アジャイルと形式手法

ライフロボティクス株式会社
川口　順央

自己紹介

川口 順央（かわぐち　まさてる）
@masateruk
ライフロボティクス株式会社
CORO開発プロジェクト プロジェクトリーダー

2004年から形式手法に興味を持ち、勉強と試行の繰り返し
個人レベルでは何度か現場に導入したことあり（Spin, Alloy, CSPなど）

ライフロボティクス、吉野家に食器洗いロボ納入　時短・安全両立

ツイート　いいね！1　LINEで送る　(2017/3/28 05:00)

日刊工業新聞

ライフロボティクス（東京都江東区、尹祐根〈ゆん・うぐん〉社長、０３・６４５８・８２５８）は２７日、吉野家の食器洗浄工程に自社製多関節型ロボット「ＣＯＲＯ＝写真」を納入したと発表した。１日当たり２・３時間かかっていた食器の浸漬・洗浄から格納までの工程が０・５時間と大幅削減を見込む。従業員の腰・肩への負担や、食器の破損によるけが・手荒れなども大幅に軽減できる。

日刊工業新聞より
http://www.nikkan.co.jp/articles/view/00422414

適用プロジェクトの概要

人の近くで動く協働ロボットCORO®の開発

スタートアップゆえにスピードが求められるが、
同時に品質も求められる

アジャイルと形式手法

プロジェクト全体をスクラムで回しつつ、仕様を形式的仕様記述言語で記述する

アジャイルと形式手法を両立する？

アジャイルのいいところ

イテレーティブでインクリメンタルな開発は不確実性が高い状況ではとても有効

形式手法のいいところ

