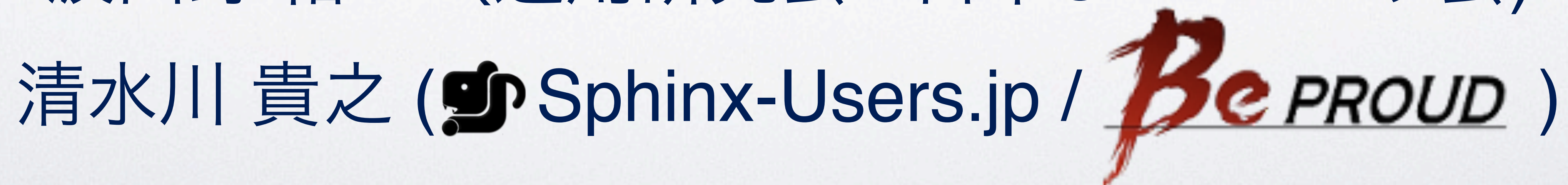
運用ドキュメントの共有と変更管理

運用ドキュメント2011 〜手軽にスピーディに継続的に保守するためのツール入門〜 Part-3

波田野 裕一（運用研究会 / 日本UNIXユーザ会)



2011-11-30 InternetWeek 2011

セッション概要 (Part-3)

3つの根幹

Why

なぜ書く?

Worth

Part-1

書く価値?

What

何を書く?

記述のHow

情報の構造化

Who

再利用

誰視点で書く?

Whom

Part-2

誰宛に書く?

Which

どれから書く?

管理のHow

ドキュメント管理

Where

どこに書く?

When

Part-3

How Much

Part-4 summary

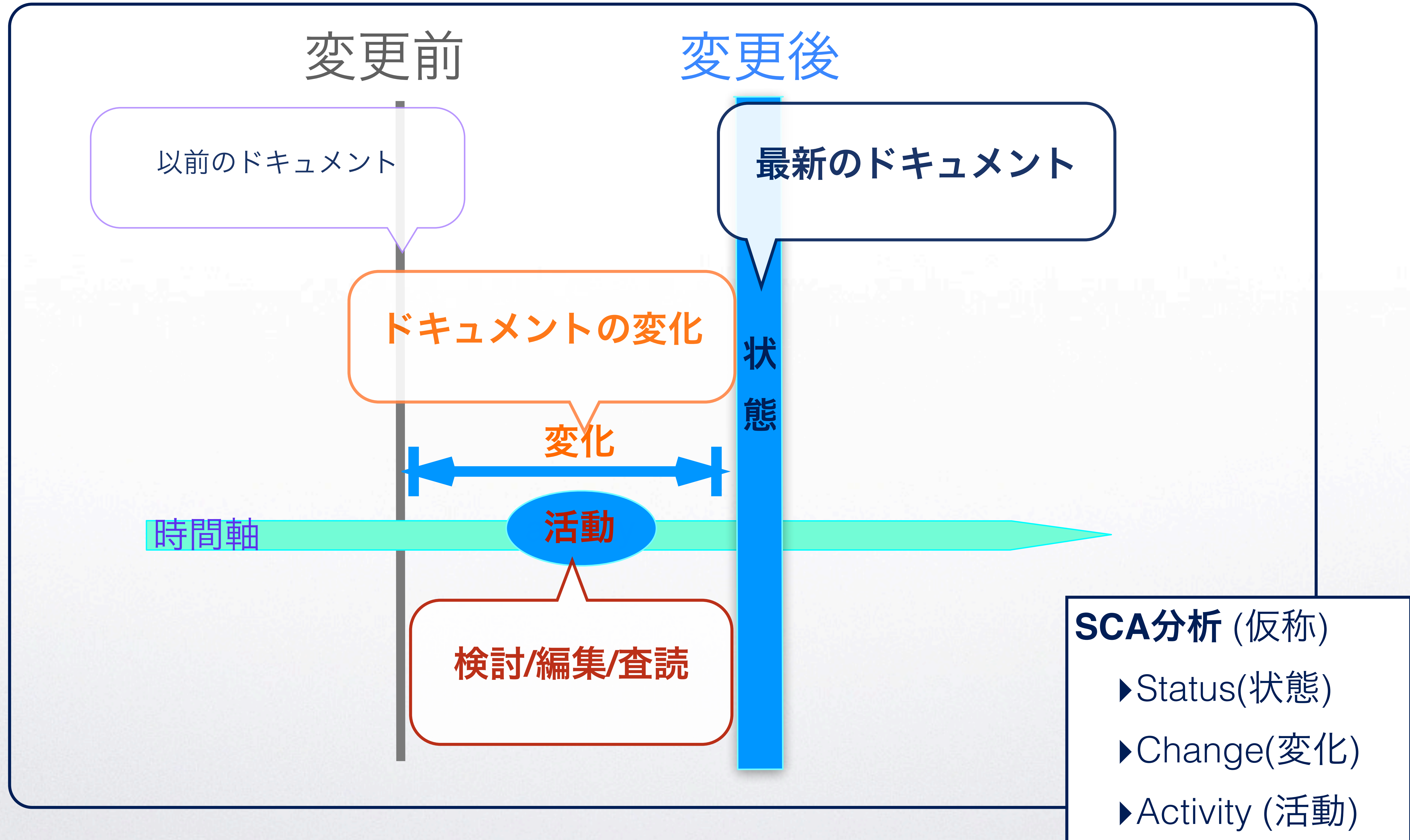
ドキュメント管理とは

ドキュメント管理

3つの「ドキュメント管理」がある

- ドキュメントの**状態**の管理
- ドキュメントの**変化**の管理
- ドキュメントに関する**活動**の管理

ドキュメント管理対象の全体像



SCA分析 (仮称)

- Status(状態)
- Change(変化)
- Activity (活動)

ドキュメント管理の対象

状態

- 最新版はどれか。どれを編集すればいいのか。(書き手)
- 最新版はどれか。どれを読めばいいのか。(読み手)

