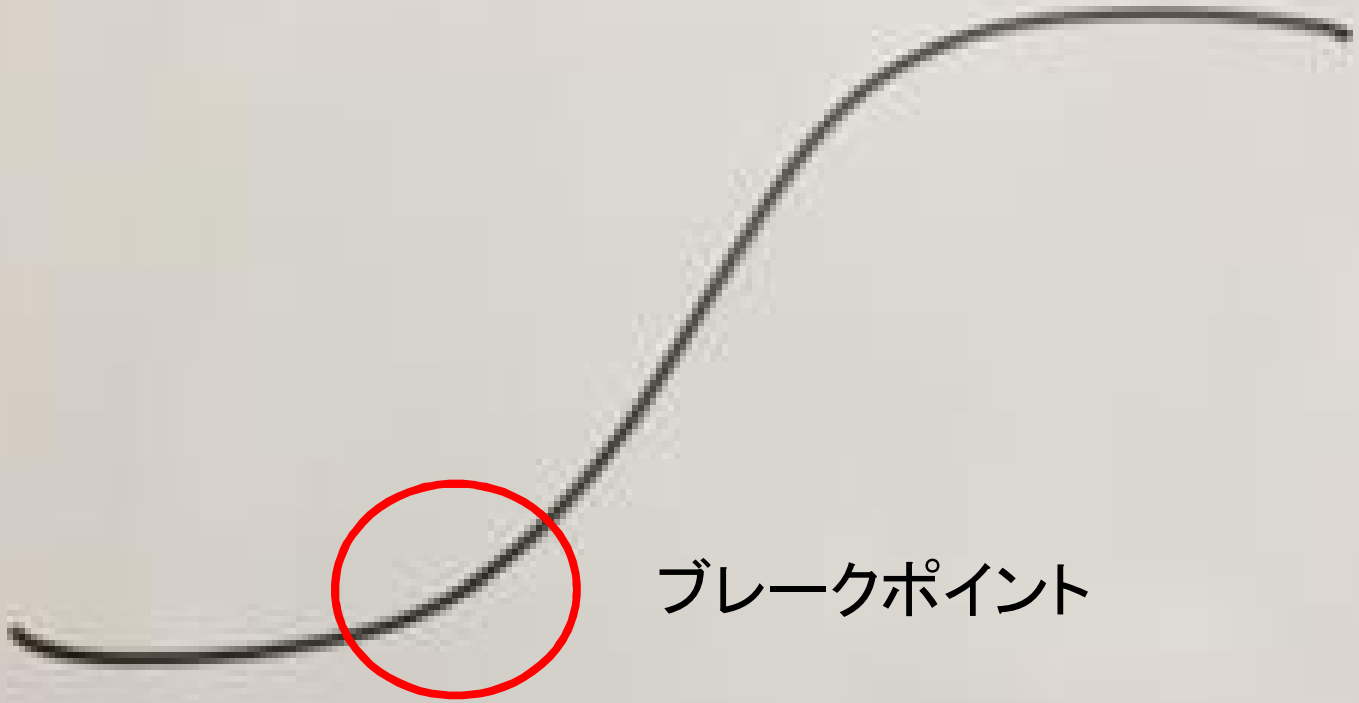
