携帯 JAVA プログラム設計

- サイズへの挑戦 -

第１版
２００２年2月8日

H13 年度　OISA　JAVA 部会

目次

１．はじめに

２００１年1月２６日、Java アプリケーション「ｉアプリ」に対応したｉモード携帯電話が発売された。ｉアプリとはｉモード携帯電話にプログラムをダウンロードして利用できるアプリケーションのことである。

ｉアプリはｉモード対応 Java で記述し、その仕様は DoJa と呼ばれている。DoJa は「NTT DoCoMo Profile」の通称である。J2ME（Java2 Micro Edition） MIDP Profile という携帯端末向けの標準プラットフォーム仕様に先行して、NTT DoCoMo により決定されたもので、MIDP Profile とは非互換である。ここで、J2ME とは Sun Microsystems,Inc. が決めた Java プラットホーム仕様である。

ｉアプリの特徴として、ユーザにダウンロード時間が長いと感じさせない配慮があり、ファイルサイズに制限がある。この制限は、プログラム本体にあたるクラスファイルと、プログラムで使用するリソースファイルを、ZIP 形式で圧縮された JAR ファイルというに変換し、その**ファイルサイズが 10Kbytes**（＝10240bytes）以内という制限である。

この制限は、「ｉアプリ」に限ったものではなく、携帯電話で動作する MIDlet プログラムには各キャリアでファイルサイズに違いがあるものの、それぞれ制限が決められている。例えば、J-Phone Java アプリは 50K（2002/1 以降の新機種より 100K）、NTT DoCoMo の FORMA は 30K である。

以降では、この制限をクリアすべく、JAR ファイルを出来るだけ小さくする方法を列挙し、それぞれ検証を行い、各手法について評価を行う。

２．JAR ファイルのダウンサイジング手法

2.1 検証する開発手法

携帯電話の Java アプリでは、JAR ファイルサイズを各キャリアの指定サイズ以内に収める必要がある。既存アプリを携帯電話に移植する場合、また、キャリア間で移植を行う場合、JAR ファイルのダウンサイジングが必要となる場合がある。JAR ファイルのダウンサイジングの手法としては、以下のような手法が考えられる。

1. **クラスファイルを少なくする**
2. **インナークラスの利用**
3. **イメージファイルの持ち方**
4. **ローカル変数の利用**
5. **配列の利用**
6. **例外処理の工夫**
7. **クラス名を短くする**
8. **メソッドのインライン化**
9. **クラスインスタンスの再利用**

第 4 章において、それぞれの手法について検証する。

2.2 目標の設定

ｉモード端末で動作する Java アプリを最終ターゲットとし、検証に利用するアプリケーション（第３章を参照のこと）の JAR ファイルサイズを１０K に収めることを最終目標とする。ただし、実行環境の多様化に合わせて下記の目標レベルを設定し、段階的にクリアしていくこととする。

表 1 ダウンサイズの到達レベル

レベル	JAR ファイルサイズ	ターゲット環境
1	Size≦100K	J-Phone（新)
2	Size≦50K	J-Phone（旧)
3	Size≦30K	FORMA
4	Size≦10K	ｉモード端末

また、ダウンサイズにあたっては既存機能の維持を条件とし、各手法の検証により得られた結果を元に評価を行う。

3．検証方法及び検証環境

各手法を検証するため、検証対象プログラムおよび開発環境を以下のように定める。

3.1 サンプルアプリケーション

昨年度の OISA　JAVA 部会で開発されたプログラム OrderTicket　Java プログラムを題材として、2 章で示したダンサイジング手法を適用し、どれだけダウンサイジングが行えるかを検証する。
以下に OrderTicket　Java プログラムの機能構成およびクラス一覧を示す。OrderTicket プログラムの機能は、ログイン、予約と予約確認、空席問い合わせ、で構成されている。