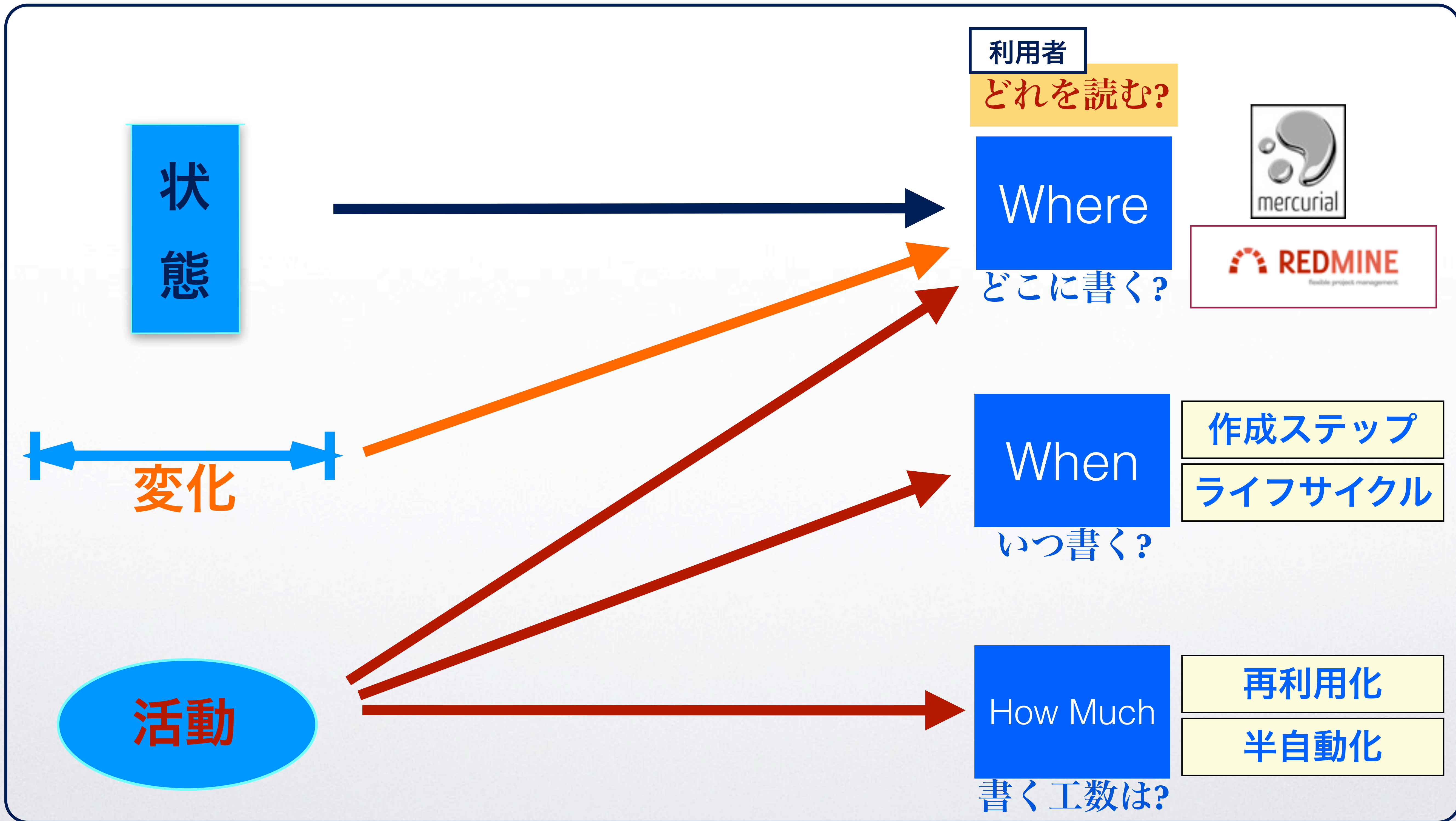
変化

- どこが変わったのか?

活動

- 何を検討し、それによってどこを変えたのか?