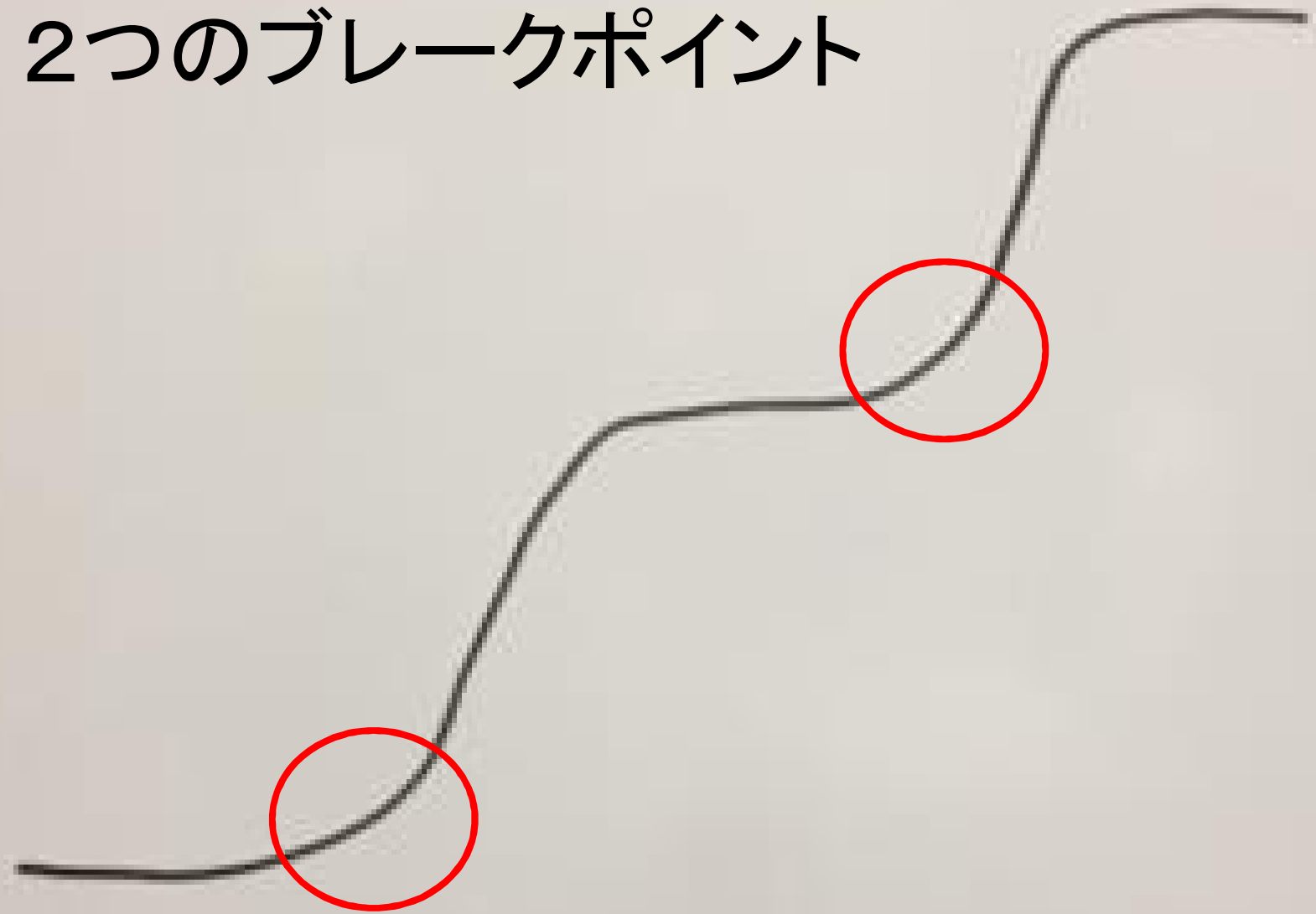
ドメイン駆動設計
学習曲線とブレークポイント

