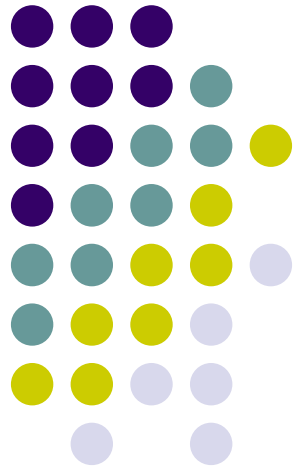
デベロッパーテスティング
～ソフトウエア開発者の基礎体力

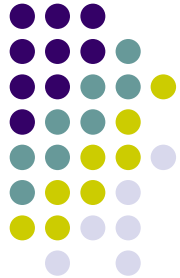


タワーズ・クエスト株式会社
プログラマ 兼 取締役社長
和田 卓人

2006年6月29日
@オブジェクト倶楽部2006夏イベント

自己紹介の前に

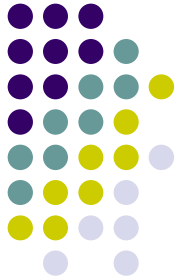
- 4年前の出来事
  - ワールドカップ
    - イングランド vs. アルゼンチンの日
  - IBMのイベント
    - テスト関係のセッション
- そこで出会った人は…



感謝

故 石井勝さん
(a.k.a masarlさん)

自己紹介



- 名前:　和田 卓人(わだ たくと)
- ブログ: http://d.hatena.ne.jp/t-wada/
- メール: takuto.wada@gmail.com

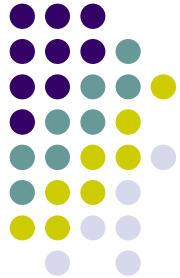
- タワーズクエストという会社を経営しています
  - 一緒に仕事する仲間を募集中!

- チームかくたに
  - XPのコーチ
  - XPで一年半開発を行う

(ちょっと宣伝) Life Hacks PRESS

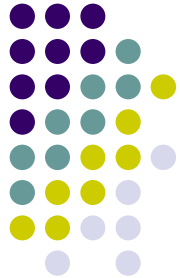
Life Hacks PRESS
B5判、168ページ
定価1,596円（税込）
ISBN 4-7741-2728-0
3月24日発売

さて、Developer Testing

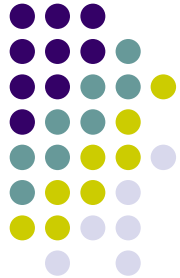


- 今日のお題「デベロッパーテスティング」
- 皆さんご存知ですか?

アジェンダ

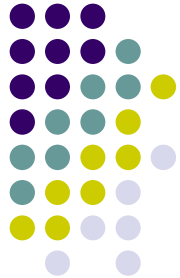


- テストの分類
- Developer Testingとは何か
- Developer Testingの目的
- Developer Testingのバリエーション
- Developer Testingを習得するには
- テスト駆動開発とは何か



序章:テストの分類

難しくなったソフトウェア開発

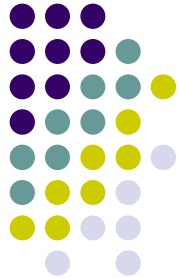


- ソフトウェアの複雑化
- ソフトウェアの大規模化
- ソフトウェアの短納期化
- オブジェクト指向もそれなりに使われだした

→ソフトウェアの品質が問題に(シンドラー社とか)

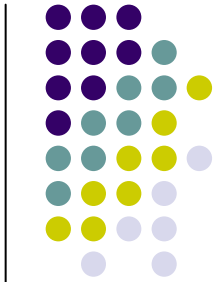
→「テスト」が注目されてきた

テスト分類の混乱



- 単体、ユニット、結合、機能、システム、…
- 単体テストとユニットテストは同じもの?
  - 単体って?
  - ユニットって?
- 結合テストは何を結合するの?
- 品質保証? 動作確認?

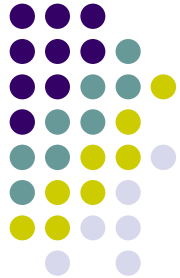
テストの目的に戻ろう



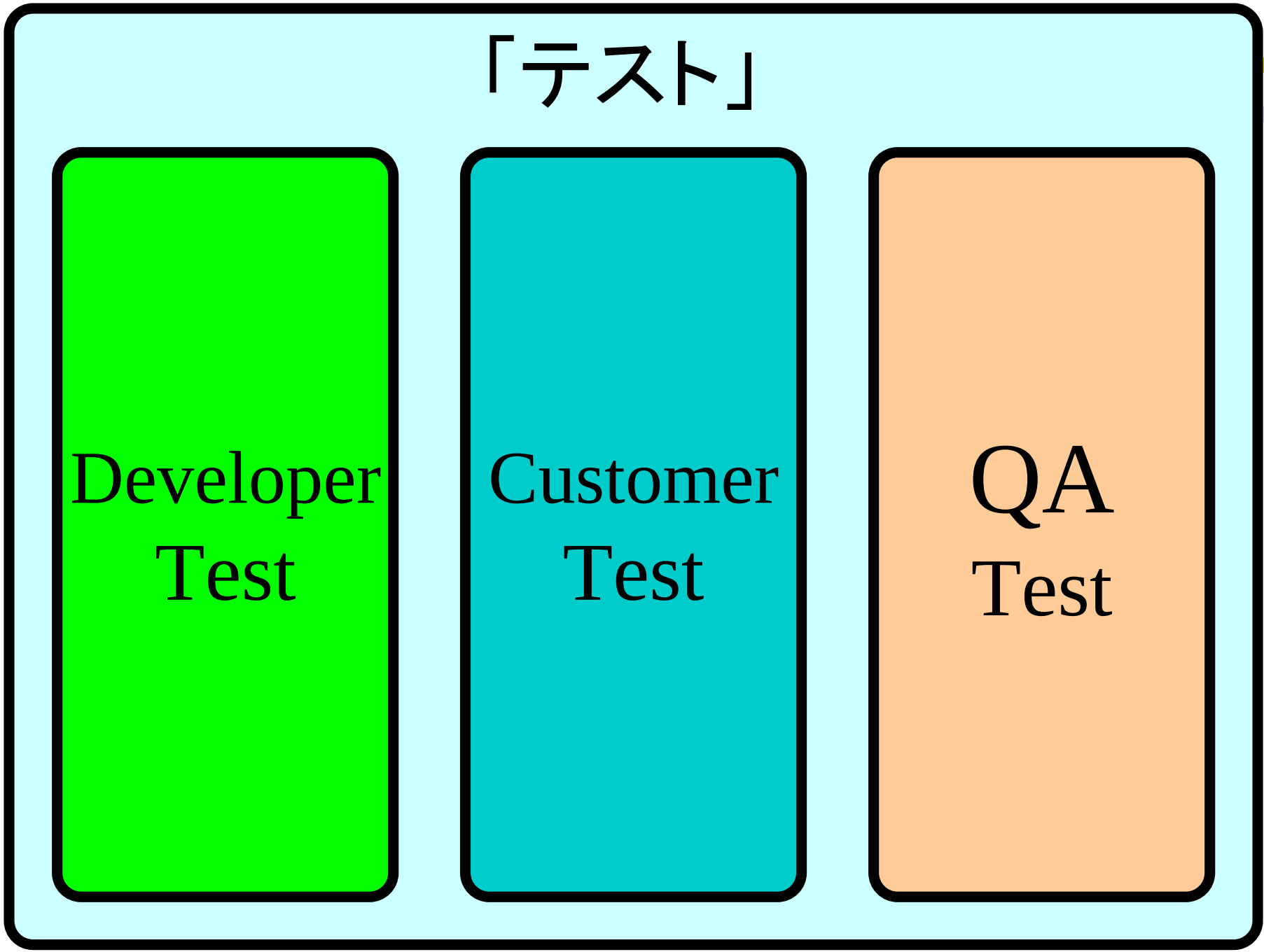
- 何のためにテストするのか
- 誰のためにテストするのか
- テストで何を知りたいのか

「結局、誰が何のためにテストを行うのか」

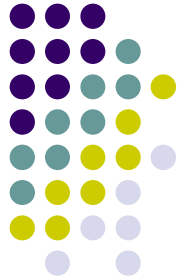
テストをロールによって分類する



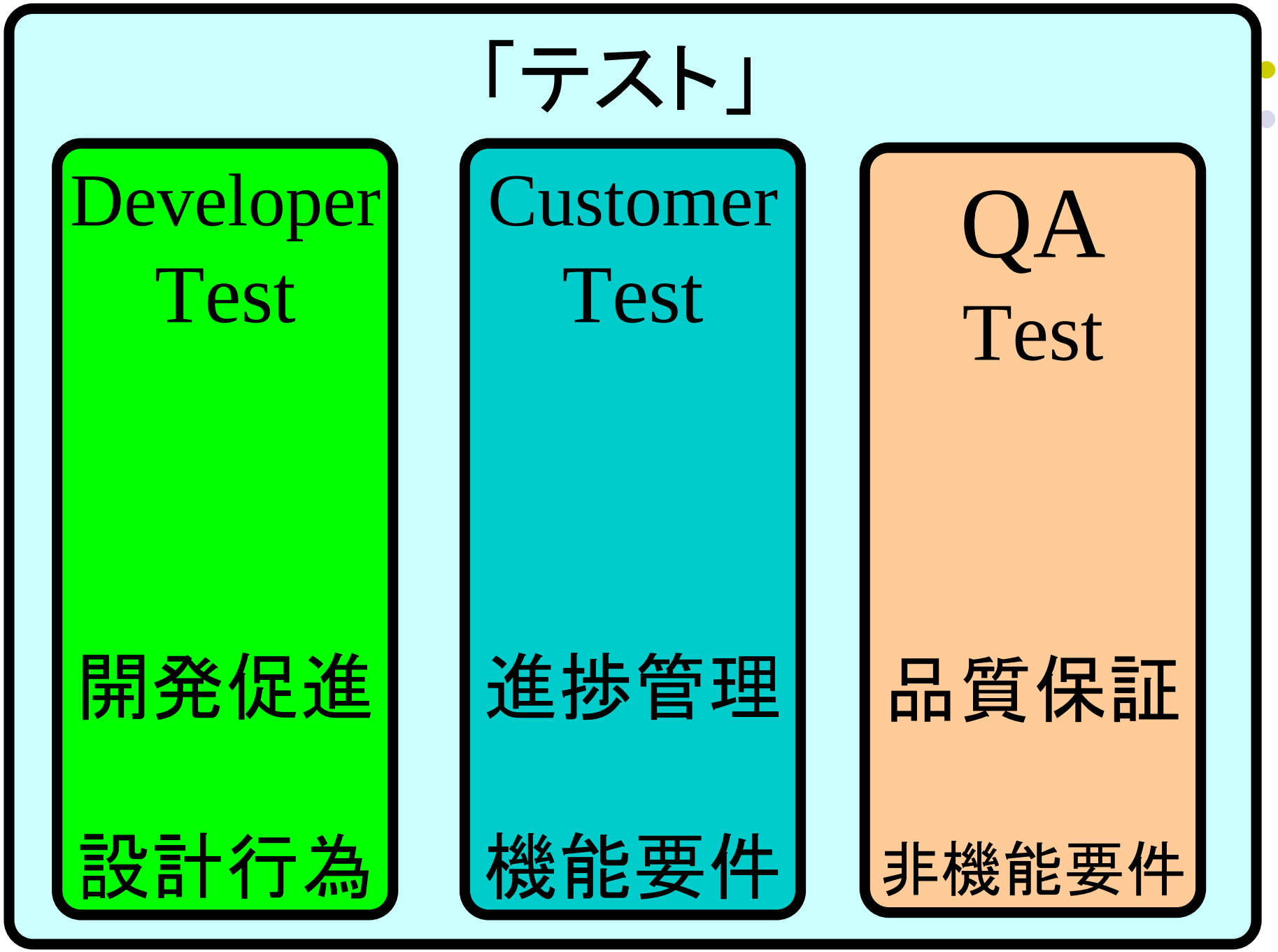
- Developer Test
  - 開発者が行う、開発促進のためのテスト
  - 単体テスト、ユニットテスト、結合テスト、…
- Customer Test
  - 顧客(のロールを担うひと)が行うテスト
  - 従来の「受け入れテスト」が多くを占める
- QA Test
  - 品質保証のためのテスト
  - 行う人はテスト担当者、もしくは開発者