ドキュメントの管理 の課題

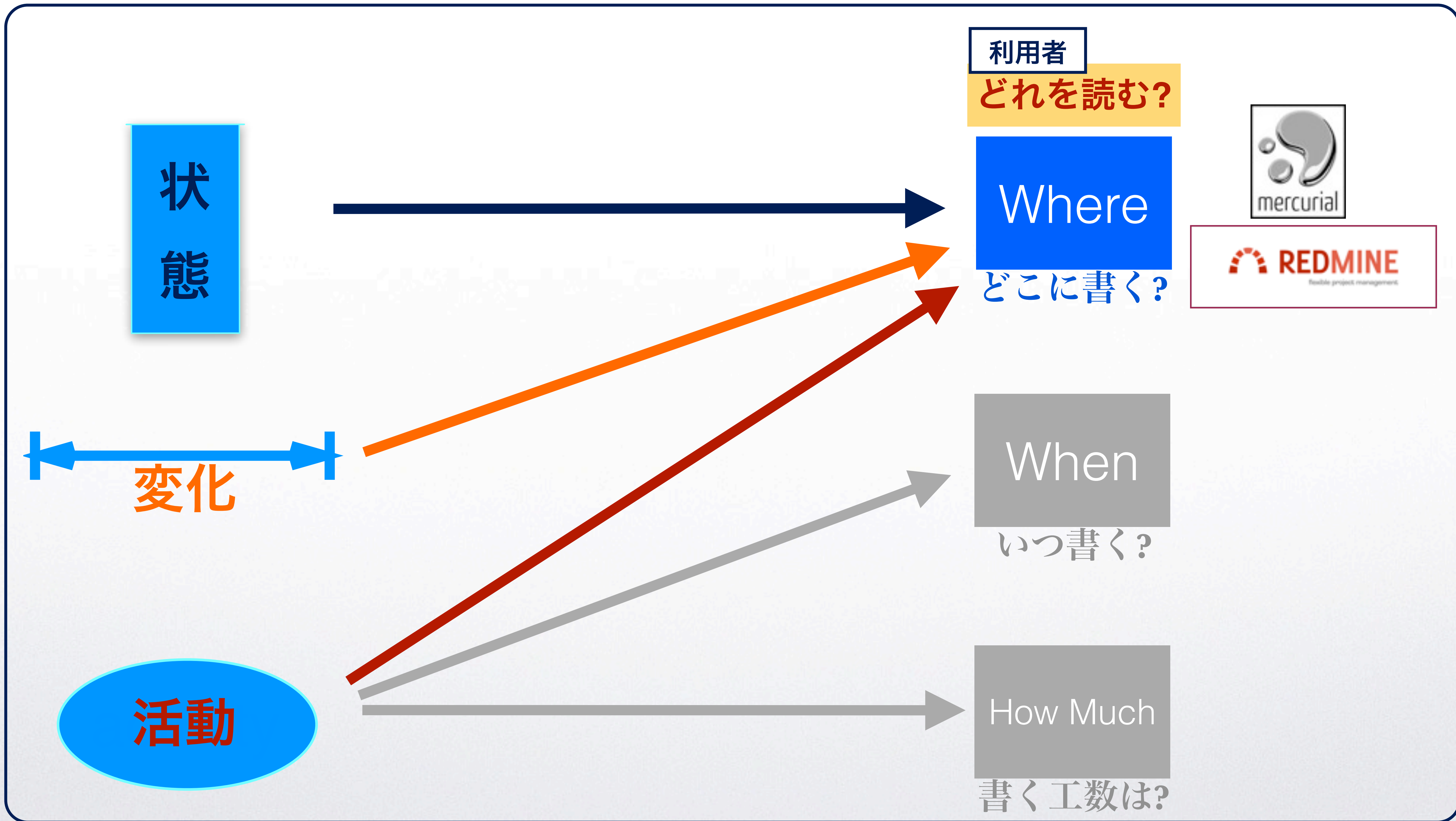


運用ドキュメントの作業/共有場所

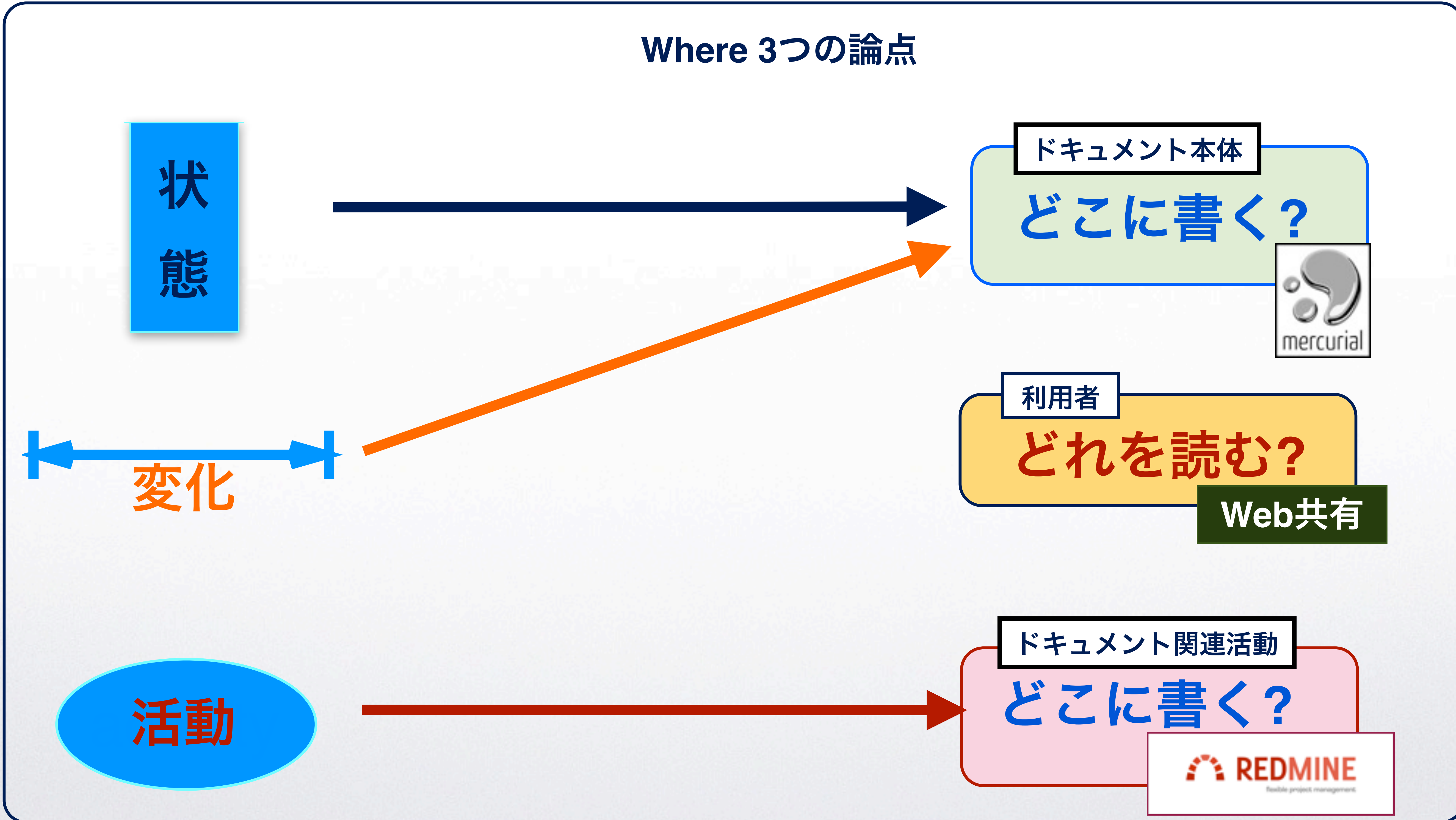
Where **どこに書けばいいのかわからない**

利用者 **どのドキュメントを読めばいいのかわからない**

ドキュメントの管理 の課題



ドキュメント管理の作業/共有場所



ドキュメント管理の作業場所 (状態 / 変化)

ドキュメント本体

どこに書く?

- **ドキュメント本体は VCS に書く**
  - VCS: バージョン管理システム
  - 構成管理、版管理を行なう。(バックアップにもなる。)
  - ドキュメント作業環境と公開ドキュメントを同期する。
  - 本セッションでは、VCSの例として **Mercurial** を紹介する。

状態

- **最新版はどれか。どれを編集すればいいのか。(書き手)**
- 最新版はどれか。どれを読めばいいのか。(読み手)

変化

- **どこが変わったのか?**

ドキュメント管理の共有場所

利用者

どれを読む?

「真正のものがどこにあるか」が最重要

- **位置が勝手に移動しない。(Permanent Position)**
- **内容が不用意に変わらない。(Read Only)**
- **外部から再利用しやすい。(Reusable)**

Web共有

シングルソース
マルチアウトプット

CUI環境では man形式
提出はPDF形式 / reSTテキスト形式
将来的には ePub形式 で配布など

状態

- 最新版はどれか。どれを編集すればいいのか。(書き手)
- **最新版はどれか。どれを読めばいいのか。(読み手)**

ドキュメント管理の作業場所 (活動)

ドキュメント関連活動

どこに書く?

‣ ドキュメント関連の活動は BTS に書く

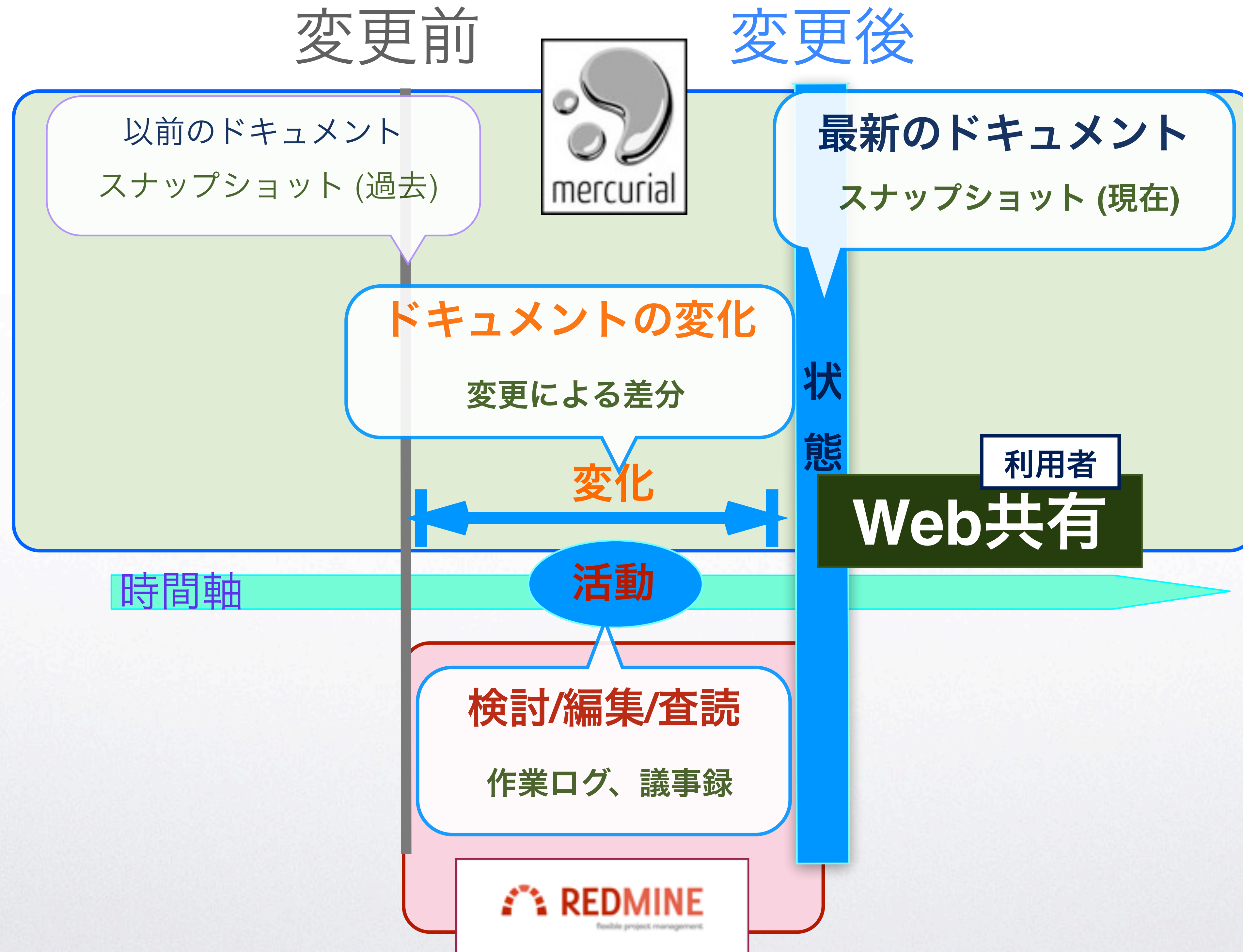
- BTS: バグトラッキングシステム (チケットシステムとも言う)。
- ドキュメントに関連するやりとりや経緯をBTSに残すことにより、**ドキュメント本体に残りにくい、Why (なぜそうしたのか?)が失われない**ようにする。
- 本セッションでは、BTSの例として **Redmine** を紹介する。

活動

- **何を検討し、それによってどこを変えたのか?**

リポジトリ連携機能により
ドキュメント本体と関連活動を紐付け

ドキュメント管理の作業場所



分散バージョン管理 (Mercurial)

Where **どこに書けばいいのかわからない**

運用ドキュメントの作業場所

ドキュメント本体

バージョン管理の歴史

1. **ディレクトリ時代** (単なるコピー)
2. **ZIP 圧縮時代** (アーカイブで管理)
3. **RCS時代** (ローカルでバージョン管理)
4. **CVS時代** (ネットワーク越えで管理)
5. **Subversion時代** (ファイル管理ではなく版管理へ)

そして...

