JaSST07札幌

探索理論に基づく合理的なテストとは

～テスト技法の選択と活用の根拠は何か～

DebugEng デバッグ工学研究所
http://www.debugeng.com/
松尾谷 徹
連絡先　CFD@debugeng.com

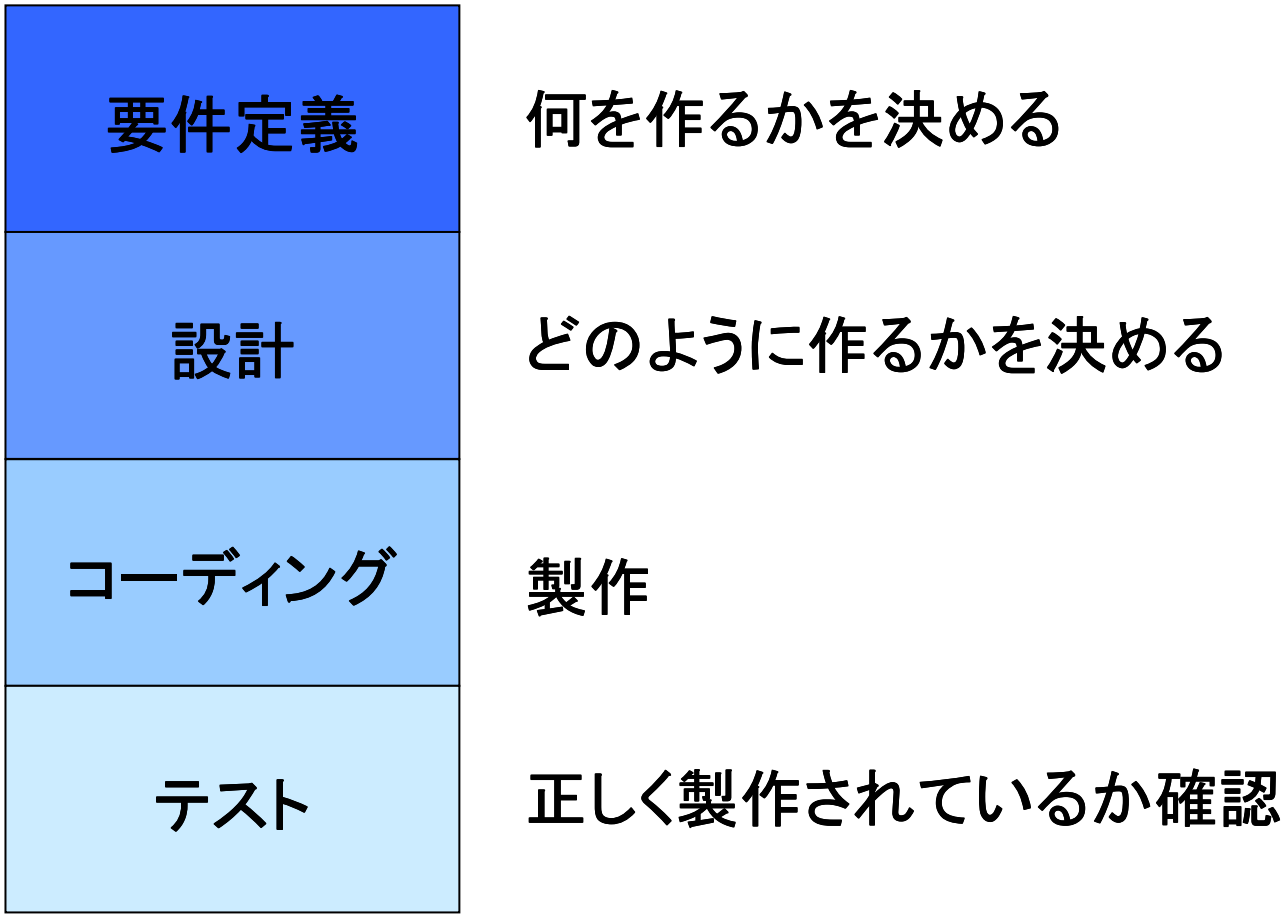
R1.0

目　次

＜＜合理的なテストとは＞＞

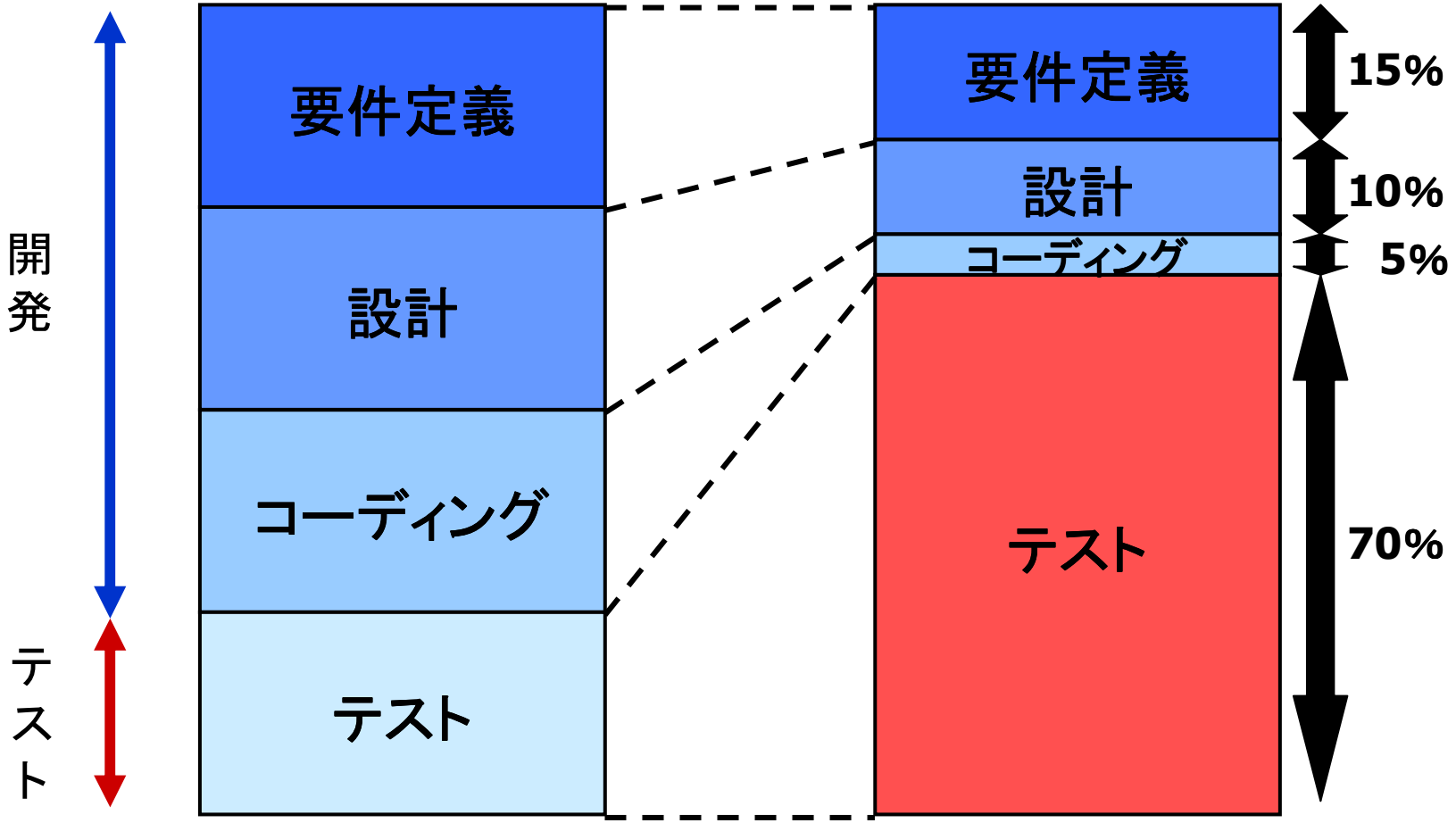
1. プロローグ:
開発ライフサイクルとテスト

■ソフトウェア開発ライフサイクル



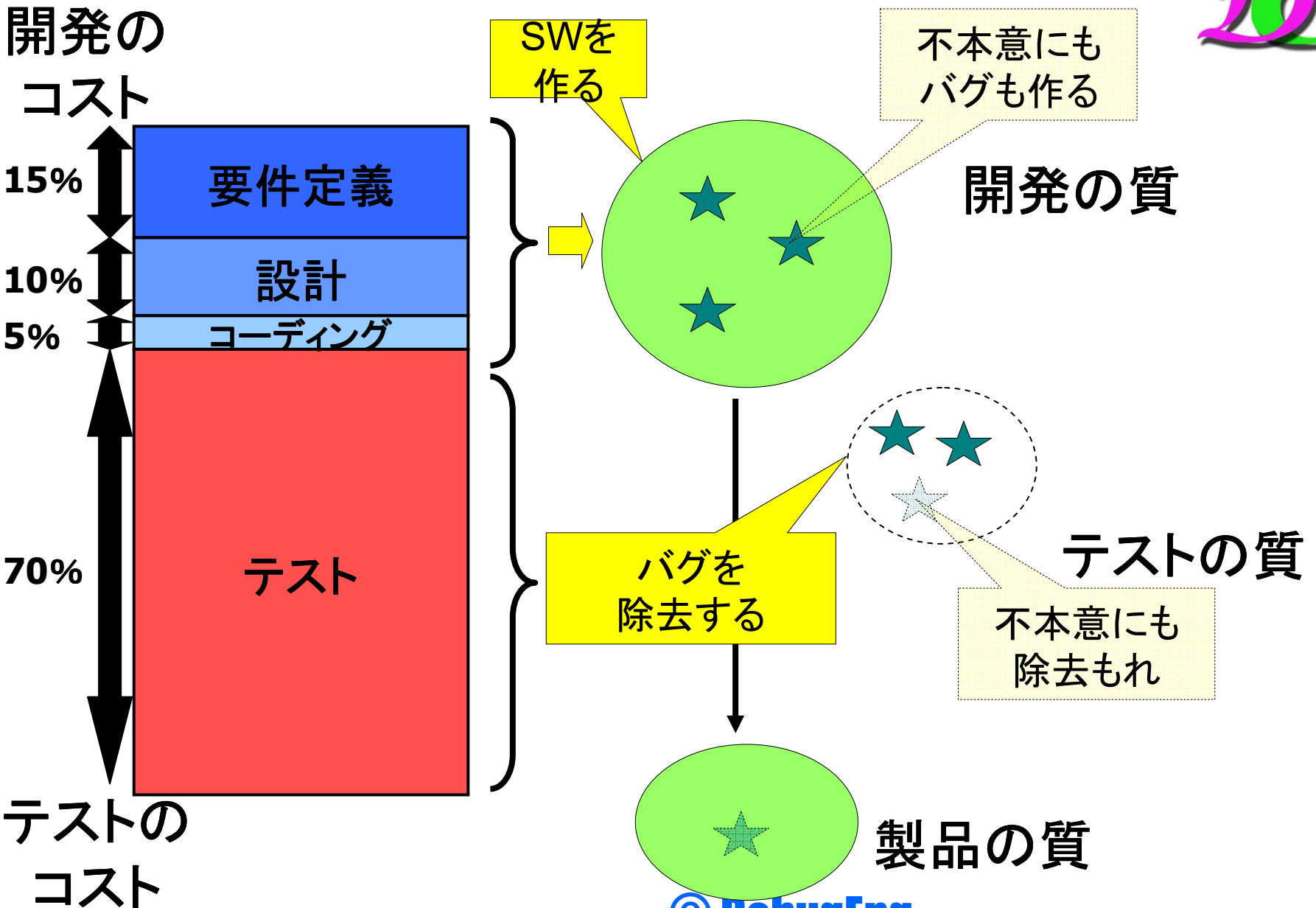
コストの割合（全体）

■開発コストが一番多い工程は？