「テスト」
Developer Test
Customer Test
QA Test

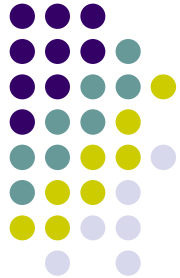
それぞれのテストの目的



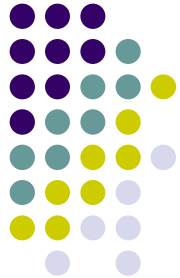
- Developer Test
  - 開発促進
  - フィードバックを伴う設計行為
- Customer Test
  - 進捗管理
  - 機能要件の検証
- QA Test
  - 品質保証
  - 非機能要件の検証


「テスト」
Developer Test
開発促進
設計行為
Customer Test
進捗管理
機能要件
QA Test
品質保証
非機能要件

再度、Developer Testingとは

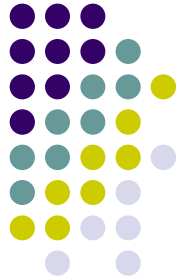


- プログラマの
- プログラマによる
- プログラマのための
- プログラムとして書かれたテスト



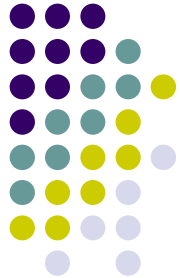
Developer Testing とは何か

Developer Testとは



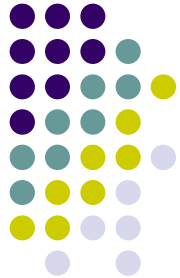
- コードとして書かれ、
- すばやく動作し、
- 自動化されており、
- テスト対象コードの特定の機能を検証するため、
- プログラマによって書かれたテスト

コードとして書かれたテスト



- これまでも書かれてきた
  - テスト用mainメソッドとか
  - テスト用のスクリプトとか
- テスティングフレームワークの登場
  - xUnitファミリー
  - テスト方法が共有された
    - クリティカル・マスを超え、大きな効果を挙げた
    - アジャイルブームに乗り、テスト人口増大

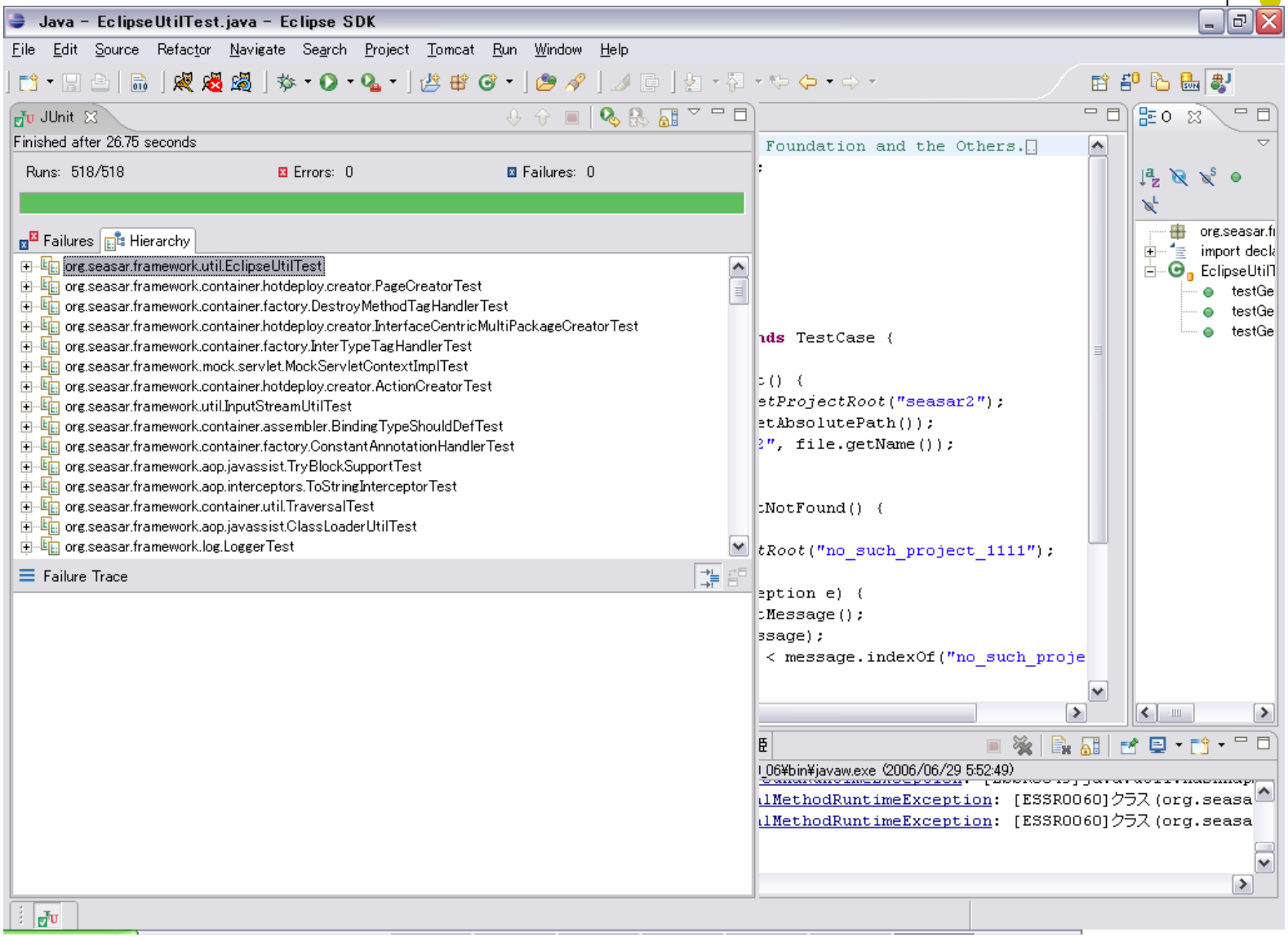
自動化されたテスト



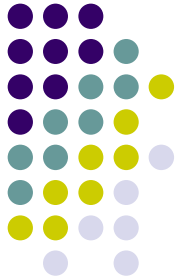
- テスティングフレームワークがもたらしたもの
  - テスト記述方法の共通化
  - テスト実行方法の共通化

- テストを書いた人でなくとも、同じようにテストを実行できるようになった

自動化されたテスト


Java – EclipseUtilTest.java – Eclipse SDK
File Edit Source Refactor Navigate Search Project Tomcat Run Window Help
JUnit
Finished after 26.75 seconds
Runs: 518/518
Errors: 0
Failures: 0
Failures
Hierarchy
org.seasar.framework.util.EclipseUtilTest
org.seasar.framework.container.hotdeploy.creator.PageCreatorTest
org.seasar.framework.container.factory.DestroyMethodTagHandlerTest
org.seasar.framework.container.hotdeploy.creator.InterfaceCentricMultiPackageCreatorTest
org.seasar.framework.container.factory.InterTypeTagHandlerTest
org.seasar.framework.mock.servlet.MockServletContextImplTest
org.seasar.framework.container.hotdeploy.creator.ActionCreatorTest
org.seasar.framework.util.InputStreamUtilTest
org.seasar.framework.container.assembler.BindingTypeShouldDefTest
org.seasar.framework.container.factory.ConstantAnnotationHandlerTest
org.seasar.framework.aop.javassist.TryBlockSupportTest
org.seasar.framework.aop.interceptors.ToStringInterceptorTest
org.seasar.framework.container.util.TraversalTest
org.seasar.framework.aop.javassist.ClassLoaderUtilTest
org.seasar.framework.log.LoggerTest
Failure Trace
Foundation and the Others.
TestCase {
("seasar2");
AbsolutePath());
", file.getName());
NotFound() {
Root("no_such_project_1111");
eption e) {
Message();
ssage);
< message.indexOf("no_such_proje
06¥bin¥javaw.exe (2006/06/29 5:52:49)
lMethodRuntimeException: [ESSR0060]クラス (org.seasa
lMethodRuntimeException: [ESSR0060]クラス (org.seasa
org.seasar.fr
import decl
EclipseUtilT
testGe
testGe
testGe