「分散バージョン管理(DVCS)」の時代が来た!

分散バージョン管理 三国時代

- Mercurial
- Git
- Bazaar

オフラインで全ての機能が使える

- ネットワークアンリーチャブルな状態でのコミットが可能。
- 全ての履歴を各自で持つのでオフライン & 高速な履歴参照が可能。

分散VCS 何が違う

マージが楽

- 非分散VCSは「最新の状態」という面でマージを行なう。
- 分散VCSは、「チェンジセット(差分)」の積み重ねでマージを行なう。

分散VCS 何が違う

ブランチ管理がしやすい

- Mercurial / Git はブランチをディレクトリに展開しない
- Subversion は「わかりやすさ」に対する反論
- Bazaar は Subversion と同じモデル

分散VCS 何が違う

頻繁なバックアップが不要

- アクティブな時は複数人が最新のリポジトリを持つ。
- 非アクティブなときは月1回のバックアップで十分。

分散VCS 何が違う

その他

- **Redmineなどのチケットシステムからのリポジトリ連携の場合にパフォーマンス影響が無い。**
- **巨大ツリーの枝の部分だけチェックアウトすることはできない。**(リポジトリ全てを clone してしまう)

分散VCS 何が嬉しい

ユースケース

1. とりあえず手元にいくつかのファイルやディレクトリを作った
2. 変更履歴を取りたくなった(バックアップしたくなった)のでinit
3. ファイルへのちょっとした変更の理由をコメントに書いてコミット
4. 他の人や他の環境で同時に使いたくなったのでclone
5. 複数の環境で更新したのでpush/pull/merge

分散VCS 何が嬉しい

分散VCS のここが嬉しい

- コミットが軽いMe (ネットワーク負荷がない)ので気分が軽い
- 管理者に頼らずに手元のファイルをとりあえず管理し始められて気分が軽い
- コミットが軽い(ネットワーク負荷がない)ので気分が軽い
- 論理的にコンフリクトしにくいので気分が軽い
- 目的別にリポジトリのcloneを作成し、アクセス権を個別に設定出来る。これによって、開発用、リリース用、運用変更用と用意出来る。

Mercurial なにが嬉しい

Mercurial のここが嬉しい

- ``hg serve`` コマンドでサーバが起動し、Webで履歴を参照したり、他の端末からのコミットを受け付けたり出来る
- 過去のVCSツールの反省を活かし、コマンドやサーバー設定などが全体的により使いやすくなっている
- CVS に慣れている人に違和感が少ないコマンド体系

Mercurial コマンド例 I

リポジトリの初期化

- **hg init**

リポジトリの複製

- **hg clone** [clone元] [clone先]

変更をリポジトリに反映

- **hg commit**

別リポジトリとの差分の送受信

- **hg push / pull** [送受信先]

作業領域の内容更新 (pull後に実行を求められる)

- **hg update**

Mercurial コマンド例 2

ファイルのリポジトリへの追加

- **hg add** [対象ファイル]

ファイルのリポジトリからの削除 (対象ファイルは消える)

- **hg remove** [対象ファイル]

ファイルのリポジトリからの削除 (対象ファイルは消ない)

- **hg forget** [対象ファイル]

ファイルのリポジトリへのファイル追加・不在ファイルの削除

- **hg addremove**

Mercurial コマンド例 3

作業領域のファイル状況の表示

- **hg status**

親リビジョンの状態でファイルを復旧 (変更を戻したい時など)

- **hg revert** [対象ファイル]

ブランチのマージ

- **hg merge**

リポジトリ内容をローカルウェブに公開する (default: http://localhost:8000/)

- **hg server**

参考文献 (Mercurial)

- 『分散リポジトリ型』時代のソフトウェア構成管理
  http://www.lares.dti.ne.jp/~foozy/fujiguruma/scm/scmbc-201111/speech.html
- Mercurial 入門セッション資料 - SCM Boot Camp #2
  http://troter.jp/scmbc-201111-mercurial-introsession/
- How to install mercurial
  http://troter.jp/how-to-install-mercurial/
- Mercurial チートシート
  http://troter.jp/mercurial-cheatsheet/

チケットシステム (Redmine)

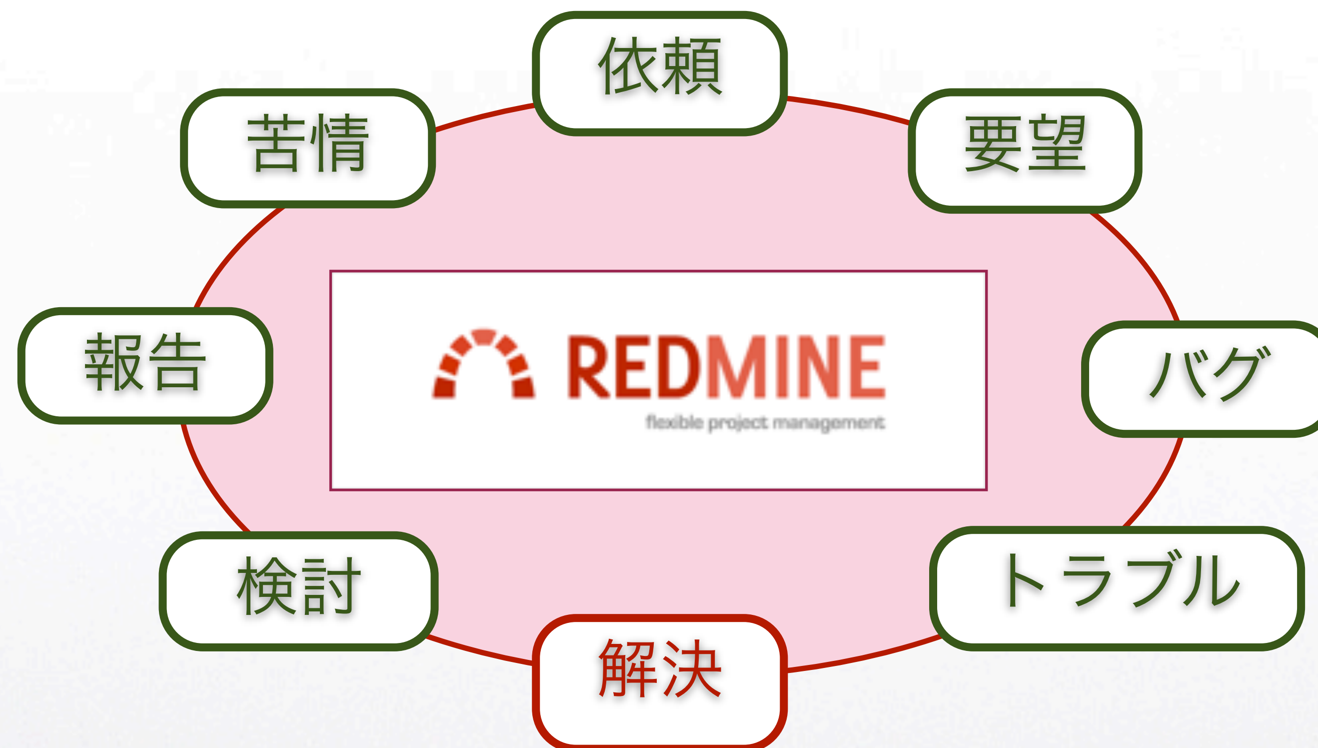
Where **どこに書けばいいのかわからない**

運用ドキュメントの作業場所

ドキュメント関連活動

Redmine とは何か

- **BTS: バグトラッキングシステム**
  - チケットシステム / 課題管理システムとも言う



多機能だが、特に活動の記録に長けている

Redmine の特徴

- Webベースのプロジェクト(タスク)管理アプリケーション
- 複数の利用者が並行して閲覧や更新の作業ができる。
- タスクを多様な方法で絞り込んだり、依存関係を設定することができる。
- タスクの親子関係を設定することでWBS風にツリー構造でタスクを管理することができる。
- タスクについて、カレンダー表示やガントチャート表示ができる。
- タスク工数の記入、合計表示ができる。
- バージョン管理システムとチケットの連動が可能である。
- 簡易WiKi / フォーラムなどのコミュニケーション機能もある。
- REST API により、外部システム/スマートフォンアプリとの連携も可能になってきた。