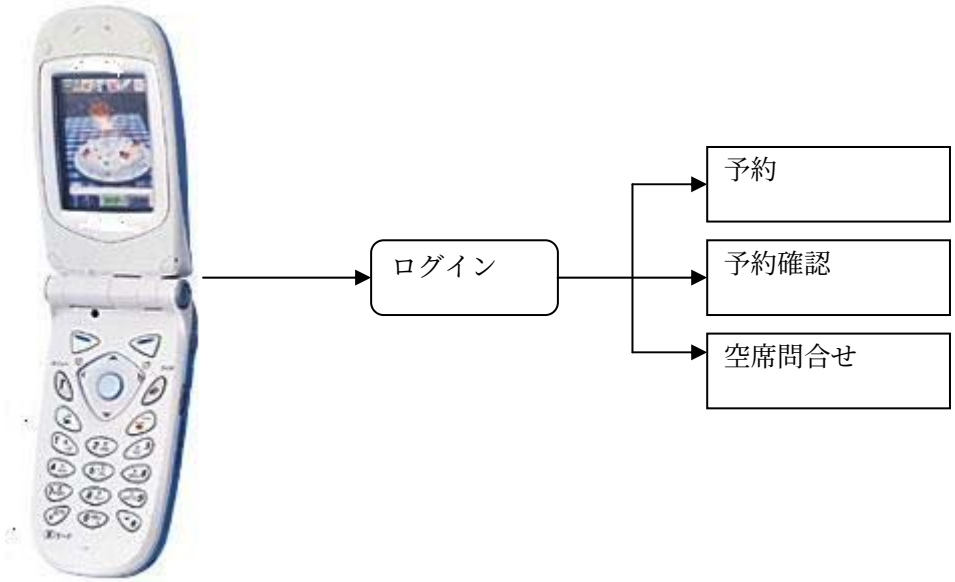

図 1 機能構成の概要

このプログラムの特徴として、OrderTicket を除くすべてのクラスが Inner Class として構成されている。また、ダウンサイズ前の JAR ファイルサイズは **114K** である。

表 2　OrderTicket クラス一覧

構成クラス名	役割
OrderTicket.class	OrderTicket MIDlet の本体全体制御を行っている。
OrderTicket$1.class	無名の Inner Class
OrderTicket$2.class	無名の Inner Class
OrderTicket$About.class	About 情報
OrderTicket$Artist.class	チケット情報(TicketItem)一覧テーブルを持つクラス、多分 DB のかわり、デモ用の実装。
OrderTicket$Confirm.class	問合せ結果確認（表示）：購入内容証明画面表示
OrderTicket$CreditCard.class	クレジットカード情報
OrderTicket$GaugeDisplay.class	ゲージ表示クラス

OrderTicket$Inquire.class	購入チケット確認（問合せ）：購入済みチケットを購入時に設定したパスワードで確認する
OrderTicket$Login1.class	ログインクラス（予約可否確認中)、予約するか否かを確認させた後、予約処理を行う。
OrderTicket$Login2.class	ログインクラス（購入中)、クレジットカード ID とパスワードを入力した後、チケット購入処理を行う。
OrderTicket$NumberedHashtable.class	ハッシュテーブル
OrderTicket$PersonalOisaData.class	スクラッチパッドクラス（予約情報を格納するために RMS：Record Management System を使っている）
OrderTicket$TicketItem.class	チケット情報（チケット情報を格納)、１つの公演に対応する。
OrderTicket$TimerClient.class	タイマー（クライアント）

3.2 コンパイル・実行環境

コンパイル環境として、JDK1.3.1_01、J2ME-Wireless_ToolKit1.0.3(以降、WTK と略す)をインストールし利用する。WTK に添付されている KToolBar というツールを用いてコンパイル、JAD ファイル作成、JAR ファイル作成を行い、JAR ファイル／クラスサイズの比較・検証を行う。

（註１）ｉモードアプリを開発するには、Sun が公開している WTK ではなく、NTT DoCoMo が公開している、「J2ME-Wireless_ToolKit for the DoJa」が必要である。NTT DoCoMo のホームページからダウンロード可能。