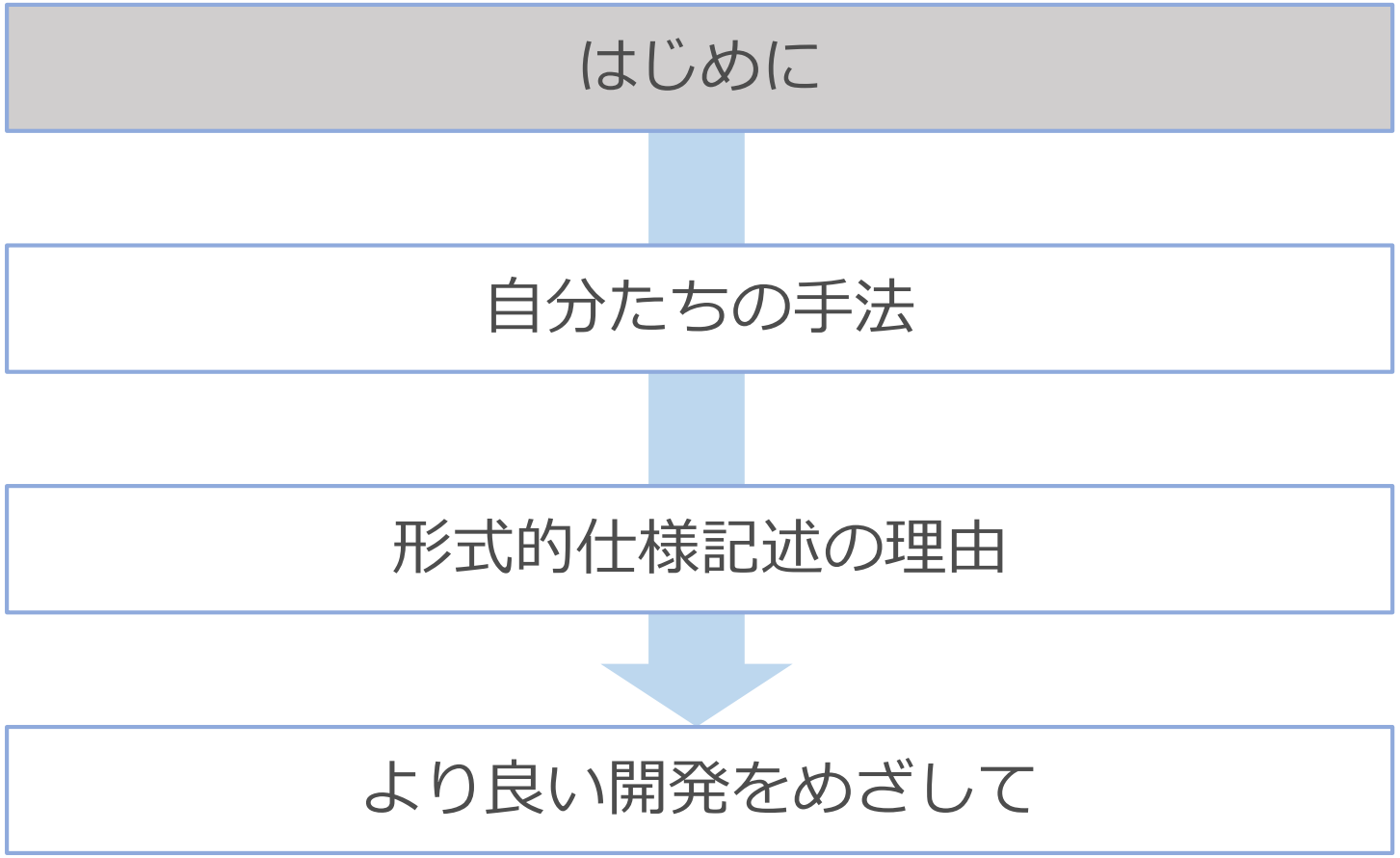
形式手法は数学を道具としてソフトウェアの品質を飛躍的に高められる

“ドキュメントより動くソフトウェア”って言うけど

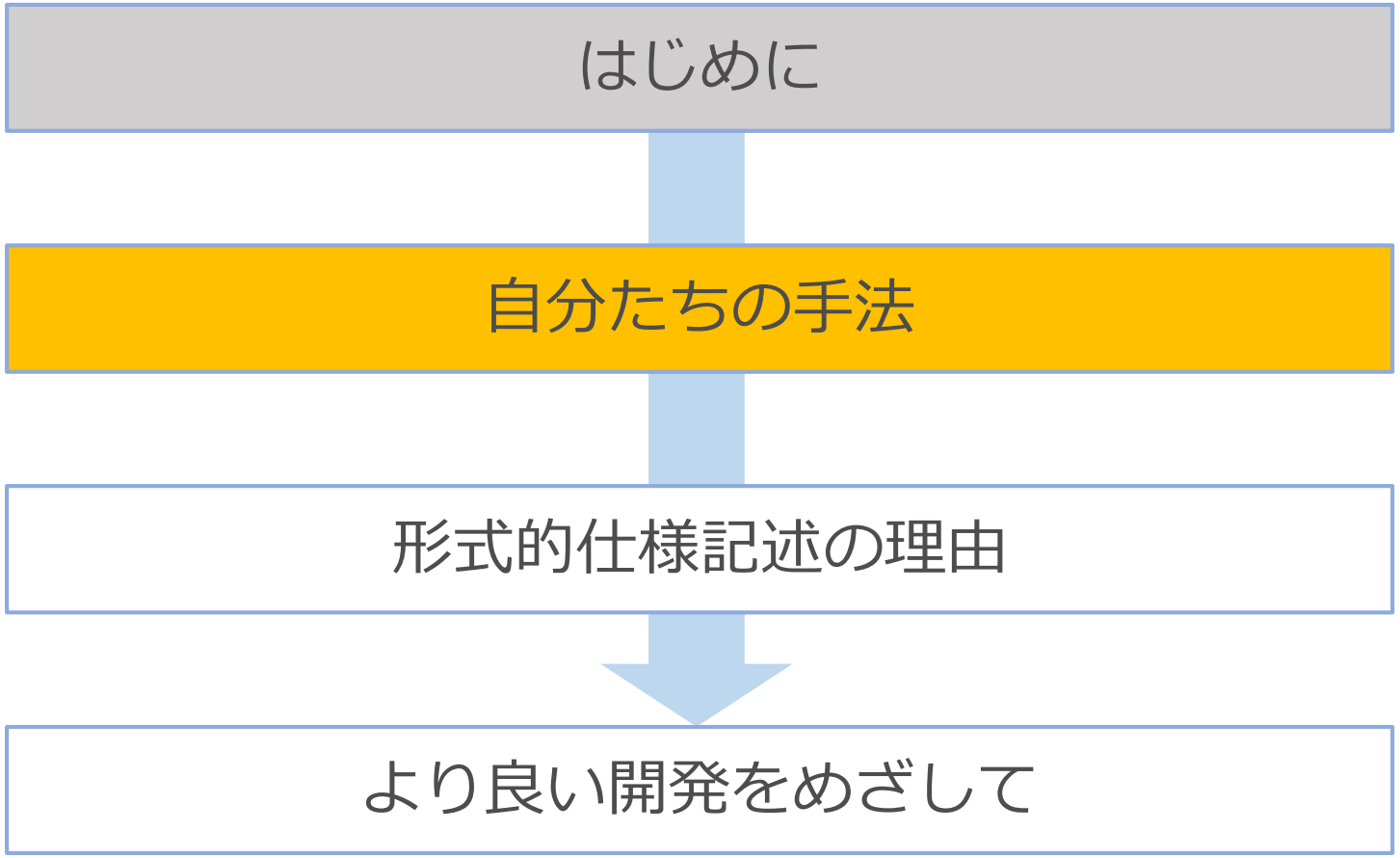
相性が良くない？

導入コストが高い？

発表の構成



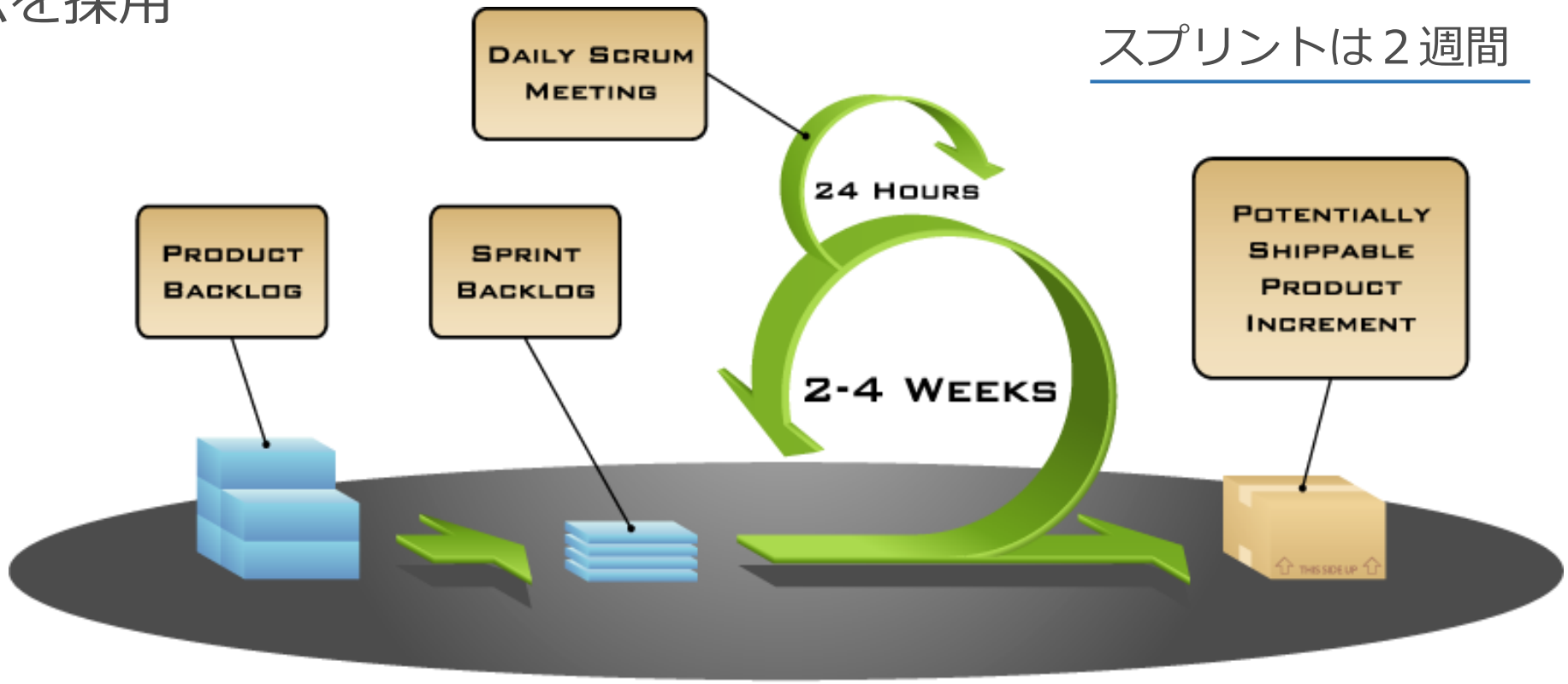
発表の構成



自分たちの手法 ー プロジェクト全体の運用

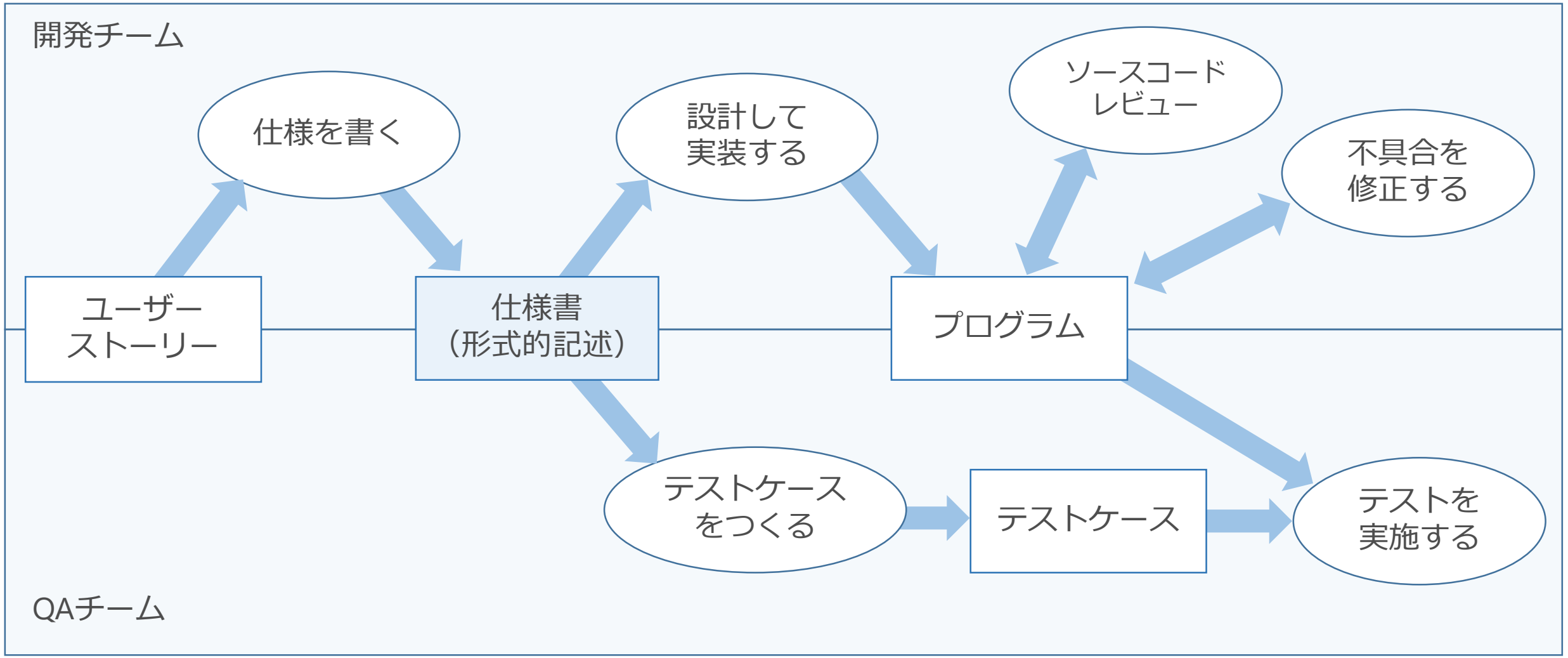
- スクラムを採用

スプリントは２週間



顧客の価値を表すユーザーストーリーをバックログに

自分たちの手法 ー ユーザーストーリーの実現の仕方



自分たちの手法 ー 仕様の書き方

- 仕様は状態遷移モデルで記述する。遷移の構成要素は、
  - 遷移元の状態、イベント（パラメータを含む）、ガード条件、事後条件、遷移先の状態
- 状態は変数をもつことができる。ガード条件、事後条件の記述を**論理式**と**自然言語**で書く。
  論理式は、独自の仕様記述言語**KML**で書く
- 独自の言語を作った理由
  - 気に入ったものがなかった。VDMも候補だったが、補助関数については関数型プログラムのように書けるものがほしかった。

KMLの記述例


```
// 例）ロボットの移動点の編集状態の状態遷移
state Edit@(移動点の編集状態) {

    var points : Point List = [];