ギルドワークス
増田　亨

学習曲線



ドメイン駆動設計
2つのブレークポイント



ドメイン駆動設計

ドメイン駆動設計の効果

役に立つソフトウエアを

確実に

効率的に

ドメイン駆動設計でやらない何がおきるか？

ドメイン駆動設計の基本の活動

第1章 **知識をかみ砕く** 開発者が業務を知る

第2章 **言葉を活用する** チームで認識を合わせる

第3章 **モデルと実装を一致させる** 要求とコードを一致させる

ドメイン駆動設計の基本

1章
2章
3章

第1章
ドメインの知識をかみ砕く

第3章
モデルと実装を結びつける

ドメインの
重要な関心事を
「言葉」で鋭く説明する

ドメインの
オブジェクト
モデル

選び抜かれた
重要な「言葉」を
コードで表現する

重要な「言葉」を
チームで合意する

会話を繰り返して
「要点」を明確にする

第2章
言葉を使った意図の伝達

ドメイン駆動設計の原理

開発者が業務を広く深く理解すればするほど、
役に立つソフトウェアを確実に効率的に作れる

チーム内の認識が合っていればいるほど、
役に立つソフトウェアを確実に効率的に作れる

要求とコードが一致していればいるほど、
役に立つソフトウエアを確実に効率的に作れる

ドメイン駆動設計を支える技術

オブジェクト指向

データクラス＋機能クラスの世界から、ドメインオブジェクトの世界へ

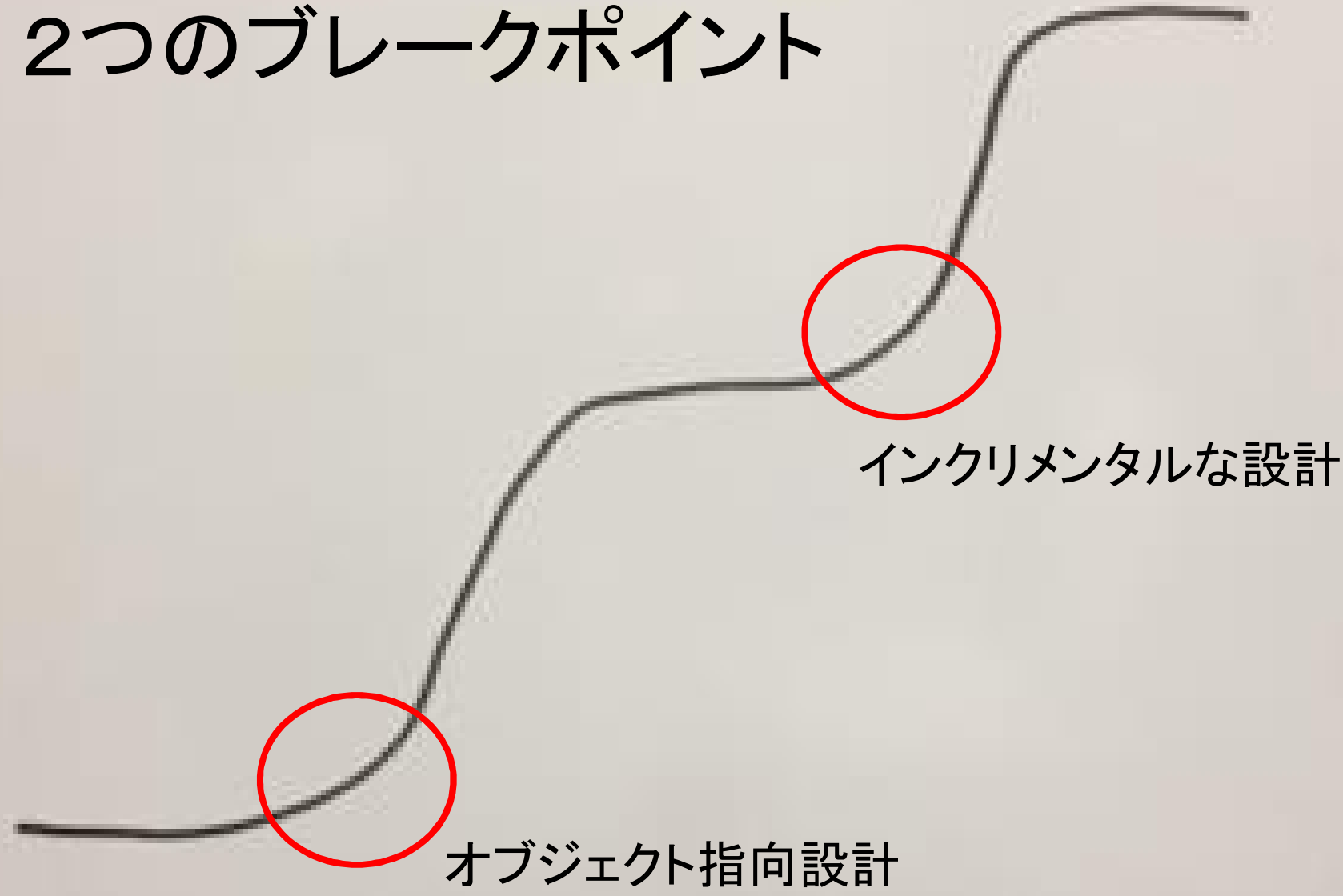
(JPA,Active Record,Entity Framework…)