（註２）コンパイルを行う際、Class ファイルの削除を必ず行い、必ず最新のクラスファイルで実行できるようにしている。

コンパイルしたアプリケーションは KToolBar あるいは同添付の”Run MIDP Application”を利用して実行する。すなわち、MIDP Profile をベースとするエミュレータで実行する。開発したアプリケーションを実際の携帯電話に読み込んで実行することは環境の問題で実施しない。

４．検証方法と結果

4.1 クラスファイルを少なくした場合の検証

[検証方法]

１５ある class ファイルの中で Artist.class は、アーティストデータをクラス内部に抱えている。このアーティストのデータが 1 件増えるごとに Artist.class のサイズが５０バイト増え、ＪＡＲファイルも１５バイトづつ増加する。よって Artist.class をＪＡＲファイルに含まないようにし、ＪＡＲファイルの外から Artist.class を呼ぶようにプログラムを修正する。この修正により、ＪＡＲファイルと同じディレクトリに Artist.class を置き、そこからアーティストデータを読込むことができる。

[検証結果]

Artist.class を含めずＪＡＲファイルを作成すると、ＪＡＲファイルのサイズが１１０，０３３バイトとなり、元のＪＡＲファイルと比べ**約４Ｋバイト縮小**した。

表 3　データ１件あたりの JAR ファイルサイズへの影響

データ件数（追加）	Artist.class のサイズ	JAR ファイルのサイズ
１	６,７０９	１１４,９８７
２	６,７５９	１15,００２
３	６,８０９	１１５,０１７
４	６,８５９	１１５,０３２
５	６,９０９	１１５,０４７
１０	７,２０９	１１５,１３７

＊JAR ファイルは Artist.class を含めた値である。

[考察]

上記の検証により、JAR ファイルにプログラムで利用するデータを含めない方が良いということが分かる。今回の検証では、ローカルファイルからデータを読み込むようにしているが、実際はサーバからの読み込みになるので通信コスト（パケット料）が発生する。この為、設計段階で通信コストとデータ量を考慮する必要があると考えられる。

4.2 インナークラス利用に関する検証

[検証方法]

JAR ファイルサイズを縮小するために、Ｊａｖａプログラムのクラスサイズを小さくしなければならない。
具体的には、あるクラスをインナークラスとした場合と、通常のクラスとした場合で JAR ファイルサイズを比較し、その結果を昨年度デモプログラムに反映する。

[検証結果]

インナークラスの利用とプログラム JAR ファイルサイズの関係を調べた結果以下のようになった。

<名前付インナークラスの対応>

下記のようなクラスをインナークラスとした場合と、外クラスとした場合の JAR ファイルサイズの違いを比較する。

```
class Xn {
    Xn() {
        return;
    }
}
```

図 2　サンプルクラス

各 Xn.class の各サイズ　　191bytes　（n=1～5）
各 OrderTicket#Xn.class の各サイズ　　341bytes　（n=1～5）

表 4 インナークラスと JAR ファイルサイズの関係

Xn の クラス数(n)	インナークラスの時の JAR File サイズ(bytes)	外クラスの時の JAR File サイズ(bytes)
1	23609	23463
2	24002	23739
3	24404	24009
4	24797	24283
5	25187	24556

<無名インナークラスの対応>

OrderTicket.class には２つの無名クラスがあるが、その用途は抽象クラスのメソッドの実装である（例；Thread クラスの run メソッドなど）。無名インナークラスの１つを独立クラスとした場合、以下のような結果となった。

```
private void    displayConnectAction(Command        c, Formobj) {
        Thread thread = new Thread(){   //ここが無名クラス
                public  void    run() {
                        try {
                                detail.setString("Wait...");
                                URL = new       String();
                                URL = BASE_URL + getURL();
                                detail.setString(readPage(URL));
                        } catch (IOException e) {
                                detail.setString("Fail..");
                        }
                }
        };
        thread.start();
}
```

図 3修正前：無名クラス利用

```
private void     displayConnectAction(Command        c, Formobj) {
        actThread       thread = new actThread();
        thread.start();
}

  // 新しい名前付インナークラス actThread を定義する。
       class actThread extends Thread    {
        public  void     run() {
                try {
                        detail.setString("Wait...");
                        URL = new       String();
                        URL = BASE_URL + getURL();
                        detail.setString(readPage(URL));
                } catch (IOException e) {
                        detail.setString("Fail..");
                }
        }
       }
```