チケット駆動開発

ここ数年、RedmineなどBTSを活用した「チケット駆動開発」という考え方が普及してきている。

- **No ticket, No commit** 「チケットの無いコミットはしてはいけない」という考え方。
- 作業全てをチケットで管理する。(チケットシステム中心の業務)

- 作業状況の見える化 (進捗状況、負荷状況)
- コード変更の見える化 (コミット単位での変化、関連する議論や経緯)
- 作業にリズムが生まれた (Open -> Assign -> Fix -> Close)
- 優先順位の見える化により、突発事項への柔軟な対応が可能に

チケット駆動ドキュメント管理

- **No ticket, No commit** 「チケットの無い作業はしてはいけない」
- 作業全てをチケットで管理する。(チケットシステム中心の業務)

運用作業やドキュメント管理も本質的には同じ。

- チケットの無い「闇作業」や「闇変更」は百害あって一利も無い。
- 時間とともにその経緯が誰にもわからなくなってしまう。
- ドキュメント管理については、下準備をチケット上でやることにより、執筆/レビュー前に関係者からコメントが得られる可能性がある。また、後日類似の検討があった場合に、ゼロからの検討が必要なくなり短期化できる可能性も出てくる。

チケット駆動うれしいところ

どこに書く?
Where

- チケットにすることで、やることが明確になる、頭から忘れてもかまわないという安心感がでてくるため、**作業に集中しやすい。**
- チケットにまとめておくことで、**過去のやりとりメールをあさらなくて良い**
- 親子チケットにより、大きいタスクを細分化することで、**目先何をやればいいのかが見えてくる**ため、進めやすくなる。
- 質の良いチケットは経緯を残しているため、**ドキュメント本体に残りにくいWhy(なぜそうしたのか?)が残り**やすい。
- チケットにしておけば、気付いた人が進めてくれる可能性が出てくる。
- 優先順位を付けやすくなる。(期限 vs. 優先度)
- 工数を都度入れておけば、**実測に基づく実績として有効**なデータとなる。
- **作業にリズム**が出てくる。
- 蓄積により、チケット全体が一次ナレッジデータベースの価値を持つようになる。

チケット駆動で主に使う機能

どこに書く?
Where

- 課題管理 (チケットトラッキング)
  - 題名、説明 (まとめ)、コメント
  - ステータス、優先度、担当者
  - 期限、予定工数、進捗率
- ロードマップ、ガントチャート、カレンダー
- リポジトリ連携
- 活動、時間トラッキング(工数管理)
- ニュース、通知機能 (ウォッチ、メール)
- 外部連携機能 (REST API)

参考文献 (Redmine)

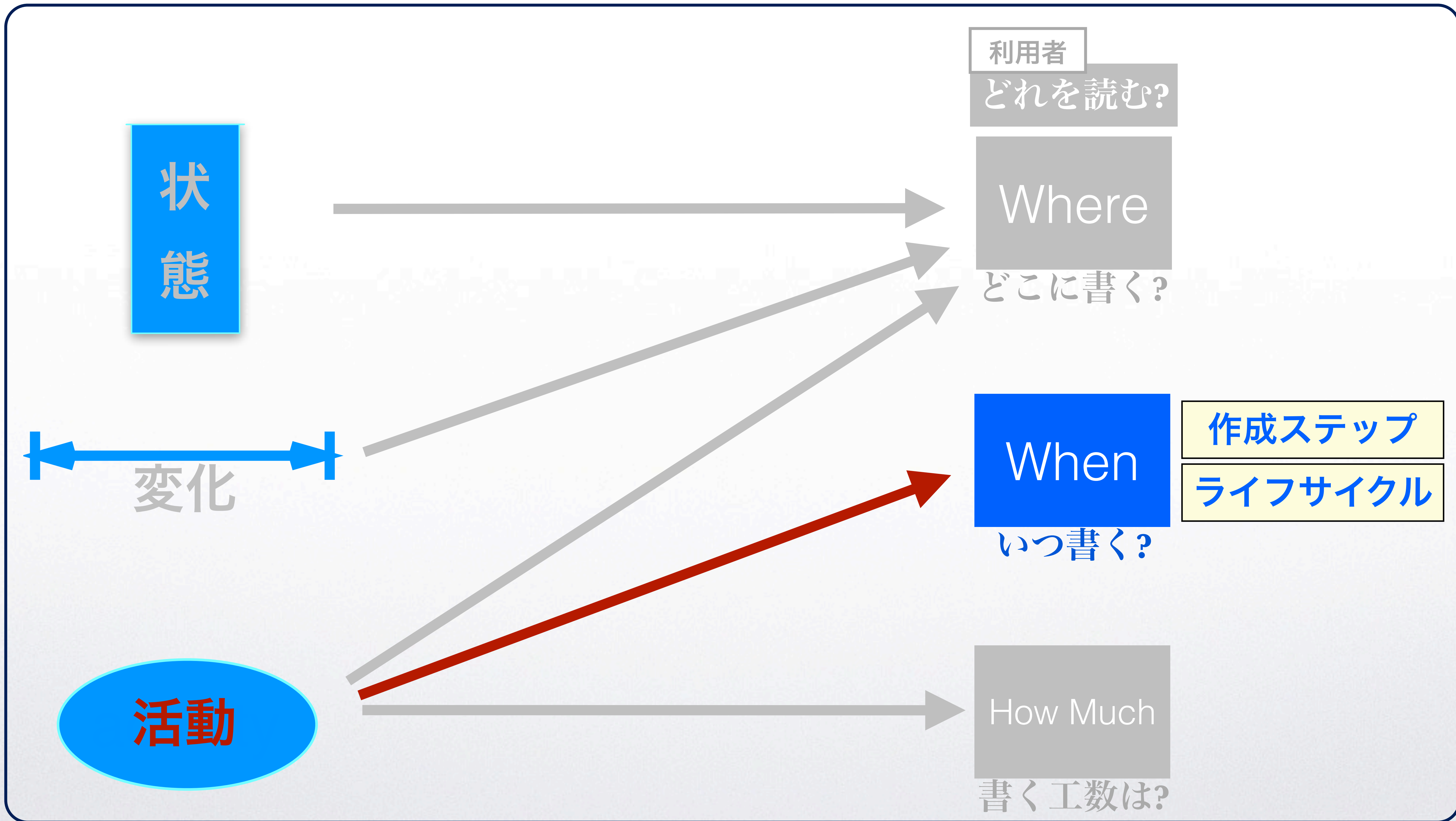
- **Redmine JP**
  - http://redmine.jp/
- **インストール**
  - http://redmine.jp/install/
- **チケット駆動開発**
  - http://ja.wikipedia.org/wiki/チケット駆動開発
- **Redmineによるタスクマネジメント実践技法**
  - http://www.amazon.co.jp/exec/obidos/ASIN/4798121622/hatena3000-22/

