ざっくりわかる“XDDP”

派生開発カンファレンス2012

XDDPチュートリアル講演資料

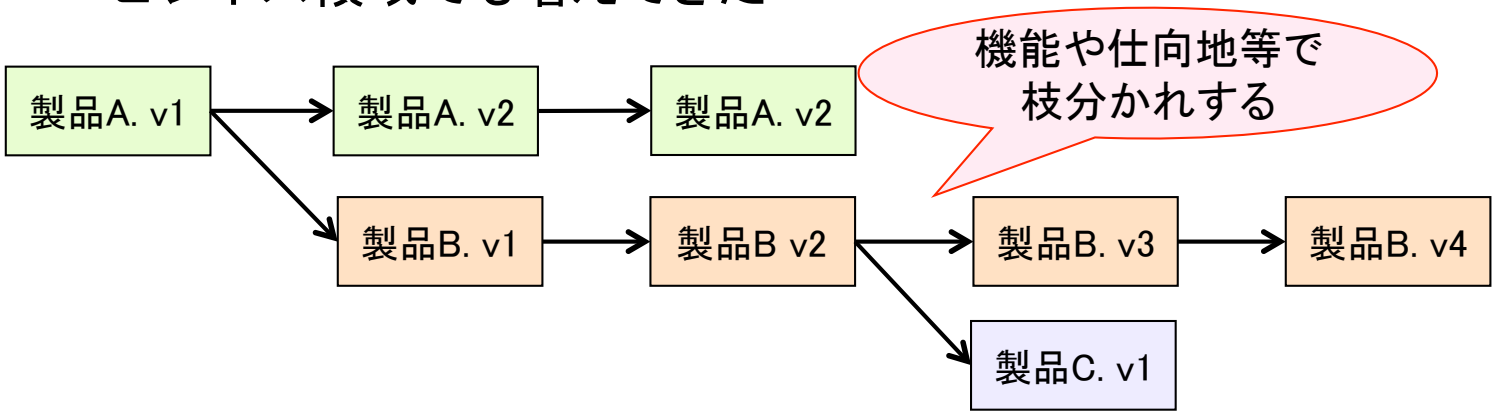
派生開発推進協議会
代表 清 水　吉 男
URL=www.xddp.jp

1.「派生開発」のおさらい

- 保守開発との違い
- バグの出方から見える派生開発の違い
- 「派生開発」の特徴
- 今のやり方のどこが問題なの？

保守開発との違い

- 保守開発・・・基本的に機能は現状維持または小さな変更
  - 不具合の訂正
  - 環境変化への対応・・・変更する箇所は概ね決まっている
  - 2008年に「機能追加」を扱うように変更されたが・・・
- 派生開発・・・機能は変化する
  - 保守開発の範囲（不具合訂正、環境変化対応）を含む訂正と変更
  - 競争のための新しい機能の追加や操作性等の変更
  - ビジネス領域でも増えてきた



保守開発との違い

- 変更依頼時に「要求仕様書」は書かれているか？

種類		依頼時の仕様	
機能追加		あり	追加したい機能の要求仕様書として書かれている
変更	追加を受け入れるための変更	なし	追加の担当者に任されている？
	本来の変更	あり	変更して欲しいところとして書かれている

- どこをどのように変更して、機能追加を受け入れるかは、担当者に任されている？
  - 既存機能が壊れていないことはテストされるが
  - 変更内容がわからない状態ではテストは不十分になる

バグの出方から見える派生開発の違い

- 新規開発と派生開発のバグの出方が違うことに気付いていますか？

種類		バグの姿
新規開発		仕様もれ：仕様に記述されていない 仕様の矛盾：仕様間で矛盾している 不適切な設計：設計が仕様を満たしていない （不適切なデータ構造など） 勘違い実装：実装時の方法が不適切
派生開発	機能追加	
	変更	変更箇所の漏れ、変更忘れ 間違い変更：依頼内容の勘違い／ソースコードの理解不十分 不適切な変更：変更によって品質が低下した 変更でここに影響がでることに気付かなかった

- 求められるレビューの観点も同じでは効果はでない

機能追加と変更と同じプロセスですか？

- 派生開発には、

機能追加
変　更
} の2つの異なった要求が含まれている

性質が異なる
↓
異なるプロセスで対応すべき

- 要求への対応方法
- レビューのタイミング／観点
- バグの性質も異なる

「派生開発」の特徴

- 新規開発と派生開発は異なる要求
  - 機能追加は新規開発に類似

種類		特徴
新規開発		• 要求仕様書に書かれていること（合意したこと）を実現する 　• 書かれていないことは実現しない • 精度の高い要求仕様書を作成する技術　＋　設計（アーキテクチャ設計を含む）・実装技術で実現できる 　• 機能追加には通常アーキテクチャ設計技術は不問
派生開発	機能追加	
	変更	• 変更を依頼された箇所を変更する 　• 他の機能の仕様と食い違いが生じる 　• 追加機能を受け入れるための変更で他の仕様を壊す • 依頼された変更箇所だけですまない 　• 影響を受けて変更しなければならない箇所がある • 依頼者が変更内容を正しく伝えているとはかぎらない

