。(**feature/***)
  - ブランチ名は、feat-login-formやfix-header-bugのように、**何をしているかが分かる簡潔な名前**にしましょう。

コミットメッセージ

- Conventional Commits の形式に従います。
  - **feat**: 新機能の追加
    これは主にMinerレベルです
  - **fix**: バグ修正
    これは主にpatchレベルです
  - **docs**: ドキュメントの変更(MDファイルなど)
  - **style**: コードフォーマットの修正
  - **refactor**: リファクタリング
  - **chore**: ビルドプロセスやツールなど、雑多な変更
  - **ci**: CIのファイルの変更、環境関連
  - **test:** テストコード(Vitestなど)関連
  -
- **スコープ(範囲、変更箇所)**を付けて、どの変更か分かるようにします。
  - 例: feat(api): ユーザー削除エンドポイントを追加
  - 例: fix(frontend): ログインボタンの表示崩れを修正
  - 例: chore(types): Slideの型定義にプロパティを追加
- 破壊的な変更がある場合
- スコープおよび破壊的変更を目立たせるための ! を持つコミットメッセージ
  - feat(api)!: send an email to the customer when a product is shipped

プルリクエスト (PR)

- **全てのmainへの変更は、プルリクエストを通じてmainにマージします。**
- PRを作成したら、もう一人の開発者にレビューを依頼します。
- 簡単な変更でも**main**にセルフマージせず、**必ず相手の承認を得てからマージ**するようにします。

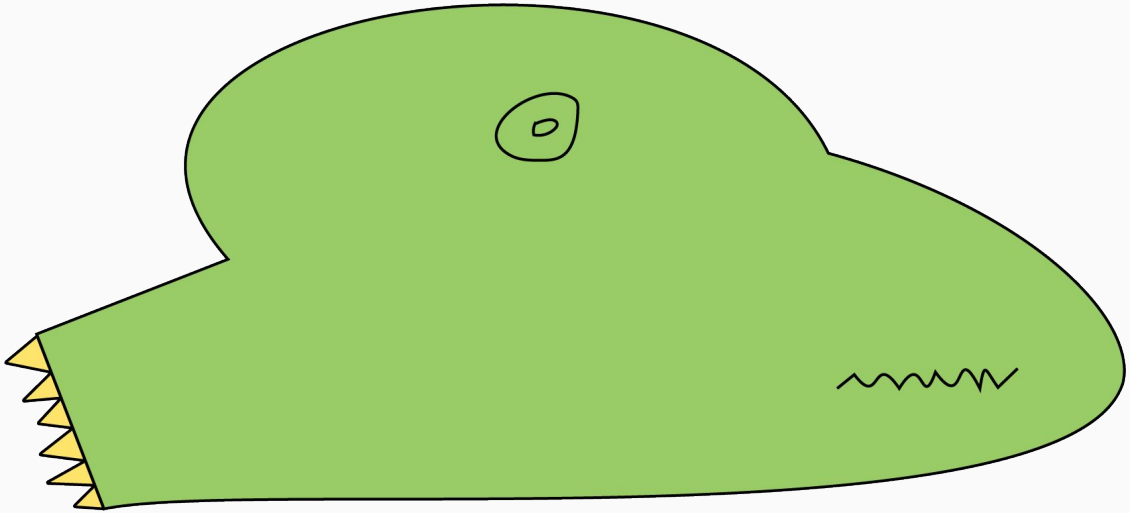
- ブランチ戦略
  迷子にならず進められる。(**具体的な例がある**)
  - ルールが決まっているので、命名規則やブランチ戦略で毎回悩む必要がありません。
- コミットメッセージ (**スムーズになる**)
  - コードを見なくても「何についての」「どんな種類の」変更なのかがレビュアーに一目で伝わります。
- プルリクエスト（**作業の進め方が明確**）
  - 個人の判断で迷う場面をなくすのが、このルールです。「必ずレビュー依頼を出す」と決まっていることで、作業の進め方が明確です。

まとめ

マニュアルに従うことは、一見すると、面倒と感じるかもしれません。
しかし、ルールはチーム開発には不可欠です。結果的に自分自身の助けにもなります。

また、問題に気づいた時、あなたの役割は問題を解決することではありません。「ヤバいかも」とチームに知らせる、ただそれだけで十分です。

ご清聴ありがとうございました！



問い合わせ窓口
お問い合わせ

作成者: 小林 栄太