運用ドキュメントのライフサイクル

When

いつ書けばいいのかわからない

ドキュメントの管理 の課題

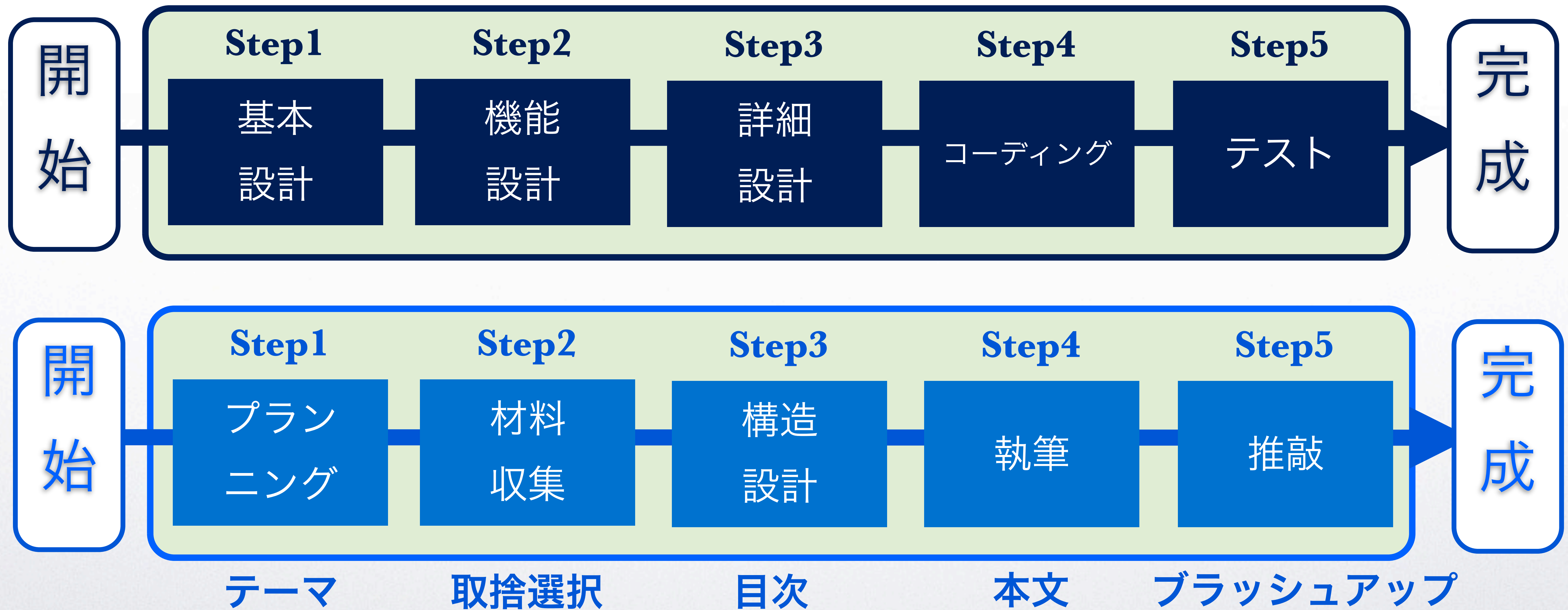


運用ドキュメントはいつ書くのか?

- **「思い立ったが書き時」**
  - と言えば、それっぽく聞こえるが、それだけではやはり書けない。
- **ドキュメント作成のためのステップ**
  - **書くためのステップ**を意識する必要がある。
  - 結果として無駄な手間がかからず確実に進みやすい。
- **「書き時」は突然やってくる**
  - 運用業務の節目(導入/変更)や運用作業はドキュメントを書く機会。

ドキュメント作成のステップ

プログラム開発とドキュメント作成の工程は似ている



(参考: 佐藤 健「SE のための「構造化」文書作成の技術 (技術評論社 2008年) P44)

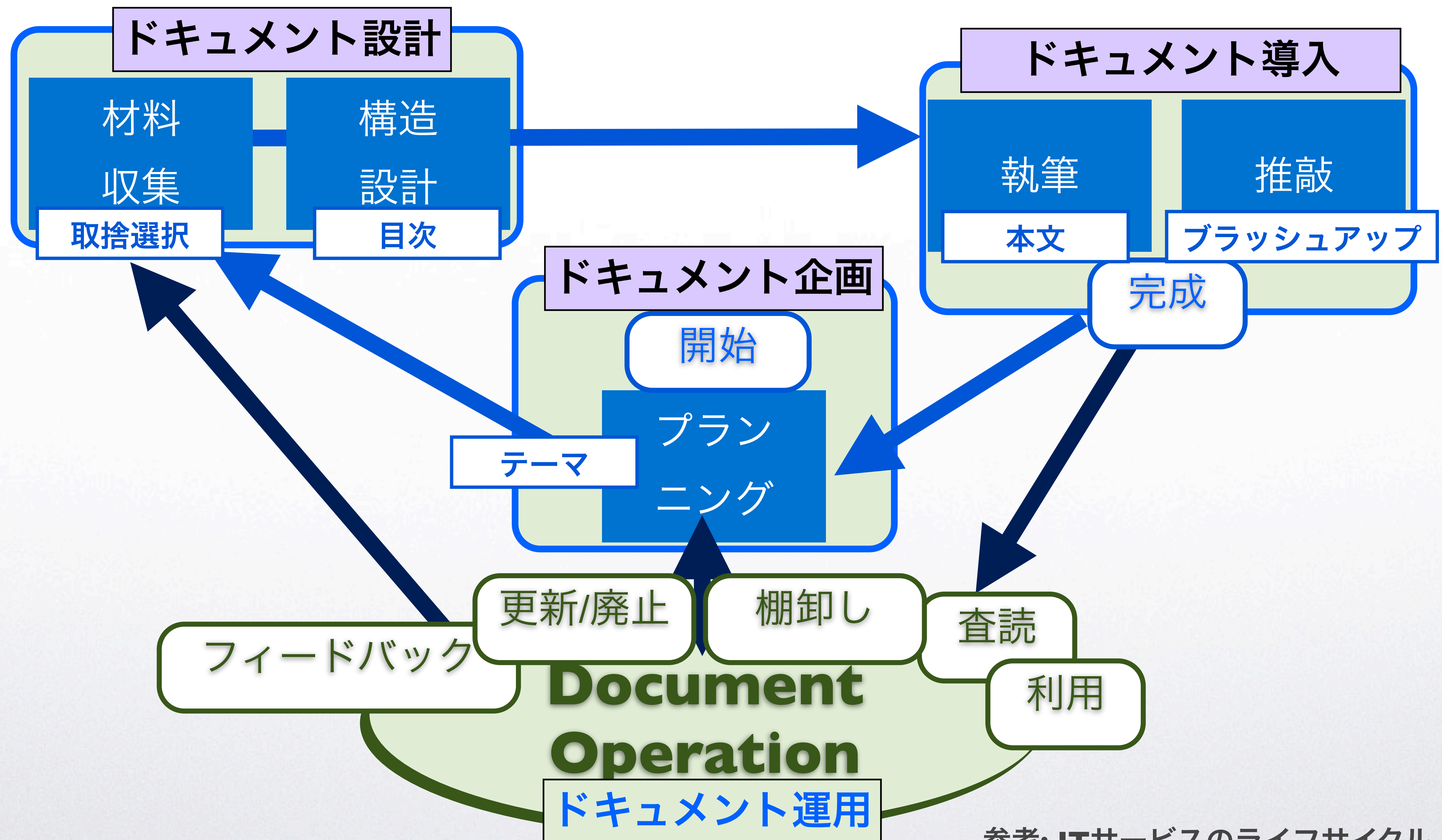
ドキュメントのライフサイクル

いつ書く？
When

- **運用ドキュメントは「完成したら終わり」ではない。**
  - むしろ完成してからがスタート。(使いながら作る事も珍しくない)
  - 更新や廃止など、ライフサイクルを意識した継続的な維持管理が必要となる。
- **「ライフサイクル」には「お葬式」も重要**
  - ドキュメントの廃止が明示的に行なわれることは少ない。
  - 不要なドキュメントが廃止されないことで、無用のトラブルを生じたり、全体の保守性を下げ無駄な保守工数がかかる可能性がある。
- **定期的な棚卸は必須**
  - 廃止すべきドキュメントや、更新されていないドキュメントを洗い出す「定期的な棚卸し」が「ドキュメントのライフサイクル」では重要となる。