工作機械の組込みソフトウェア開発

開発とテスト：　質とコスト（生産性）



■下流工程の混乱

昔も、今も混乱は続いている・・・・現状認識

アプローチの結果

■*上流遡行アプローチ*<開発の質を高める>

- *成熟期:技術、管理など百花繚乱で多くの成果を出した*
- *しかし下流工程の混乱を解決できていないから*
- *何故解決できないのか？*
- *それは:　上流工程の品質向上は、テストのコストを下げないから*

■*下流アプローチ*<テストの生産性を高める>

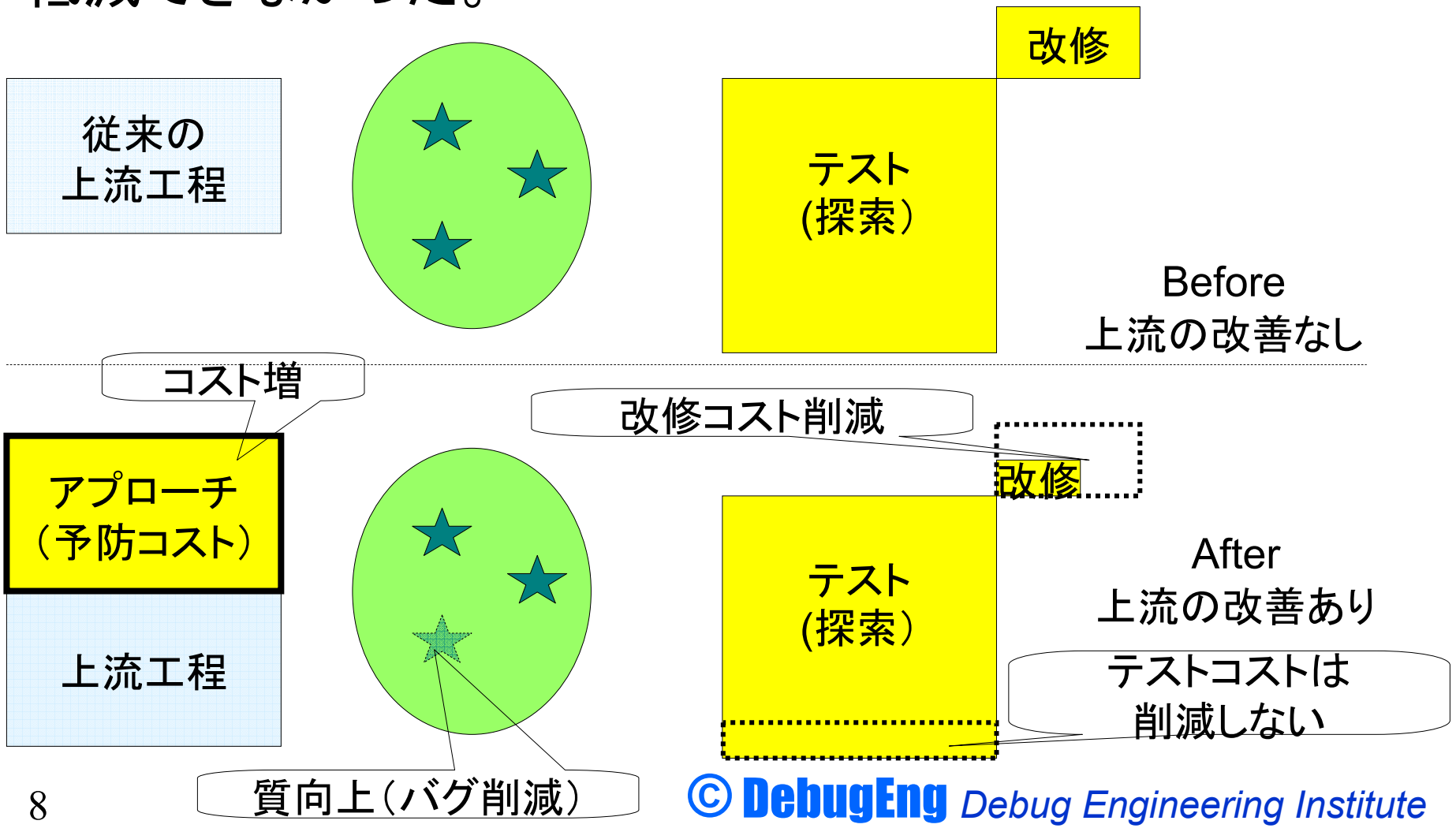
- *展開期:先進企業は合理的に品質を達成できている*
- *しかし、多くの企業は昔のまま(格差拡大期)*

