

ヒューマンキャピタル2008

# ITアーキテクトの重要性とその育成法

～アメリカの成功事例から学ぶ～

**水野匡章**

カンザス州立大学

コンピュータサイエンス学科教授

ラーニングツリー・インターナショナル講師

# IT時代の特異性

ITの時代は今までの機械・電気中心の産業の時代に比べて、以下の大きな特徴がある

## 1. 「賞味期間の極端な短縮」


- 技術者の持つ技術の賞味期間の短縮
- 製品の賞味期間の短縮
- 職業の賞味期間の短縮

## 2. 「長期間の機能の追加による製造物（ソフトウェア）の中心部のライフサイクルの長期化」

- 以上の2点は、製造業の理想的な形態、技術者の育成法などに大きな影響を与える
- しかし多くの日本の企業は、機械・電気の時代に作り上げた仕組みを変えていない



# アウトライン

- 
1. 技術者の持つ技術の賞味期限の短縮
  2. ソフトウェア製品の中心部のライフサイクルの長期化
  3. 最新のソフトウェア設計法と管理法
    - ITアーキテクトを中心においた設計・実装とプロジェクト管理
    - 日米におけるケーススタディ
  4. アーキテクトの育成法
  5. まとめ



# IT技術者の持つ技術が通用する期間の短縮

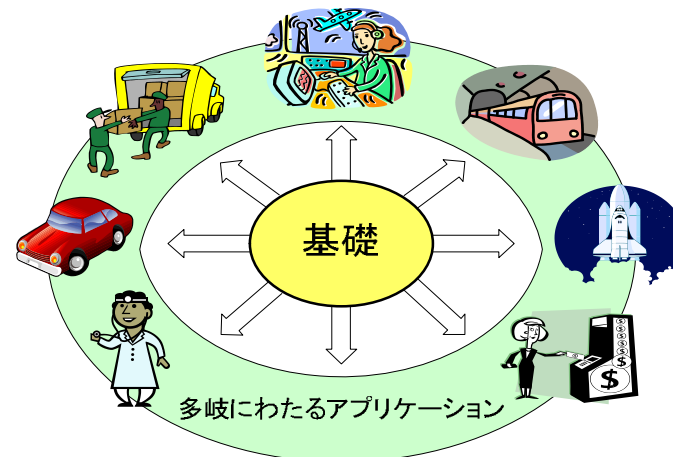
- 機械系製造業大手のA社、B社の部長
  - 機械系では大学で基礎を学び、それを元にオン・ザ・ジョブ・トレーニング（OJT）で仕事に必要な技術を磨いていく
  - 一方コンピュータの分野では大学で学んだことはすぐに陳腐化してしまうが、コンピュータに「基礎」はないのか？



# IT技術者の持つ技術が通用する期間の短縮（続き）

コンピュータ科学に基礎はないのか？

- 論理（ロジック）、アルゴリズム、開発言語及びプログラム技法などは基礎である
- しかしコンピュータは応用分野（アプリケーションの範囲）が極端に広いため、それらの最大公約数である基礎と各アプリケーションとのギャップが大きすぎる
  - 基礎は知っていてもそれだけで個々のアプリケーションを開発するのは不可能
  - 「基礎 + アプリケーション固有の知識」が必要
- しかもアプリケーションの領域は日々目覚ましい速度で広がっている
  - 「アプリケーション固有の知識」を学び続けなくてはならない



## IT技術者の持つ技術が通用する期間の短縮（続き）

- コンピュータ以前の技術者は学校で基礎を学び、OJTで技術に磨きをかけた
- コンピュータ技術者にはこの技術習得のモデルは通用しない
  - このような最新の「アプリケーション固有の知識」を深いレベルでOJTで学ぶのはほとんど不可能
    - 周りに最新の知識がある人たちがいない
    - 知識の無いところにOJTはありえない
- さらに、第一線のIT技術者であり続けるためには、（単なる概要ではなく）プログラム（ソースコードのレベルで）に接し続ける必要がある
  - 中島聡氏（Windows 95のユーザ・インターフェースのチーフ・“アーキテクト”）のブログ[2]より
    - 「自信を持って言えるのだが、“どんなに優秀なエンジニアでも、決してプログラムを自分自身で書かずに良い詳細仕様を作ることは出来ない”という絶対的な法則があるのだ。私の知っている優秀なエンジニアは、皆それを知っており自ら実行している。」
    - 「実際にプログラムを書き始めて初めて見えてくること、思いつくことが沢山あるので、それを元に柔軟に設計を変更しながらプログラムを書き進めるのである。作っているプログラムが予定通りに動き始めてやっと、設計も完成に近づくのである。」



