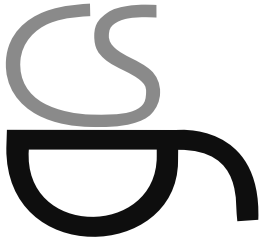
CMSI計算科学技術特論C (2015)

千葉　滋
東京大学情報理工学系研究科
創造情報学専攻



ソフトウェア工学の視点から 後編

Shigeru Chiba, U. Tokyo

最近の潮流

- 開発スピードが重要
- 独自部分に注力、それ以外は極力汎用品 (commercial off-the-shelf) を再利用
  - ソフトウェアのモジュール化が大切
    - 全て内製コードでもモジュール化は有用
  - 車輪の再発明は避ける
- 内製コードは保守性を意識
  - 再利用性、拡張性、bug-fix し易さ
  - Code clone は基本、悪

Shigeru Chiba, U. Tokyo

Lessons

- 再利用性と使いやすさ、で競争力をえる
  - そのソフトウェアを使う人は果たして多いか？
  - 利用機会の多い機能 (function) の提供
  - 再利用性の高いコード
  - 最初のとっつきやすさ
    - Use case を明確に
    - ドキュメント整備が大切
  - 保守の継続性
    - 長寿命なコード
      - 依存技術、配布形態 (CVS, SVN, git, ..)、ライセンス
  - 効果的な宣伝

Feedback

- 化学系では「単体動作アプリ」が売れる
  - 同じような計算をするアプリが溢れかえって互いにお客を奪い合う
  - Webアプリ
    - サイトごとに特注のアプリケーションを開発
    - Wordpress 等は単体アプリといえますが…

Shigeru Chiba, U. Tokyo

保守の継続は難しい

- 解？ Open Source Development
  - コミュニティ開発
    - 世界中から無償で人が集まって開発に貢献
    - 開発段階なら？
      - 新機能の追加は楽しい
      - 自身の能力の宣伝になる
      - ユーザ目線からも信頼感が高まる
      - ソフトウェアは永遠に進化し続けないといけない

Shigeru Chiba, U. Tokyo

JBoss Inc. のビジネスモデル

- 基幹ソフトウェア
  - オープンソース・ソフトウェアで無償公開
- 基幹ビジネス
  - （法人向け）チュートリアル
  - 保守サービス
    - ２次、３次受け保守サービス会社を配下にもつ
- 開発者（従業員）
  - 世界中に拡散

Apple のビジネスモデル

- 基幹ソフトウェア　MacOS X
  - ほとんど無料
- 基幹ビジネス
  - Hardware 販売
  - iTunes, AppStore, …

  - 昔の Microsoft の基幹ビジネス
    - 基幹ソフトウェアの販売

Low-cost な保守のために

- Modular design
  - 活発に機能拡張し続けるためにも
  - Separation of Concerns
    - 関心事 (Concern) ごとにコードをまとめる
      - 分担開発が可能に（人月の神話）
      - Bug fix も無駄がない
    - どこが将来変わりうるかを、まず分析
      - 差分を探す

Information Hiding

- あるいは Encapsulation

- Module interface
  - モジュールは内部を隠す
  - 外部とは限られた interface を通じて
    - 実装の自由
    - 利用者にとっての理解しやすさ

Shigeru Chiba, U. Tokyo

Modularity is not free

- 実行速度は遅くなりがち
- Crosscutting concerns の存在
  - いくつかのモジュールに跨がらないと実装できない concern
    - 例えば高速化のための機能
    - 解決策　コンパイル時に織り込む？
      - modular code を tangling code に
      - C++ templates や DSL の利用

Shigeru Chiba, U. Tokyo

Separation of Concerns の技法

- オブジェクト指向
  - 副作用（代入）を用いたプログラミングでは
    - データ（状態）と関数（手続き）は
      セットで提供したい
    - 例

```
MPI_Request req;
MPI_Status status;
MPI_Irecv(buf, len, MPI_DOUBLE, …, &req);
MPI_Wait(&req, &status);
```

Separation of Concerns の技法

- オブジェクト指向
  - データが中心
    - 計算を物理現象として捉える
    - 言語サポートがあるとよい
  - データと関数をセットで提供
    - 組み合わせを間違えない
    - 目的の関数を利用者が探しやすい

```
Req r = new Req();
r.irecv(…);
r.wait();
```

    - 例

```
class Req {
    MPI_Request req;
    MPI_Status status;
    void irecv(…) { … }
    void wait() { MPI_Wait(&req, &status); }
        :
};
```

Shigeru Chiba, U. Tokyo

Fortran

- データを整数で指定
  - c　内部に受信状態を表すデータがあり、
    c　その番号が req

```
Integer req
call MPI_Irecv(array, N, …, req, err)
call MPI_Wait(req, ...)
```

Fortran

- データ中心に名前を変えてみる
  - Ireceiver で関数を探せる？
    - Wait が Ireceiver_get, Isend_get, … と違う名前に
  - Integer irecv
    call MPI_Ireceiver_make(irecv, array, N, …)
    call MPI_Ireceiver_get(irecv, ...)

