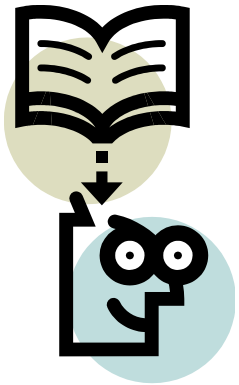
ソフトウェア・テストの30年前と30年後

ソフトウェア・テストの未来を探る



2012年1月26日
東海大学大学院組込み技術研究科
山浦 恒央 (YAMAURA, Tsuneo)
yamaura@keyaki.cc.u-tokai.ac.jp

1. ソフトウェア開発技術の歴史

1964年
OS/360
の開発

1968年
「ソフトウェア
工学」の誕生
（言葉のみ）

1975年
「ソフトウェア
開発の神話」

1978年
「ソフトウェアテストの技法」

1986年
「No Silver Bullets」

1960年　1970年　1980 年　1990年　2000年

メインフレーム全盛期　過渡期　PC全盛期

オンラインプログラミング

ソフトウェア・メトリクス

アジャイル開発方式

高級言語

構造化プログラミング → オブジェクト指向

第3世代言語

CASEツール

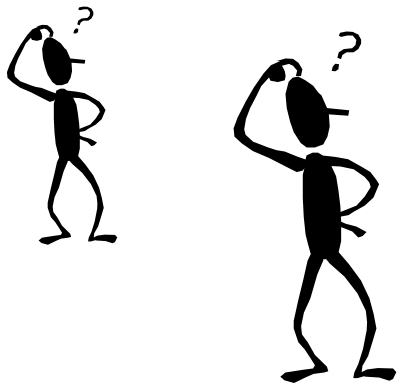
人工知能・エキスパートシステム

2．30年前のプログラミング開発

●重長広大型のソフトウェア開発が中心
・銀行のオンラインシステムや、国鉄（現JR）の緑の窓口
・ソフトウェア・パッケージを使わず、全て自前で開発した。

●システム寄りのプログラミングが花形扱いを受ける。
・例えば、OSやコンパイラ

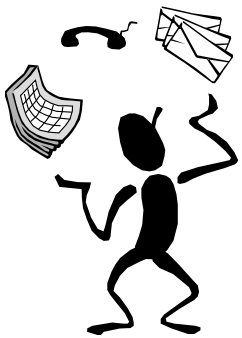
●メインフレームが全盛
⇒PCは、ごく一部のマニア間でもてはやされた。



2. 30年前のプログラミング開発

- 開発モデルは、ウォーターフォール・モデルしかなかった。
- 言語は、アセンブラーとCOBOLが中心
- 生産性は、1,000LOC/人月、バグ密度は、10個/1,000LOC
- 品質は、スケジュールやコストより圧倒的に重要だった。