## IT技術者の持つ技術が通用する期間の短縮 ( 続き )

- 特に日本では、社内での地位の向上とともに、優秀なプログラマでもプログラムを書かなくなってしまう場合が多い
- したがって、日本の企業は5年前、10年前、20年前の技術しか(深いレベルで)知らない技術者を大量に抱え始めていて、その傾向は加速している
  - アメリカではコンピュータ・サイエンス学科は1960年代から存在
  - しかもプログラミング経験20年以上というスーパー・プログラマ(アーキテクト)が多数おり、彼らが企業の中核にいる
  - 日本ではコンピュータ・サイエンス学科の歴史が浅く、「コンピュータ・サイエンス」という学問自体を理解している人が極端に少ない
- IT技術者には、OJTではなく生涯教育(学び続ける姿勢、学び続けられる環境及び機会)の提供が重要



# アウトライン

1. 技術者の持つ技術の賞味期限の短縮

➡ 2. ソフトウェア製品の中心部のライフサイクルの長期化

3. 最新のソフトウェア設計法と管理法

- ITアーキテクトを中心においた設計・実装とプロジェクト管理
- 日米におけるケーススタディ

4. アーキテクトの育成法

5. まとめ



# ソフトウェア製品の中心部のライフサイクルの長期化

## 対処療法と内部構造改善（リファクタリング）

- ソフトウェアの開発は、「根本的なつくりかえをせずに、長時間にわたり、少しずつ機能の追加を行う」という特質がある
  - この様なことは、機械（歯車やポンプからなるメカ）や電気（電子基板：ソフトを含まない）であったか？
- そのため、長期的に中心構造がだんだんと現在の機能のセットに合わなくなってしまうたり、中心部がブラックボックス化したりしてしまうケースが多い
- このようなことに対する方策として、リファクタリング（機能を変えずに中身をきれいにする）があるが、リファクタリングの概念自体がソフトの専門家以外には理解されていないのではないか？
  - 従ってリファクタリングに費用や人材をつぎ込まない
  - 一見動いているソフトのパーツは新たなバグを出すのが怖くていじりたくない
- 巨大なブラックボックスの表面に少しずつ機能を追加していくという対処療法をとり続け、大きな変革が必要になったときに対応できなくなる危険がある
- 「巨大なブラックボックス化」を防ぐにはどうしたらよいか？



# アウトライン

1. 技術者の持つ技術の賞味期限の短縮
2. ソフトウェア製品の中心部のライフサイクルの長期化
3. 最新のソフトウェア設計法と管理法
  - ITアーキテクトを中心においた設計・実装とプロジェクト管理
  - 日米におけるケーススタディ
4. アーキテクトの育成法
5. まとめ



# 最新のソフトウェア開発法（RUP）

- デジタル機器のメーカーはその中枢であるソフトウェアの生産性をあげるべき：  
ソフトウェア
  - の信頼性をあげる
  - を効率良く作る（費用、期間）
  - をそのライフサイクルにわたって効率よく管理する
- ここで、生産性をあげる上で筆者がもっと理にかなっていると考え、最新のソフトウェア開発法であるRational Unified Process (RUP)[1]を紹介する
  - さらに筆者の身近で、その方法を使いうまくいっている例を紹介する（ケース・スタディ）
- RUPの3つの柱
  - ユースケース駆動
  - アーキテクチャ中心
  - 繰り返しによる開発