    var index : Int = 0;

    invariant { // この状態で成り立つ不変条件
        (length points != 0) =>
            (0 <= index && index < length points);
    }@{
        点のリストが空でなければ、インデックスは0以上、
        かつ、点のリスト長より小さい
    }

    transition addPoint(p : Point)
          when length points < 10
               @[- 点のリストが10未満のとき -] --> {
        post {
            target points;
            points' = points ^ [p];
        }@{-
            事後の点のリストは、事前の点のリストの末尾に
            点pを追加したもの
        -}
    }

    transition move
          when length points != 0
               @[- 点のリストが空でないとき -] --> {
        post {
            state' = Moving(points);
        }@{-
            移動状態へ遷移する
        -}
    }
}

state Moving@(再生状態)(ps : Point List) {
…
```

状態変数

不変条件

ガード条件を持つ遷移

異なる状態への遷移

※ 赤字で書いた部分は必須の注釈

注釈がない場合はコンパイルエラー

KMLの記述例 ― 状態変数と不変条件


```
// 例）ロボットの移動点の編集状態の状態遷移．// でコメントが記述できます
state Edit@(移動点の編集状態) {

    var points : Point List = [];
    var index : Int = 0;

    invariant { // この状態で成り立つ不変条件
        (length points != 0) => (0 <= index && index < length points);
    }@{
        点のリストが空でなければ、インデックスは0以上、かつ、点のリスト長より小さい
    }
```

KMLの記述例 ― 状態遷移

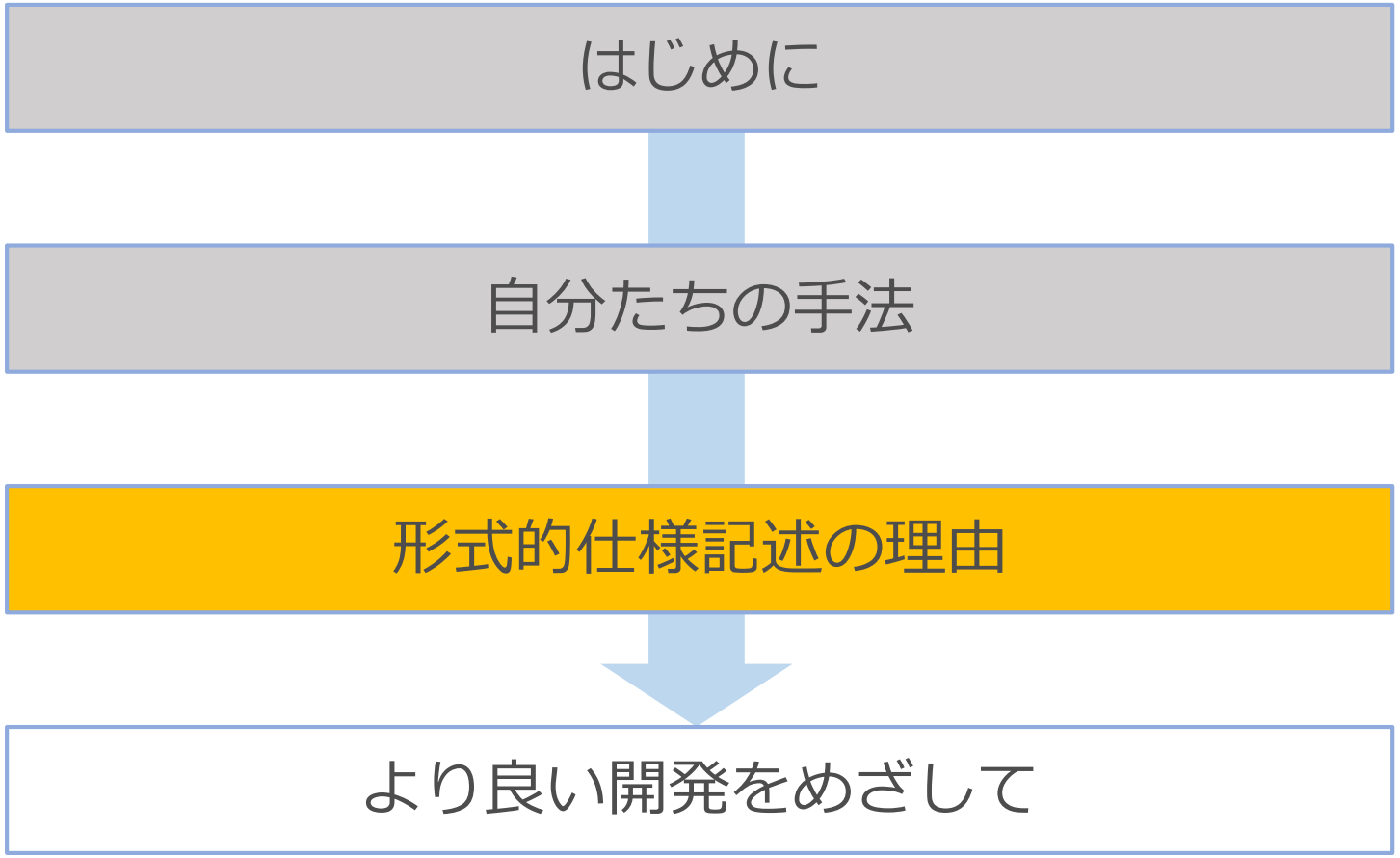

```
transition addPoint(p : Point) --> {
      when length points < 10
           @[- 点のリストが10未満のとき -] --> {
    post {
        target points;

        points' = points ^ [p];
    }@{-
        事後の点のリストは、事前の点のリストの末尾に点pを追加したもの
    -}
}
```