自働化されたテスト

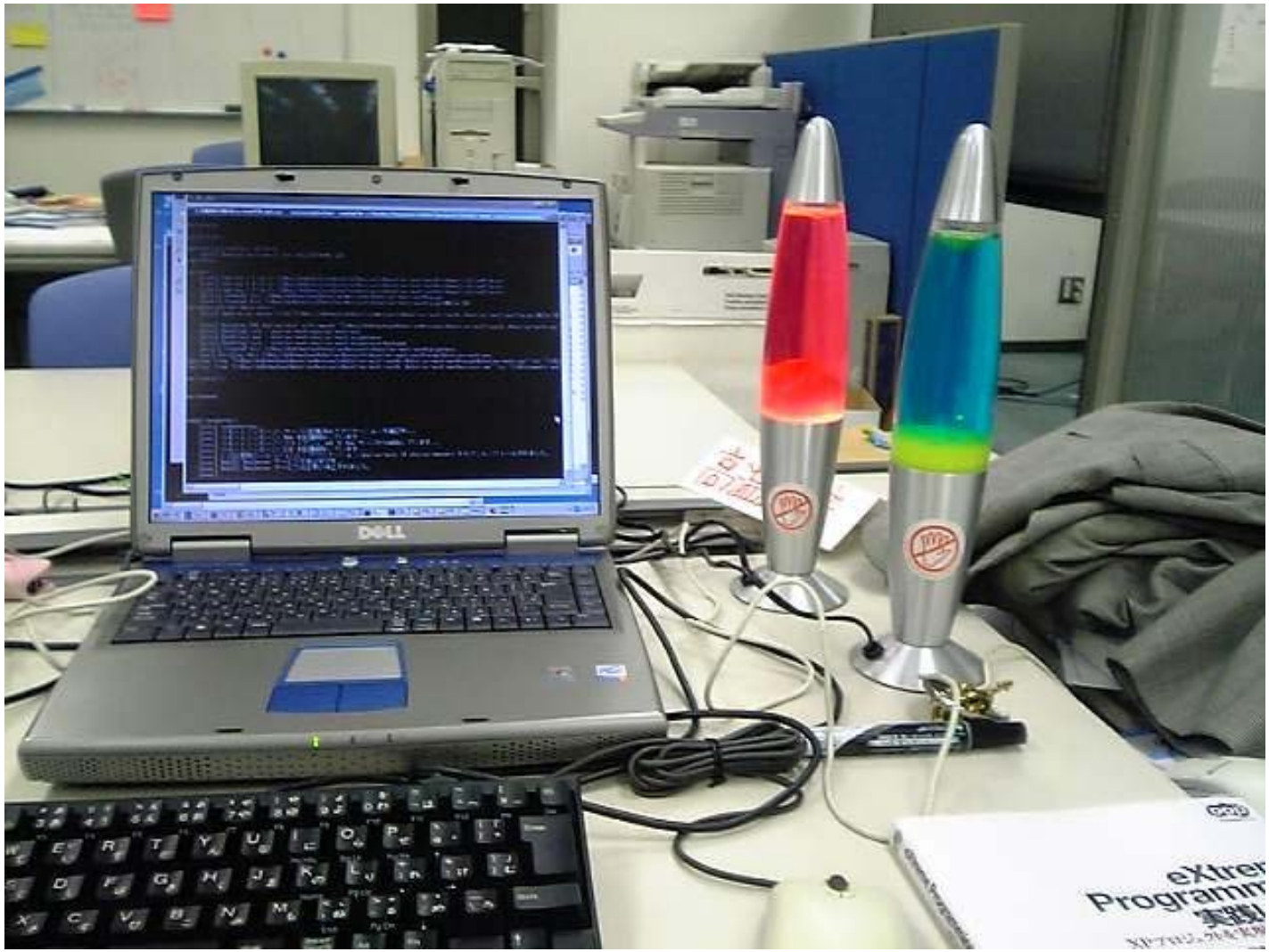


- テスティングフレームワークがもたらしたもの
  - テスト記述方法の共通化
  - テスト実行方法の共通化
- テストを書いた人でなくとも、同じようにテストを実行できるようになったということは…
- 機械にも実行させることができる
  ⇒ テストの自働化

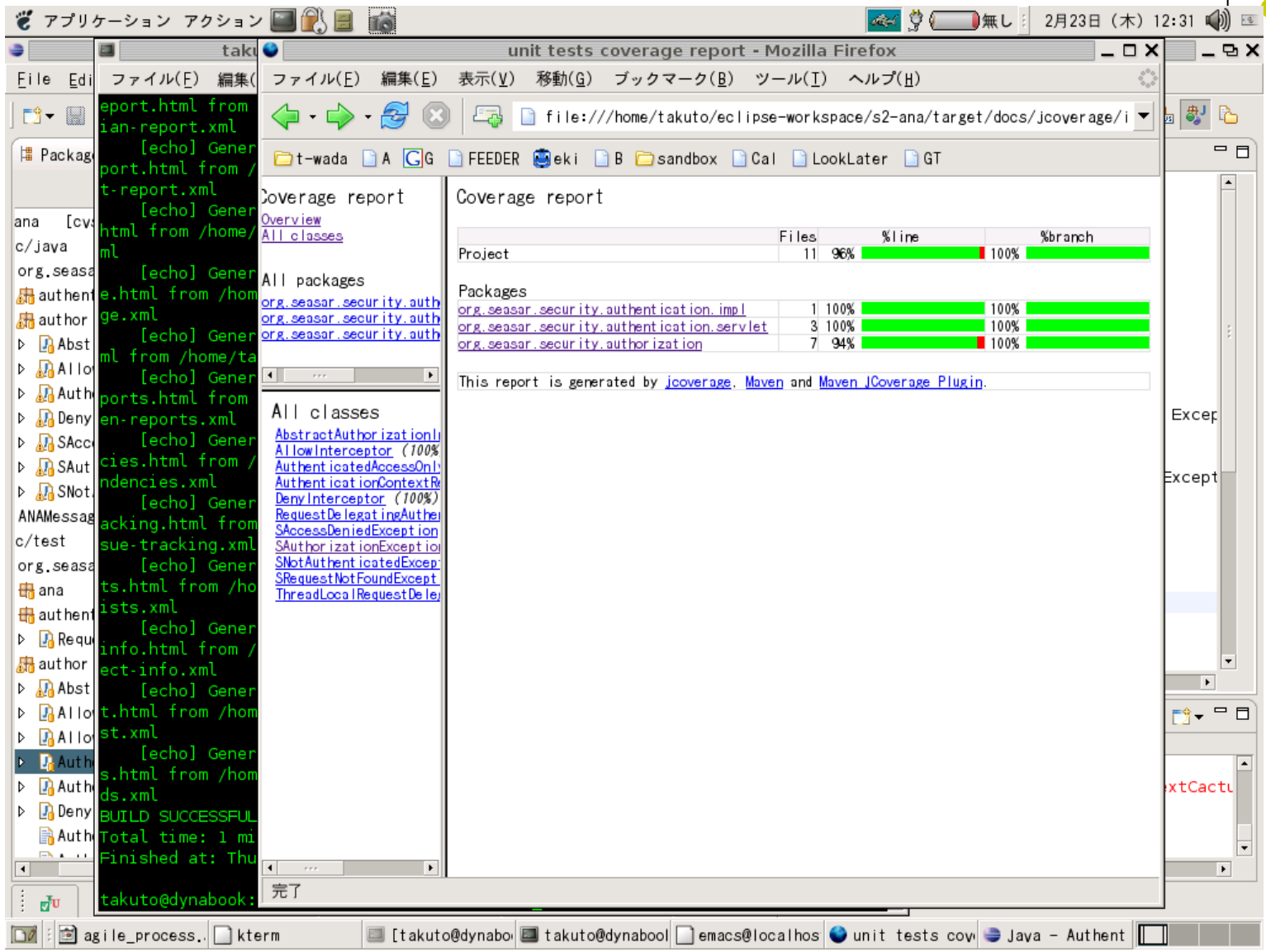
なぜ自動化(自働化)?

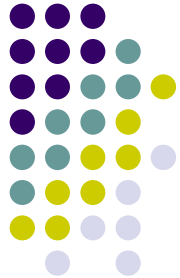
- 機械(コンピュータ)は
  - 正確
  - 一貫している
  - 疲れない
- 時間の有効利用
  - プロジェクトで人間がやることはまだ沢山ある
- 現状の「見える化」

ソフトウェアあんどん



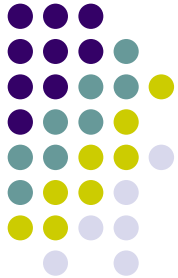
テストカバレッジレポートの生成





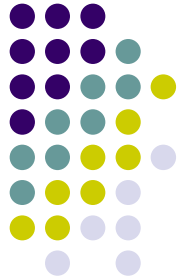
Developer Testing の目的

なぜテストするの? (優等生編)



- 意図したとおりに動作するか確認する
- 常に意図したとおりに動作するか確認する
- コードが信頼できるか確認する
- 意図を表現する(実行可能なドキュメント)

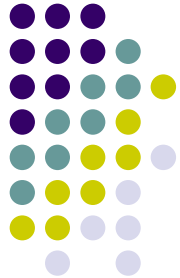
で、だ。
なぜDeveloper Testingするの?



- ソフトウェア工学的なメリットもいろいろあるけれども、最大の理由は心理的なもの

- 即時にフィードバックを得るため
- 書いたコードに自信を持つため
- これから書くコードを自信を持ってコーディングするため

心理的な効果



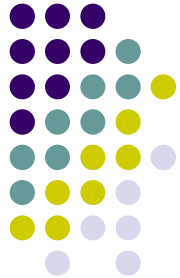
自働化されたテスト群があると…

⇒より自信を持って開発できる

⇒より良い設計ができる

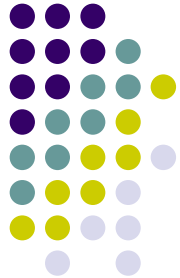
⇒心に余裕ができ、他のメンバーの様子を見ることができる

Developer Testingの真の目的



- 目的は健康(Health)!
- 何の健康?
  - ソフトウェアの健康
    - 動くコード
    - 無駄の無いコード
    - シンプルな設計
    - 変化に適応できる
  - 開発者の健康
    - 体の健康
    - 心の健康

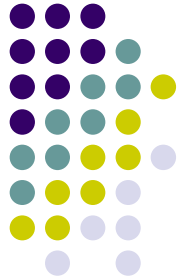
プロジェクト健康宣言



- 「Developer Testingとは、我々開発者が心身共に健康な状態で、健康なソフトウェアを開発するための手段である」

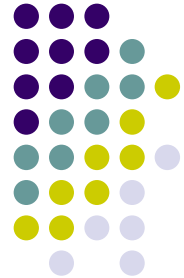
- 「健康は信頼をもたらす。信頼とは、コードへの、自分への、仲間への、マネジャーへの、お客様への信頼である」

プロジェクト健康宣言



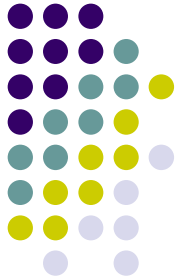
- 「健康というだけでなく、健康であり続けることを目標とする。ソフトウェア、メンバー、お客様のビジネスが共に健康であり続けることが目標である」

- 「健康であるとは、意識が行き届いており、柔軟で、活力があるさまである。健康なプロジェクトにおいては、メンバーの心身の不調、コードの不吉な匂い、お客様の懸念を察知し、それらを改善すべく自己適応する」



Developer Testing のバリエーション

Stub/Mock

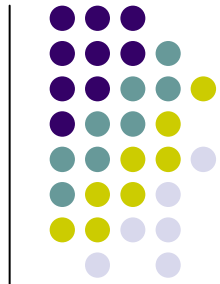


- [Context] Developer Testingの障害になるもの
  - ネットワーク
  - ファイルシステム
  - データベース

⇒ スピードが遅く、テスト毎のセットアップも難しい

- メモリ上でシミュレートするテスト用のクラスを作成してテストする

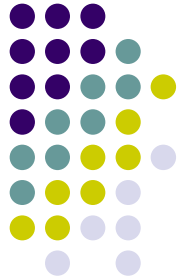
Debugging Test



- [Context] バグを見つけたとき

- そのバグを再現するテストを書き
- そのテストを失敗させ
- しかる後にそのテストを成功させることによって
  バグを修正すべし

Learning Test

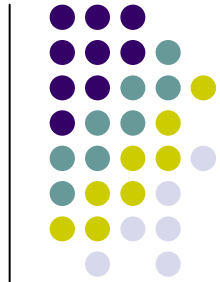


- [Context] 未知のライブラリを使って作業するとき

- ライブラリの使い方を習得し、活用してコードを書くことになる
  ⇒ 二手以上使ってしまう
  ⇒ 一歩が大きい

- ライブラリの使い方の学習目的のみのテストを書く
  - 学習結果がテストとして残る
  - 学習のみを目的とするので像がブレにくい

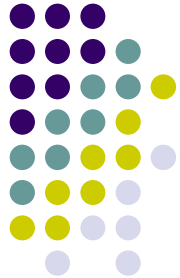
Assumption Test



- [Context] 外部ライブラリに依存した開発を行う際

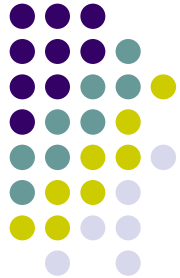
- こう動くだろうという想定をテストにしておく
- 仕様が変わったらテストが失敗する
  - テストがセーフティーネットとして作用する

Test as documentation



- ソースにはHow
- テストにはWhat
- ドキュメントにはWhyを書く!