図 4修正後無名クラス（actThread と名づける）

JAR ファイルサイズとの関係は以下のようになっている。

表 5　無名クラスと名前つきクラス

	無名クラスのとき	有名クラスのとき
OrderTicket.class	13767	13787
#1.class	1070	―
#actThread.class	―	**1087**
JAR サイズ	23197	23231

+17

単位：byte

上記の結果から、OrderTicket プログラムにおいて簡単に修正可能な５つのクラスについてインナークラスの利用をやめ、それらのクラスを外クラスとした。あわせて、一部の変数スコープを static に変更した。なお、無名クラスに名前を付けることは、JAR サイズ増加が見込まれるため実施しなかった。

（インナークラスをやめたクラス）

PersonalOisaData
NumberedHashtable
CreditCard
TicketItem
Artist

以上の結果、**24K あった JAR ファイルが 1K 削減され 23K**になった。

[考察]

ダウンサイズの際、インナークラス利用では、クラス名の有無が JAR ファイルサイズに影響することに注意が必要である。実作業では、インナークラスのメリットと差し引きして、実装の設計を行う。

注意点としては２点ある。

① インナークラスはなるべく使わない（インナークラスにすると、クラスサイズ以上の何かが付加され JAR ファイルサイズ増大に貢献している。また、インナークラスは、単体クラスサイズも大きくなる。）クラスの働きが同じであれば、外クラスの方がサイズは小さい。

② インナークラスにおける無名クラスと名前付クラスの使い分けは、名前付のクラスのほうが、若干サイズが大きくなる程度で、ほとんど変わらない。実装上、無名クラスで問題なければそれでもよい。

インナークラスのデメリットとしては、実装後のクラス分割を考えると、修正が大きくなる可能性が高いことがあげられる。

今回、プログラムを見直して感じたことであるが、以下のような項目を徹底することでもプログラムサイズ縮小に貢献できるのではないかとか考える。

③ JAR ファイルの中に不必要なファイルを持たない。

④ リファクタリングを行い、必要ないコーディング（未使用の変数、メソッド、クラス、リソース、データのハードコーディング）は削除する。

4.3 イメージファイルの持ち方についての検証

[検証方法]

イメージファイルの形式、ファイルの持ち方を検証し考察する。

[検証結果]

１） イメージファイルの形式は可能であれば jpeg がよい。実際 png よりコンパクトである。

表 6　ファイル形式によるイメージファイルサイズの違い

ファイル	サイズ
Fukuokadome.png	13k
Fukuokadome.bmp	21K
Fukuokadome.gif	3K
Fukuokadome.jpg	3K

２） ソースファイル中に静的データとしてハードコーディングされているイメージデータをイメージファイルの利用に改めた。この修正により、**34K あった JAR ファイルサイズが、24K** になった。

３）イメージファイルのＪＡＲファイル内削除

昨年度版の OrderTicket クログラムのＪＡＲファイルのサイズは１１４,９８７バイトである。実はこのＪＡＲファイルの中にはイメージファイルが２２も含まれているので、このイメージファイルをＪＡＲの外から読み出すように変更した。この変更では、ＪＡＲファイルと同じディレクトリにイメージファイルを置くフォルダを設け、そこから読み出すこととする。イメージファイルを含めずＪＡＲファイルを作成すると、**ＪＡＲファイルの**

サイズが３４,７２９バイトとなり、イメージファイルを含めていた従来のＪＡＲファイルと比べると**約７０％縮小した**。

[考察]

イメージファイルの持ち方では、JAR に含めるものと外付けにすべきものを切り分けなければならない。使用頻度の低いイメージデータを JAR 内部に抱え込むのは、JAR ファイルサイズに大きく影響する。JAR ファイルにイメージファイルを含める場合、上記のように jpeg 化すると効果的である。なお、今回のプログラム改造では、全てのイメージファイルを JAR 外に持つので、ファイル形式の対応は未実施である。