Shigeru Chiba, U. Tokyo

継承

- 関数の分類をクラスで表現
  - 共通部分はスーパークラス(親クラス)
    - 自動的に別名がつく？

```
class MPI_Nonblocking {
   get(Integer i, …)
}
```

```
class MPI_Ireceiver {
   make(Integer i, Real buf(:), …)
}
```

```
class MPI_Isend {
   make(Integer i, Real buf(:), …)
}
```

Shigeru Chiba, U. Tokyo

差分を探す

- Code clone を避ける
  - Separation of concerns につながる
  - 複数のプログラムの共通部分をモジュールに
  - コード中の値が異なる
    - パラメタ化して関数に
  - コードが部分的に異なる
    - 差分を関数引数として受け取るように

Shigeru Chiba, U. Tokyo

Functor (in ML)

- 関数型言語 ML, Ocaml, Haskell, …
  - いくつかの関数の集まり(組)をまとめて引数として渡せる
    - 差分に対応する関数を事前に何組か用意しておく
    - 差分に対応する関数の組を新たに書く

```
module Stencil(kernels: KernelSig) = struct
    let update grid = function
        ... kernels.halo i j k ...
        ... kernels.func subarray ...
end
```

Shigeru Chiba, U. Tokyo

Method overriding

- オブジェクト指向言語（メソッドの上書き）
  - スーパークラスのメソッドを置換

```
class Stencil {
    void update(Grid g, …) {
        … halo(i, j, k) …
        … func(i, j, k) …
    }
    float halo(int x, int y, int z) { … }
    float func(SubArray a) { … }
}
```

```
class Diffusion extends Stencil {
    float halo(int x, int y, int z) {
        return 0.0F;
    }
    float func(SubArray a) {
        return a.east() + a.west() …
    }
}
```

Shigeru Chiba, U. Tokyo

その他

- 色々なプログラミング技法
  - オブジェクトを module と見なして functor 風にする技法もある
- C++ templates
  - functor 風にも使える
  - ただし型検査はない
  - が、実行が高速になると期待できる

コードの再利用性を高める技術

- Application software
  - 基本は as is で利用。設定ファイルでカスタマイズ。
  - ソースコードを読んで自分で直接書き直してカスタマイズ。
- Library
  - 関数、手続き単位で部品として再利用
- Framework
  - 半完成品。
  - 部分的にコードを差し替えて完成品に。
- Domain Specific Language (Code generator)
  - 特定のアプリケーション分野専用の「言語」
  - （部分的に）コードを自動合成

Shigeru Chiba, U. Tokyo

Framework での再利用技術

- 半完成品
  - 部分的なコード差し替え、追加で完成
    - Main 関数を含むライブラリ？
    - カスタマイズするので再利用の範囲が広い
  - 例
    - 積分サブルーチンに被積分関数を引数で渡す
  - 言語サポートがあるとよい
    - functor, method overriding など
    - カスタマイズ性を上げるには、何でも関数引数に
    - かえって使いにくくなる
      - ほとんどの関数引数は標準品 (default) でよい

Shigeru Chiba, U. Tokyo

DSL での再利用技術

- DSL: Domain Specific Language
  - コード生成機能つきライブラリ？
  - External DSL
    - 完全に独立した新プログラミング言語
      - プログラムはその言語だけで完結して書ける
  - Embedded DSL
    - 既存言語のプログラム中に埋め込んで書く
    - 専用コンパイラで処理、実行コードを生成
      or 構文だけDSL風だが実体は既存言語のライブラリ

Shigeru Chiba, U. Tokyo

コード生成機能つきライブラリ

- 利用者は boilerplate code を書かずにすむ
  - ライブラリ設計の基本
    - 複数のプログラムの共通部分がライブラリ
    - コード中の値が異なる
      - パラメタ化して関数に
    - コードが部分的に異なる
      - 差分を関数引数として受け取るように
    - コードは異なるが類似性が高い
      - 機械的に自動生成する
      - さもなくば手で書く
        慣れれば単純作業だが、慣れるまでが大変
        単純作業は間違いをおかしやすい

Shigeru Chiba, U. Tokyo

コード生成のための道具

- コンパイラ作成 library/framework
  - ROSE compiler infrastructure
    http://rosecompiler.org
- 言語作成 workbench
  - The Spoofax language workbench
    http://metaborg.org/spoofax/
- プログラム変換 library
  - Javassist
    http://www.javassist.org