自分たちの手法 ー まとめ

- スクラムを採用
- ユーザーストーリーをもとに仕様を形式的に書く
- 仕様を書いた後は、開発チームとQAチームは並行に作業する

発表の構成



”ドキュメントよりも動くソフトウェア”の理由

こんな文書化は嫌われる

- すでに終わった作業についての単なる文書化（創造的な活動ではない）
- 編集環境が気に入らない。WordやExcelが嫌いだ
- 意味のわからないフォーマットや体裁に従わないといけない
- 書いたドキュメントを誰も使わない。用途が不明

まとめると

- かけた時間に対して効果が少ない。**費用対効果**が悪い
- 楽しくない

本来ドキュメントがもつ力

考えをアウトプットさせられる

- 頭の中で何となく考えていたものが明確になる
- 考えが足りなかったことに気づく
- 間違いを発見できる

書いたものが手に入る

- 人にレビューしてもらえる
- あとで読み返すことができる

形式的記述は、本来ドキュメントがもつ力を最大限に引き出す

考えをアウトプットさせられる

- 頭の中で何となく考えていたものが明確になる
- 考えが足りなかったことに気づく
- 間違いを発見できる

書いたものが手に入る

- 人にレビューしてもらえる
- あとで読み返すことができる

形式的記述は、本来ドキュメントがもつ力を最大限に引き出す

考えをアウトプットさせられる

- 頭の中で何となく考えていたものが明確になる
- 考えが足りなかったことに気づく
- 間違いを発見できる

書いたものが手に入る

- 人にレビューしてもらえる
- あとで読み返すことができる

形式的記述は、本来ドキュメントがもつ力を最大限に引き出す

考えをアウトプットさせられる

- 頭の中で何となく考えていたものを**厳密**にアウトプットさせられるため、より明確になる
- 考えが足りなかったことに気づく
- 間違いを発見できる

書いたものが手に入る

- 人にレビューしてもらえる
- あとで読み返すことができる

形式的に仕様を書くとより明確になる

自然言語だけで書いた仕様

```
# 移動点に点 p を追加したとき
  移動点のリストに点 p を追加する
```

挿入？末尾に追加？先頭に追加？

KMLで書いた仕様

```
transition addPoint(p : Point) --> {
    post {
        target points;

        points' = points ^ [p];
    }@{-
        事後の点のリストは、事前の点のリストの末尾に点pを追加したもの
    -}
}
```

末尾に追加することが明確

形式的記述は、本来ドキュメントがもつ力を最大限に引き出す

考えをアウトプットさせられる

- 頭の中で何となく考えていたものを**厳密**にアウトプットさせられるため、より明確になる
- 考えが足りなかったことに気づく
- 間違いを発見できる

書いたものが手に入る

- 人にレビューしてもらえる
- あとで読み返すことができる

形式的記述は、本来ドキュメントがもつ力を最大限に引き出す

考えをアウトプットさせられる

- 頭の中で何となく考えていたものを**厳密**にアウトプットさせられるため、より明確になる
- 未定義な言葉で説明したつもりの文章は書けないため、考慮不足を**ごまかせない**
- 間違いを発見できる

書いたものが手に入る

- 人にレビューしてもらえる
- あとで読み返すことができる

形式的に仕様を書くと未定義な言葉でごまかせない

自然言語だけで書いた仕様

```
# 移動点に削除操作したとき
  インデックスに対応した点を削除する
```

未定義になりやすい言葉

- ○○に対応した
- ○○に応じた
- ○○に一致する

KMLで書いた仕様

```
transition deletePoint
          when length points != 0 @[- 点のリストが空でないとき -] --> {
        post {
            target points, index;
            points' = [ points # i |
                        i : Int & 0 <= i && i < length points && i != index ];
            index' = if index + 1 = length points then max(index - 1, 0) else index;
        }@{-
            事後の点のリストは、事前の点のリストからindex番目を除いたもの
            インデックスは末尾を指していた場合は1つ小さい値と0の最大値、そうでなければ変わらない
        -}
    }
```