- ドキュメントになるテストを書こう!
- 読まれることを意識したテストコードにする



Developer Testing を習得するには

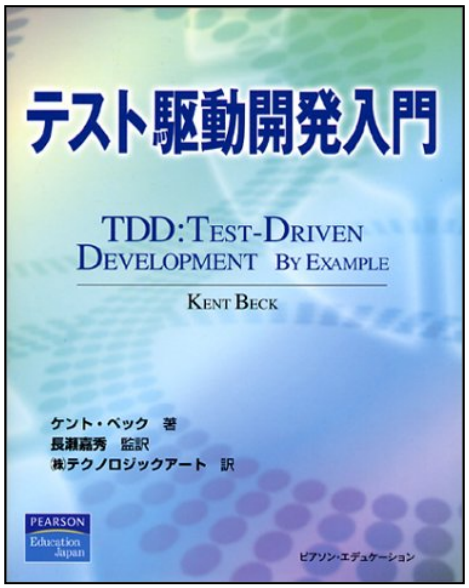
方法論ではなく、スキル

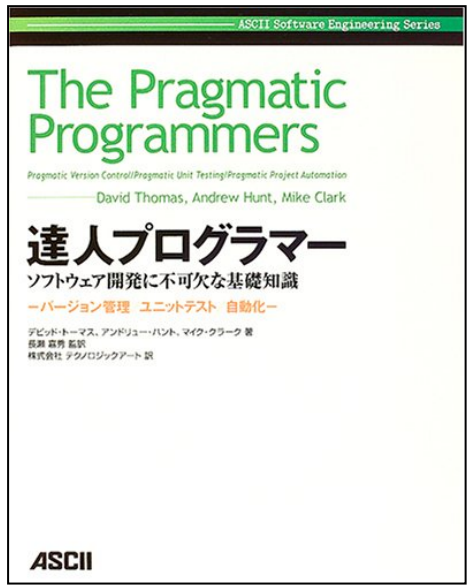
ということは…

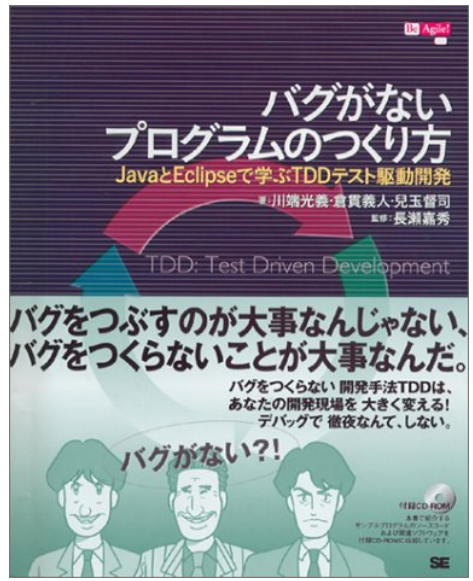
- 才能にかかわらず、習得可能
- 量は質に転化する(by はぶさん)

本を読もう、写経しよう

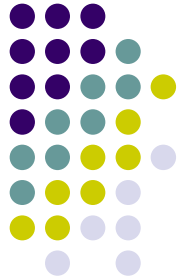
- テスト駆動開発入門
- 達人プログラマ
- バグがないプログラムのつくり方





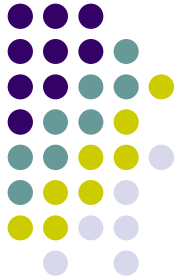


オープンソースソフトウェアのテストコードを読もう

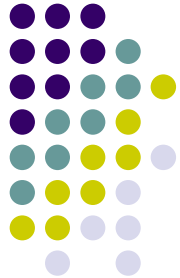


- Codehaus.orgの各プロダクト
- Seasar.orgの各プロダクト
- また、JUnitはJUnit自身によってテストされています

経験者に会おう

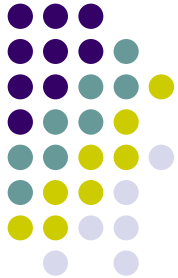


- セミナーや勉強会に参加する
- 経験者とペアプロする



テスト駆動開発 (TDD: Test Driven Development) とは何か

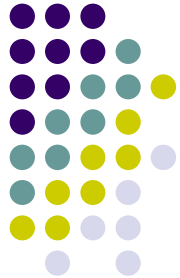
TDDの流れ



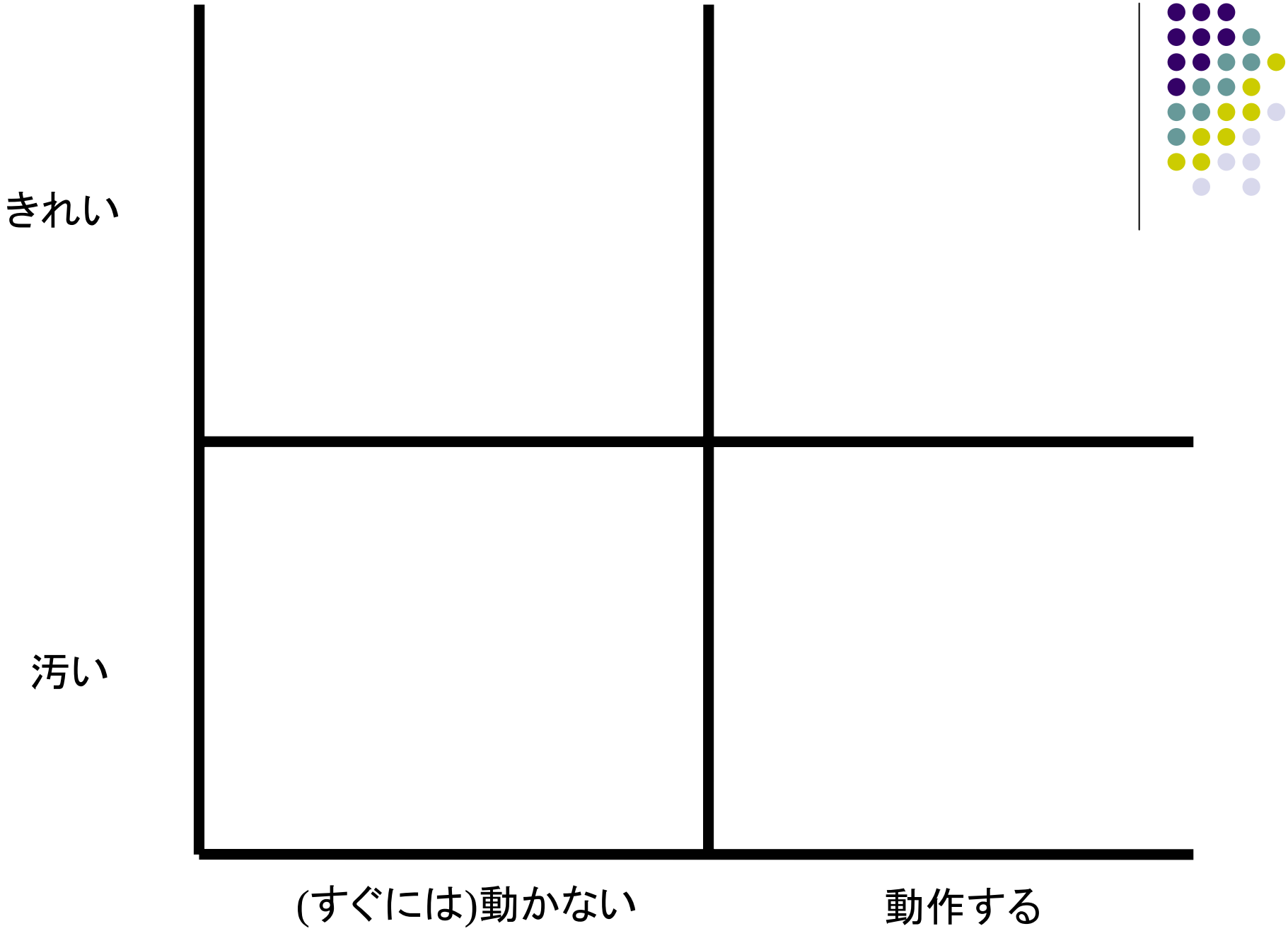
- コードを書いてからテストを行うのではなく
  1. ユニットテストを書き
  2. そのテストを実行して失敗させ
  3. 目的のコードを書き
  4. 2で書いたテストを成功させ
  5. リファクタリングを行う