4.4 ローカル変数化による検証

[検証方法]

クラス内で宣言された変数は、メモリ上にその変数の領域がとられ、変数の内容はそこを用いて扱われるようになる。それに対し、メソッド内ローカル変数として宣言された変数は、スタック領域に変数の内容を格納するようになる。アセンブラレベルで処理を見た場合、メモリ領域へのアクセスの処理のほうが、スタック処理にくらべ多くなるので、Java のクラスファイルのサイズも、必然的に変数をメソッド内ローカル宣言したほうが、小さくなると考える。

[検証結果]

```
class A {
    private String aaa;

    void method() {
        // aaa を使用した処理
    }
}
```

⇨

```
class A {
    void method() {
        String aaa = null;
        // aaa を使用した処理
    }
}
```

図 5 テストプログラム例

昨年度のサンプルプログラムにはクラス内変数の宣言が４２ありましたが、このうちの３つの変数をローカルメソッド内に移動させ、結果、クラス内変数の宣言を３９にし、ＪＡＲファイルのサイズの違いを検証した。以下にファイルサイズを比較した結果を示す。**３フィールドローカル化により、４５バイトの減少**が見られたことがわかります。

表 7　実施結果

	昨年度のプログラム		反映後のプログラム
フィールド数	４２		３９
ＪＡＲファイルサイズ	５３８０６ byte	⇨	５３７６１ byte

[考察]

コーディングを行うにあたっては、変数のスコープは常に意識しなければならない。今回、クラスファイルサイズの削減方法として、変数のローカル変数化を挙げましたが、本来きちんとしたリファクタリングを行っていれば、変数に無駄なスコープが発生することはあり得ない。つまり、クラスファイルのサイズ削減として、本項目を意識するのではなく、きちんとしたリファクタリングを行うことこそが重要である。

4.5 変数を配列に置き換えた場合の検証

[検証方法]

本節では､同じ型のスカラー変数を複数宣言する場合と配列で宣言した場合のサイズ比較を行い、変数を配列で置き換えた場合にサイズダウンに繋がるかどうか検証し、その結果を考察する。

[検証結果]

サンプルプログラム中の下記スカラー変数を配列に変換を行い、JAR ファイルのサイズ比較を行った。

●Alert 変数

変更前変数名	変更後配列名		変更前ｻｲｽﾞ	変更後ｻｲｽﾞ
altCrdMsg1	altMsgxxxxx[7]	[0]	42,664	42,570 (－94)
altCrdMsg2		[1]		
altPwdMsg1		[2]		
altPwdMsg2		[3]		
altTktNtg		[4]		
altTktMiss1		[5]		
altTktMiss2		[6]		

●int 型変数

変更前変数名	変更後配列名		変更前ｻｲｽﾞ	変更後ｻｲｽﾞ
imgPindex	imgXindex[2]	[0]	42,555	42,560(+5)
imgHindex		[1]		

※単位は ﾊﾞｲﾄ

図 6 実施結果１

int 型の変数を配列に変換した場合、逆にサイズが大きくなる傾向がありそうなので、int 型のダミー変数をサンプルソースに追加し調査を行う。

調査方法として、変数を１つずつ追加し、サイズを測定する。その後，配列を作成し，配列要素を１つずつ追加し、サイズを測定する。追加した変数の数と配列要素の数が一致するところの JAR ファイルのサイズ比較を行う（下記サンプルソース参照)。結果を下表にまとめて示す。

```
/*   ｽｶﾗｰ変数を用いた場合  */

class A {
   int A1,A2,A3,A4,A5,A6,A7
      A() {
      }
}
```

⇨

```
/*   配列を用いた場合  */
class A {
   int A1[];

      A() {
         A1 = new int[7];
      }
}
```

変更前変数名	変更後配列名		変更前ｻｲｽﾞ	変更後ｻｲｽﾞ
追加前			42,550	42,550
A1	A1[n]	[0]	42,557	42,580
A2		[0] － [1]	42,562	42,580
A3		[0] － [2]	42,568	42,580
A4		[0] － [3]	42,573	42,580
A5		[0] － [4]	42,577	42,580
A6		[0] － [5]	42,582	42,580
A7		[0] － [6]	42,586	42,580