■*統合アプローチ*<検証指向>

- *黎明期:さらなる品質のため今後の課題*

上流遡行アプローチの限界

■上流工程の品質向上は、下流工程のコストをそれほど低減できなかった。



下流アプローチ： テストの生産性向上

- テストの生産性　＝　テストの成果／費やしたコスト

- コストは簡単に測定できる
- テストの成果

　　ここで誤解がある

初期のモデル

1. 見つけたバグの数／隠れているバグの数

　　隠れているバグの数は不明、推測困難

現在のモデル

2. 探したエリア／探さなければいけないエリア

　　探索の合理化が必要

Debug Engineering

＜＜合理的なテストとは＞＞

2. テストの効果と効率
～探索モデル～

古典的なモデル

- テストの効果
  ＝発見バグ数／潜在バグ数
- 潜在バグ数を正確に知る方法は無い
  そこで、推測する：　何から推測するのか？

1. 過去のデータ（統計量）から推測
   - *同じソフトウェアは作らない、開発者の影響、などなど不正確*
2. 発見バグの加速度から推測する
   - *信頼度成長モデル*

テストに使える探索理論

テストは目標物を探索する行為

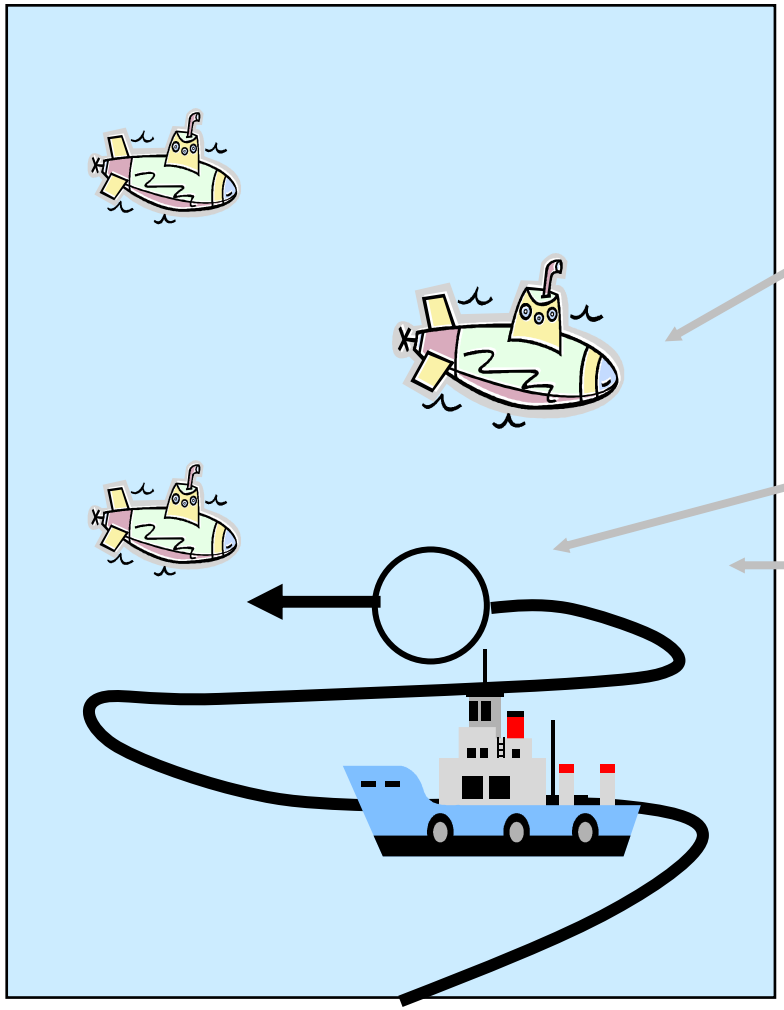
OR（オペレーションズ・リサーチ）では探索理論

- *第二次世界大戦における3大OR上の発見*
- *米海軍の独国Uボート探索から生まれた*

■ 探索要素（探索性能を決める要素）

- *探索空間：　駆逐艦が担当する海域　広さ、海流・・etc*
- *探索目標：　Uボート（潜水艦）　速度、数（存在密度）*
- *有効探索幅：　駆逐艦のソナー探知域*
- *探索速度：　駆逐艦の運動能力（速度、操舵性）*