# ユースケース駆動

- ユースケース駆動：要求仕様（ユースケース）をはっきり決めて、それを基に以降の設計・実装・テストから成る開発プロセスを推進する
- 日本では、ソフトウェアの「設計」プロセスについて誤解されている場合が多い
  - 設計というのは、いかにして（how）仕様（what）を実現するかということを決めるプロセスである
    - 日本では仕様書（what）を設計書という呼ぶ場合が多い
    - 「仕様設計」による丸投げ文化
- 設計の生成物（設計図）がUML図（クラス図、ユースケース実現）
  - 最新の設計法（RUP）における設計とはUML言語（グラフィック言語）によるプログラミング
  - 設計図が最終コードをはっきり規定している
    - きちんとした設計をすればそれはほぼ機械的にコードに落ちる
  - 設計図は最終コードよりはるかに「変更を加えやすく」「理解しやすい」
    - ソフトウェアの「見える化」
- I. Jacobson博士（RUPの開発の中心、UML2000での基調講演）：「現在はプログラミングとデバッグにかける時間が多すぎる。将来はその比率を8（設計）対2（プログラムとデバッグ）位にするべき。」



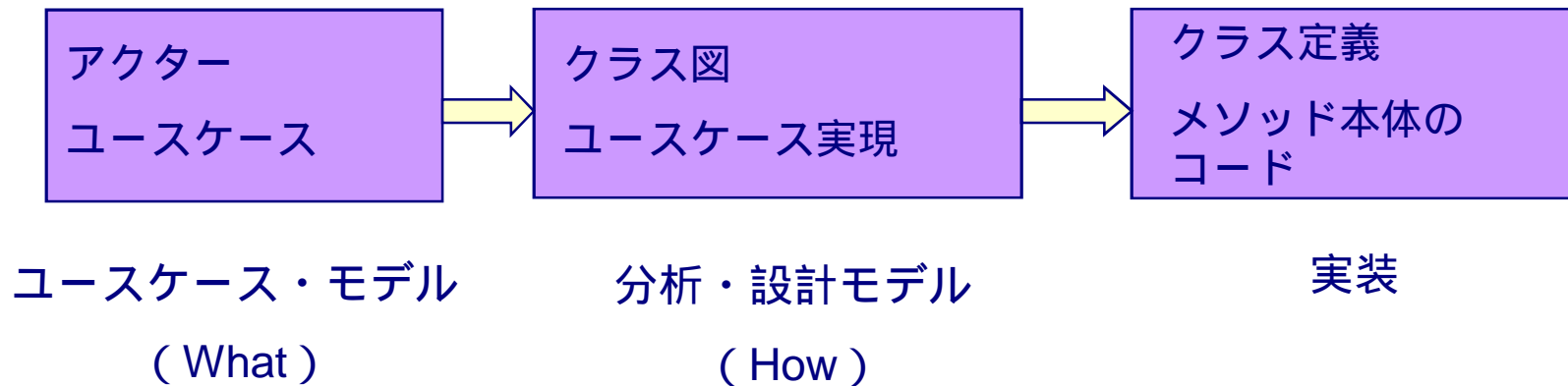
## ユースケース駆動 ( 続き )

- 多くの日本の企業で
  - 設計とっているプロセスは設計ではなく ( 仕様補足であり )、
  - プログラミングとっているものの多くが ( 非常に効率の悪いやり方をしているが ) 設計なのである
- つまり、プログラマを育てないということは設計者を育てないということである
  - ソフトウェア産業における危機的な「技術の空洞化」