「部分理解」の中で変更作業が強いられる

① 変更箇所を見つけるための適切な文書が不足

② ソースコードの「読解技術」や「スペックアウト技術」が不足

全体を理解してから変更に取り掛かれる状況にない

「思い込み」「勘違い」が入り込む余地が大きい

本人は正しいと
思っている状態

このままソースコードを
変更ればバグになる

表現する

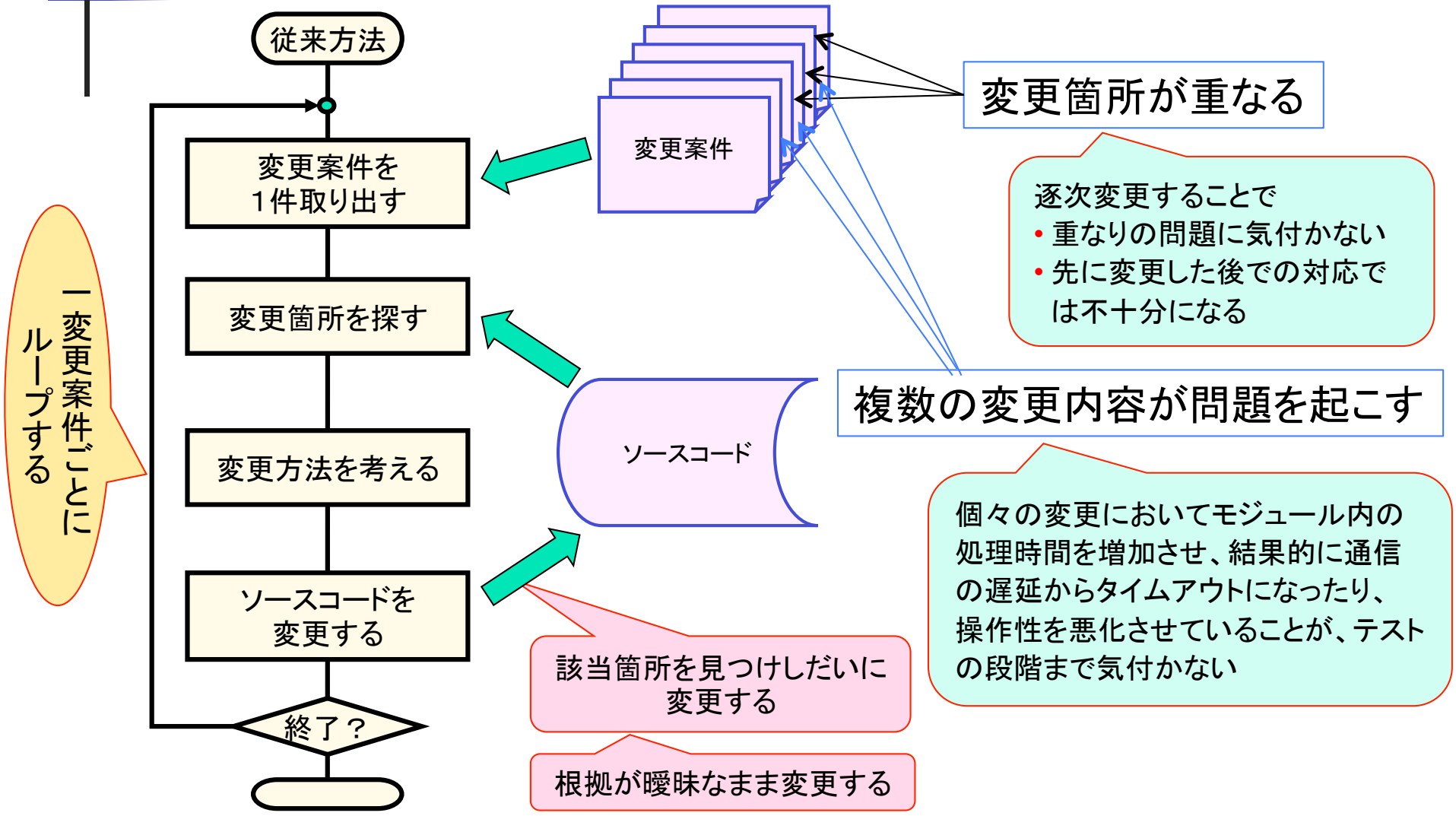
レビューする

担当者の理解の状況
を早い段階に検証する
仕組みが必要

2. 今のやり方のどこが問題なの？

- 現行の作業のやり方は？
- テストでバグが見つかったら修正すればいい？
- あとでより良い変更箇所が見つかったときは？
- 変更の記録が残っていないことの影響は？

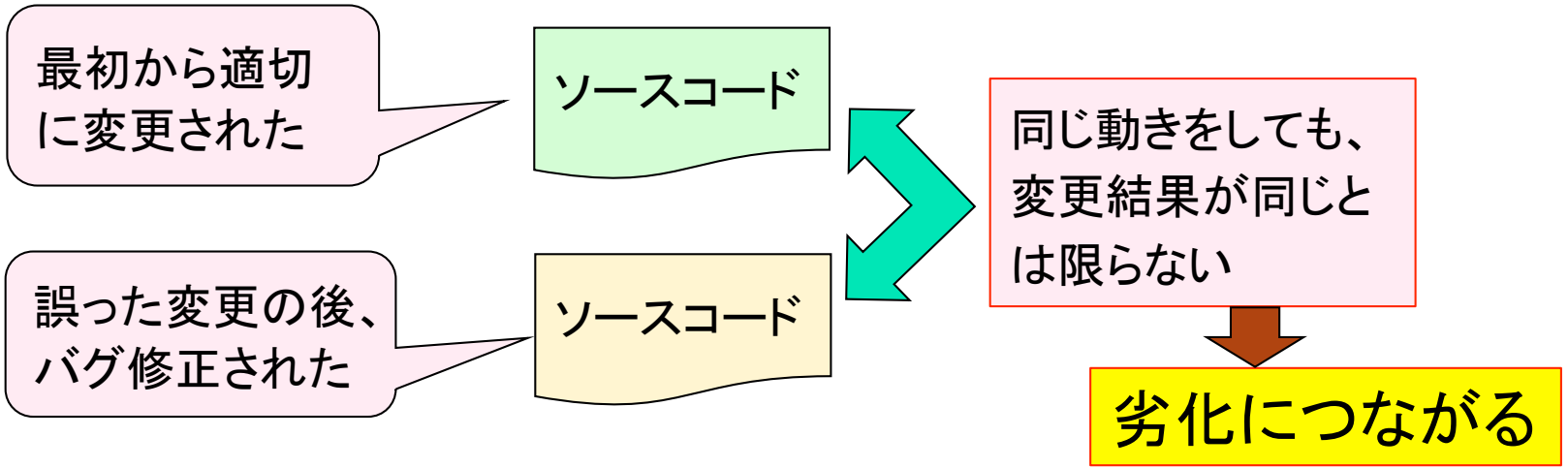
現行の作業のやり方は？



テストでバグが見つかったら修正すればいい？

- 安易な考え方でソースコードを変更しているので、次々とバグが発生する
- バグを発生させたのと同じプロセスでバグを修正している

テストでバグを出しきれないまま出荷し、問題を大きくする



あとでより良い変更箇所が見つかったときは？

逐次、ソースコードを変更していくなかで、後になって、
より良い変更方法に気付くことがある

- この時、変更方法を変更することができますか？

(ケース1) 先の変更方法が間違えていた → 正しい変更方法に変更し直す

(ケース2) 先の変更方法でも間違いではない → 変更方法を変更しない

「気付かなかったことにする」ことの問題の重要性
(人格の毀損)に気付いて欲しい

変更の記録が残っていないことの影響は？

派生開発の特徴・・・
ベースのソースコードは既にテストされていて稼働している

- バグは、今回変更した箇所に絡んで発生している
  - バグの対応（デバッグ）が新規開発と同じになっている
- 変更の記録（場所、理由を含む）があれば，原因究明が短時間で可能
  - ソースコードに変更日付を入れるだけでは役に立たない

3.「XDDP」ってどんな方法なの？

- XDDPの誕生に「XDDP」」の本質がある
- 追加機能要求仕様書について
- 派生開発に特化したプロセス
- 事前調査のやり方に注意
- 変更要求仕様を作成する
- 変更設計書を作成する
- ソースコードは一気に変更する