探索モデル　探される側／探す側



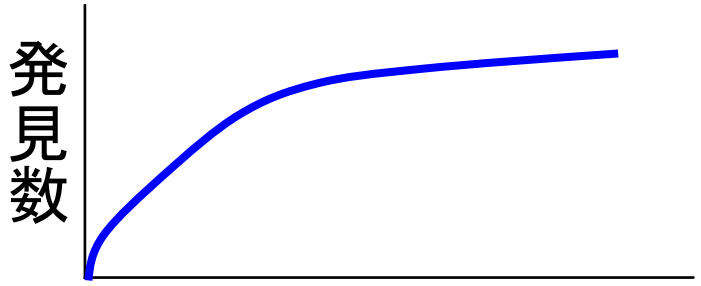
探索空間

探索目標

探索有効幅

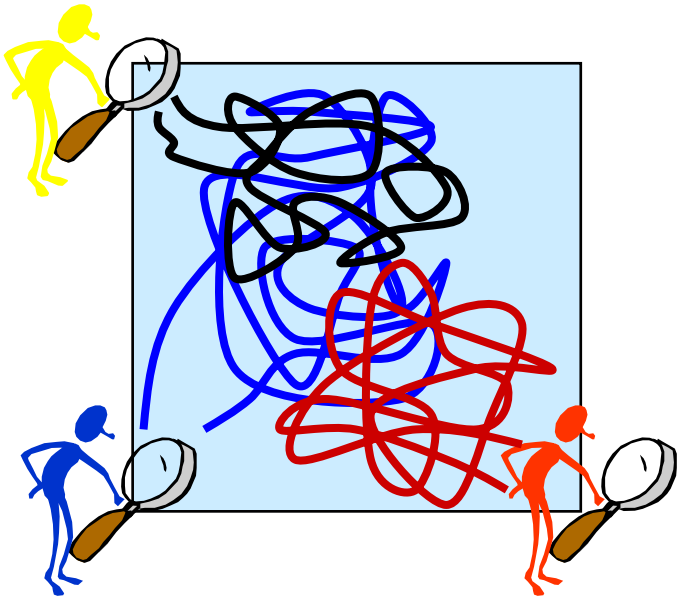
探索速度

（発見確率）



探索時間

探索戦略・ランダム探索



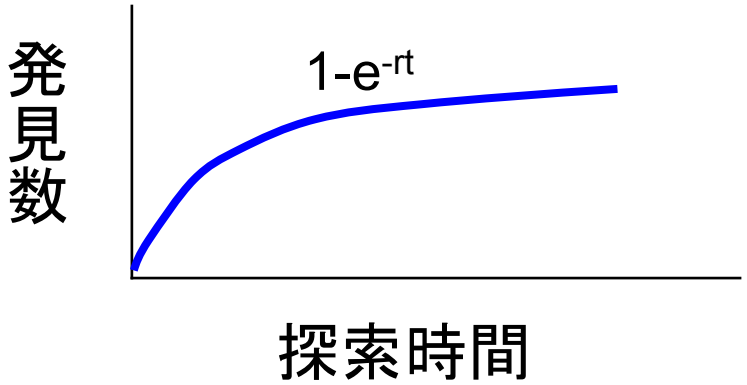
手当たりしだいに探す
航路を見ると酔歩型の探索
＜通常の探索＞

ランダム探索

発見確率　$1-e^{-rt}$

この方法は、最も効率の悪い
探索方法である.

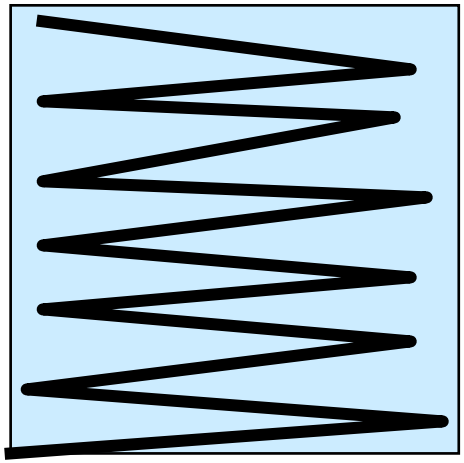
（発見確率）



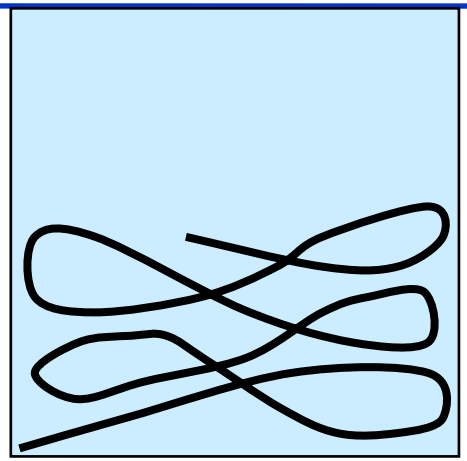
むだ：探索の重複がある

むら：未探索個所が不明

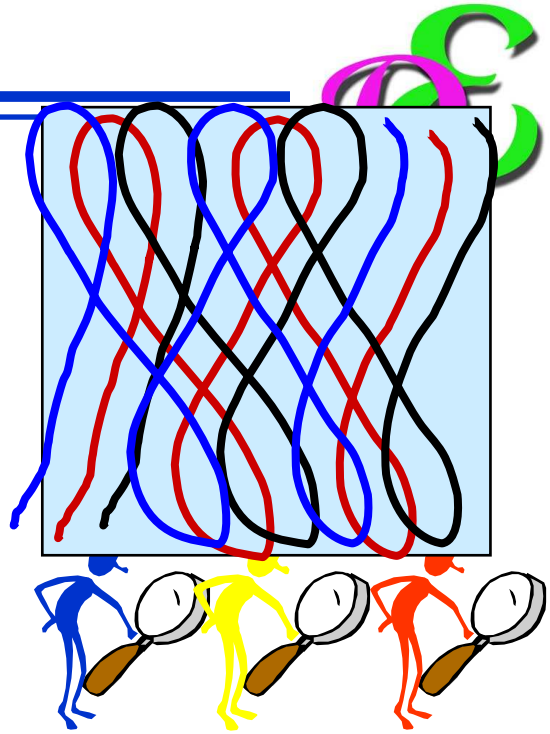
効率的な探索　平行探索

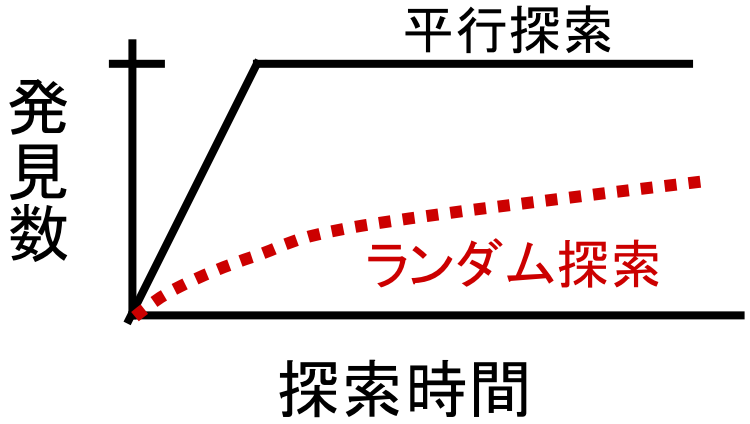


平行探索



8の字航法





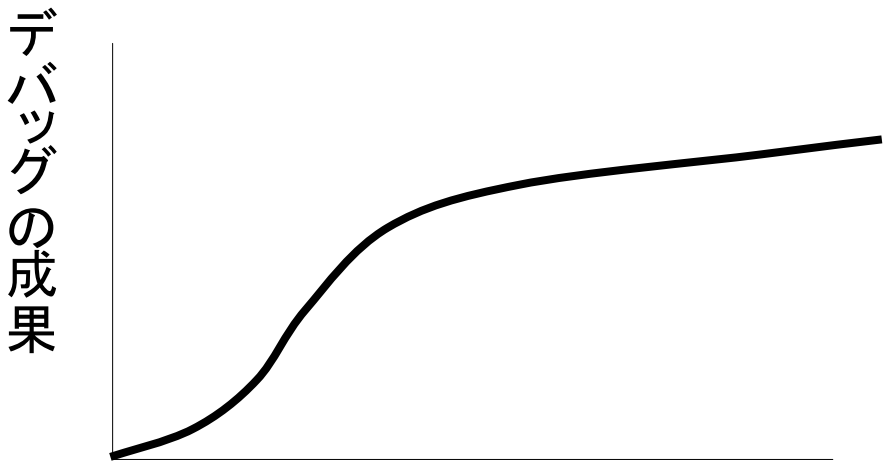
平行探索
効率の良い探索

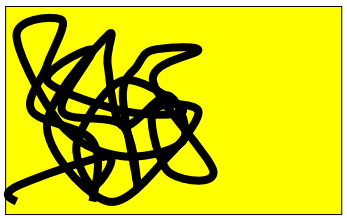
重複探索がない
未探索箇所が解る

網羅的な探索

統計で個別ソフトウェアの品質管理が出来るか？

■信頼度成長モデル(SRGM)