  ⇒品質目標に達しない限り、出荷しなかった（スケジュールとコストを犠牲にした）。



2. 30年前のプログラミング開発

●ソフトウェア危機（Software Crisis）が叫ばれた。

⇒品質と生産性を上げようと、いろいろ試行した。

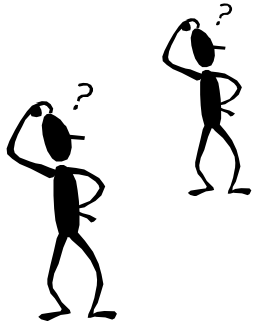
⇒最も期待されたのが「再利用（部品化）」だが、挫折した。

・正確には、「小規模再利用は成功したが、大規模再利用は失敗」

●構造化プログラミングが最先端の開発技術だった。

・あらゆるものに、無条件に「構造化」が付いた。

構造化技法、構造化ツール、構造化モジュール、……



2. 30年前のプログラミング開発

●メモリーが非常に高価だった。

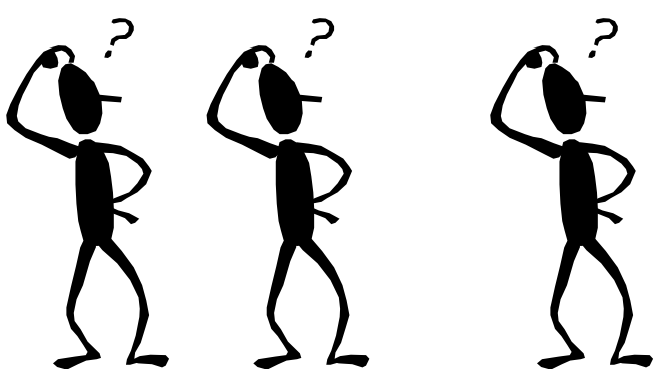
●実メモリーに制限がある。

⇒オーバレイを使い、自分で「仮想記憶もどき」を実装した。

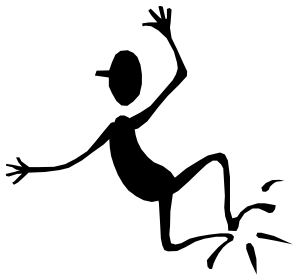
⇒メモリー占有量が少ないプログラムほど、優れたプログラミングであると称賛された。

●作りやすさや理解容易性より、メモリー効率や性能を追求した。

⇒「理解容易性 vs 効率」はソフトウェア開発の永遠のテーマ



当時のアセンブリ言語による(職人的な)プログラミング例



●自分が理解できればよい。

⇒自分が作っても、3ヶ月経てば、理解が難しい

●「1ステップでも短く、1マイクロ秒でも速く！」の職人芸がまだ全盛

```
          ・・・・・・
          LD R9, TOKYO_AVE01      ⇒レジスタ9に「東京の平均点」をロード
          LD R8, JAPAN_AVE01      ⇒レジスタ8に「日本の平均点」をロード
          LD R7, STUDENT1         ⇒レジスタ7に「生徒の平均点」をロード
COMP01    CR R7, R9               ⇒レジスタ7とレジスタ9を比較
          BG END1                 ⇒レジスタ7の方が大きい場合、END1へ
          ・・・・・・
          LI  R6, X'AF78'         ⇒レジスタ6に「AF78」をセット
          ST R6, COMP01           ⇒レジスタ6の内容「COMP01」番地へセットし、
          ・・・・・・                 「 CR R7, R9 」を「 CR R7, R8 」に書き替え る。
```