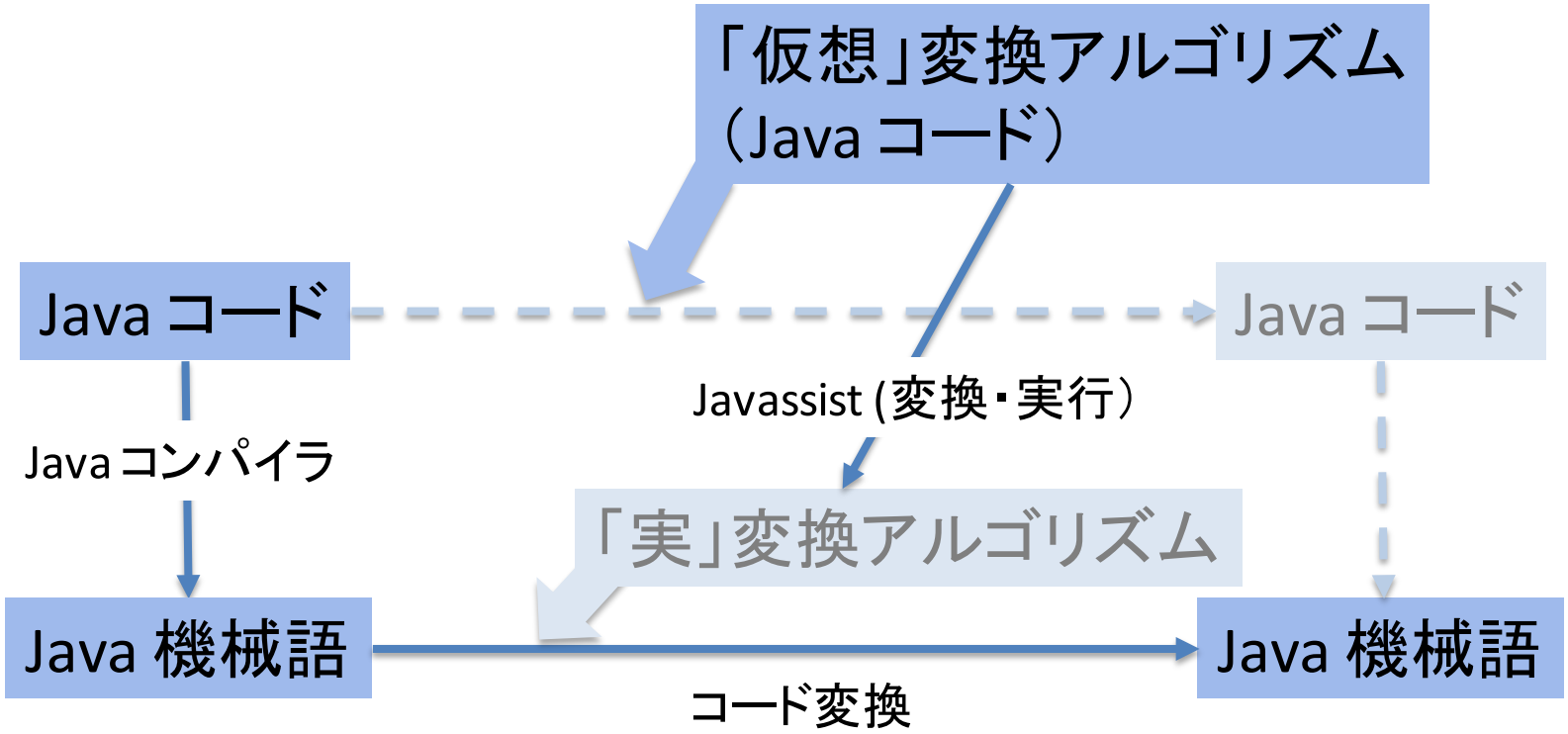
Javassist

- Java 機械語コード変換・生成ライブラリ
- 利用例
  - プログラマは仮想的なインタフェース(API)を提供するライブラリを使って書く
    - 仮想的なAPIは利用が簡単
      - 例: GPU kernel を呼び出しの前後に CPU/GPU 間メモリ転送をしない
  - ライブラリはコンパイル時などにコード変換
    - 本物のAPI向けにプログラムを自動的に書き換える
      - 例: 転送を最適化する位置に CUDAmemcpy() を挿入

Shigeru Chiba, U. Tokyo

Javassist の特徴

- 少ない前提知識で簡単に利用可能
  - Java の知識があれば Java 機械語の知識がなくてもコード変換を記述できる



まとめ

- 広く利用される library/framework
  - 高い再利用性
    - 少々の差分であればカスタマイズして利用可能に
  - 利用の仕方のわかりやすさ
    - ドキュメント整備
    - サブルーチン・関数の探しやすさ
    - 学習の容易さ
      - なるべく少ない前提知識で利用可能に

Shigeru Chiba, U. Tokyo