インクリメンタルな設計

「設計不要」でもなく「上流工程で設計する」でもない世界へ

技術的にはこの2つ越境が、ドメイン駆動設計を実践するブレークポイント

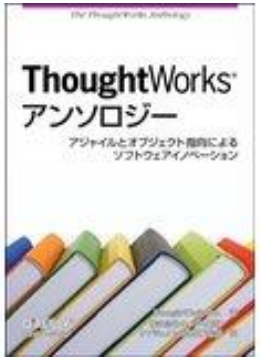
ドメイン駆動設計
2つのブレークポイント



体で覚える
オブジェクト指向設計

頭でわかっているつもりではだめ
オブジェクト指向らしい設計が
直観的にわかり
オブジェクトらしいコードに
自然に手が動くように

実践的に練習する

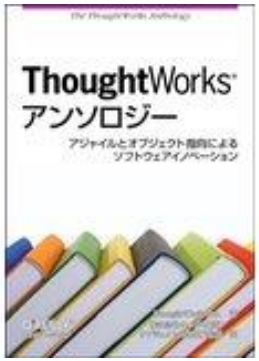


オブジェクト指向エクササイズ

ソフトウェア設計を改善する9つのルール

リファクタリング

既存のコードの設計を改善する



オブジェクト指向エクササイズ

オブジェクト指向の良さを生み出す9つのルール

９つのルール

1. 一つのメソッドでインデントは一段階
2. else 句は使わない
3. すべてのプリミティブ型と文字列をラップする
4. 一行につきドットは1つまで
5. 名前を省略しない
6. すべてのエンティティを小さくする
7. 一つのクラスのインスタンス変数は2つまで
8. ファーストクラスコレクションを使う
9. getter, setter, プロパティを使わない

名前をつける

1. 一つのメソッドでインデントは一段階
2. else 句は使わない
3. すべてのプリミティブ型と文字列をラップする
4. 一行につきドットは1つまで
5. 名前を省略しない
6. すべてのエンティティを小さくする
7. 一つのクラスのインスタンス変数は2
8. ファーストクラスコレクションを使う
9. getter, setter, プロパティを使わない

パッケージ
クラス
メソッド

小さい単位で
名前をつける

データとロジックを凝集させる

1. 一つのメソッドでインデントは一段階
2. else 句は使わない
3. すべてのプリミティブ型と文字列をラップする
4. 一行につきドットは1つまで
5. 名前を省略しない
6. すべてのエンティティを小さくする
7. 一つのクラスのインスタンス変数は2つまで
8. ファーストクラスコレクションを使う
9. getter, setter, プロパティを使わない

値オブジェクト

ファーストクラス
コレクション

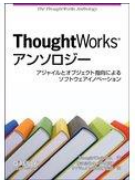
データとロジックを別の場所におかない

複数の関心事を持たない

ルールに違反している時は、複数の関心事が混在している

1. 一つのメソッドでインデントは一段階
2. else 句は使わない
3. すべてのプリミティブ型と文字列をラップする
4. 一行につきドットは1つまで
5. 名前を省略しない
6. すべてのエンティティを小さくする
7. 一つのクラスのインスタンス変数は2つまで
8. ファーストクラスコレクションを使う
9. getter, setter, プロパティを使わない

オブジェクト指向エクササイズ



９つのルールで、１０００行くらい書いてみると
オブジェクト指向らしい設計を体感できる

既存のコードから9つのルールの適用例を探し、
値オブジェクトやファーストクラスコレクションを
実際に作ってみると
before/afterでオブジェクト指向設計を体感できる

リファクタリング

オブジェクト指向の良さを生み出す設計改善のガイドライン

いやな臭い

- コードが重複している
- メソッドが長い
- クラスが大きい
- 引数が多い
- データとロジックの置き場所が別のクラスに分かれている

設計の改善

- 名前の変更
  - 業務の関心事と実装を一致させる
- メソッドの抽出
  - 手続きに名前をつける
- クラスの抽出
  - 手続きのグループに名前を付ける
- メソッドの移動／フィールドの移動
  - データとロジックの置き場所を同じクラスにする