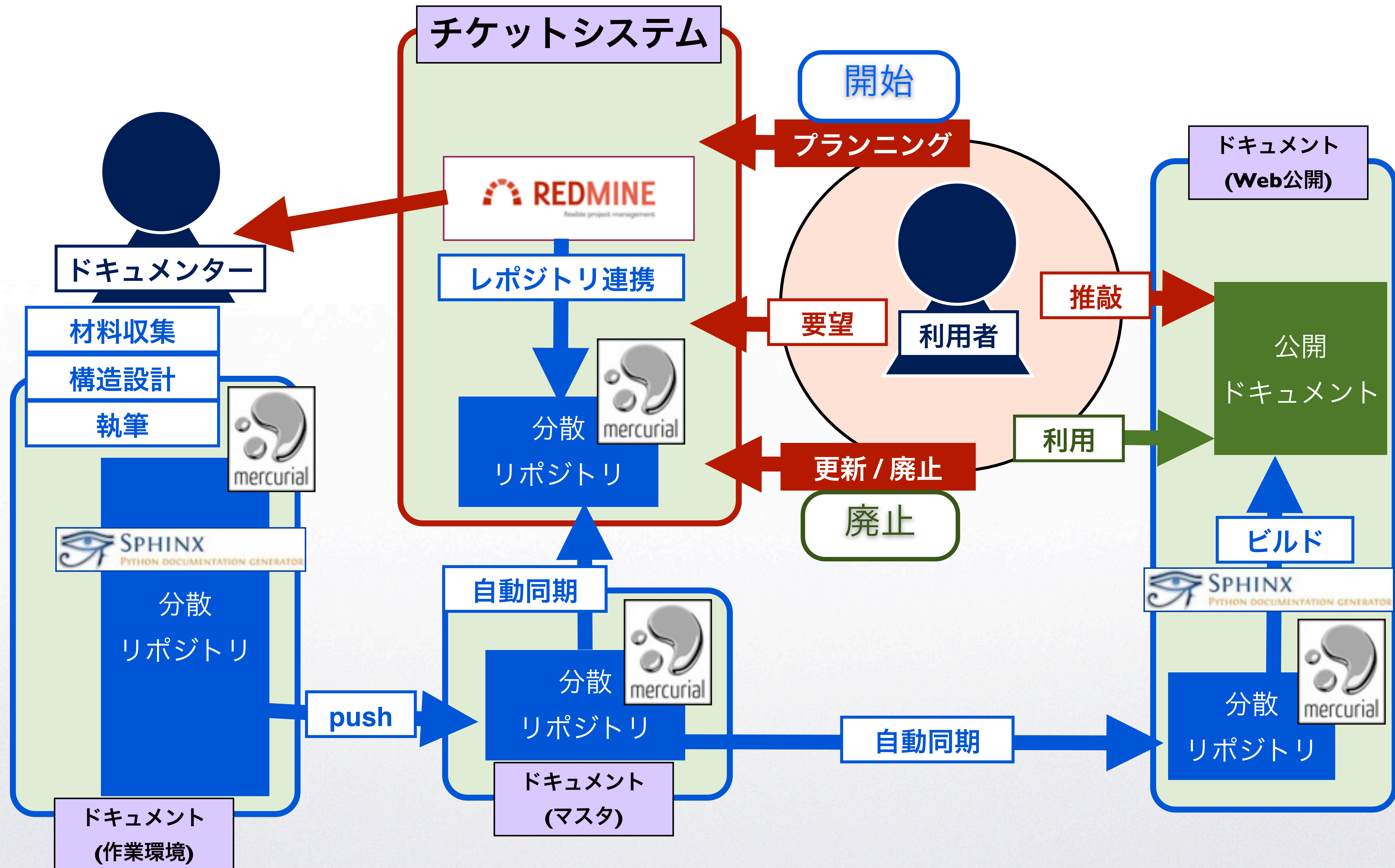
ドキュメントのライフサイクル

いつ書く?
When



参考: ITサービスのライフサイクル (ITIL v3)

参考: ドキュメントのライフサイクル (例)

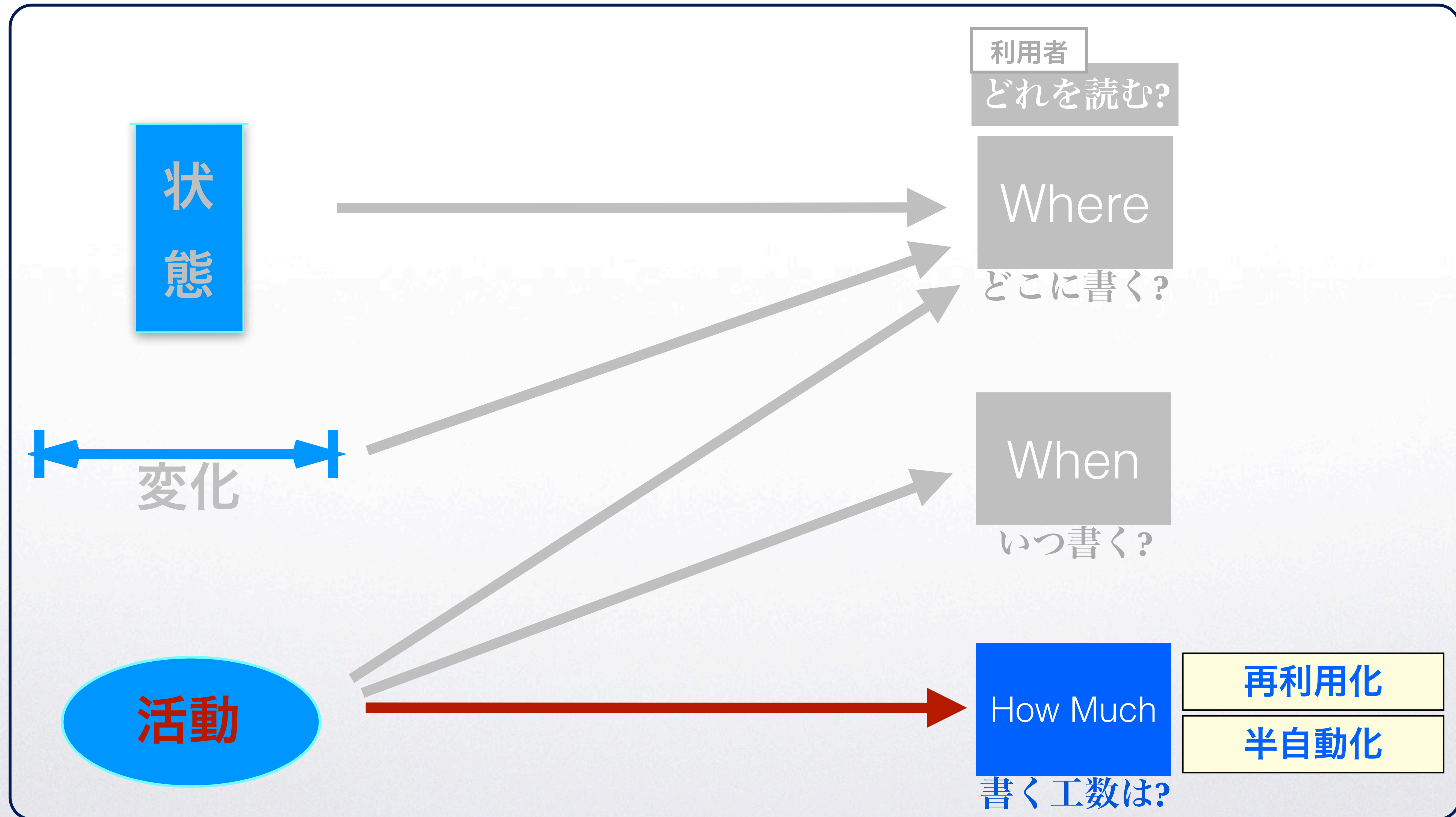


運用ドキュメントの工数確保

How Much

書くための工数が取れない

ドキュメントの管理 の課題



どう工数を捻出するか?