形式的記述は、本来ドキュメントがもつ力を最大限に引き出す

考えをアウトプットさせられる

- 頭の中で何となく考えていたものを**厳密**にアウトプットさせられるため、より明確になる
- 未定義な言葉で説明したつもりの文章は書けないため、考慮不足を**ごまかせない**
- 間違いを発見できる

書いたものが手に入る

- 人にレビューしてもらえる
- あとで読み返すことができる

形式的記述は、本来ドキュメントがもつ力を最大限に引き出す

考えをアウトプットさせられる

- 頭の中で何となく考えていたものを**厳密**にアウトプットさせられるため、より明確になる
- 未定義な言葉で説明したつもりの文章は書けないため、考慮不足を**ごまかせない**
- **ツール**の力で間違いを発見できる

書いたものが手に入る

- 人にレビューしてもらえる
- あとで読み返すことができる

形式的記述は、本来ドキュメントがもつ力を最大限に引き出す

考えをアウトプットさせられる

- 頭の中で何となく考えていたものを**厳密**にアウトプットさせられるため、より明確になる
- 未定義な言葉で説明したつもりの文章は書けないため、考慮不足を**ごまかせない**
- **ツール**の力で間違いを発見できる

書いたものが手に入る

- 人にレビューしてもらえる
- あとで読み返すことができる

形式的記述は、本来ドキュメントがもつ力を最大限に引き出す

考えをアウトプットさせられる

- 頭の中で何となく考えていたものを**厳密**にアウトプットさせられるため、より明確になる
- 未定義な言葉で説明したつもりの文章は書けないため、考慮不足を**ごまかせない**
- **ツール**の力で間違いを発見できる

書いたものが手に入る

- レビューアが間違いかそうでないか判断を**正確に**できる
- あとで読み返すことができる

形式的記述は、本来ドキュメントがもつ力を最大限に引き出す

考えをアウトプットさせられる

- 頭の中で何となく考えていたものを**厳密**にアウトプットさせられるため、より明確になる
- 未定義な言葉で説明したつもりの文章は書けないため、考慮不足を**ごまかせない**
- **ツール**の力で間違いを発見できる

書いたものが手に入る

- レビューアが間違いかそうでないか判断を**正確に**できる
- あとで読み返すことができる

形式的記述は、本来ドキュメントがもつ力を最大限に引き出す

考えをアウトプットさせられる

- 頭の中で何となく考えていたものを**厳密**にアウトプットさせられるため、より明確になる
- 未定義な言葉で説明したつもりの文章は書けないため、考慮不足を**ごまかせない**
- **ツール**の力で間違いを発見できる

書いたものが手に入る

- レビューアが間違いかそうでないか判断を**正確に**できる
- **曖昧さがない**。そのため解釈が多様にならない

費用対効果の点でおいしい形式的仕様記述 #1

- 仕様をすべてのはじまりにすることで仕様の利用頻度（参照回数）が多くなる
  - 開発者が設計をするとき
  - QAがテストケースをつくるとき
  - バグっぽい動きを見つけた時に、仕様かバグかを判断するとき
  - 仕様変更をうけたときに今の仕様を確認するとき
  - 問い合わせをうけたときに仕様を確認するとき
  - マニュアルをかくとき