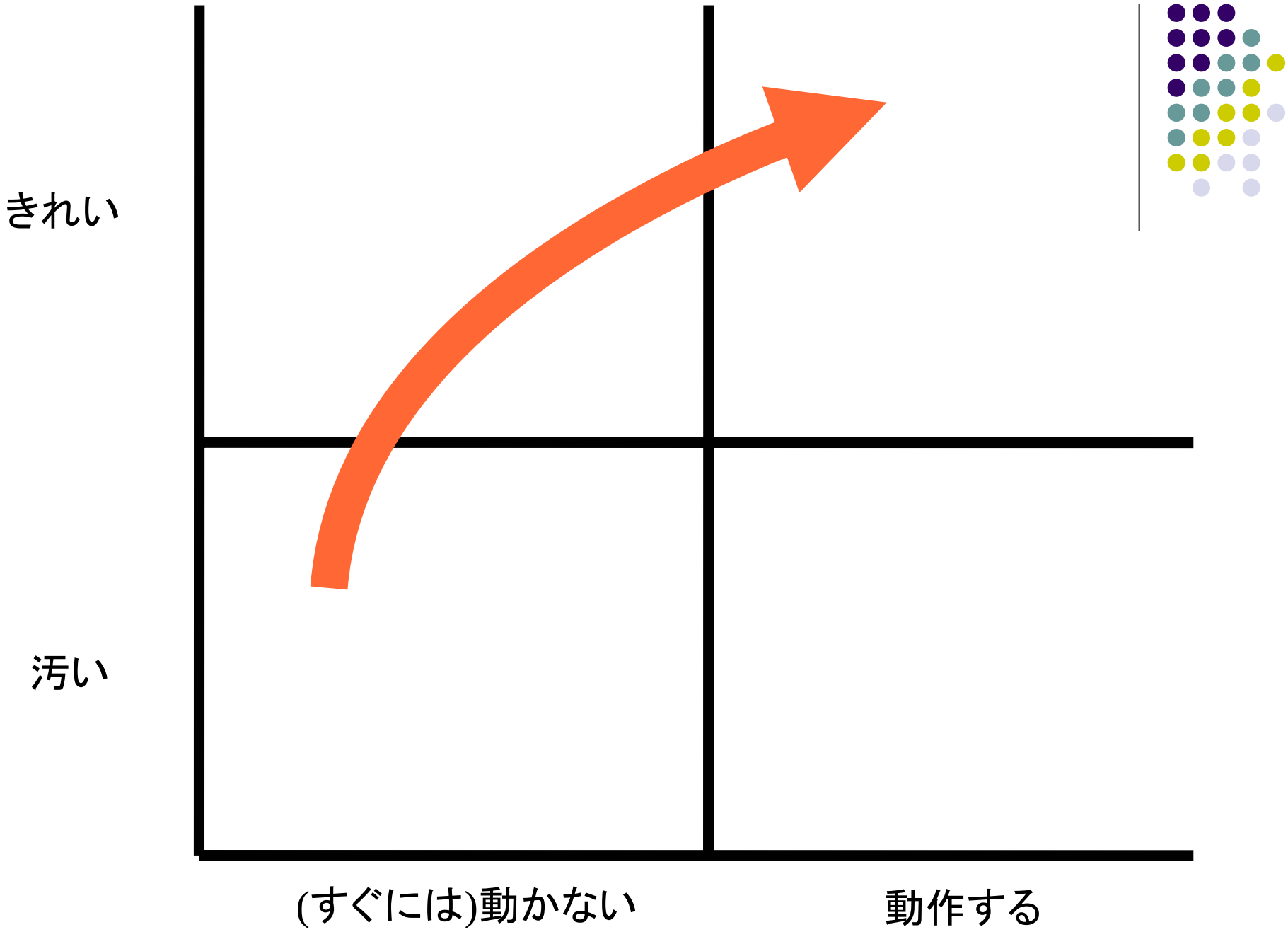
という流れで開発を行っていく開発手法

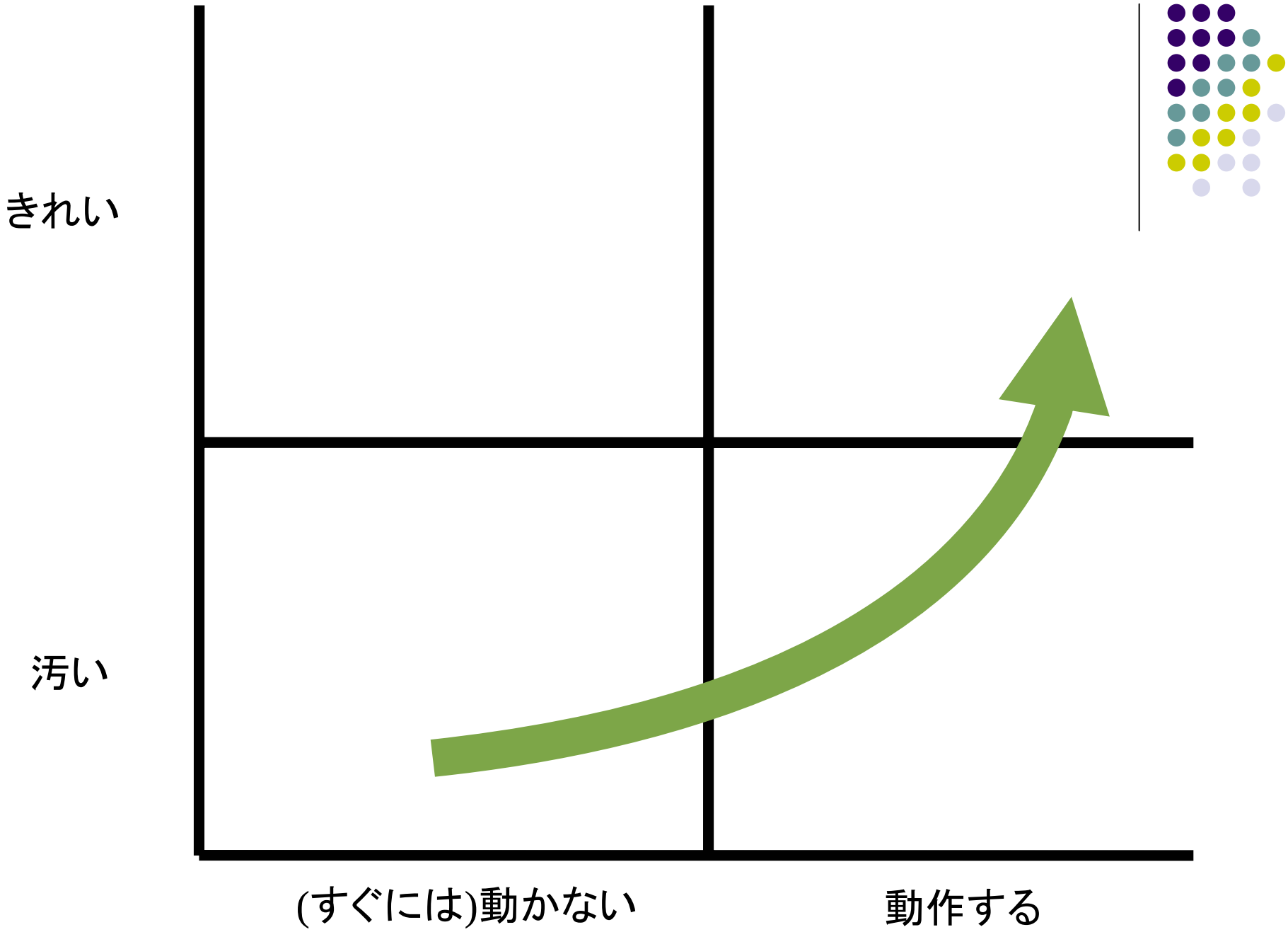
道はひとつではない

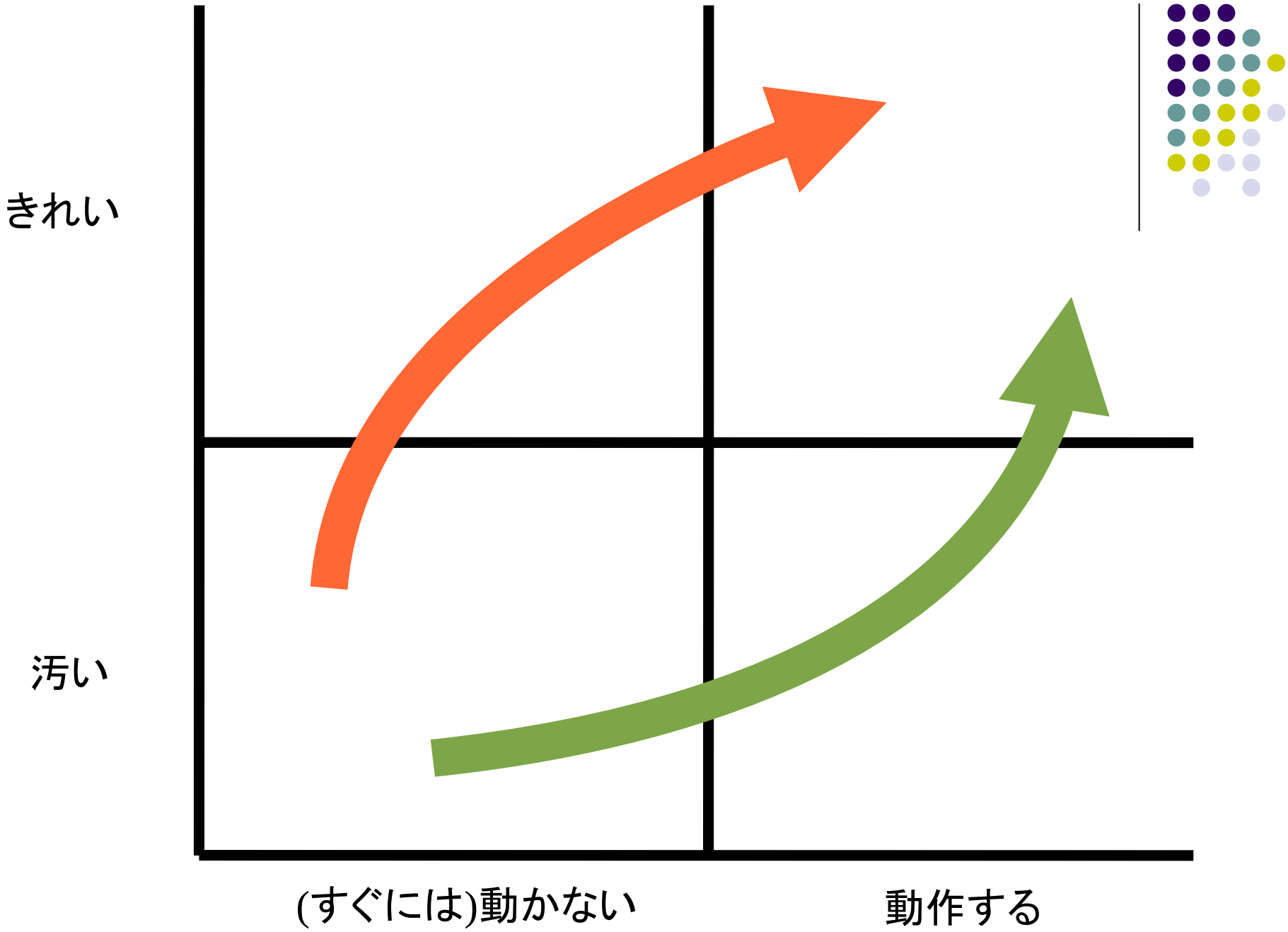


- 目標は「動作する、きれいなコード」
- きれいな、から攻めるか
- 動作する、から攻めるか

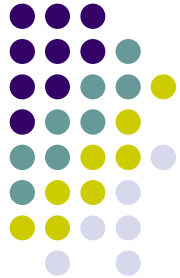

きれい
汚い
(すぐには)動かない
動作する


きれい
汚い
(すぐには)動かない
動作する


きれい
汚い
(すぐには)動かない
動作する

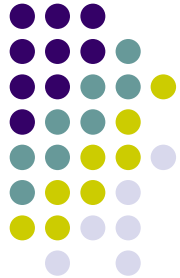

きれい
汚い
(すぐには)動かない
動作する

TDDの最小サイクル



- 失敗させるテストを書く
- そのテストが通るような実装を行う
- テストが通る状態のまま、重複を取り除く(リファクタリング)