リファクタリング

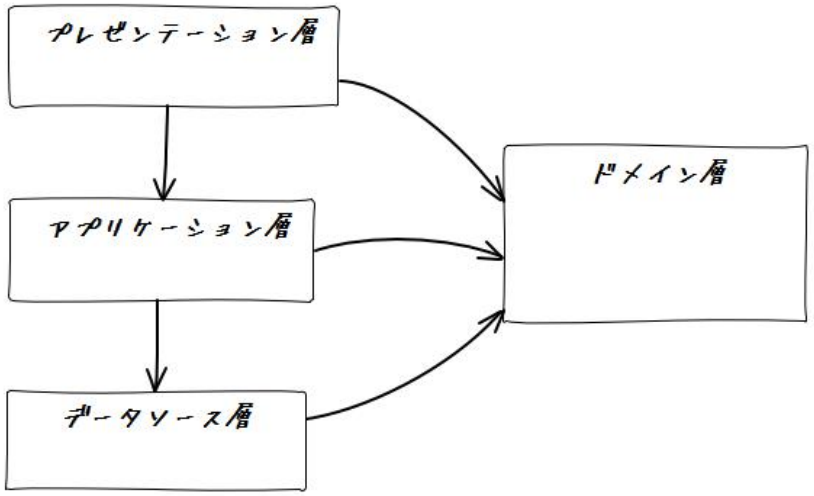
既存のコードで、いやな臭いを見つける

リファクタリングを適用して
設計改善のbefore/afterを体感する

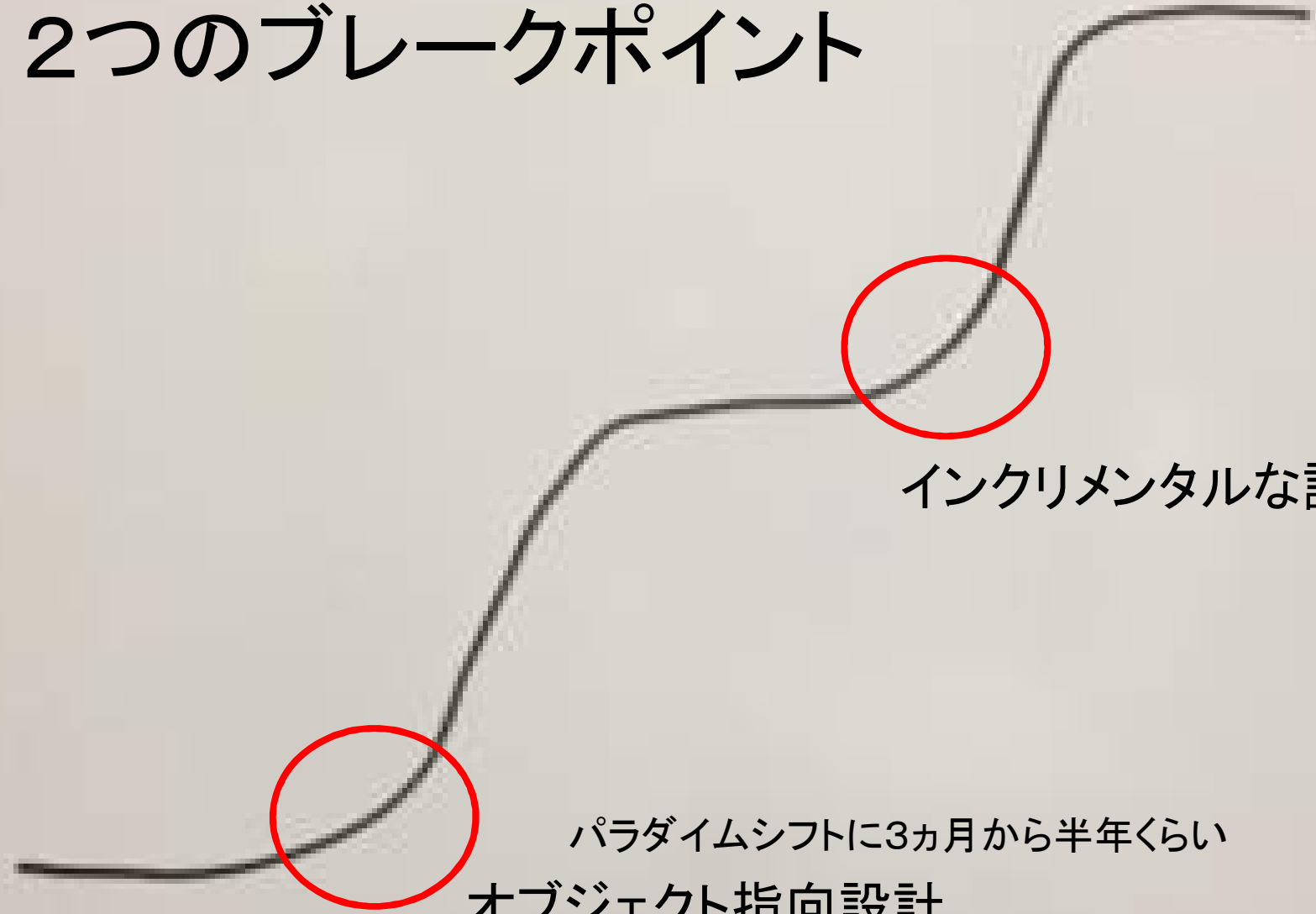
これを繰り返しているうちに
・いやな臭いに敏感になる
・最初から短いメソッド/小さなクラスを書きはじめる