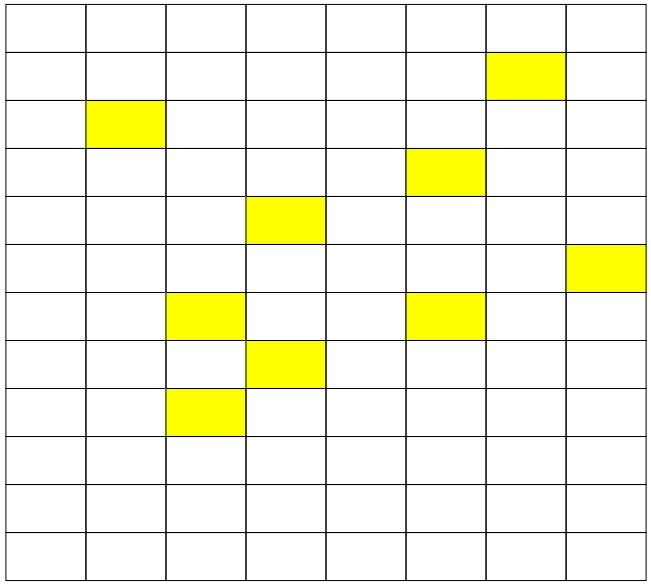




ランダム探索

未テスト要素

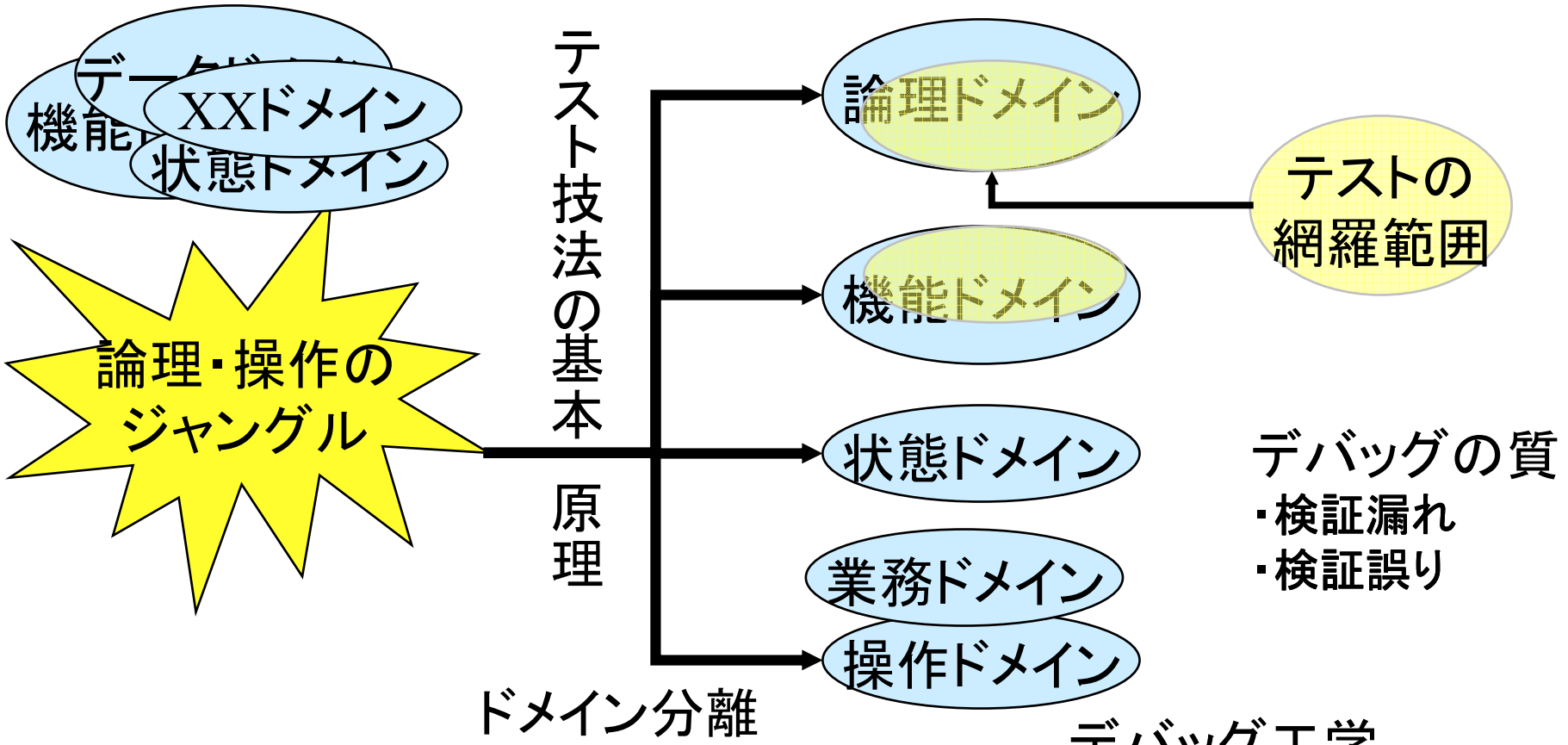
テストが必要な要素



必要な要素をくまなくテストする

バグ探索はドメイン網羅から

- 被テスト側の論理空間をドメインに分離する
- 各ドメインを網羅する検証



デバッグ工学

Debug Engineering

＜＜合理的なテストとは＞＞

3. テストの局面と技法
～技法選択を評価する～

マトリックステスト

■方法

- *原因(入力)を同値分割する*
- *原因を因子、同値を水準として*
- *マトリックスを作成する*
- *すべてのマスに結果を予測*

■利点

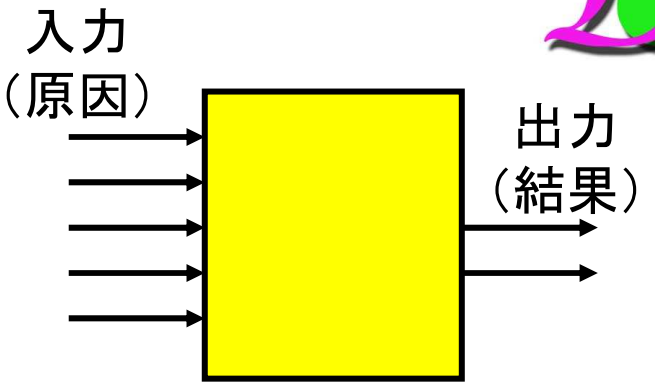
- *有則(機能がある場合)も*
- *無則(何も起こらない仕様)も網羅*

■欠点

- *因子i、水準j(i)とすると*
  テスト項目数は j(1) ×j(2) ×・・・j(n)

■実践

- *単体テスト*
  なら1番



		因子1		因子2			因子3		因子4		
		水準									
因子1	水準										
									■		
因子2					■						
							■				
因子3						■					
因子4											