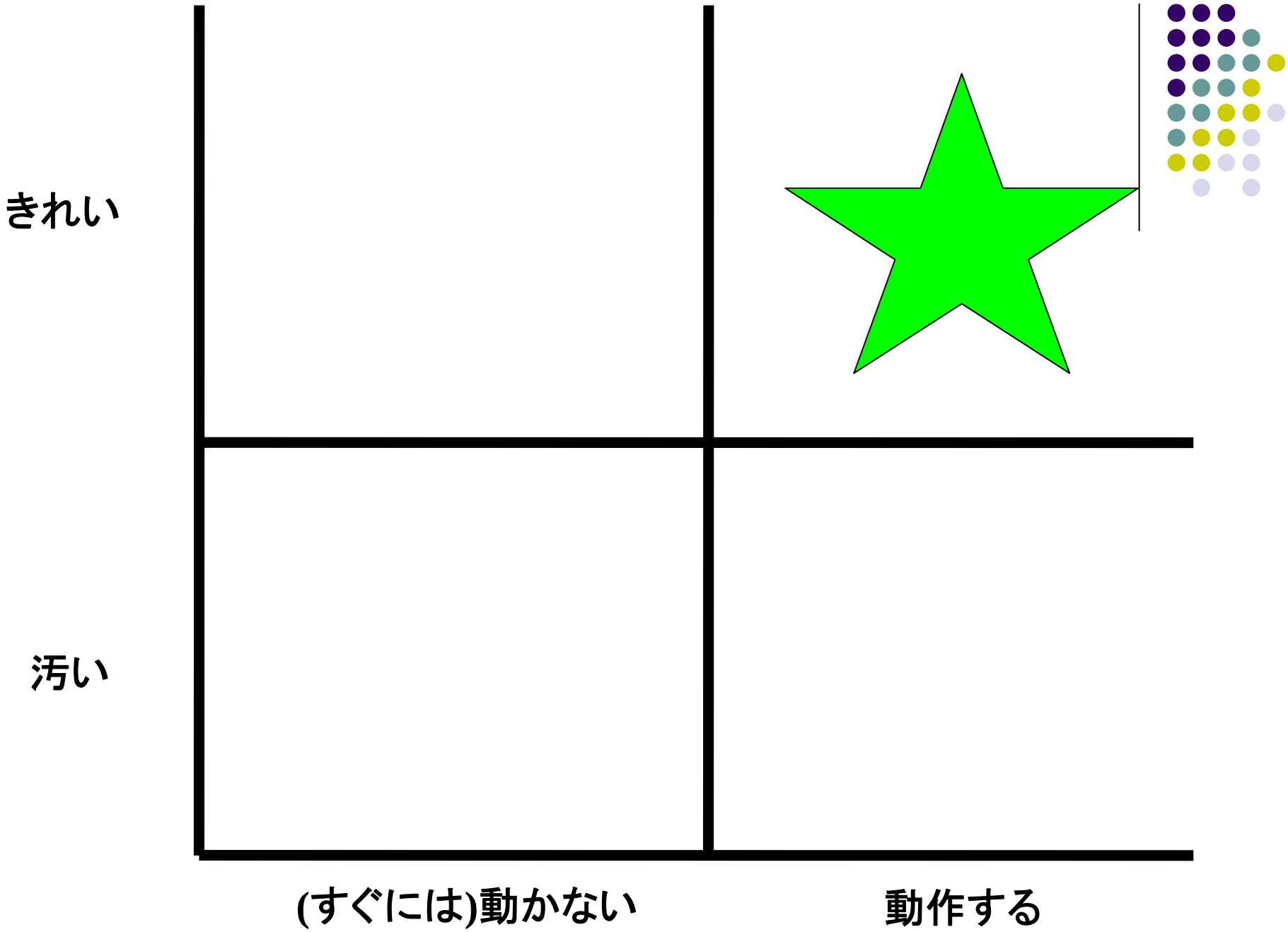
TDDのリズム

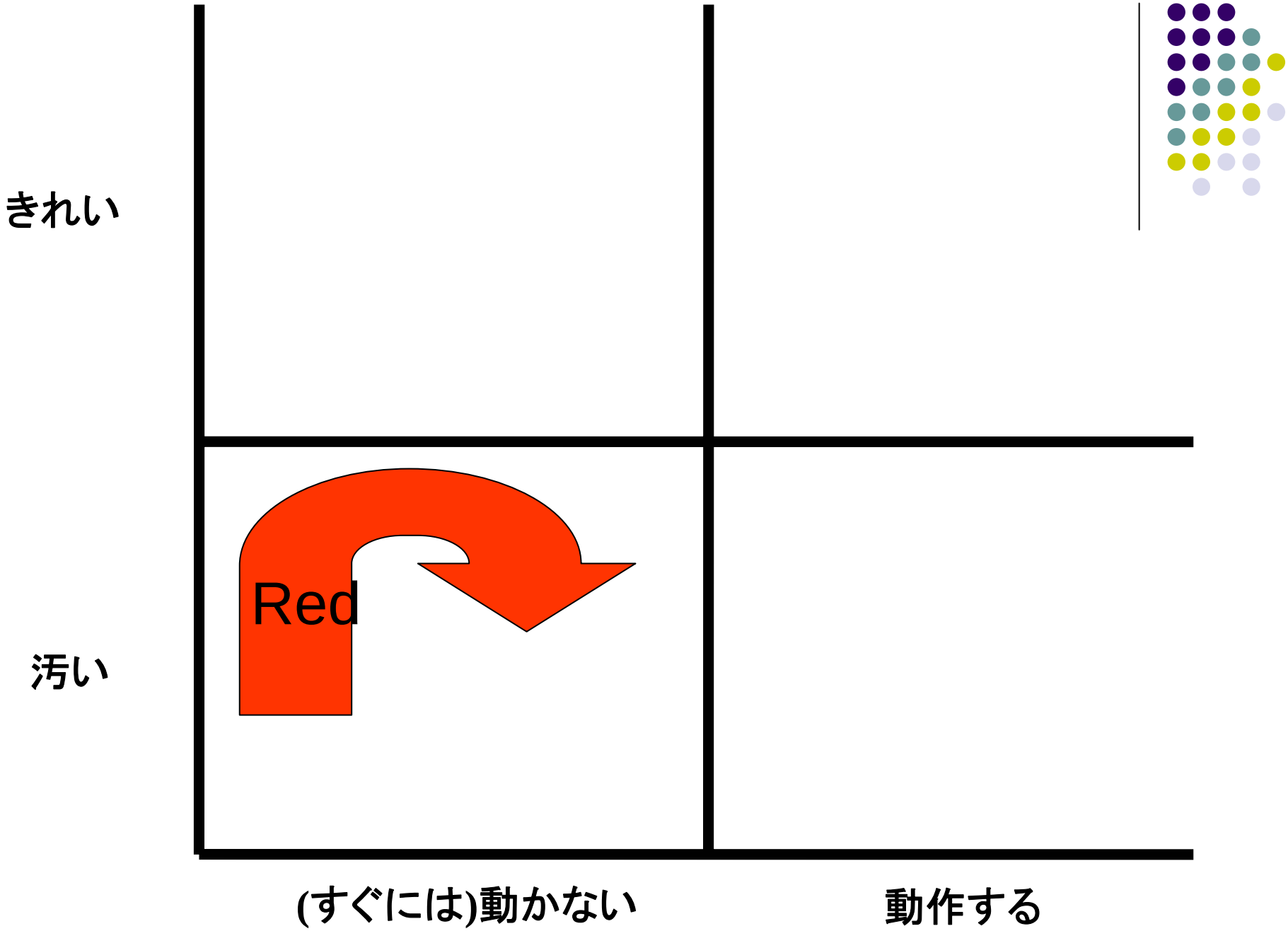


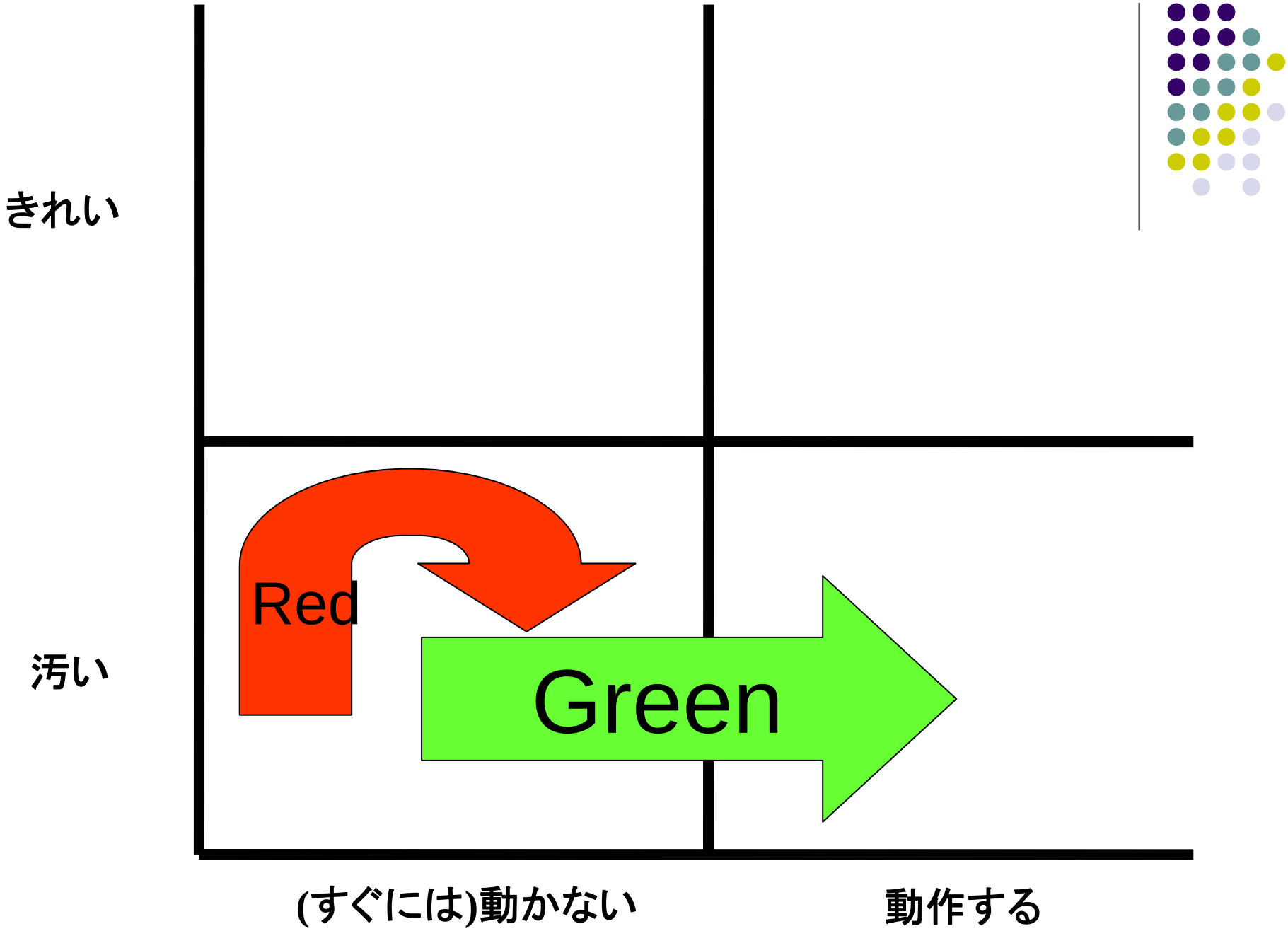
- Red, Green, Refactor
- Red, Green, Commit, Refactor, Green, Commit