「XDDP」の誕生に本質がある

４０数項目の機能追加と変更を３ヶ月で！

- アメリカの顧客からの要求（1978年）
  - 初めてのドメイン、初めて見るソースコード、言語も初
  - 国内の客先には、その製品の仕様を知る人はいない

「保守のプロセスでは不可能」と判断

- １週間で、新しいプロセスを設計してシミュレーションで確認
  - 機能追加と変更の２種類のプロセスに分けて品質と生産性を確保

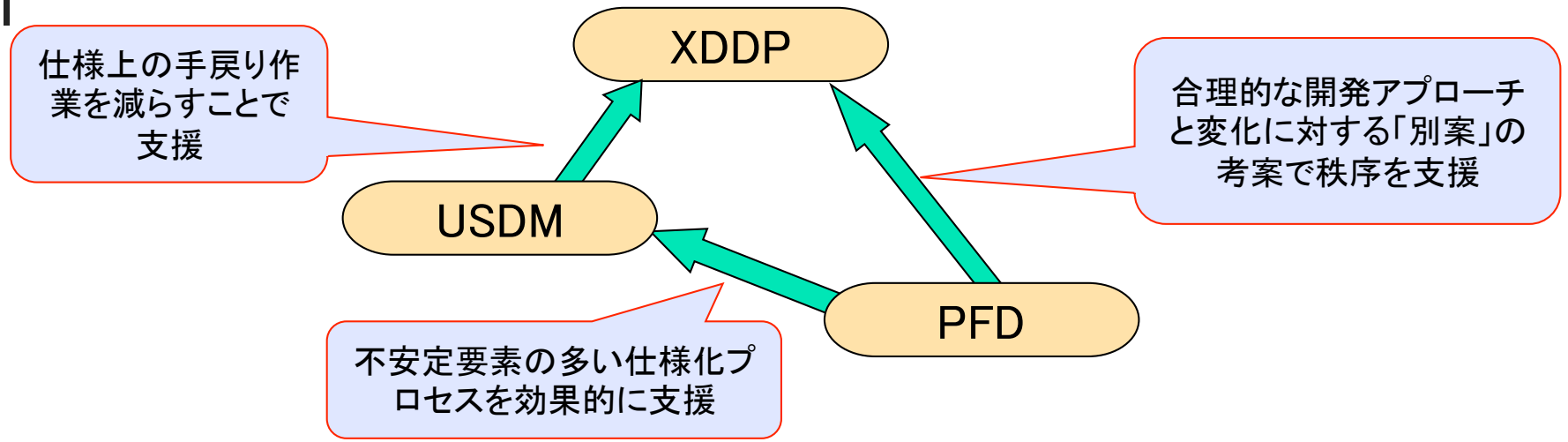
品質	変更箇所（理解した仕様を含む）を「before/after」で記述し、Faxでレビューを依頼
	「before」は、現状の仕様を私が理解した状態 → レビューで確認
生産性	必要不可欠な最小限の成果物とプロセスの連鎖で構成（by DFD）

３ヶ月の納期を達成

XDDPトライアングル

XDDPを考え出したときは、既にUSDMもPFD(DFD)も習得済み

- 「XDDP」は「USDM」と「PFD」の支援の上で成り立っている

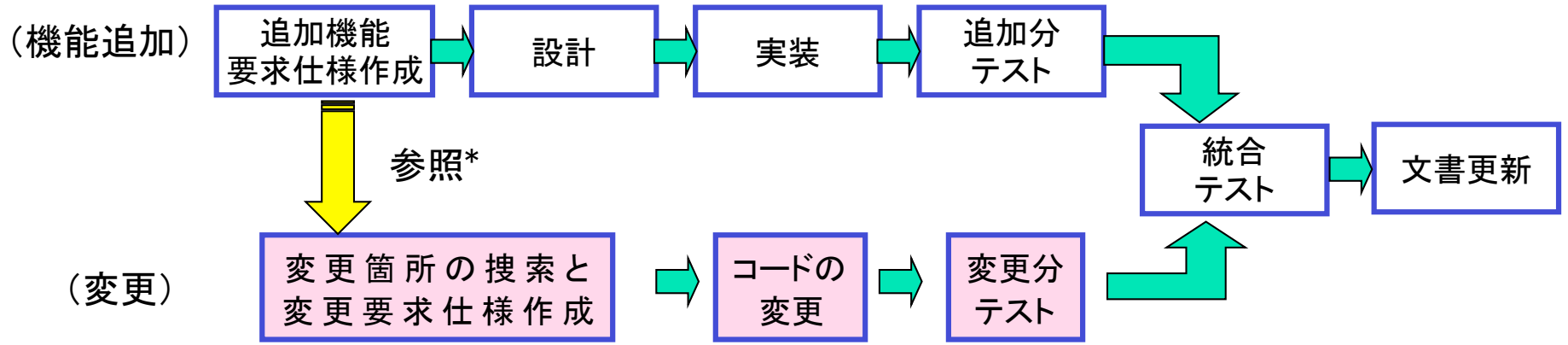


- 「USDM」
  - 追加機能の仕様モレを減らしたり、変更箇所（変更仕様）の抽出モレを減らすことで、短納期などの制約の中での開発作業を支援する
- 「PFD」
  - ムダのない合理的な開発アプローチを設計し「計画書」に繋げる
  - 途中で生じる変化に対して適切にプロセスと成果物を調整する

Page 16

派生開発に特化したプロセス

- 「XDDP」は機能追加と変更を異なるプロセスで対応する



今日は、
以下、変更の
プロセスについ
て説明します

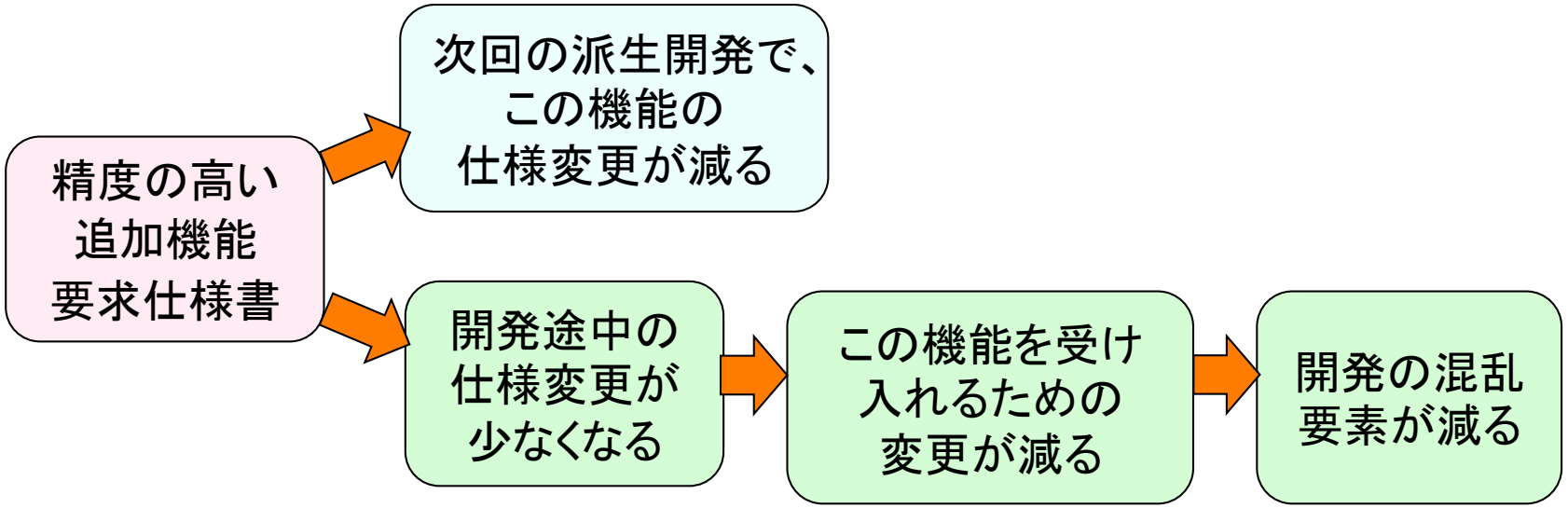
- 「追加」と「変更」では、要求の性質が違うためにプロセスを分ける
- 「変更」は「差分」で進める
- 公式文書の更新はテスト後（後半）に行う