図 7 テストプログラム例とその実施結果

上記、Alert に対する検証結果を反映して、JAR ファイルサイズは**９４バイト縮小した。**

[考察]

スカラー変数を配列にまとめることにより、ダウンサイジングの効果がみられますが、その兆候は配列にまとめる変数の個数に依存すると考えられる。これは int 型の変数だけにかぎられた傾向ではなく，全ての変数でこのような傾向がある。まとめる変数の数が少ないとダウンサイズの効果が得られないが，より多くの変数を配列にまとめるとよりで、より大きな効果が得られることが予測できる。

4.6 例外処理をまとめた場合の検証

[検証方法]

例外処理をまとめた場合を検証し、その結果を考察する。Java では、**try{ }catch(Exception){ }**で例外をキャッチすることができるが、この try{ }catch(){ }を増やすとファイルサイズが大きくなる為、プログラム開発作業の終盤、エラーが出ないことが分かった場合には、同じメソッド内の try{ }catch(){ }を 1 つにまとめ、そのメソッドより上のメソッドへ例外を投げるようにすると try{ }catch(){ }が少なくなり、ファイルサイズを縮小することができる。

```
void method() {
    try {
        例外を発生する処理 A
    } catch (Exception e) {};
    try {
        例外を発生する処理 B
    } catch (Exception e) {};
}
```

⇨

```
void method01() {
    try {
        method();
    } catch (Exceprion e) {};
}
void method() throws Exception {
    例外を発生する処理 A
    例外を発生する処理 B
}
```

図 8 テストプログラム例

[検証結果]

上記のようなメソッド内に同一の例外が発生する処理を、上位のメソッドに Throw するように変更した場合には、ファイルサイズにどの程度の違いが発生するか検証する。サンプルプログラムを作成し、クラスファイルサイズを検証した結果を以下の表に示す。

表 8　Try-Catch 文とクラスファイルサイズの相関

例外処理の記述回数	同一メソッド内に記述した場合(byte)	上位メソッド内に記述した場合(byte)	差分（byte）
1	６９７	７１２	－１５
2	７６７	７２３	４４
3	８３７	７６９	６８
4	９０８	７８０	１２８
5	９７９	７９２	１８７
6	１０５０	８０４	２４６

[考察]

上記表より、同一例外を一つにまとめ上位のメソッドに投げた場合は、ダウンサイジングの効果があることがわかる。しかし、「例外処理の記述回数」が１回の場合の結果をみるとわかるように、単純に上のメソッドに投げた場合では、ダウンサイジングの効果が無く、メッソッド内で一つにまとめた方が効果は大きい。なお、この手法については、ソース中に適応できる箇所がなかったため利用していない。

4.7 クラス名を縮小した場合の検証

[検証方法]

クラス名を縮小した場合を検証し、その結果を考察する。

[検証結果]

昨年度のＪＡＶＡ部会の OrderTicket プログラム内のクラス名を１Byte ずつ短くした場合のクラスファイルと、JAR ファイルのそれぞれの変化の推移を下記の図に示す。