結果1が生起 ■

結果2が生起 ■

何も生起しない ■

機能テスト

■方法

- *機能仕様から結果を求める*
- *結果が生ずる原因(入力)を求める*

■利点

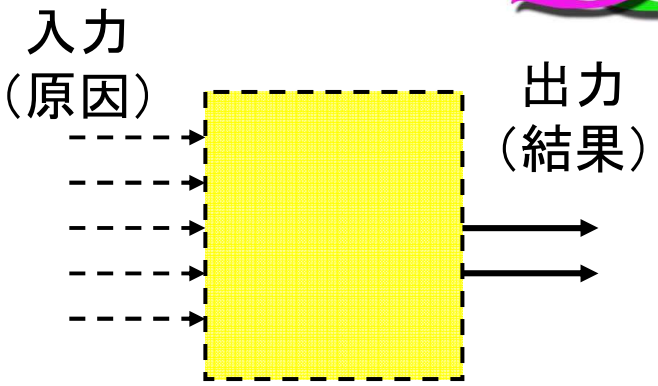
- *仕様にある有則だけテスト*
- *簡単にテストできる*

■欠点

- *有則でも原因の組合せが漏れる*
- *実装仕様から生ずる有則が漏れる*
- *無則はすべて漏れる*

■実践

- *原因の組合せが無く、かつ、無則について別の方法で確認できれば使える*



		因子1		因子2			因子3		因子4		
		水準									
因子1	水準										
									■		
因子2					■						
							▨				
								■			
因子3						▨					
因子4											

結果1が生起 ■ 結果2が生起 ■

テスト漏れ ▨

デシジョンテーブル

■方法

- *機能仕様から結果を求める*
- *原因の同値分割を行う*
- *機能が生起する原因の組合せを表現*

■利点

- *機能テストより、有則の網羅性が高い*
- *論理的な矛盾を検出できる*

■欠点

- *実装仕様から生ずる有則が漏れる*
- *無則はすべて漏れる*

■実践

- *無則について別の方法で確認できれば使える*

			1	2	3	4		
原因∧条件∨	因子1	同値	Y					
	因子2							
			Y					
	因子3			Y				
	因子4		N					
				Y				
結果	結果1		Y					
	結果2			Y				

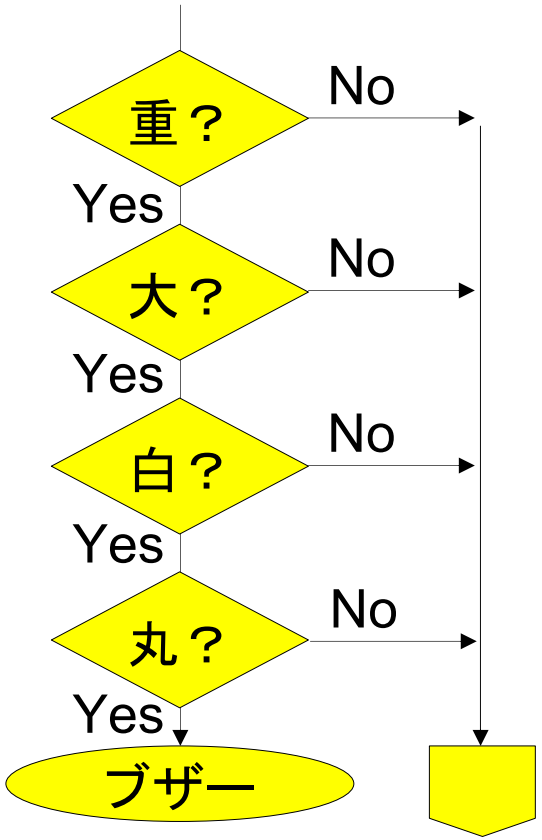
CFD（原因流れ図）によるデシジョンテーブル

■無則の網羅が問題！！

■　例　4つの原因からなるマトリックス（16項目）

- *原因1：色{赤、白}、原因2：型{丸、星}、原因3：大きさ{大、小}、原因4：重さ{重、軽}*
- *有則：重たい大きい白丸の時にブザーを鳴らす（1項目）*
- *無則：ブザーを鳴らさない（15項目）*

		赤		白	
		丸	星	丸	星
大	重			ブザー	
	軽				
小	重				
	軽				



■制御構造からテストすると5項目

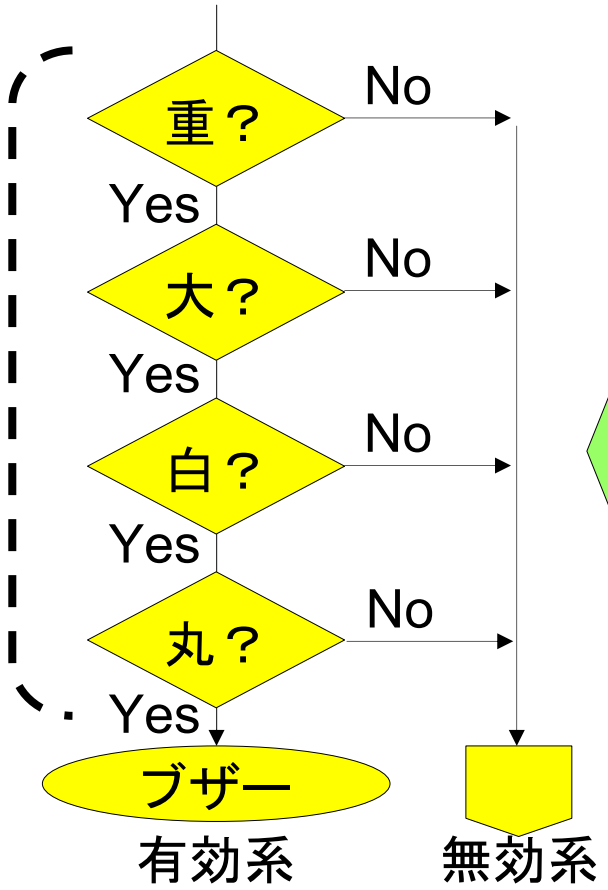
- *有則：1項目　　無則：4項目*

制御構造が明らかであれば

■制御構造からテストを設計すると5項目

- *有則:1項目　　無則:4項目*

実装仕様から生ずる流れはパス計測で検出



有則のテスト項目
仕様から抽出

無則のテスト項目
制御構造から抽出

		同値	1	2		4	5	6
原因	重さ	重	Y		Y	Y	Y	
		軽		Y				
	大きさ	大	Y			Y	Y	
		小			Y			
	色	白	Y				Y	
		赤				Y		
	型	丸	Y					
		星					Y	
結果	ブザー		Y	N	N	N	N	