費用対効果の点でおいしい形式的仕様記述 #2

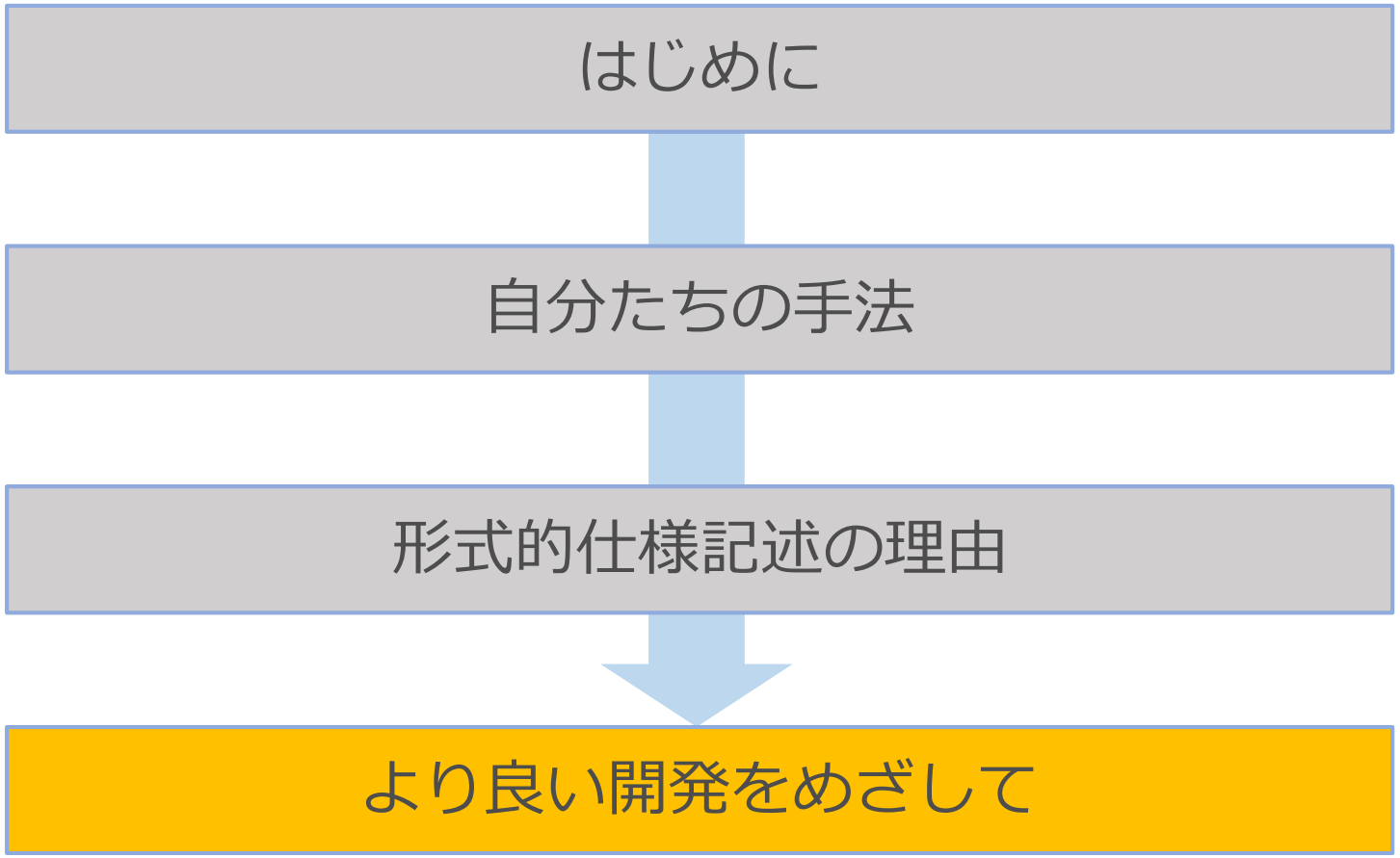
- 仕様を形式的に書くことで得られるおいしさ
  - 形式的に書くと、たくさんの間違いや考慮漏れに気づく
  - コードを書く前に気づくことができる。 Fail Fastを仕様で実現する
- 意思決定がはやくなる。何をやるべきか明確になる
  - 仕様に違反した実装であれば、バグなのでプログラムを修正する
  - 仕様が妥当（望ましいもの）でなければ、仕様変更を検討する

効果が見えて楽しいのが継続のカギ

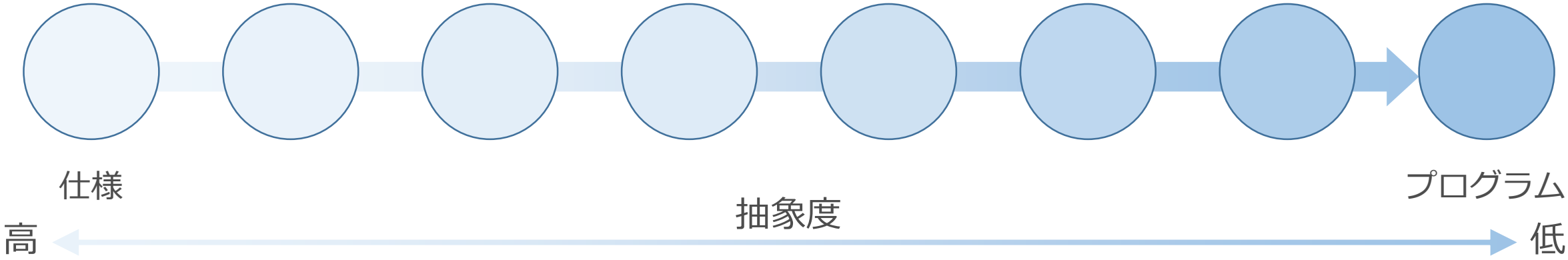
メンバーの感想

- ✓ 自然言語で書かれると読む人によって解釈が多様になるけど、論理式だと曖昧さがない
- ✓ 日本語と併記している仕様を見ると、日本語の不明瞭さがよくわかる
- ✓ 齟齬が生じにくいのでコミュニケーションが取りやすかった
- ✓ 仕様の漏れに気づきやすい
- ✓ 書いていて楽しいKML

発表の構成



ソフトウェアの開発は段階的詳細化



- どの程度段階を踏むのか、それぞれのモデルを書く、書かない、などさまざまな選択肢があるが、すべての開発で陽に陰に行なっている
- モデルを記述することには効果がある。問題は費用対効果である
- 対象を抽象的に記述したり、書いたものを分析する技術が向上すれば費用は抑えられる

形式手法を通して得る抽象化能力

- 数理論理学をベースとしている。数学は抽象的な記述に向いている
- 詳細化関係など形式的な定義を理解することで得られる自分の作業のより**明解な理解と認識**
  - たとえ非形式的に作業を行うとしても形式的な定義に照らし合わせると自分が何をやっているのかを明確に認識できる
  - 何を行なって、何を行わないかを意識的に選択することができる
- 記述したモデルの分析と検証が計算機で可能。検証を正確に、網羅的に、高速に行える
- 使っているうちに検証能力がいつの間にか身につく