３．30年前のテスト技術

●「品質の高いソフトウェアを作らねばならない」との意識はあった。

●しかし、テスト技術は自己流。

⇒組織全体で採用した「テスト技術」は存在しない。

⇒各部署で、先輩（職人？）から引き継いだ技術を伝承。

●ワープロ用ソフトウェアから、原子力発電所の制御プログラムまで、同程度の高品質を求めた。

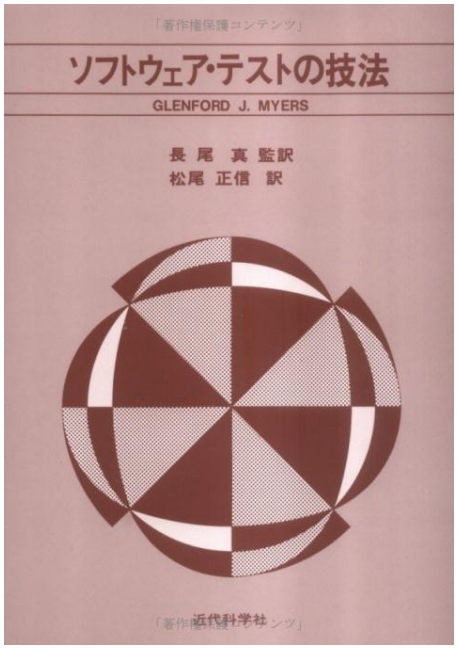
⇒リスク・ベースの考え方は不在。

●開発プロセス、テスト・プロセスの意識はない。

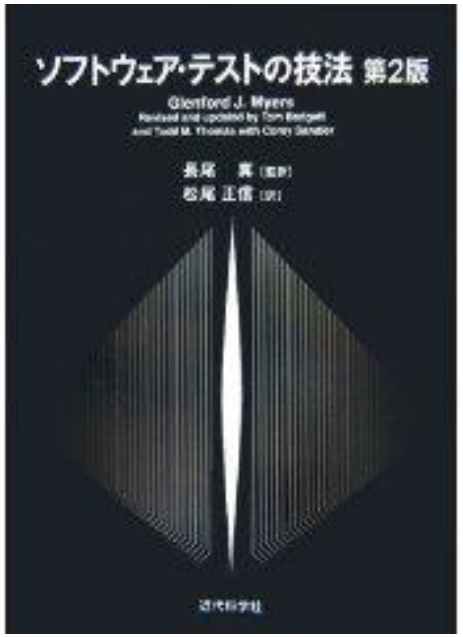
●定量的な品質測定を模索

3. 30年前のテスト技術

1980年登場の「テスト関連書籍」のベストセラー



オリジナル版



改訂版

『ソフトウェアテストの技法（The Art of Software Testing）』

●原著：グレンフォード・マイヤーズ著（1978年）

●訳書：長尾真訳(1980年、2,835円) 近代科学社

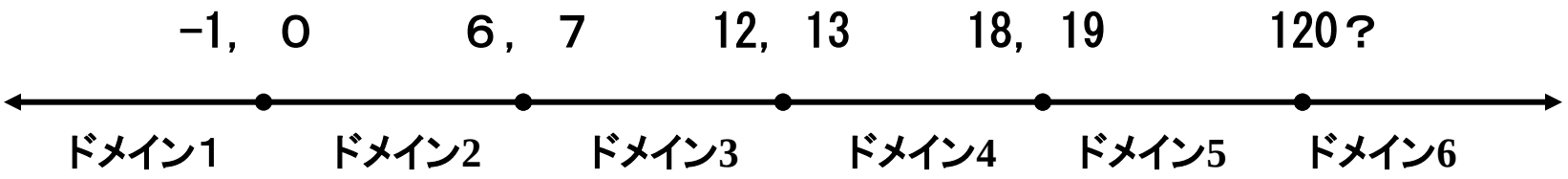
本を読む技術者は少なく、一般に浸透するには時間がかかる。

4. 30年前から不変のテスト技術（その1）

●同値分割

入場料: 6歳以下　:無料　　7歳～12歳: 500円

13歳～18歳:800円　19歳以上: 1,000円



・各入力ドメインから、1つの値を選び、それをドメインの代表値とする。

・例えば、ドメイン1から－2、ドメイン2から5、ドメイン3から10、
ドメイン4から15、ドメイン5から100、ドメイン6から150を選ぶ。

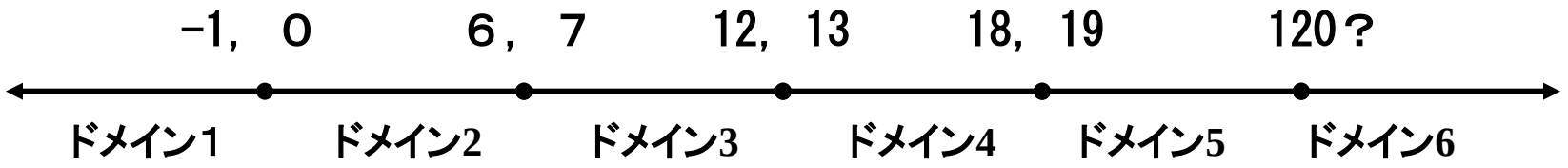
⇒ 必要以上にテスト項目を増やさないための戦略

４．30年前から不変のテスト技術（その２）

●境界値分析（バグは境界上に密集している）

入場料: ６歳以下　:無料　　７歳～12歳: 500円

13歳～18歳:800円　19歳以上: 1,000円



各入力ドメインの境界値をテスト項目として選ぶ。

上記の場合、－１、０、６、７、１２、１３、１８、１９、(１２０？)