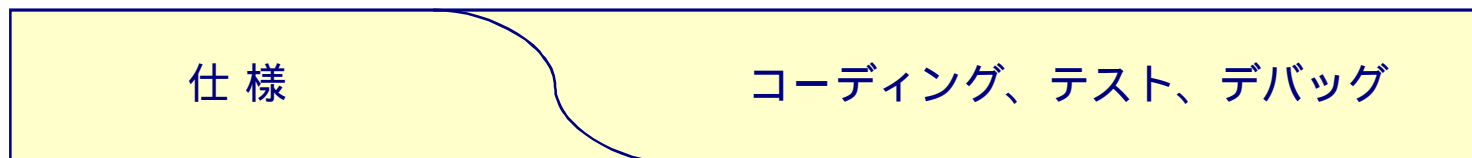


# ユースケース駆動 ( 続き )

## 最新型 ( RUP ) :



## 旧来型 :



# アーキテクチャ中心

- アーキテクチャ中心

- アーキテクチャとはシステムの最重要機能をカバーする、あるいは開発工程で最もリスクが大きいと思われる要求仕様（全体の10%くらいといわれている）の設計・実装
  - アーキテクチャが最終プロダクトの構成の核になる
- この開発を行うのが優秀なプログラマであるアーキテクト
  - リスクを正しく見つけ出す力は、アーキテクトの経験、能力に基づく
  - 細部まで詰めないと（コーディングまで行わないと）、リスクの発見及び対処はできない
- アーキテクチャができた時点で、致命的なリスクは無くなっている
  - アーキテクトが重要で複雑な部分の実装まで行う
  - 他のプログラマが、その他の部分を実装する
- これは、製造の上流（設計）と下流（組み立て）を分けるコンピュータ化以前の製造法や、古いソフトウェア工学の手法とは異なる



# 繰り返しによる開発

- 繰り返しによる開発：以下の4つのフェーズを、ユースケース駆動を何度も繰り返しながら進行する
  - 方向付け：ビジネス観点からの評価
  - 推敲：アーキテクチャの決定及び次の実装フェーズの詳細な見積もりの決定
    - アーキテクチャが安定した時点で殆どのリスクはなくなっている
    - 正確な見積もりが立てられる
  - 実装：残り（90%）の仕様の分析・設計・実装・テスト
    - このフェーズが最も期間が長く多くの開発要員が投入されるが、見積もり通りにスムーズに進行する
      - トータルの開発時間は短くなる
    - 必要ならば、アーキテクトはアーキテクチャの改善を行う
      - 全体の構成が、きれいで効率よく保たれる
      - ブラックボックス化を防ぐ
        - 製品の保守が容易になる
  - 移行：製品リリース
    - アーキテクトは、ベータ・リリース（テスト・リリース）からのフィードバックに基づく変更に対しても、アーキテクチャの整合性を保つ



# アーキテクト不在の問題点

- 航空機のソフトなど、高度な信頼性が要求されるようなシステムをアーキテクトを置かずに開発することは可能か？
  - 優れた全体構成を決定でき、常に全体を把握し続けるアーキテクトが不在では、巨大ソフトのブラックボックス化は避けられないと思う
  - 各プログラマが全体の一部しか知らず、システムの多くの部分をブラックボックスとしてしか見ることができないような人員構成で、安全なシステムを開発することは可能か？
    - 開発側から見ずに、顧客の立場になって、例えば「自分はそのように開発された旅客機に乗るか？」という目で見してほしい



# 日本の現状

- 日本では、アーキテクトのキャリアパスを設けている会社は多くない
- 日本では非常に多くのテストを行い、バグが見つかったとそれに対して対症療法を付すという方法をとる場合が多いようだが
  - それは、テストがカバーしたケースに関して正しく動くという保証にしかない
  - 巨大なブラックボックス化したソフトウェアは、最後には対症療法も不可能になり、崩壊するのでは？
- 日本のマネージャからよく聞く意見：
  - 本体の給与は関連会社のそれより高いので、コストの点から本体でプログラミングを行うわけにはいかない
  - 個々のプログラムの生産性には数十倍（あるいはそれ以上）の差があるので優秀なプログラマー（アーキテクト）ならば本体で雇用していてもコスト的な問題は無いであろう



# アウトライン

1. 技術者の持つ技術の賞味期限の短縮
2. ソフトウェア製品の中心部のライフサイクルの長期化
3. 最新のソフトウェア設計法と管理法
  - ITアーキテクトを中心においた設計・実装とプロジェクト管理
  - 日米におけるケーススタディ
4. アーキテクトの育成法
5. まとめ