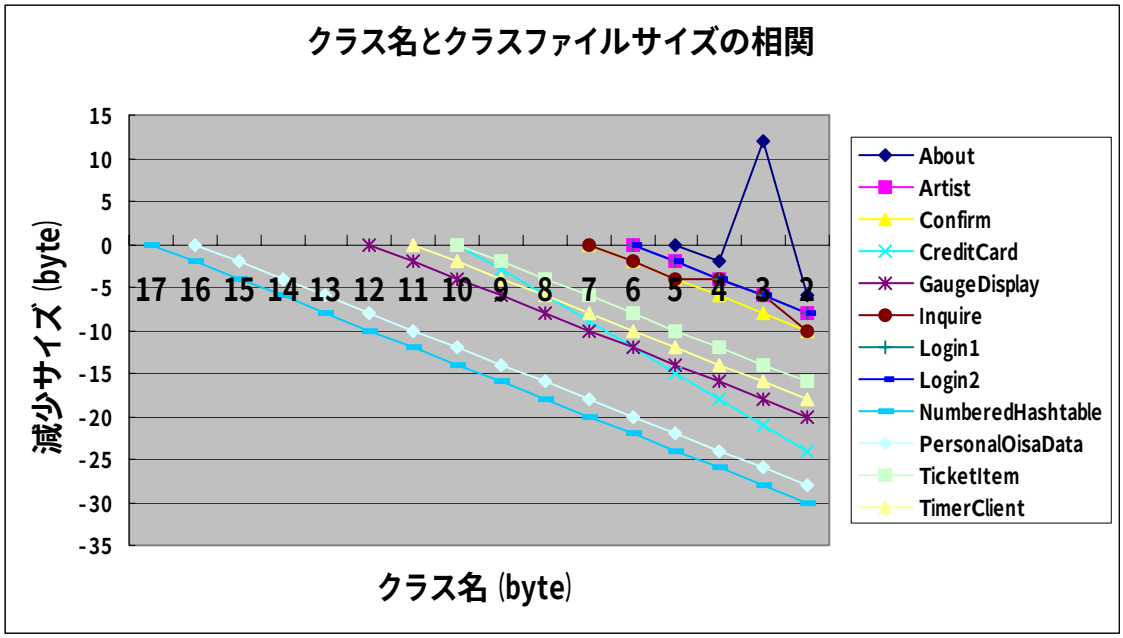

図 9 クラス名とクラスファイルサイズの相関

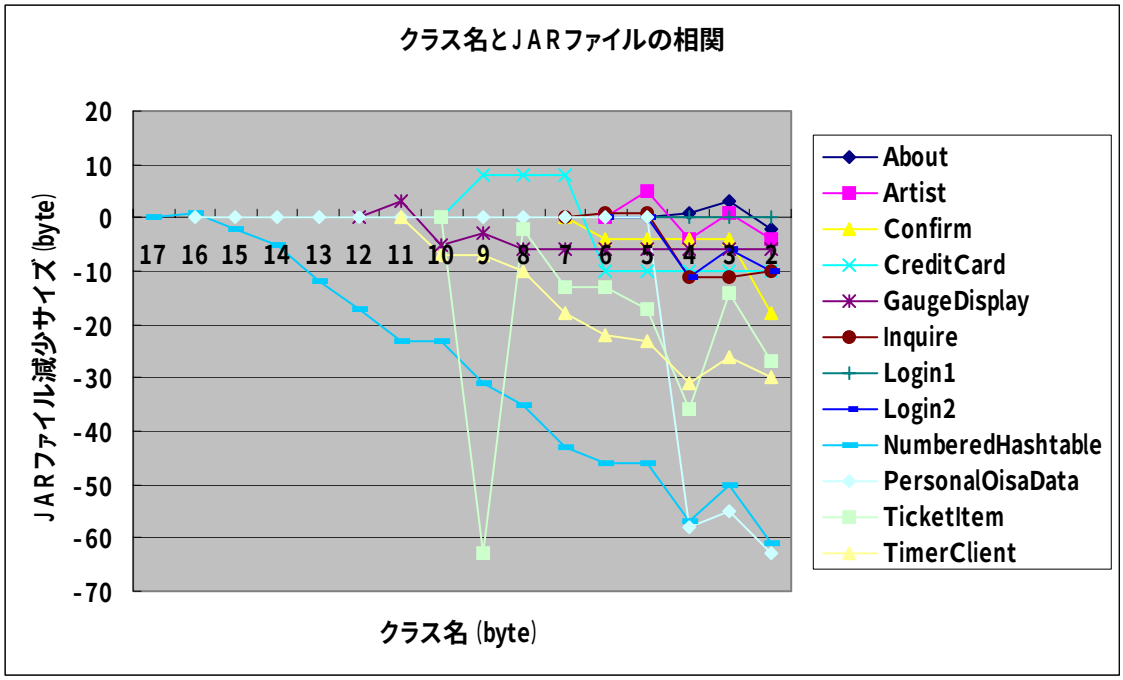

図 10 クラス名とＪＡＲファイルサイズの相関

この手法を適用（すべてのクラス名を X バイトに）した結果、OrderTicket を除くクラス名を全て２バイトに変更した場合 Jar ファイルのサイズは **241 バイト 減少**する。（また、OrderTicket → OT に変更すると、さらに**約 550 バイト減少し，Total で約 800 バイト減少**する。）