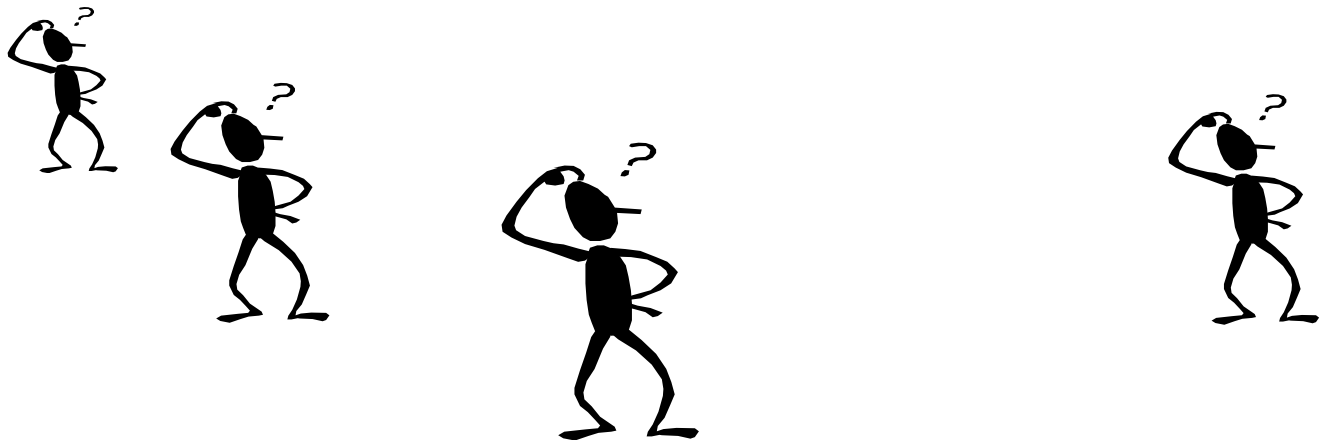
5. プログラム開発に対する当時の態度

(1)ソフトウェア開発は科学である。

⇒数学が苦手な理科系と、謎解きが好きな文科系が集まった

(2) ソフトウェアの開発は、自動車の製造、ビルの建築などと同類の技術である。

(3) 将来、技術が発達すると、ソフトウェアの自動生成が可能になり、生産性や品質が急激によくなる。



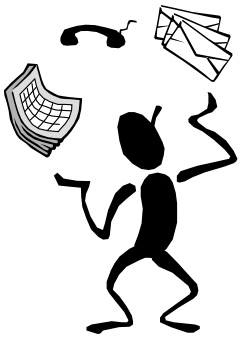
6. 現在の生産性とテスト技術

●30年前から、以下はほとんど変化していない。

- ソフトウェアの生産性（1,000LOC/人月）
- プロジェクト管理技術
- テスト、デバッグ技法、品質制御法

●いまでも、「ソフトウェアテストの技法」はベストセラー

●品質は、バグ密度から見ると、10個/1,000LOCから5個/1,000LOCに改善された。これは、高級言語の導入の成果。



7. ソフトウェアの開発がこれほど難しい理由（ブルックスの説）

① プログラムは、規模が大きい。

一人乗りのボートは簡単に作れても、1,000人乗りの船は難しい。

② ソフトウェアは、目に見えない。

手で触れないし、大きさ、重さもないので、全体像を簡単には、捉えられない。開発の難しさも実感できない。

③ プログラムは、簡単に変更できる。

物理的製品の開発では、後で変更がないよう、注意して作るが、ソフトウェアでは、変更が簡単でコストがかからないため、注意しない。

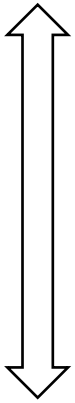
④ 制約となる自然界の法則が少なく、自由に作れる。

重力、磁力、電流などの法則に束縛されないため、考え方が正しければ、正解となる(芸術の世界)。

7. ソフトウェアの開発がこれほど難しい理由

●プログラミング開発のプロセスには、順序性があるため（1つのプロセスが完了しないと、次へ進めない）、分業が簡単ではない。

分業可能（人数を倍にすると、時間は半分になる）



●穴を掘る、皿を洗う、・・・

●家を建てる

●プログラムを開発する

●子供を産む(10人が手伝っても、10ヶ月かかる)