＊追加機能を受け入れるための変更方法を探すために参照する

追加機能要求仕様書について

- 「USDM」を導入することで精度の高い要求仕様書を書く

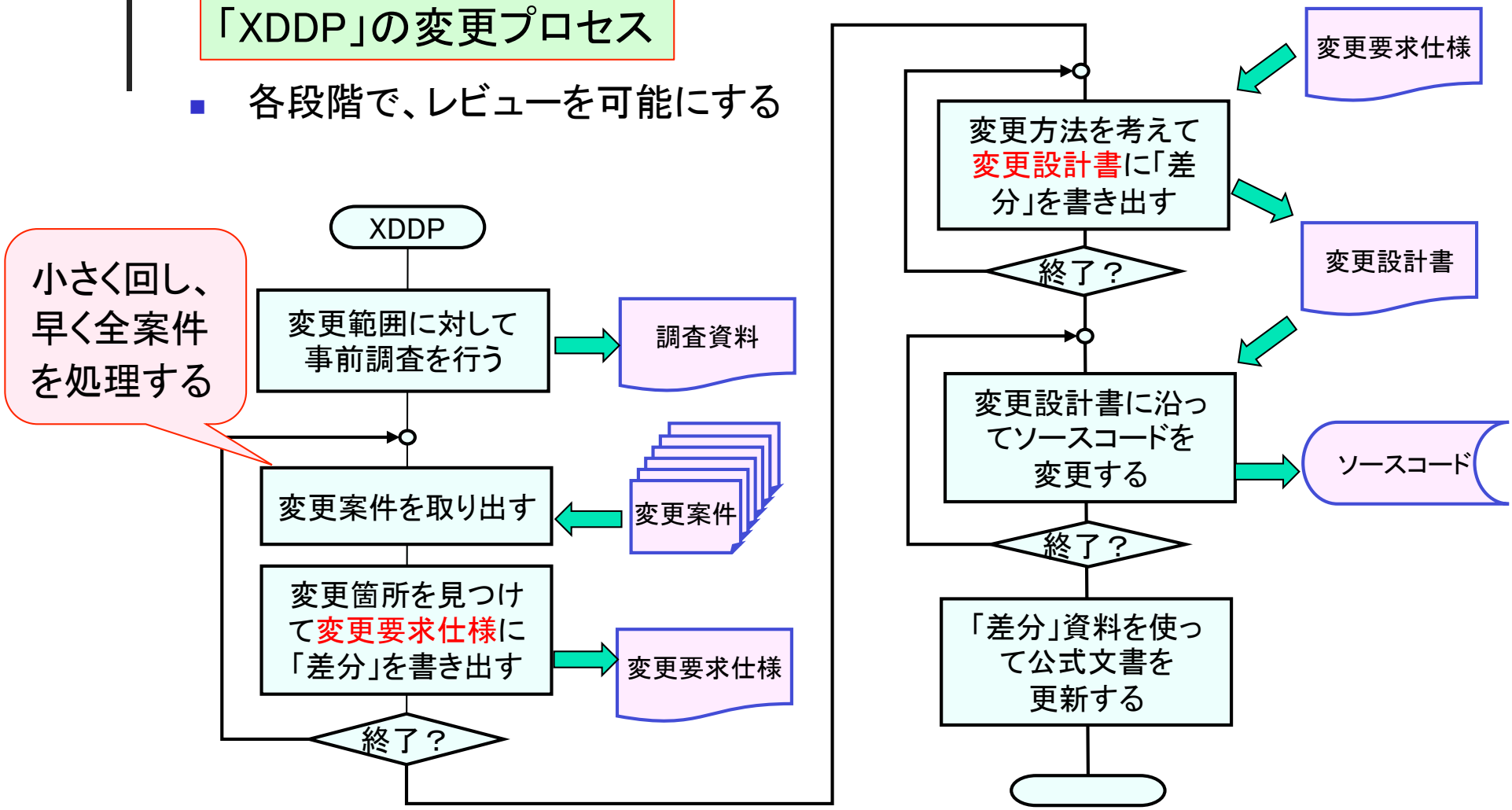
「USDM」では、「設計しながら仕様を抽出する」という考え方を指示しない



派生開発に特化したプロセス

「XDDP」の変更プロセス

- 各段階で、レビューを可能にする



Page 19

事前調査のやり方に注意

- 注意・・・ある範囲を「理解」するために調査するだけ
  - 変更箇所を探すための調査ではない
  - 調査資料は、現在バージョンのベースの設計書を補うもの
- 処理構造は、ある深さのレベルで止める
  - あとで変更箇所を探す時にどこを探せば良いかを判断するために、どこでどういう処理が行われているかが分かればよい
  - 「構造図」の表現に工夫

- 事前調査で起きる問題
  - 一般に歯止めがないために時間を使い過ぎる
  - 「探検隊」・・・調べていると次々と新しい（未知の）洞窟に入り込む
  - 成果物・・・後で変更箇所を探すときに、ここで調査したことが役に立っているか？

変更要求仕様を作成する(1)

- 変更を「変更要求」と「変更仕様」の階層で表現する
  - 変更要求
    - その変更の意図するところを表現する＝変更の本質
  - 変更仕様って？
    - 関数仕様の記述レベル・・・ほとんどソースコードに近いケースもある

変更要求	KUB01	データ区分を3種類から5種類に増やして欲しい	
	理由	扱う商品が増えたことで3区分では不足してきた	
	変更仕様	KUB01-1	現在データ区分の1～3に対して8と9を追加する
	変更仕様	KUB01-2	D1画面とD4画面の区分表示欄を3行から5行に増やす
	変更仕様	KUB01-3	区分8と9を表示欄の4行目と5行目に対応させる処理を追加する
変更要求	DOR01	ドアの開閉ボタン操作の後すぐに動作しないで，少し「間」を置いて欲しい	
	理由	すぐに反応されて慌ててしまう	
	変更仕様	DOR01-1	ドアの開閉ボタン押下検出後、動作開始前にを200ms遅延処理を追加