補足：技術方式や規約

- 三層構造ではなく、三層＋ドメインモデル
- O-Rマッピング
  - JPA, Active Record, Entity Framework、…
  - データクラス＋機能クラスになりがち
  - オブジェクト中心のもっと軽量のツールに切り替える
- プログラミング言語
  - 静的な型付き言語
  - 独自の型の定義
  - 型名の明記
  - 型の設計変更に強い



ドメイン駆動設計
2つのブレークポイント



インクリメンタルな設計

オブジェクト指向の良さを生み出す設計手法

インクリメンタルな設計

コードを書いて動かしてから設計を改善する

最初から良い設計は見つからない
動いていても良い設計とは限らない

コードに書いて動かしてみると学びが多い
その学びをコードに反映する

そういう設計のほうが確実で効率的

インクリメンタルな設計

発想の転換

体制とやり方

体で覚える

インクリメンタルな設計
発想の転換

システム企画書
プロジェクト計画書
要件定義書
基本設計書
画面デザイン
項目定義書
データベース設計書
ユーザーガイド
テスト計画書
…

どれもいきなり完成しない
アウトラインからはじまり
少しずつ加筆し整形していく

コードもいっしょ

プロジェクトの早い段階から
アウトラインのコード書いて
少しずつ加筆し整形していく

初日から

それがもっとも確実で効率的

ソフトウェア設計の主たる表現手段はコード
（図や文書は補助）

インクリメンタルな設計
体制とやり方

コードを書く人間が要件のヒアリングと分析をする
それができる人材を選ぶ／育てる

コードを書かない人間に設計やモデリングをさせない

初日からコードのアウトラインを書く
毎日コードを加筆し整形していく

それがもっとも確実で効率的

インクリメンタルな設計
実プロジェクトで体で覚える

初日からコードを書こうとしてみる

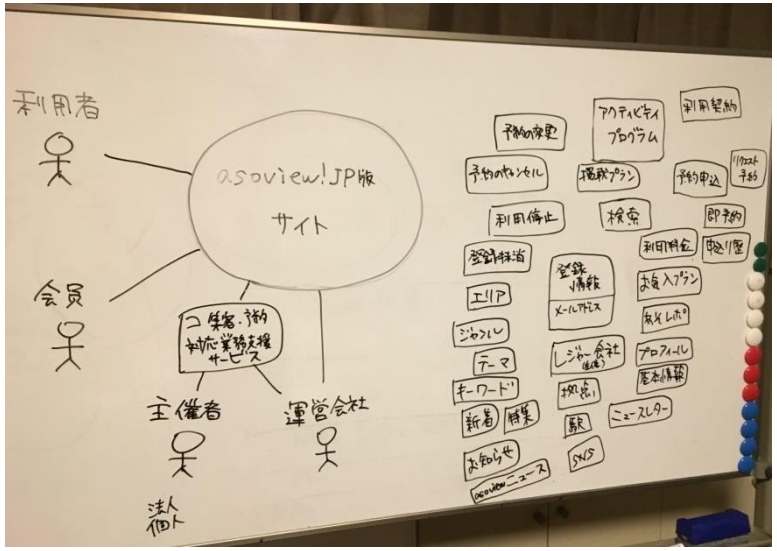
書けなかったら
時間を制限してラフスケッチ
コードで書いてみる

コードが書ける時は、コードだけ書く
コードが書けなくなったら会話や図を使う

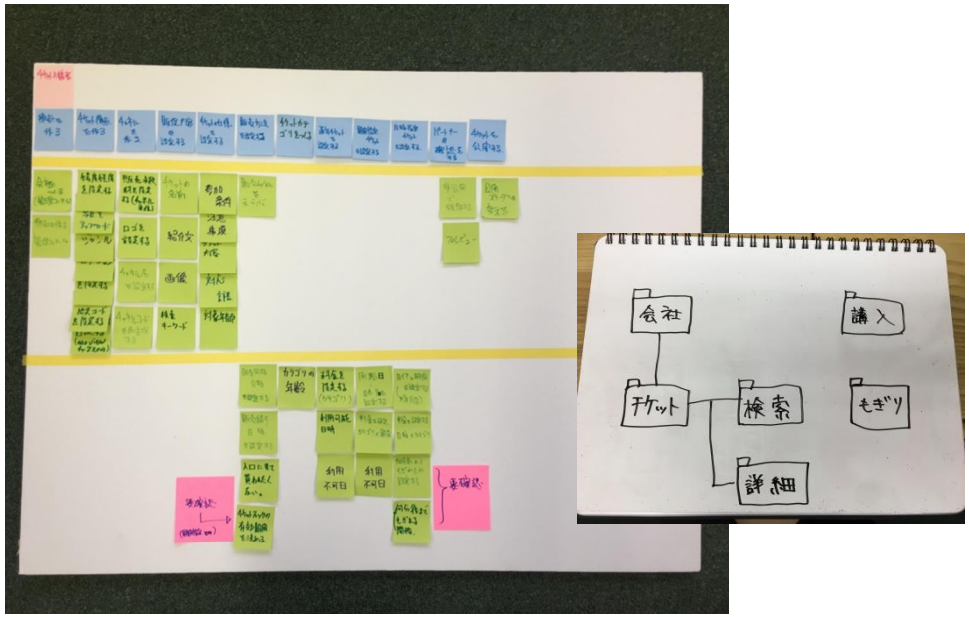
動いたらコードを見直す
業務の知識が増えたらコードに反映する

インクリメンタルな設計
スナップショット

コードを書きだすための準備



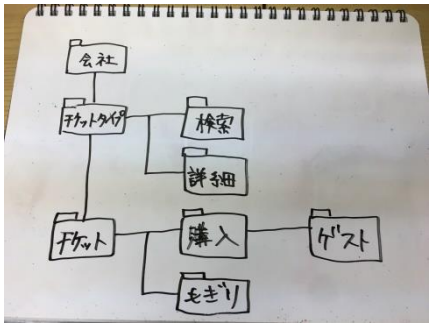
コンテキスト図と言葉の洗い出し