分業不可能（増員しても、期間に影響しない）

7．ソフトウェアの開発がこれほど難しい理由

●ソフトウェアの開発は、「電子製品を設計・製造する」よりも、
「小説を書くこと」に似ている。

●時代が進んで、ワープロが発達しても、小説執筆の「速度」、
「品質」、「生産性」が上がらない。

⇒紫式部の時代から、ほとんど改善は見られない。

８．30年後のソフトウェア開発技術とテスト技術

●ソフトウェア開発が「小説の執筆」に似ているなら、画期的な開発技術や品質制御技術は期待できない。

●生産性は、新規開発の場合、1,000LOC/人月、バグ密度は、5個/1,000LOCで、今とほとんど変わらないだろう。

●ただし、応用する経験値が上がったり、開発プロセスが進化する。

8. 30年後のソフトウェア開発技術とテスト技術

●30年後に成熟することを期待する品質制御関連の技術

(1) 品質のレベル分け

(2) 応用分野別の品質情報データベースの確立

(3) リスク管理をベースにした品質制御プロセスの確立

(4) 品質のカプセル化

８．30年後のソフトウェア開発技術とテスト技術

(1) 品質のレベル分け

●日本の製品規格

・工業製品はJIS規格

・農林水産農業製品はJAS規格と食品表示（品質表示基準）

●ソフトウェアにも、品質規格が必要では？

8. 30年後のソフトウェア開発技術とテスト技術

(1) 品質のレベル分け

●ソフトウェア品質規格に期待できる効果

・投資では、債券や国債にAAA、Baaのように格付けし、投資者が投資リスクを選べるのと同様、ユーザが品質を含めて、ソフトウェアを選べる。

・将来、必ず発生する「ソフトウェアのバグにより被った損害の賠償、および、社会的責任問題」に対応するためには必須。

・ソフトウェア開発を発注する場合、要求する品質レベルを明示できる。

8. 30年後のソフトウェア開発技術とテスト技術

(1) 品質のレベル分けの具体例

レベル1：レベル2から5に該当しない（＝正常ケースも動作しない場合がある）。
C0網羅、C1網羅とも計測せず。

レベル2：正常系はすべて、異常系は代表的なケースでのみ正しく動作する。
C0網羅80%、C1網羅は計測せず。

レベル3：正常系、異常系はすべて、組合せ系は代表的なケースでのみ正しく動作する。C0網羅80%、C1網羅50%以上

レベル4：正常系、異常系、組合せ系がすべて正しく動作する。C0網羅100%、C1網羅80%以上

レベル5：過負荷テスト、最小構成テスト、長時間耐久テスト等をすべて実施。
C0網羅、C1網羅とも100%

8．30年後のソフトウェア開発技術とテスト技術

(2)応用分野別の品質情報データベースの確立

・ソフトウェアの応用分野が広範囲に渡るため、品質制御方式を一般化するのは困難で、非現実的。

・デジタルカメラ制御用ソフトウェアと携帯電話用プログラムでは、同じ組込み系ソフトウェアであっても、「常連バグ」や、バグが出やすい状況は大きく異なる。

・各応用分野の「常連バグ」をデータベース化し、それを自社製品用にカスタマイズすることで、応用分野ごとの品質制御のレベル上げる。

8. 30年後のソフトウェア開発技術とテスト技術

(3)リスク管理をベースにした品質制御プロセスの確立

・携帯電話のソフトウェアの規模が10Mステップを越える今、30年前のように、じっくりテストする時間的な余裕のない企業は多い。

・機能に優先度をつけ、優先度に合わせてテストの質や量を変えたり、システム破壊、環境破壊などを引き起こす重大不良を重点的にテストするなど、限られた「テスト資源」をリスク・ベースで割り当てる品質制御プロセスが増えるはず。