CFDの網羅　制御構造による禁則の利用

■方法

- *モジュール構造を利用する*
- *有効系は処理の流れに沿って、結果の連鎖*
- *次処理に結合しないものを無効系*

■利点

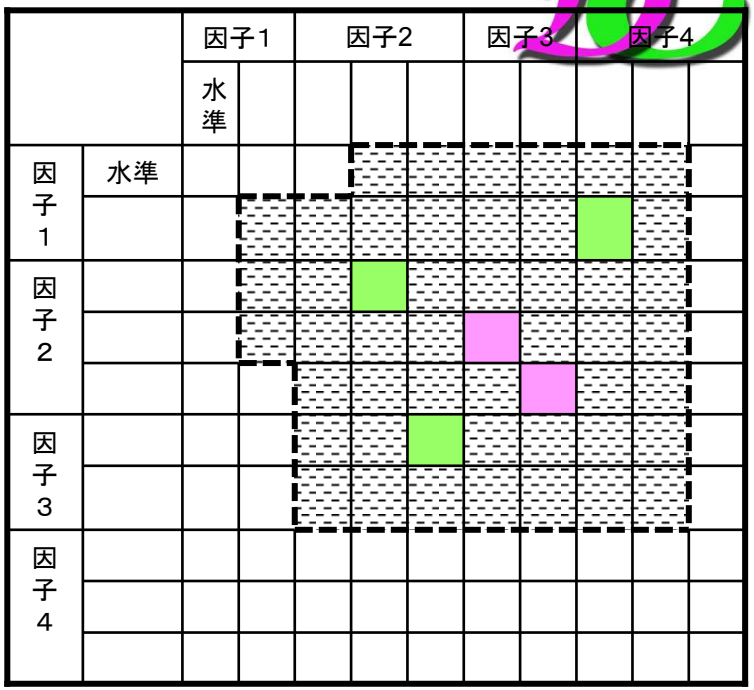
- *デシジョンテーブルの欠点を補う*
- *マトリックスと同等で、テスト量は激減*

■欠点

- *制御の結合が無い場合（すべて無則）は使えない（統合テストなど）　他の技法も同じ*

■実践

- *大きめの単体テストから結合テストまで使える*



結果1が生起

結果2が生起

何も生起しない

実装（構造）による禁則 ……

実験計画法、All-Pair法

■方法

- *独立した機能の組合せについて*
- *原因を因子*
- *原因のパラメータや同値を水準*
- *表かツールで組合せを求める*

■利点

- *機械的に組合せを生成*
- *漏れがない(条件あり)*

■欠点

- *有則があると別にテスト設計*
- *禁則があると避ける工夫が必要*

■実践

- *統合テストなどで応用範囲が広い*

		因子1		因子2			因子3		因子4		
		水準									
因子1	水準			1	1	1	1	1	1	1	1
				1	1	1	1	1	1	1	1
因子2							1	1	1	1	1
							1	1	1	1	1
							1	1	1	1	1
因子3									1	1	1
									1	1	1
因子4											

組合せによる
無害をテストする

Debug Engineering

＜＜合理的なテストとは＞＞

これから進むべき方向は
4. 統合アプローチ＜検証指向設計＞

テスト技法だけでは限界

- テストの仕事量は何で決まるか？
- 答え：探索空間の大きさ
- では、誰が探索空間の大きさを決めているのか？
- 答え：開発フェーズ（主に設計とコーディング）