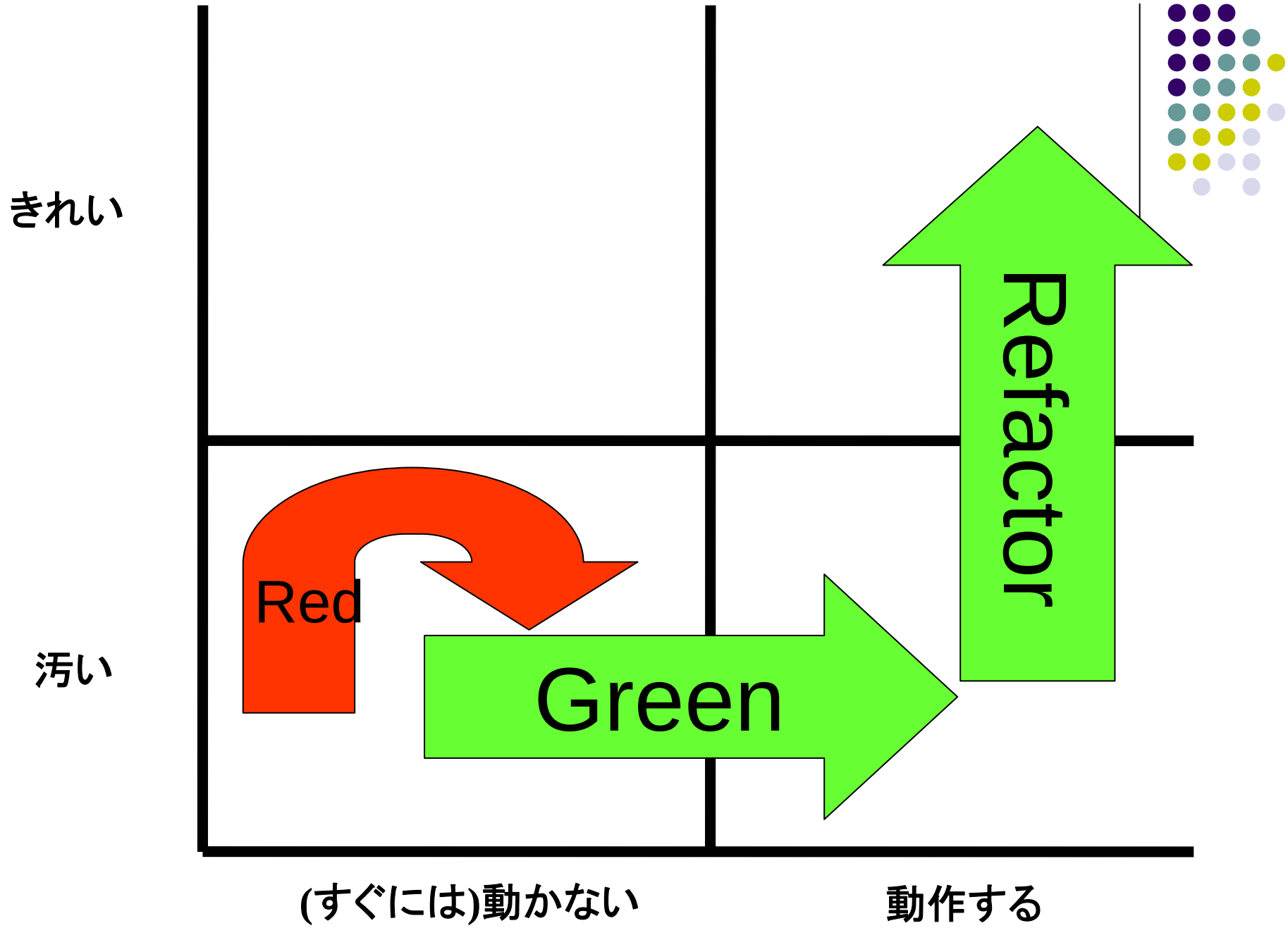
「動作する」を満たしてから

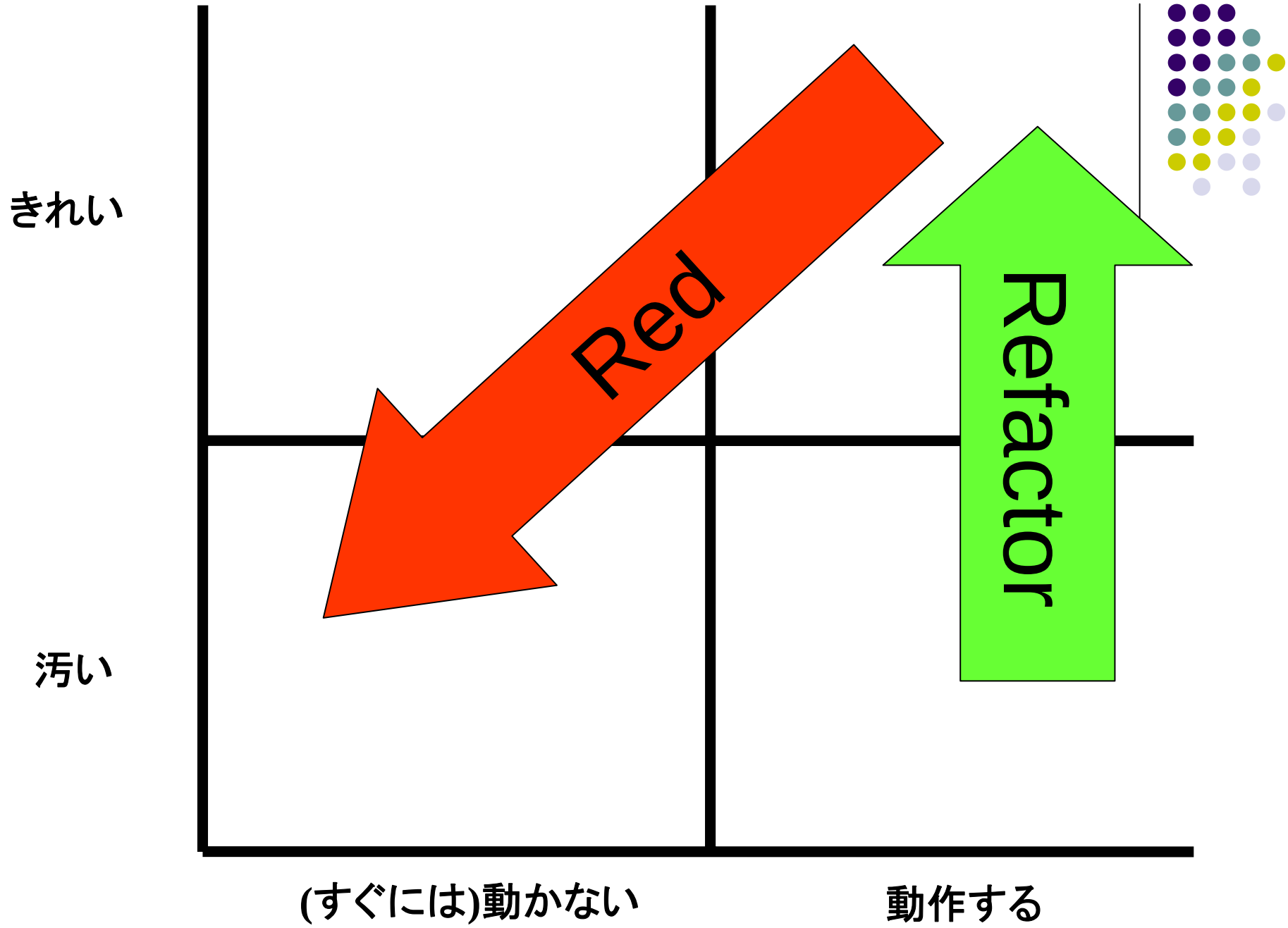
「きれいな」にとりかかる

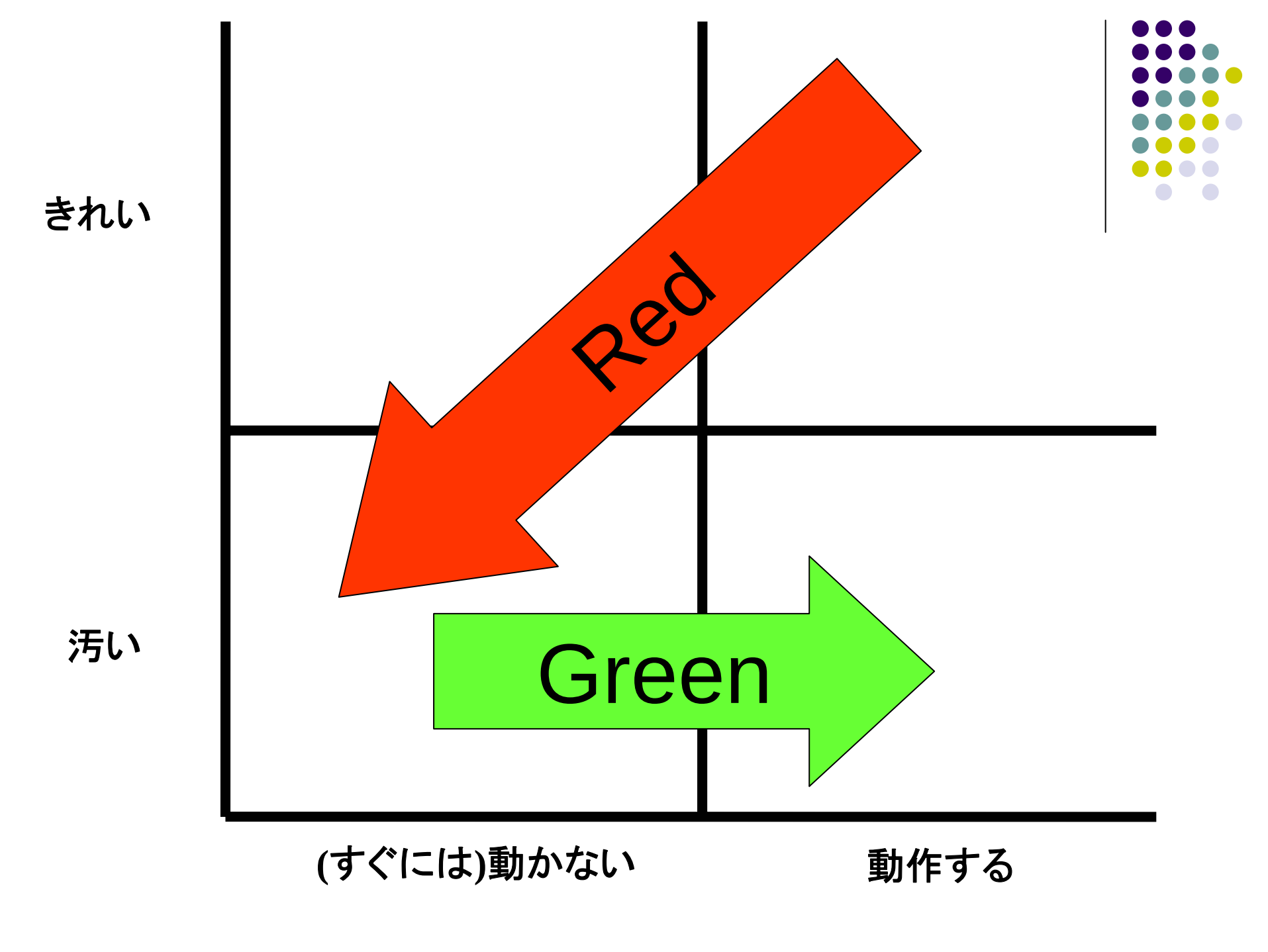

きれい
汚い
(すぐには)動かない
動作する

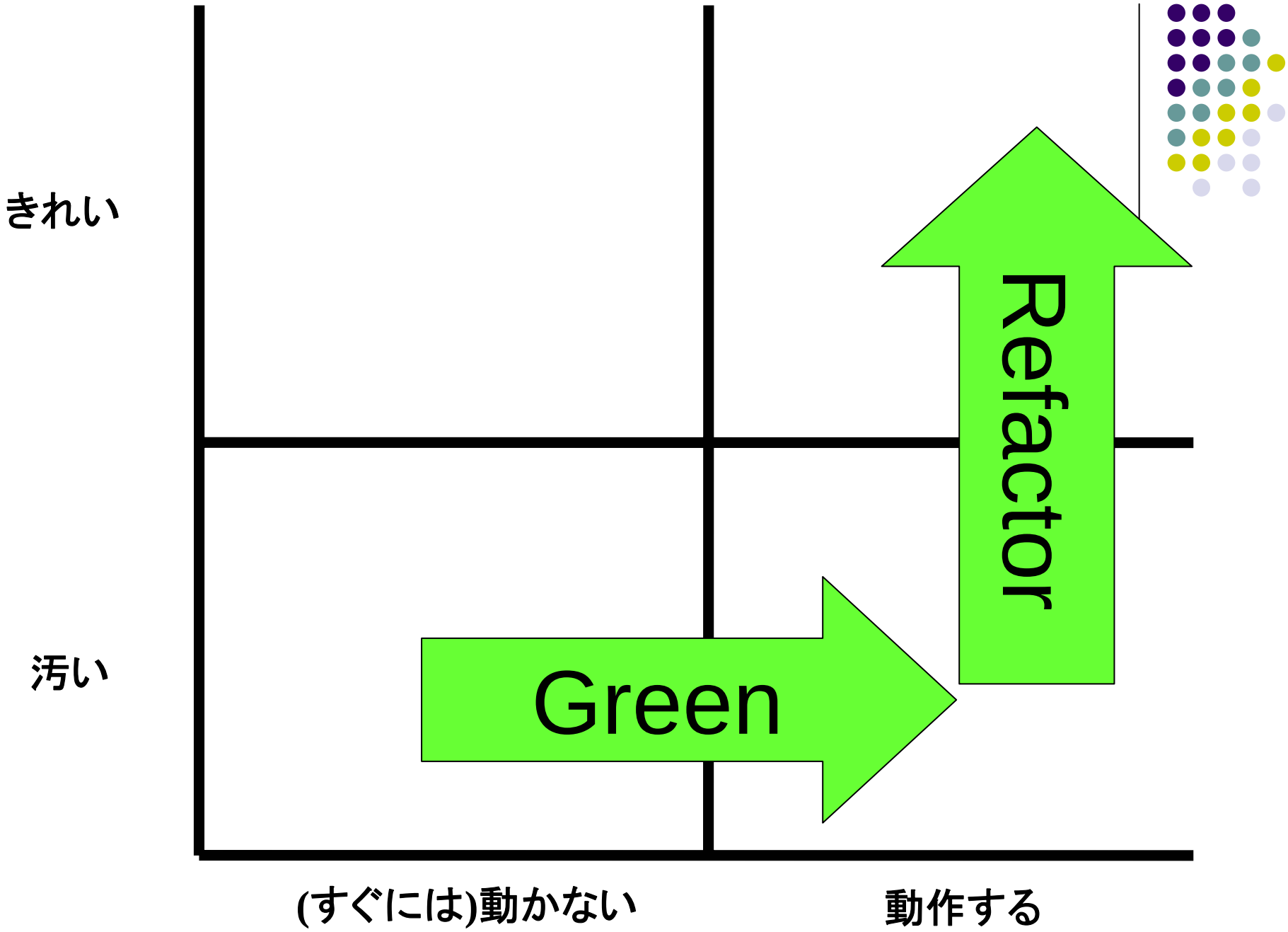

きれい
汚い
Red
(すぐには)動かない
動作する

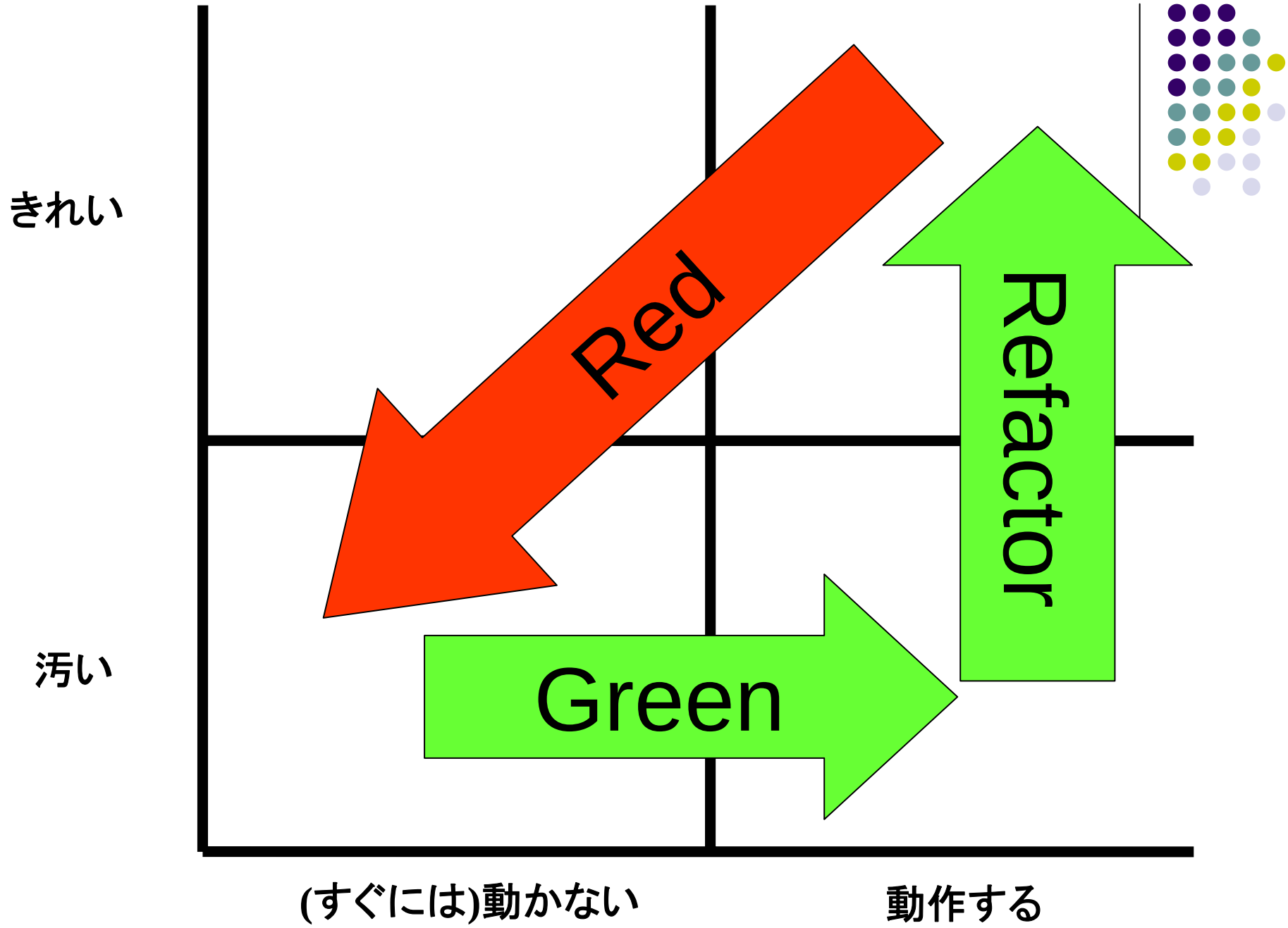

きれい
汚い
Red
Green
(すぐには)動かない
動作する


きれい
汚い
Red
Green
Refactor
(すぐには)動かない
動作する

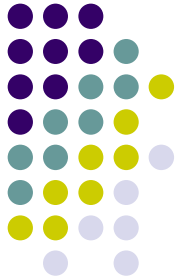

きれい
汚い
Red
Refactor
(すぐには)動かない
動作する


きれい
汚い
Red
Green
(すぐには)動かない
動作する


きれい
汚い
Refactor
Green
(すぐには)動かない
動作する


きれい
汚い
Red
Green
Refactor
(すぐには)動かない
動作する

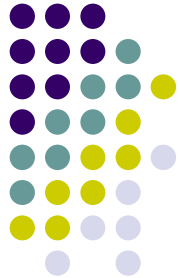
TDDは設計技法です



- 文脈
  - プログラミングは設計行為である(J.Reeves)
  - REDは仕様の設計
  - GREENは仕様の実装
  - Refactoringは内部設計の改善

参考「ソフトウェア設計とは何か」
http://www.biwa.ne.jp/~mmura/SoftwareDevelopment/WhatIsSoftwareDesignJ.html

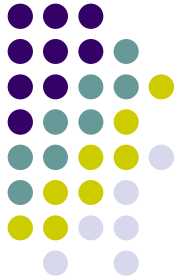
テストリストを書く



- 実装モードになった頭を設計モードに戻す

- Test ResultはLocationを
- Test ListはDirectionを示す

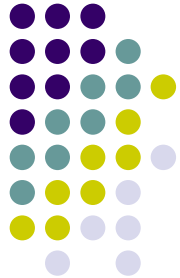
テストを先に書く意義とは



- 実装とインターフェイスを分けて考えることができる
  - HowではなくWhat
- 利用者の視点を最初から得ることができる
  - “Eat your own dog food”

- テスト可能なコードになる
- 実装がないのでグレイボックステストになる