- **小さく作り、小まめに更新する。**
  - まず業務範囲の目次と見出しを作成し、重点ポイントを中心に工数を掛けずに小まめに更新する。
- **ドキュメントの再利用化**
  - テンプレート化による再利用性向上を図り、日常作業や新規ドキュメント作成工数の低減化サイクルを確立する。
- **ドキュメント作成の半自動化**
  - 構造化された一部データを自動生成することで、ドキュメントの作成自体を半自動化する。

「工数を捻出するための工数」が必要なのが現状

小さく作り、小まめに更新する

- **最初に重要なものは、目次と見出しの作成。(業務全体の構造化)**
- 業務で必要になったときに、細切れ時間を使って見出し以下を膨らませていく。
  - **「ドキュメント作成のステップ」を細かい周期**でまわしていく。
  - 各ステップについて「やり切らない」でどんどん進めていく。
  - 分散バージョン管理ツールで、細かい粒度で commit する。
- 更新のチャンスを逃さないために、**手軽に更新できる状態を維持する。**
  - Sphinx は手軽に更新できる環境の一つ
  - reST(Sphinx)環境を整え、精神論による作成努力から脱却する。

目次と見出しを「地図」として、アジャイルに
ドキュメント(短い周期で細かく)を仕上げていく

ドキュメントの再利用化

- **内容が同じものを複数のフォーマットに都度書く**ことを避ける。
  - 例: 内部向けドキュメント(WiKi) / 他部署向けドキュメント(PowerPoint) / アジェンダ(メール)に同じ内容をそれぞれ違う書式で書くと3倍以上の工数が発生。更にそれぞれの内容が分岐していくリスクがある。
  - Sphinx: HTML / PDF で出力するか、 ソース(整形テキスト)のまま提出可能。
- **形式が同じものを複数回使う**場合は、テンプレート化して再利用する。
  - 対象例: 手順書、メールテンプレート、申請書など
  - Sphinx: テンプレートパーツの組み合わせ(include)、変数部分の置換(replace)が可能
- 過去に使ったものを保存しておくことで、**似た作業のときの雛形**として活用する。
  - 検索可能な状態で保存されていることが重要
  - Sphinx: テキスト形式なので、過去数十年にわたる資産が活用でき、今後数十年後も使える可能性が高い。

「同じものを二度書かない」「似たものは転用する」

ことで再利用可能なドキュメント資産を増やしていく。

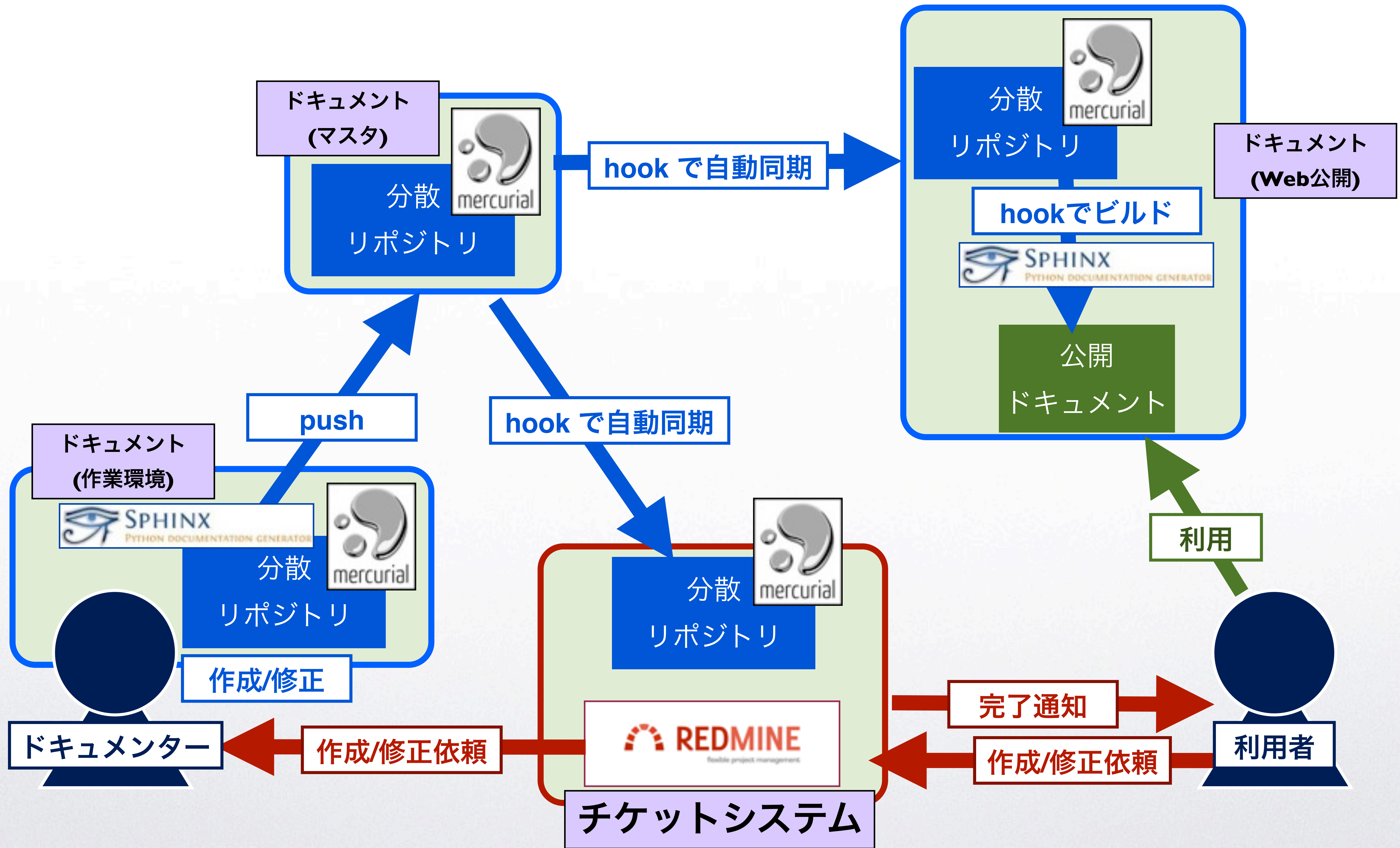
ドキュメント作成の半自動化

- **人が手で書く必要の無いものは、システムに作らせる。**
- **公開されているデータは、システムに自動取得させる。**
  - 監視システムのデータ (例: MRTGのグラフ画像)
  - 社内の統計データ(Web/API) (例: csv形式の月間データ)
  - 総務省の祝日データ
- **取得したデータはシステムに加工させる。**
  - Sphinx本体: Python で書かれているので自作拡張機能を書けなくもない
  - Sphinx実行: Make で実行されるので UNIXコマンドでの事前/事後処理が可能
- **加工したデータをWebで公開して提供する。**
  - 自部署と関連する他部署のドキュメントも半自動化が可能になる。(コミュニケーションコスト低減)

システムが作れない「人手が本当に必要」な
ドキュメントに人は注力できるようになる。

参考: 運用ドキュメントシステム (例)

参考: システム構成 (例)



参考: システム構成 (例) に必要な知識

- **Mercurial**
  - **基礎知識** (init, update, commit, push, pullサブコマンド)
  - **.hg/hgrc の記述** (push / hook 設定)
- **Redmine**
  - **インストール方法** (Apache / DBMS(MySQLなど) / Rail on Rails)
  - **基礎知識** (チケットの作成、更新、メール通知)
  - **リポジトリ設定** (プロジェクト内での設定方法)
- **RedmineチケットとMercurialコミットログの紐付け**
  - **Mercurialでのコミットログの書き方** (Redmineの設定で変更可能)
    - チケットへの紐付: "**refs** #xxx"
    - チケット解決: "**fixes** #xxx"

(xxxはチケット番号)

参考 (インストール手順)

- **Sphinx (難易度 低)**
  - http://sphinx-users.jp/gettingstarted/index.html
- **Mercurial (難易度 低)**
  - http://troter.jp/how-to-install-mercurial/
- **Redmine (難易度 中)**
  - http://redmine.jp/install/