８．30年後のソフトウェア開発技術とテスト技術

(4) 品質のカプセル化

・オブジェクト指向による「カプセル化」と「情報隠蔽」により、機能とデータをカプセル化するだけでなく、品質もカプセル化する。

・再利用において、「機能の再利用」に注目するだけでなく、「品質の再利用」も効果が大きいことを認識する。

・「品質のカプセル化」により、「いかに、多数のテストを効率よく実施するか？」から、「いかに、テストしないで済ませるか？」にパラダイムをシフトする。

８．１．ソフトウェア・テストでのパラダイム・シフト

●開発の基本思想のシフト

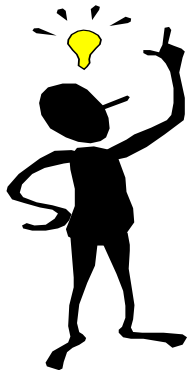
・「いかに、早く大量に作るか？」⇒「いかに、作らないで済ますか？」

●テストの基本思想のシフト

・「いかに、効率よくテスト項目を設計し、テスト項目を効率よく実施するか？」

・「いかに、テストしないで済ませるか？」

●この鍵になるのが、「再利用」「オブジェクト指向」



8．1ソフトウェア・テストでのパラダイム・シフト（再利用）

再利用における「品質の向上」効果を再認識する。

再利用で期待できる効果

●生産性の向上

・開発期間の短縮と開発コストの削減

●品質向上

・信頼性の向上

⇒既に、あるレベルの品質が確保されている。

・保守性の向上

⇒稼動しているので、同じような保守をすればよい。

●リスクの回避

・実現可能性の保証

・性能保証

８．１ ソフトウェア・テストでのパラダイム・シフト（再利用）

●再利用を成功させる3つのポイント

①再利用元と再利用先が同じ応用分野であること（例えば、デジタルカメラ同士の再利用）。

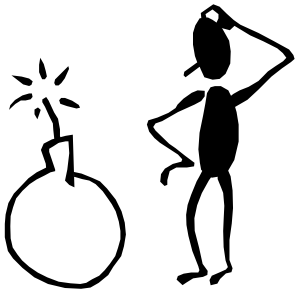
②改造量が20%以下

③設計ドキュメントが全て最新の状態で揃っている。

●再利用での最大の課題は、「ソースコードを簡単に理解して、改造箇所をピンポイントで見つけられるか？」 この解決策は以下の2つ。

①理解しやすいように作る。 あるいは、反対に、

②理解しなくても流用できるようにする。

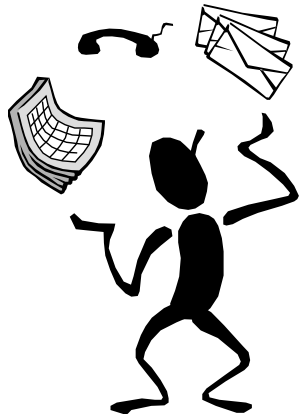


８．２　ソフトウェア・テストでのパラダイム・シフト（構造化プログラミング ）

●「ソフトウェア危機」を解決すると期待されたプログラミング方式

●「品質が高い」、「作りやすい」、「理解しやすい」、「再利用しやすい」、「保守しやすい」プログラミングを実現することを目的にした。

●プログラミング作法の元であり、プログラミング制御構造の超基本（プログラマの常識）。

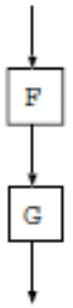


８．２　ソフトウェア・テストでのパラダイム・シフト（構造化プログラミング ）

理解しやすい、作りやすいプログラミング

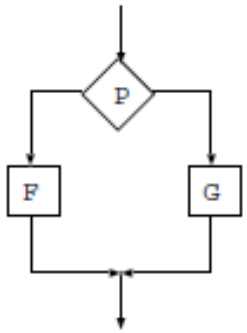
①どんなプログラムでも、連結(concatination)、選択(selection)、反復(iteration)、呼出し(call　or invocation)の４つで表現すること。