# アーキテクトの例

- ボーイング・ボールドストロークプロジェクト
  - Douglas Schmidt (ワシントン大学)、David Sharp (ボーイング)
- 米マイクロソフト
  - 各プロジェクトにはアーキテクトとプロジェクト・マネージャが“parallel”(対等)の関係で配置される
  - David Cutler (Windows NT チーフ・アーキテクト)
  - 中島聡 (Windows 95/98 ユーザ・インターフェース チーフ・アーキテクト)
  - Bill Gates (チーフ・ソフトウェア・アーキテクト)
    - 中島聡氏のブログ[2]より  
「ミーティングのたびに、Microsoft内で作られている主要なプロダクツのアーキテクチャを全て把握し、信じられないような記憶力と洞察力で、何万人ものエンジニア集団を引っ張っていくBill Gatesのすごさをつくづく感じたものである。...これこそがチーフ・ソフトウェア・アーキテクトとしてのBill Gatesの役割であり、それがMicrosoftの成功の原動力でもあった。」



## アーキテクトの例 ( 続き )

- Linus Torvalds ( リナックス開発者 )
  - リナックスの中心部を一人で開発、リナックスがオープンソース化した後も、世界中の貢献者が送るコードを吟味してリナックスに採用するかどうかを決定している
- Ken Thompson、 Denis Riche ( UNIXの開発者 )
  - 1974年にベル研でUnixを開発、1980年代にコンピュータのノーベル賞といわれるTuring賞受賞
  - 1990年初めに、当時ベル研で開発中の最新ソフトの中心部のプログラムを書いていた
- G. Holtzman ( 現在最もユーザの多いモデルチェッカーSpinの開発者 )
  - 初期のSpinを一人で実装
- Strustrap ( C++の開発者 ( Strustrap ) )
  - C++の言語仕様を決定するとともに初期のコンパイラを実装
- プロジェクト・マネージャのためのコンサルタントの話 ( K-StateでPhD )
  - 難易度の高いプロジェクトには、ほとんどの場合アーキテクトがつく



## RUPに似た開発を行っている例（続き）

- 坂村健氏（Tronの開発者）
- C社（日本）
  - プロジェクト・マネージャ: 34歳、社長賞 2 回
  - スーパー・プログラマ集団、それに認定されるとその会社の理事と同等の報酬
  - 各プロジェクトごとに、二人をスーパー・プログラマ集団から連れてきて見積もりをさせる
    - 彼らの許可が出るまでは見積もりは外部に出せない
  - プロジェクトが遅れた場合、元のスーパープログラマ達が呼び戻される。彼らにはプロジェクトを成功させる責任がある
- D社（日本）
  - チーム・リーダーの話：62人のグループのリーダー
  - アーキテクトが重要機能をその実装に最適な言語で実装し、それを設計図の形にして出す
  - 彼の設計は必ず動くという保証がある
    - 他のプログラマがそれを実装する



## RUPに似た開発を行っている例（続き）

- E社（日本）
  - 数年前、システムの変更のため既存のソフトの多くを書き直す必要が生じた
  - 関連会社の優秀なプログラマが変更する各ソフトのコアの部分を実装し（これらをプロトタイプと呼んだ）、他のプログラマがプロトタイプに肉付けをしていった
  - この方法により非常にスムーズに変更を完了した
- F社（日本）
  - 小さい支店で、人数があまりいないのでいろいろな役をこなさなくてはならない
  - 現在は、顧客に行き仕様を作成するところから始まり、その後のプロジェクトの管理をするのがメインの仕事
    - プログラム歴は長い
  - リスクがあるうちは見積もりを立てられないので、最も重要な部分を実装する（アーキテクチャ）
  - それを基に見積もりを出し、プログラマに引き渡し、後はプロジェクトの進行を管理する



# アウトライン

1. 技術者の持つ技術の賞味期限の短縮
2. ソフトウェア製品の中心部のライフサイクルの長期化
3. 最新のソフトウェア設計法と管理法
  - ITアーキテクトを中心においた設計・実装とプロジェクト管理
  - 日米におけるケーススタディ
4. アーキテクトの育成法
5. まとめ