変更要求仕様を作成する (2)

- 変更要求を階層で捉えることのメリット
  - 本来の依頼されている変更と、その変更によって影響を受けた変更を一つの「変更要求」の下でまとめることができる

変更要求	KUB01	データ区分を3種類から5種類に増やして欲しい	
	理由	扱う商品が増えたことで3区分では不足してきた	本来の変更
	＜区分の追加＞		
	変更仕様	KUB01-1	現在データ区分の1～3に対して8と9を追加する
	変更仕様	KUB01-2	D1画面とD4画面の区分表示欄を3行から5行に増やす
	変更仕様	KUB01-3	区分8と9を表示欄の4行目と5行目に対応させる処理を追加する
	＜D1画面の調整＞		
	変更仕様	KUB01-5	D1画面の〇〇〇を下に3行分下げる 【説明】 1行分は見やすさを確保するため

影響を受けた変更

- これによって、ソフトウェアシステムの様子が見える
  - 影響が拡散する？

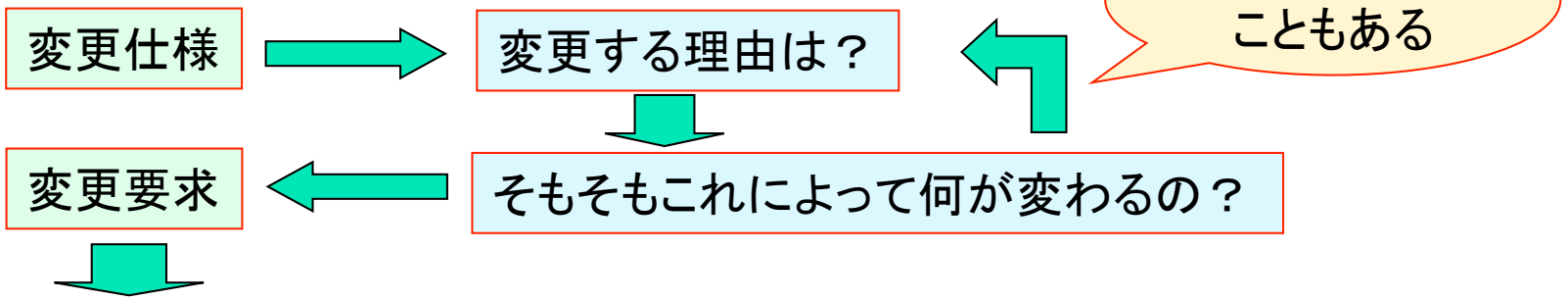
変更要求仕様を作成する (3)

- ほとんどの変更依頼は「仕様レベル」で届く

	□ □ □		防犯カメラの首振り角度を45° から60° に変更して欲しい

  - 多くの担当者は「楽勝」という反応を見せるが、このまま変更すると、変更箇所が不足することになる

- 「昇華」・・・変更仕様から変更要求を捉える
  - 変更仕様から変更要求を捉え直す



要求	撮影範囲を30%拡大し，往復時間は従来と同じにするために動作スピードをアップさせてほしい
理由	モニターも含めてトータルの設置コストを下げて販売に繋げたい

変更要求仕様を作成する(4)

- 2種類の「変更仕様」のタイプ

現状の仕様に対する変更	•依頼されている
追加機能を受け入れるための変更	•必ずしも依頼されていない •追加機能要求仕様から変更箇所を探す

受け入れ方法をレビュー
できるように記述する

- 「変更仕様」の探し方

機能仕様書や設計書から見つける	•機能仕様書や設計書が丁寧に作られている状況で可能になる •後でソースコード上の「場所」を特定する
ソースコードを解析しながら見つける	•設計書などの文書が不十分 •スペックアウトしながら変更箇所を探す

変更要求仕様を作成する (5)

- 変更は「before / after」で記述する

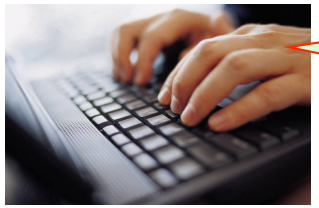
  「before」の状態を「after」に変更する

  - before・・・担当者の現状の認識を記述する
    - 担当者が，ソースコードをどのように読み取ったか？
  - after・・・変更したい状態
  - 「追加する」「削除する」は、それ自体に「before / after」を含む

大事なことは、
現状の どのような仕様をどのように変化させたい
のかを伝えること

変更要求仕様を作成する(6)

- 文章表現するのは苦手で・・・
  - 現状のやり方でも、該当箇所を見つけたときはこれと同じことをイメージしているはず



ここが区分を判定しているところ
だから、この判断の前に新しい
条件を追加して・・・っと

    - ただし一瞬後には変更方法を考えているので、ここを見つけたときの根拠などの記憶は薄れる
  - 理解が曖昧だと文章にならない
  - 文章で表現することで
    - 曖昧な状態での変更を思いとどまらせる
    - 他に変更するところに気付くことがある

変更要求仕様を作成する（7）

- ソースコードの場所（関数名）は「TM」に記述する
  - TM : Traceability Matrix
    - 「TM」に記載したところでいろいろな問題（間違い）に気付く
    - ここに変更箇所があるのなら「〇〇〇」にもあるのでは？
- 「TM」の「列」情報としてはソースファイルに限定しない
  - ビルドのパラメータ
  - 設計書等の文書　など

書き手が問題に気付く
文章は、レビューアも
問題に気付きやすい

				file	file	file
要求	CCL30	加入者データに家族データを追加して家族割りサービスを始める				
	理由	同業他社との競争に勝つため				
		＜加入者データの表示の変更＞				
	□ □ □	CCL30-05	加入者名の横に主従区分の表示を追加する			f3()
	□ □ □	CCL30-06	主従区分＝主の時は、その横に家族の加入者番号を登録数分表示する		f7()	
	□ □ □	CCL30-07	主従区分＝従の時は、主となる加入者番号を表示する		f7()	

変更要求仕様を作成する (8)

- スペックアウトで変更箇所を見つける
  - この段階でソースコードを読んで調査する作業を「スペックアウト」と呼び、その際に生成する文書を「スペックアウト資料」と呼ぶ
    - 事前調査に続けて調査することもある
    - この段階では，まだ変更を織り込まないことが大事
  - ソースコードの「読解技術」が必要
    - ソフトウェアエンジニアリングの技術を逆流用する

- 変更の性質によって調査内容や調査方法が異なる
  - 処理の変更
  - データの変更（構造の変更やデータの取る要素の変更）
  - グローバルデータの変更

変更要求仕様を作成する (9)

- スペックアウト文書の扱い
  - この段階では、変更前のベースの公式文書を補完するもの
  - 「現状」を正しく表現する
  - スペックアウト中に、変更箇所を発見したときは変更要求仕様書に記述する
    - この時点では、スペックアウトした資料は原則として変更しない
    - 変更を反映して確かめたいとき・・コピーしたものに反映して確認する
    - 理由＝その変更が適切な対応とは限らないし、他に変更が重なる？
    - 変更を盛り込んでしまえば他の変更方法に気付かなくなる