②goto文をなるべく使用せずにプログラミングすること（goto文はエラー処理に限定し、かつ、「前方goto文」に限る）。



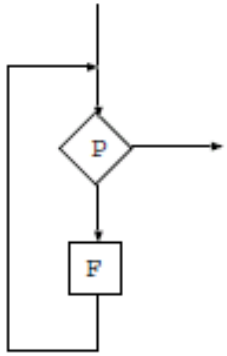
連結

F Followed by G



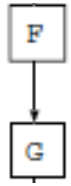
選択

if P, then F else G



反復

while P do F



呼出し

F Calls G

８．２　ソフトウェア・テストでのパラダイム・シフト（構造化プログラミング ）

品質の向上、再利用の促進、生産性の向上、保守性の向上において、

●構造化プログラミングで解決できたこと

①プログラミング構造が、作りやすく、理解しやすくなった。

②一部の再利用が容易になった(共通サブルーチン) 。

●構造化プログラミングで解決できないこと

①本来の再利用は、まだまだ簡単ではない。

②グローバル変数の問題

●これを解決するためにできたのが「オブジェクト指向プログラミング」

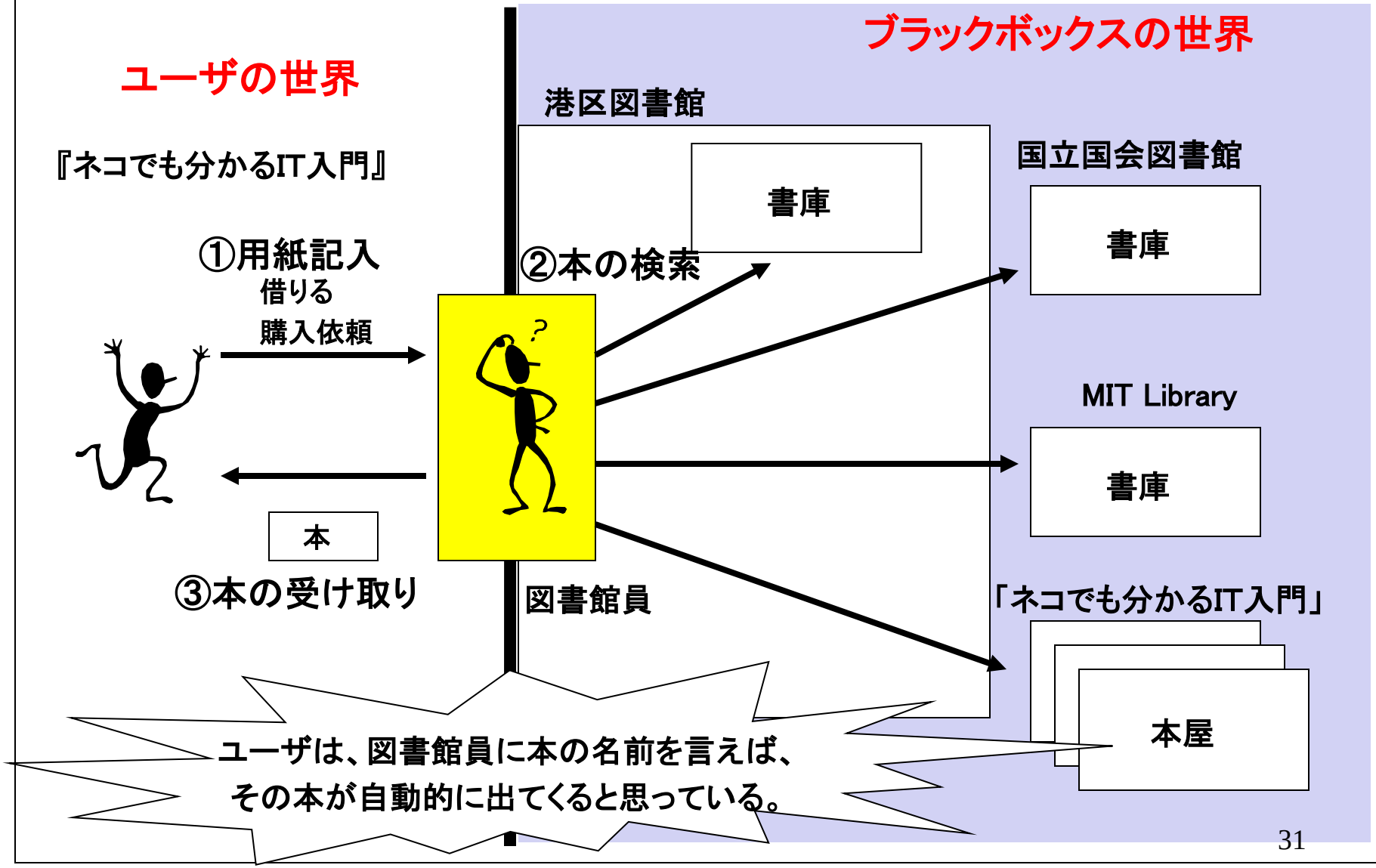
①基本思想は、「抽象化」と「情報隠蔽」

②データを構造化したのがオブジェクト指向

③オブジェクト指向は、再利用のための技術との思想が必要

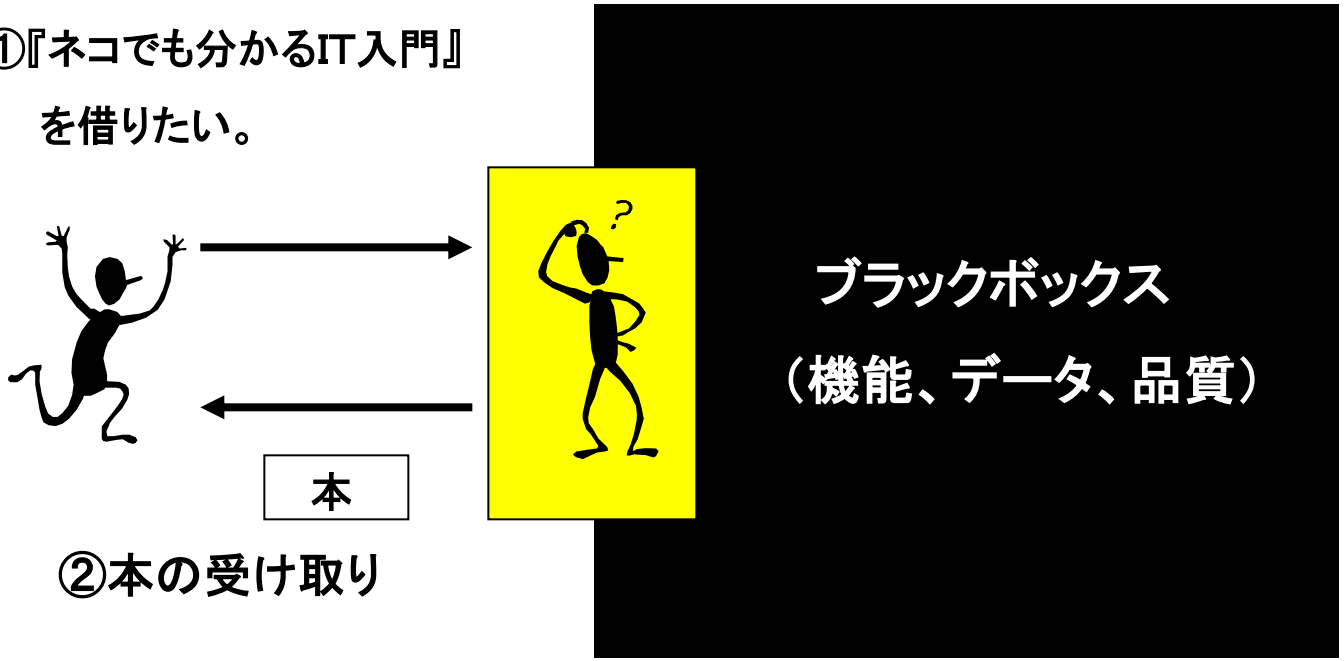
８．３　ソフトウェア・テストでのパラダイム・シフト（オブジェクト指向）

データの物理的な構造や場所を知らなくても(複雑なことを知らなくても)、データにアクセスしたい。



８．３　ソフトウェア・テストでのパラダイム・シフト（オブジェクト指向）

データと処理と品質のカプセル化



●ユーザには、図書館のデータ、処理方式はすべてブラックボックス。

●再利用する設計者には、データ、処理方式、品質をカプセル化して扱える。

⇒オブジェクト指向と再利用による品質のカプセル化