なぜリファクタリングするのか

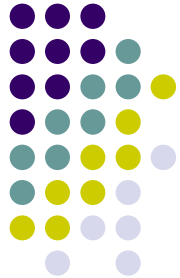


- シンプル設計
- コードを理解しやすく
- コードを修正しやすく
- コードをシンプルにすること≒設計をシンプルにすること

リファクタリングの目的

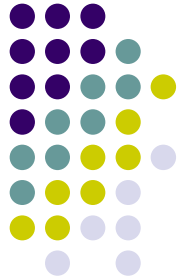
「理解や修正を簡単にするため」

呟いてみよう、囁いてみよう



- 「意図をテストにしよう。何がしたいの?」
- 「で、どうなったら嬉しいの? それをテストで表現してみよう」
- 「オレはいま何が不安なんだ?」
- 「その不安をテストで表現できないか?」
- 「次の一手が大きすぎないか?」

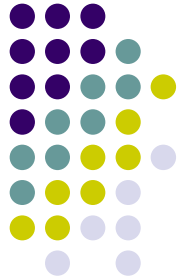
「で、どうなったら嬉しいの？」



- ゴール指向
- 「こう動いてくれたら嬉しい」という状態をテストとして書く
  - ⇒ ToBeから引っ張る

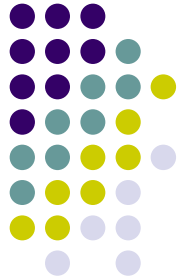
- ToBeとAsIsの距離を測り、そのギャップを埋めていくべくテストを設計する
- ToBeがAsIsに一致したら次のToBeを定義する
  - これはPDCAサイクルにほかならない

GTDとTDDが似ている点



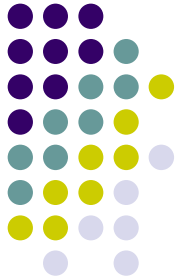
- 心に着目している
  - 開ループが生産性に与えるダメージ
- 一度に一つのことのみを相手とする
  - 「次の一手は？」

EoT (Ease of Testing)



- テストしやすい設計が、良い設計
  - 責務が少なく、はっきりしている
  - コラボレータが少ない

⇒ 一般的な良い設計(凝集度が高く、結合度が低い)



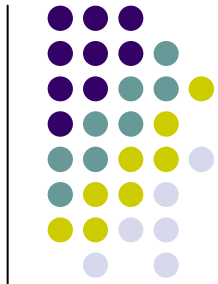
おわりに

- Developer Testingはスキルです
  - つまり習得可能です
  - 写経してみましょう

- Developer Testingで健康的なプログラマ人生を

Enjoy Testing!

Special Thanks



- かくたにさん、チームかくたにのメンバー
- JavaEE勉強会、および世話人の角田”セレブ”直行さん
- kdmsnrさん
- 溝口八郎右衛門さん
- Kent Beck氏
- オブラブスタッフの皆様

- 会場にお越しくださった皆様

- そして、masarlさん

ご清聴
ありがとう
ございました