- 変更箇所を全部見つけたと思われるところで、スペックアウト作業を止める
  - 理由＝スペックアウトが目的ではない

変更要求仕様を作成する (10)

- 機能追加を受け入れるための変更も扱う
  - 追加機能を動作させるために
    - ① 現状の操作画面への部品の追加や、
    - ② 追加したボタンの操作時の処理を追加する必要がある

追加機能を呼び出すために内部処理で条件判断の追加
機能追加に伴って増えたタスクの負荷を平準化するために，既存の処理の一部を他のタスクに移動する変更

通常は、追加機能の「要求仕様」と、
これを受け入れるための「変更要求仕様」が対応する

変更要求仕様を作成する(11)

- より効果的な変更方法に気付いたときは乗り換える

他の変更箇所を探しているときに、既に終わった変更案件に対してより良い変更方法に気付くことがある

- 「XDDP」では、この段階ではソースコードを変更していないので，変更要求仕様を書き換えるだけ
- 適切な変更箇所(方法)を選択できることで人格の毀損を免れる

見つけしだいにソースコードを変更しながら作業を進めたときは、より良い方法に乗り換えるのは難しい

変更要求仕様を作成する(12)

- いったん変更要求仕様でレビューする
  - すべての変更箇所を網羅しているか？
    - 現実には，変更箇所を拾いきっていると判断できることが多い
  - 変更の意図を勘違いしていないか？
  - 変更箇所やタイミングはそこで良いのか？
  - 同じような変更が必要な箇所はないのか？
  - この変更によって既存の処理条件が変化しないか？
  - この変更によって使い勝手が悪くなることはないのか？

- 変更を文章で書いている
- TMの情報がある

など

この段階のレビューで、変更の間違いの多くが発見される

変更設計書を作成する(1)

- 変更要求仕様毎に「TM」の交点に対して作成する
  - 具体的な変更方法を記述する
  - この時点で「関数名」が特定されている
    - ドア開閉処理(関数名)の先頭で「200ms」のディレータイマの呼出し処理を追加する
  - 関数の外の定義を変更することもある
    - データ区分に
      DATA-L(8)と
      DATA-MAX(9)を追加する

ベースの「関数設計書」の変更情報＝「差分」

(TMのセル情報)
変更の考え方
　　特殊な考えで変更する場合
構造の変更
　　データ構造
　　処理構造
関数外の変更
　　定義、マクロなどの変更
関数内の変更
　　具体的に
単体テストの内容
　　変更箇所の確認のため

変更設計書を作成する (2)

- 関数の特定まで合っていたのに，不適切な変更をしてしまうこともある
  - 区分として「定義」すべきところを定数を使って処理した
  - 分岐した先で、別の変数に対して処理していたところを見落とした
  - 不用意な変更でモジュールの尺度を悪化させてしまう

変更要求仕様の記述粒度が粗い

→ 変更設計書に異なる関数の処理が混じりやすくなる

↓

変更漏れなどのミスを招く

必要な情報だけを簡潔に書く

変更設計書を作成する (3)

- どのように書けば良いのか？
  - 2つの使用目的に適合させる
    - ① これを見てソースコードの変更がスムースに進むこと
    - ② テスト後に，これを使って公式文書を更新できること
  - この変更設計書の記述を見ることで，ソースコードの変更作業が立ち止まることなく進むことを目指す
    - 目標＝80行～100行／1時間

別の人がソースコードを変更する場合は、
記述を少し丁寧にする必要がある

変更設計書を作成する (4)

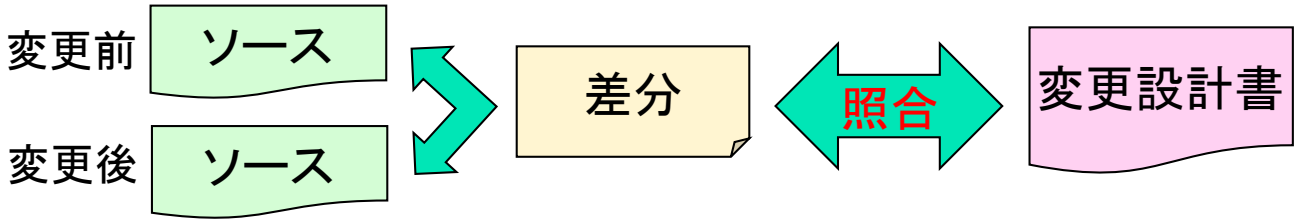
- 変更設計書のレビュー
  - 変更要求仕様書で、変更の意味や目的および「理由」に対して、個々の変更仕様が具体的変更内容としてレビュー済み
  - 後は、具体的な変更方法が適切かどうかを検証する
    - 変更の意図と一致しているか？
    - 他に関数内で変更箇所を見落としていないか？
    - その変更ではソースコードを劣化させないか？
  - 変更設計書が多いときは、レビューは選択してもよい
    - ただし、バグとの関連を判断するために、レビュー「した／しなかった」状況が記録されること
    - 変更要求仕様 あるいは TM 上に記録する

一気にソースコードを変更する (1)

- 変更設計書に基づいて一気にソースコードを変更する
  - 抜け駆け変更を許すと、後の人に変更を押し付けることも生じる
  - 実装工程の生産性が示すこと

生産性が高い	変更設計書に必要な情報が拾いきれている
生産性が低い	変更のための情報が不足していて、変更しながら考えている. この時、立ち止まって考えたことは変更設計書には記述されない

- 変更漏れが起きないように
  - 1枚の変更設計書に異なる関数の変更を扱うと漏れやすい
  - こまめに「✔」をつける
  - 最後に「ソースコードの差分」と照合する



一気にソースコードを変更する (2)

- この時、ソースコードの変更作業が立ち止まることがある
  - 原因＝変更設計書の記述情報が不足している
  - このままソースコードの変更を続けても時間を失うだけ
- 現状＝立ち止まって考えている時間・・・何も記述されない

ソースコードの変更を見合わせて、もう一度「変更設計書」の記述を見直し，不足する情報を補う

生産性の差	必要時間
生産性＝100行 / 時間	10000行の変更 → 100時間で変更できる
生産性＝ 30行 / 時間に低下	10000行の変更 → 330時間かかる
	変更設計書の補充に 200時間投入し、 100行 / 時間の生産性を出せば 300時間で可能

公式文書はテストで確認されたあとで行う

- テスト後に、「差分情報」を使って公式文書をマージする
  - 一般に、テストの後半から可能になる

公式成果物	変更要求仕様書	変更設計書
機能仕様書	○	
画面操作仕様書	○	
制御仕様書	○	
各段階の設計書（仕様書）	○	
関数仕様書	○	○
関数設計書		○
データ仕様／設計書	○	○

- ここまで公式文書を更新しない理由
  - テストでバグが出たことで、変更内容や変更方法が変わることがある
  - 当初想定した変更では問題が判明し，別の変更方法に切り替えることがある