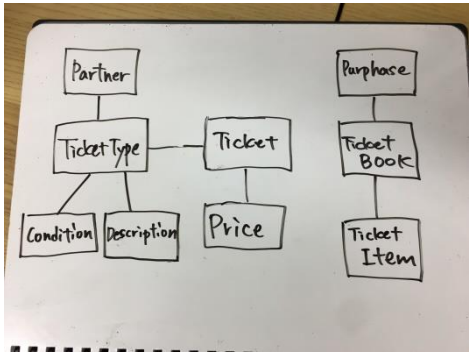
業務の流れ

主要な関心事の整理
パッケージ構成

コードを書くのが行き詰った時の会話



パッケージのスケッチ



クラスのスケッチ

インクリメンタルな設計

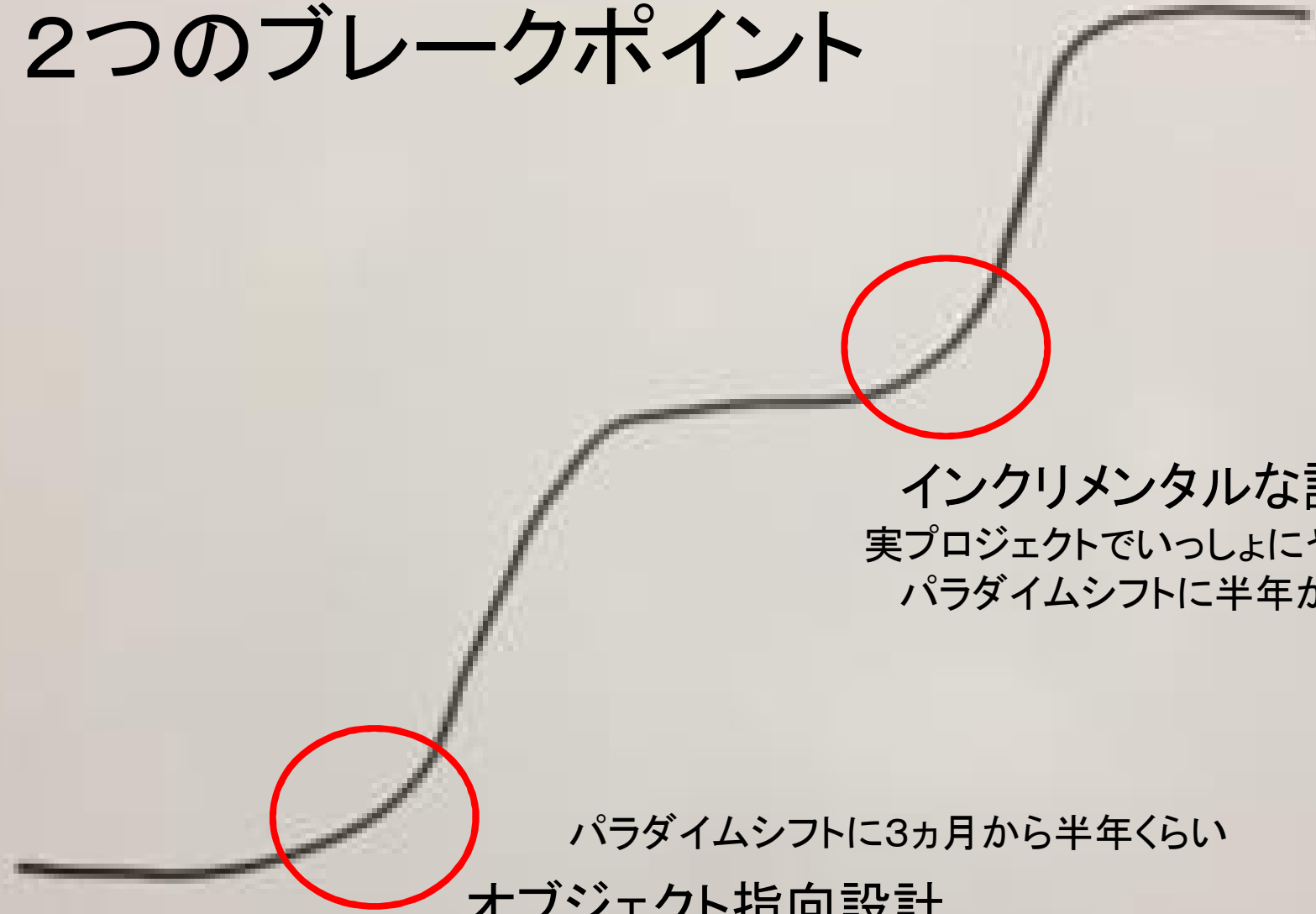
機能や画面を段階的に追加していくという意味ではない
（開発はそうやって進むが）

ソフトウェアの設計を「毎日」行う

名前の変更
メソッドの抽出
クラスの抽出
パッケージの抽出
フィールドの移動
メソッドの移動
クラスの移動

コードを書いて動かして得た知見をコードに反映する

ドメイン駆動設計
2つのブレークポイント



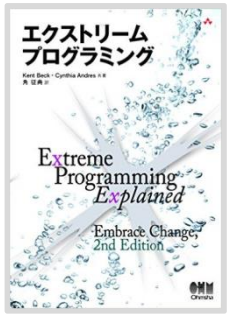
インクリメンタルな設計

実プロジェクトでいっしょにやりながら
パラダイムシフトに半年から数年

パラダイムシフトに3ヵ月から半年くらい

オブジェクト指向設計

変えるのは無理？



エクストリームプログラミングの
「はじめに」に記された
ケント・ベックのメッセージ

- どんな状況でも改善できる
- どんなときでも「あなた」から改善を始められる
- どんなときでも「今日」から改善を始められる