スモールスタートで初期コストを抑える

- まずは仕様記述から始める
  - ハードルが高い形式的な検証や分析は最初はあきらめる
- 初学者は読むことから
  - 書くより読むほうが易しい
  - 自然言語を併記すればなんとなく読めるという状況をつくる
  - ある程度読む期間を経て記述に慣れてきたら書くことに挑戦する
- わからないことは経験者にすぐに聞ける環境が望ましい

より良い開発手法への挑戦

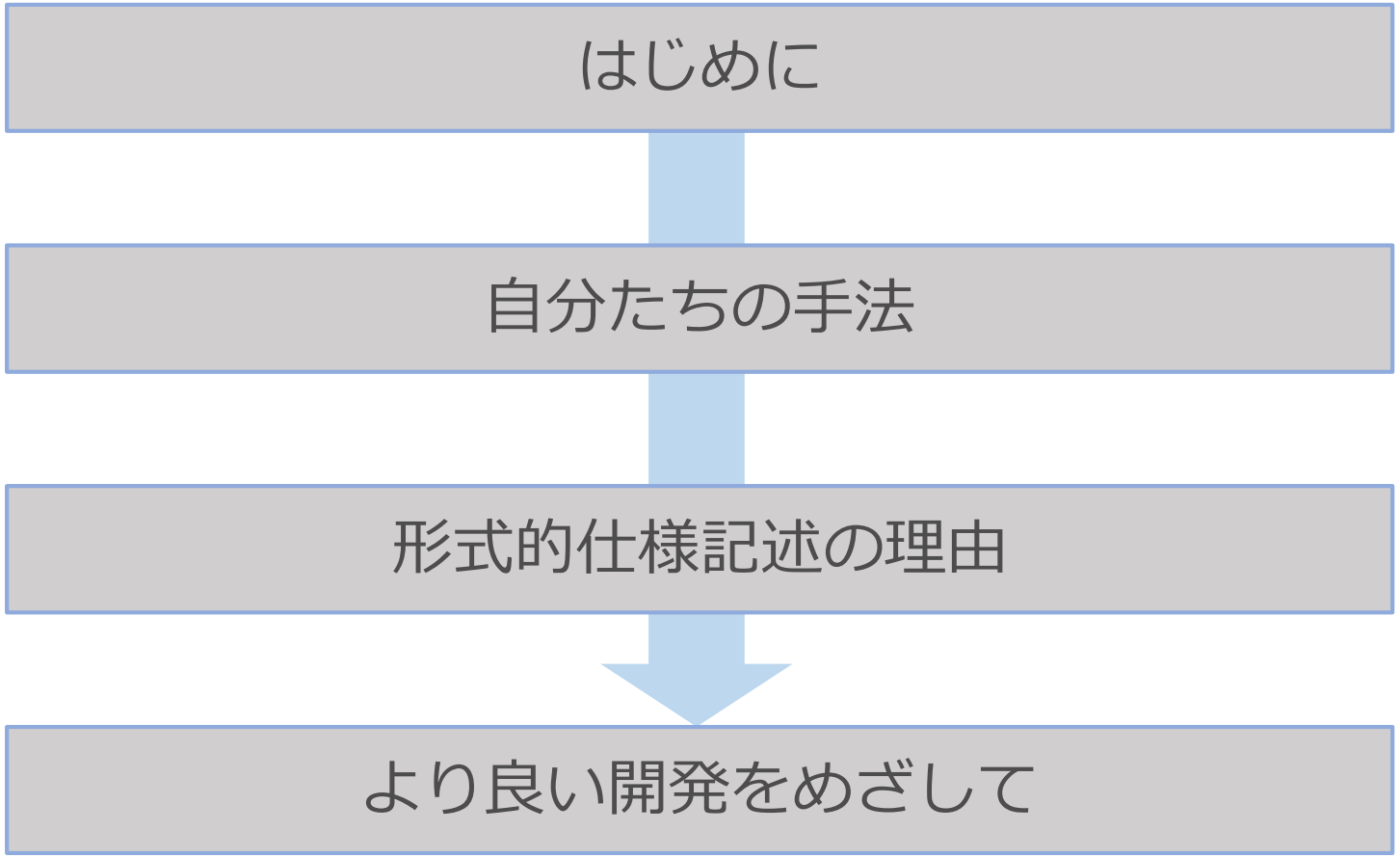
- アジャイルや形式手法に限らず、開発をよりよくする技術であれば挑戦したい
  - いつか暇になったら、というときは永遠に訪れない
  - 普段の開発をしながら自分たちを高めていく
  - 新しい技術への挑戦を当たり前のことにしたい
- アジャイルも形式手法もまだまだ道半ば

本質的な問題に注力するために

- 最近スケジュールが優先され品質が軽視される傾向があるが、実際に求められているのは品質の高い製品を早く届けること
- 品質を犠牲にしてスピードを求める人たちは、いつまでたっても品質の高いものを速く作れるようになれない
- 品質の高いものをつくる技術を持った人たちが訓練を重ねれば、速く作れるようになれる

自分たちを訓練して当たり前のように品質の高い製品が作れるようになれば、ユーザにとっての価値は何か？とか、使い勝手とか、より**本質的な問題に注力**することができるようになる

発表の構成



まとめ

- アジャイルと形式手法のいいとこ取りはできる
- 仕様を形式的に書いてドキュメントの費用対効果を高める
- 開発をより良くするために日常の中で新しい技術に挑戦する

発表の構成