4. どんな場面で使えて、どんな効果があるの？

- 小規模システムの変更案件から対応するとよい
- バグの対応にも使おう
- 慣れてくると変更規模の大きいプロジェクトに適用できる
- １時間あたりのコード生産性が低い状況に有効
- 変更作業が見積もれる
- 変更のエビデンスが残る

小規模システムの変更案件から対応するとよい

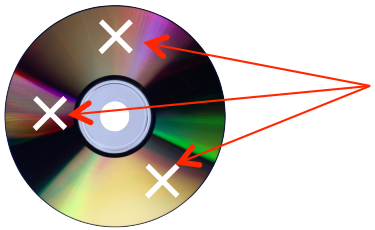
- 最初は少ないメンバーで始める

規模	様子
一人プロジェクト	✧ 最初は少ないメンバーで始める ✧ お互いの守備範囲の変更情報を見せ合う ✧ レビューは組織で対応する（特に経験者が関与する）
変更量の少ないプロジェクト	
数名のプロジェクト	

- 組織の足並みを揃える必要はない
  - 足並みが揃うことに拘ると、動きの遅い人や悪い人に引っ張られる

準備ができた人から始めればよい

バグの対応にも使おう



バグ → この後、バグ修正の作業を行う

バグを出したプロセスと同じプロセスで対応している

- ここと思った箇所（＝「思い込み」と「勘違い」のまま）を、いきなり修正する

バグが繰り返される

- 対応策
  - バグへの対応に「XDDP」を適用して
  - バグの連鎖を断ち切る

ただし、バグが多いときは、選択したり何らかの工夫が必要

変更規模の大きいプロジェクトにも適用できる

- 「XDDP」を習得した人が増えてくれば、変更規模の大きいプロジェクトにも適用できる
  - お互いの変更の「手の内」を見せ合うことで効果がでる
  - 「TM」の構成を工夫する
  - 適当な「ツール」を活用する
- チーム（メンバー）の足並みが揃っていなくてもよい
  - XDDPで対応したチームの結果が良いことが見えるはず
  - 習得の状態に応じた結果がでる

従来のような結果が出ることはない

１時間あたりのコード生産性が低い状況に有効

- 従来の変更方法ではコード生産性が低くなる

生産性を下げている行為	
個々の変更作業の中で変更の意味を顧客に確認している	１時間あたりのソースコードの変更行数が低くなる **変更行数 ≦ 10**
見つけ次第にソースコードを変更	
確認のために何度も変更した箇所を読み返す ただし、そこは既に変更された状態にある	
不適切に変更したことに気付いて変更をやり直す	

- 「XDDP」では・・・
  - 変更の依頼内容を正しく捉え直す →変更要求
  - 見つけた変更箇所を書き出す →変更仕様
  - 思い出すときはソースコードではなく変更要求仕様を見る
  - ソースコードの変更は「１回」で完了 →劣化防止
- その結果、著しい効果がでることもある

変更作業が見積もれる

- 各段階でソースコードの変更行数を見積もる（再見積り）
- 「3段見積り」＝初期見積り、再見積り、実績

段階	見積りの様子
変更要求の段階	•その変更でどれぐらいの行数の変更になるかを見積もる •見積りの確度が低い項目から着手
変更仕様の段階	•変更箇所を当たって行くなかで、変更する行数が見えてくる •当初見積りよりオーバーするかどうかを監視
変更設計書の段階	•最終的に変更方法を把握するので、非常に高い見積り精度になっている

- 各段階を経ることで見積りの精度が向上する

変更のエビデンスが残る

- 「変更要求ー変更仕様」の階層構造の効果
  - 変更要求
    - 変更の「意図」するところや「理由」を表現
  - 変更仕様
    - 変更要求を満たすために変更する箇所を集約
    - 影響を受けて変更する必要が生じた箇所も集約

変更の意図が正しく把握されているかどうかが見える

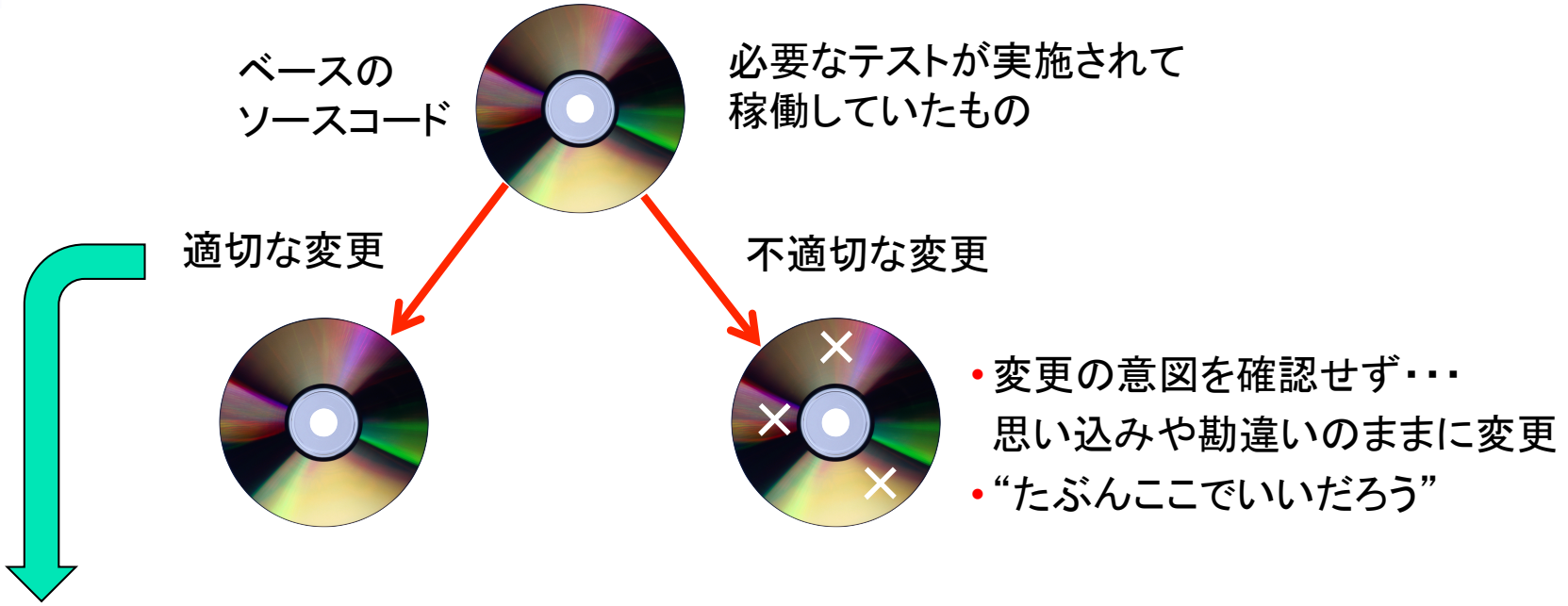
- 「変更３点セット」の効果
  - 「変更要求仕様」
    - 何を変更するかを記述する
  - 「TM」
    - 変更がどのモジュールに及んでいるかを示す
  - 「変更設計書」
    - 具体的にどのように変更したのかを記述する