# 日本の現状と課題

日本の多くの会社は

- プログラマを低賃金労働者としてみている
- プログラマを一過性のポジションとして考えている

そのため

- プログラマに教育投資をしない
- プログラマが高い地位（アーキテクトなど）に上がるためのキャリアパスが無い
- 優秀なプログラマに見合う待遇を与えていない
- 優秀なプログラマをマネージャにし、技術力を磨き続けることを困難にしてしまう



## アーキテクトを育てるには（続き）

- 企業がプログラマはチープ・レーバーではなく、知的職業であるという認識を持つ
  - 知的職業：加算が利かない、リプレースが利かない
    - 一人のスーパープログラマを複数の平凡なプログラマで置き換えることはできない
  - ITの時代では、プロスポーツのように非常に高い技術を持つスター技術者の果たす役割がより大きくなる
    - 米マイクロソフトのチーフ・アーキテクト制
    - Googleの強さは先進的なアイデアを出せると同時に、そのアイデアをすぐに具現化する技術力にある
  - プログラマに残業をさせるより、勉強する時間や機会を与えたほうが、短期間に生産性は圧倒的に向上する
    - 渡部昇一(20数年前の文芸春秋の記事からの要約)
      - 大学教授は有り余る時間を与えられている。そのうち本当に研究をするのは3、4割くらいかもしれない。後の人たちは遊んでしまうかもしれないが、それでもその方が全員を時間で拘束して働かせるよりもはるかに生産性があがるのである
    - ITの時代では、学び続ける姿勢、学び続けられる環境（時間も含めて）・機会の提供が最重要（生涯教育の充実）



# アーキテクトを育てるには（続き）

- 企業がマネジメントのシステムを変える
  - － 櫻井よしこ 「日本の危機2」より[3]
    - － オリックス宮内社長：21世紀の労働事情を考える時、私が考えるのはマネジメントとマネージされる人というだけで、労使という考えは全くありません。マネジメントは職能で、もう一方は知識労働者です。マネージャが偉くて部下の知識労働者が下であることは全くないわけで、双方共に別々の職能を果たしている。企業はこの二者によって機能します
  - － マネージャとアーキテクトの別々のキャリアパスを用意する
    - － 数年ごとに新たな「アプリケーション固有の知識」が生まれてくるソフトウェアの分野では、ソフト技術出身者でも一旦マネージャになったら、技術の第一線に居続けることは難しい
    - － 優秀なマネージャと優秀なアーキテクトの両方を、別の人材で育てていく必要がある



# アウトライン

1. 技術者の持つ技術の賞味期限の短縮
2. ソフトウェア製品の中心部のライフサイクルの長期化
3. 最新のソフトウェア設計法と管理法
  - ITアーキテクトを中心においた設計・実装とプロジェクト管理
  - 日米におけるケーススタディ
4. アーキテクトの育成法
5. まとめ



## まとめ

- システムの最重要部の設計・実装（アーキテクチャ）を行うのがアーキテクトであり、アーキテクトは複雑なソフトウェア開発には欠かせない
  - 良いアーキテクトは、全体のソフトウェア構成を効率よくきれいなものにする（アーキテクチャ）
  - アーキテクトは機能追加ごとに、必要ならばそれに適するようにアーキテクチャの改善を行い、ソフトウェア製品のブラックボックス化を防ぐ
- 優秀なアーキテクトを育てるには
  - 企業はプログラマを知的職業として認識し、技能に見合った待遇を与える
  - OJTに頼るのではなく、生涯教育（学び続ける姿勢を持ち続ける、学び続ける時間および機会を提供する）が重要である
  - 優秀なプログラマで、希望するものにはアーキテクトの道を用意する



## 参考文献

1. I. Jacobson, G. Booch, and J. Rumbaugh, "The Unified Software Development Process," Addison Wesley, 1999
2. <http://satoshi.blogs.com>(ブログ)
3. 桜井よしこ、「日本の危機 2 」新潮社、2002
4. 水野匡章、コース123Pテキスト「実践的オブジェクト指向分析・設計と実装 - 実装演習を中心としたシステム開発」Learning Tree International、2000