品質管理は、原点に戻って、上流遡行

しかし、上流のバグを少なくする

・・・30年の戦いで戦果なし

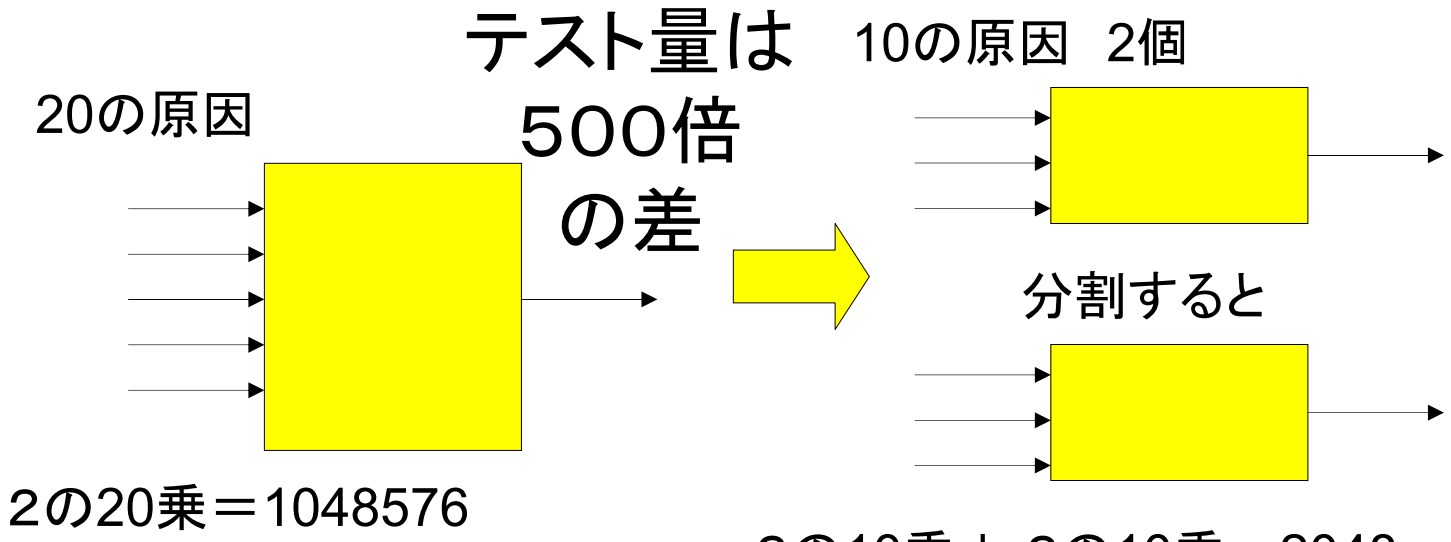
そうではなくて

1. バグの住む探索空間を小さくし、探索の合理化を行う

1番目の課題は探索空間

■適切なモジュール分割

- *プログラミングの規模制約より、テストの規模制約の方が小さい*
- *プログラミングだけ考えるとテストで困る　＜素人プログラマー＞*



2番目は、実装仕様の探索空間

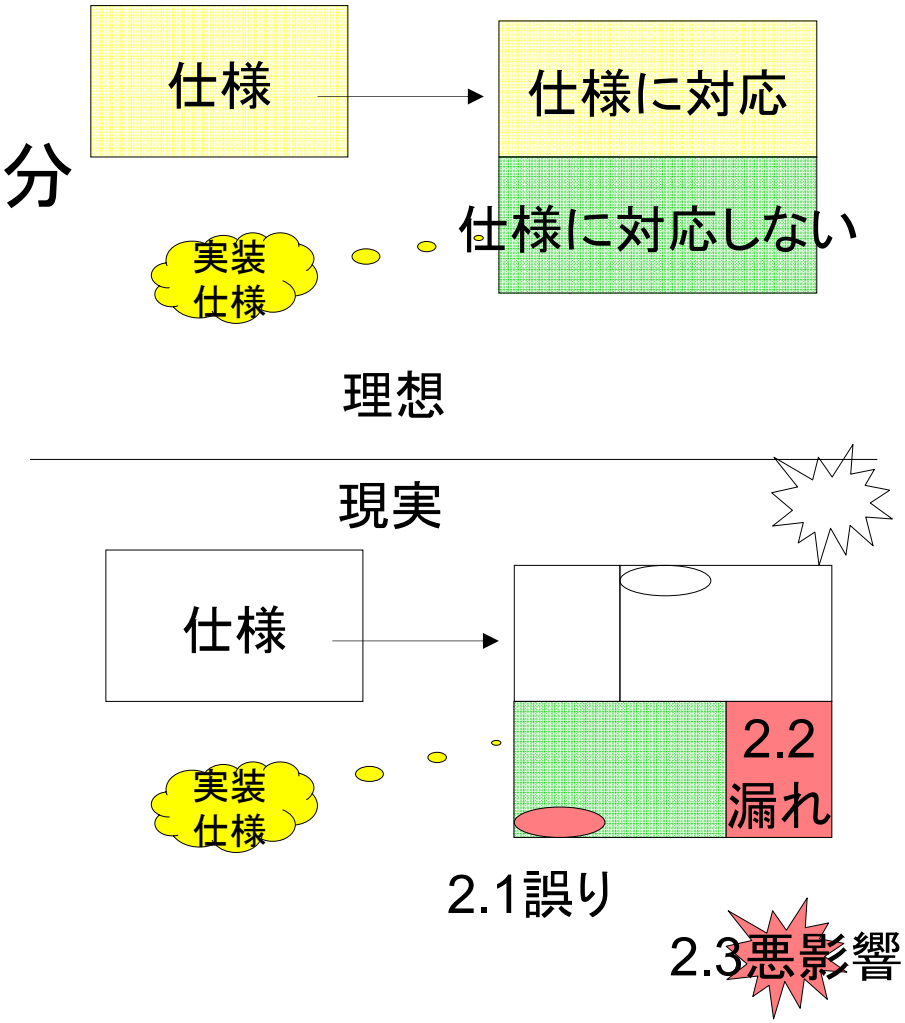
＜プログラミングの影響＞

さらに発見が困難

- 仕様に対応しない実装部分
  暗黙的に要求される実装
  実装仕様と呼ぶ

その誤り

1. *機能／論理　の実装**誤り***
2. *機能／論理　の実装**漏れ***
3. *悪影響(**冗長**)*



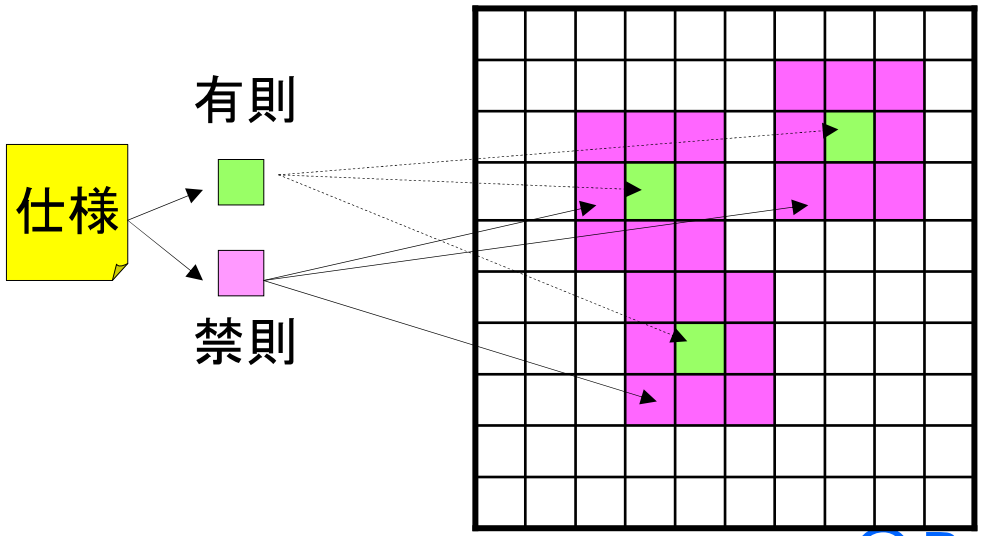
3番目は積極的な禁則

安全技術で実績あり

禁則：結果が生起しない機能

- *インターロックなど*

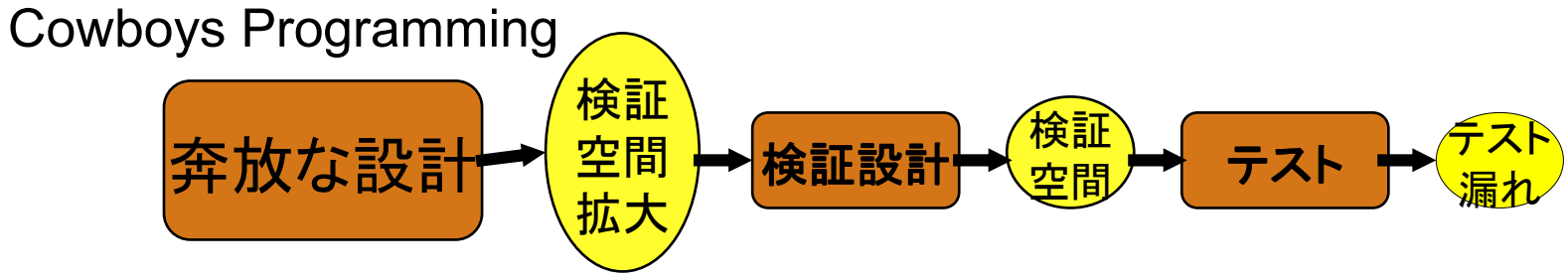
禁則により、悪影響を断つ

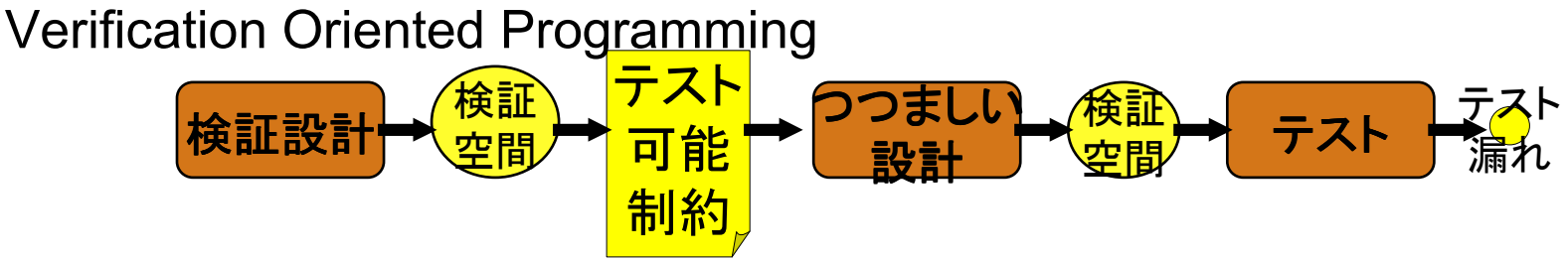


検証指向設計(1992)

■テストを効率的に行える工夫を設計段階で組込む

- XPのテストファースト(1998)も同様な考え方





統合アプローチ

- 高品質の実現には、確実な検算プロセスしか無い
- 開発とテストは、両者で検算プロセスを構成
- 検算プロセスを作り上げることを第一とするアプローチ

まだまだ黎明期

一つのアプローチとして、

検証指向プログラミング、検証指向設計・・・1992年から

Inverted Studyテストから設計を研究する・・2005年から

何れもテストを深く理解する必要がある

　CFDは1988年発表から20年・・・まだ発展中

まとめ

■テストを取り巻く課題

経営的な課題
テスト軽視

管理的な課題
近視眼的作業管理

技術的な課題
スキル不足

意欲の課題
劣悪な環境

トップダウンアプローチ

バグがなければ
テストは
不要な失敗コスト

後ろ向き作業
としての
テスト

■明日に向かって

モデル

戦略的
テスト

技法の
選択

技法実践
スキル

テスト
仕事満足

成功事例

開発

必要
不可欠
な
検算
プロセス

テスト

DebugEng
はこれらをサポート