変更がモレなく実施されたかどうかが見える

5. 取り組みは難しいの？

- きちんと変更すれば混乱しない
- 見つけた変更箇所を書き留めるだけ
- なぜ、公式文書の更新を後にするのか？
- 「差分」だけでできるのか？

きちんと変更すれば混乱しない



- ベースの理解
- 変更依頼の内容を理解
- 変更箇所を整然と探索

- 変更箇所を「３点セット」で記述し
- 組織をあげてレビューする

初めての外注SEの成果が示す意味

- 開発途中で「3名のリソース不足」の申し出あり　→別外注先から1名確保を指示
- 外注の担当者は「5年目」
- 始めて・・「派生開発」そのものが初体験
- 準備・・・・「XDDP」の学習、製品の機能やソースコードの構成の把握に「3週間」
- 作業内容・・・複数のモデルのソースコードの合体（流用）＋仕様変更
- レビューには発注側の経験者（1名）が参加
- 期間2．5ヶ月で、予定の期限内に完成
- 外注担当者の責任のバグ＝0件（結合テスト以降）

	変更（全）	削除	生産性	
変更行数	7000行	1500行	140行/H	110行/H
実装工数	50H		全体	削除行除外

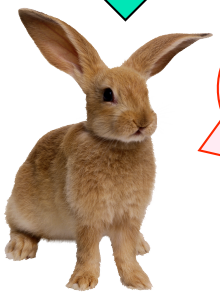
最高の結果が出た理由は・・・？

なぜ、公式文書の更新を後にするのか？

- 初期の段階・・・変更内容が不安定
  - その変更が適切とは限らない
  - 他にもっと良い変更方法が見つかることがある
  - 工数不足などから途中でその変更を中止することもある
  - 公式文書には変更対象の仕様が必ずしも記述されていない
- 変更を反映してしまうと、反映した箇所は変更できるが、他に影響する箇所が見えない
- 複数の担当者（チーム）が並行して作業ができる
- テスト作業と重ねることで、公式文書の更新作業をテストの裏に隠すことができる

「差分」だけでできるのか？

- 今まで・・・
  変更を反映した文書を見ていた
- でも・・・
  変更漏れを繰り返してきた



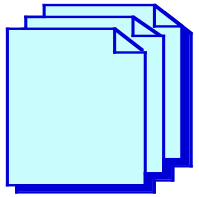
差分だけでは
変更後の様子が
見えないヨ～

公式文書と
両方見てネ

公式文書

ベースの
ソースコード

＋



今回変更分
（差分）

原則として、これで作業を進める

おまけ

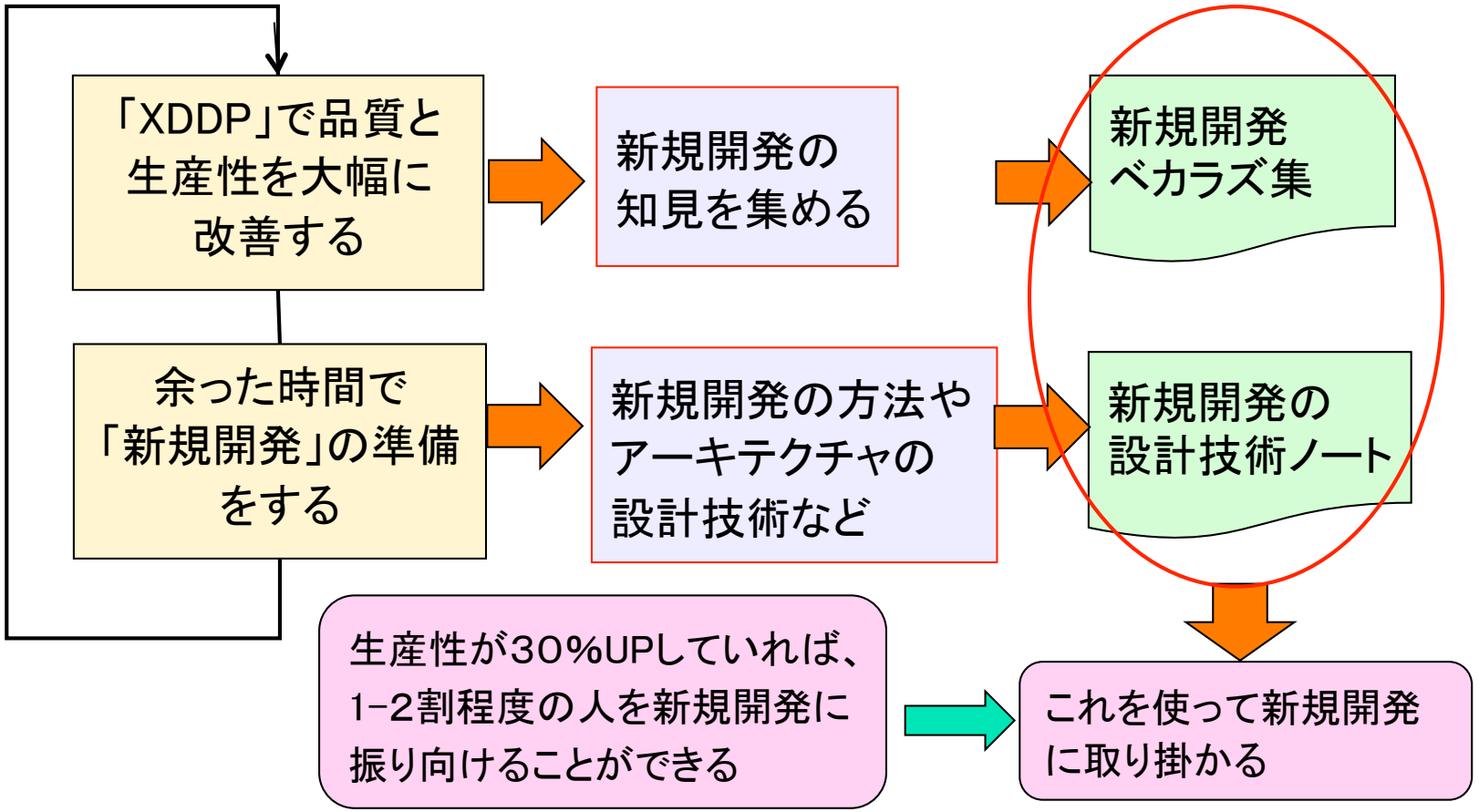
派生開発の技術

新規開発の技術

一体で手に入れておかないと
競争に勝てない

「次」への準備

- 「XDDP」を活用して派生開発の混乱を鎮め、「次」の準備を並行させる



（参考）XDDPトライアングルの展開

- 「XDDP」の3つの技術は、それぞれいろいろな取り組みに展開する