[考察]

表２、３より、クラス名を１Ｂｙｔｅ短くすると、クラスファイルは概ね２Ｂｙｔｅずつ減少していることがわかる。ＪＡＲファイルサイズに対しては、規則性は無いようである。クラス名が２Ｂｙｔｅの場合が最もＪＡＲファイルサイズが小さくなる。

4.8 メソッドのインライン化による検証

[検証方法]

処理をインライン化することで、クラス内のメソッド数を削減し、クラスファイルのサイズを小さくすることができる。インライン化とは呼び出すメソッドの処理を呼び出し元の中に埋め込んでしまう処理である。インライン化をすることにより、全体としてのメソッド数が減少することになる。

```
class A {
    void method01() {
        …
        method02();
        …
    }
    void method02() {
        // method02 の処理
    }
}
```

⇨

```
class A {
    void method01() {
        …
        // method02 の処理
        …
    }
}
```

図 11　テストプログラム例

[検証結果]

昨年度の OrderTicket クラスには、public、private 合わせて３７のメソッドがある。上記で説明したインライン化を行い、メソッド数を１７にまで減らし、ファイルサイズの検証をおこなった。以下にファイルサイズを比較した結果を示す。２０メソッドのインライン化により、**１０９３バイト分減少した**ことが分かる。

表 9 実施結果

	昨年度のプログラム	反映後のプログラム
メソッド数	３７	１７
ＪＡＲファイルサイズ	５３８０６ byte	５２７１３ byte

[考察]

メソッドとは、処理の実行単位である。その単位をどの粒度でみるかにもよるが、基本的には１メソッド１処理としたほうが、ソースは見やすくなる。メソッドをまとめるということは、１メソッドに複数の処理を埋め込むということになり、メソッド内のステップ数が増加する。可読性が落ちるので基本的には行わないことが望ましいのであるが、比較的楽に行えるので、開発が完了した時点での最終手段の１つとして、実施することができる。

4.9 インスタンスを再利用した場合の検証

[検証方法]

通常、メソッドの戻り値（即値）はメソッドの戻り値の型で受け取りますが、それをそのまま、別のメソッドの引数にすることにより、余分なインスタンスを生成することを防ぐことができる。余分なインスタンスの生成がなくなるので、結果的にクラスファイルのサイズは小さくなる。

```
void method01() {
    …
    String aaa = method02()
    int a = Integer.parseInt(aaa);
    …
}
String method02() {
    return new String(“100”);
}
```

⇨

```
void method01() {
    …
    int a = Integer.parseInt(method02());
    …
}
String method02() {
    return new String(“100”);
}
```

図 12 テストプログラム例

[検証結果]

昨年度のプログラムのコード中で１５個所、即値を利用した形に変更をしてみて、ファイルサイズの変化を検証した。以下にファイルサイズを比較した結果を示す。既値利用の結果、**２８５バイト分の減少**がみられたことがわかる。

表 10 実施結果

	昨年度のプログラム	反映後のプログラム
ＪＡＲファイルサイズ	５３８０６ byte	５３５２１ byte

[考察]

検証結果をみると、たしかにクラスファイルサイズの減少がみられるが、その効果は微々たるものである。時間をかけて適用できる部分を見つけ出し、実際に反映させる手間を考えると、あまり効率的ではないと判断できる。

5．まとめ

5.1 最終結果

ダウンサイジング手法適用の最終結果を以下にまとめて示す。当初 114K あった **JAR ファイルが 18K にまでサイズダウンできた**ことがわかる。機能削減を実施することなく、当初目標のレベル３をクリアできたが、最終目標レベルに到達することはできなかった。

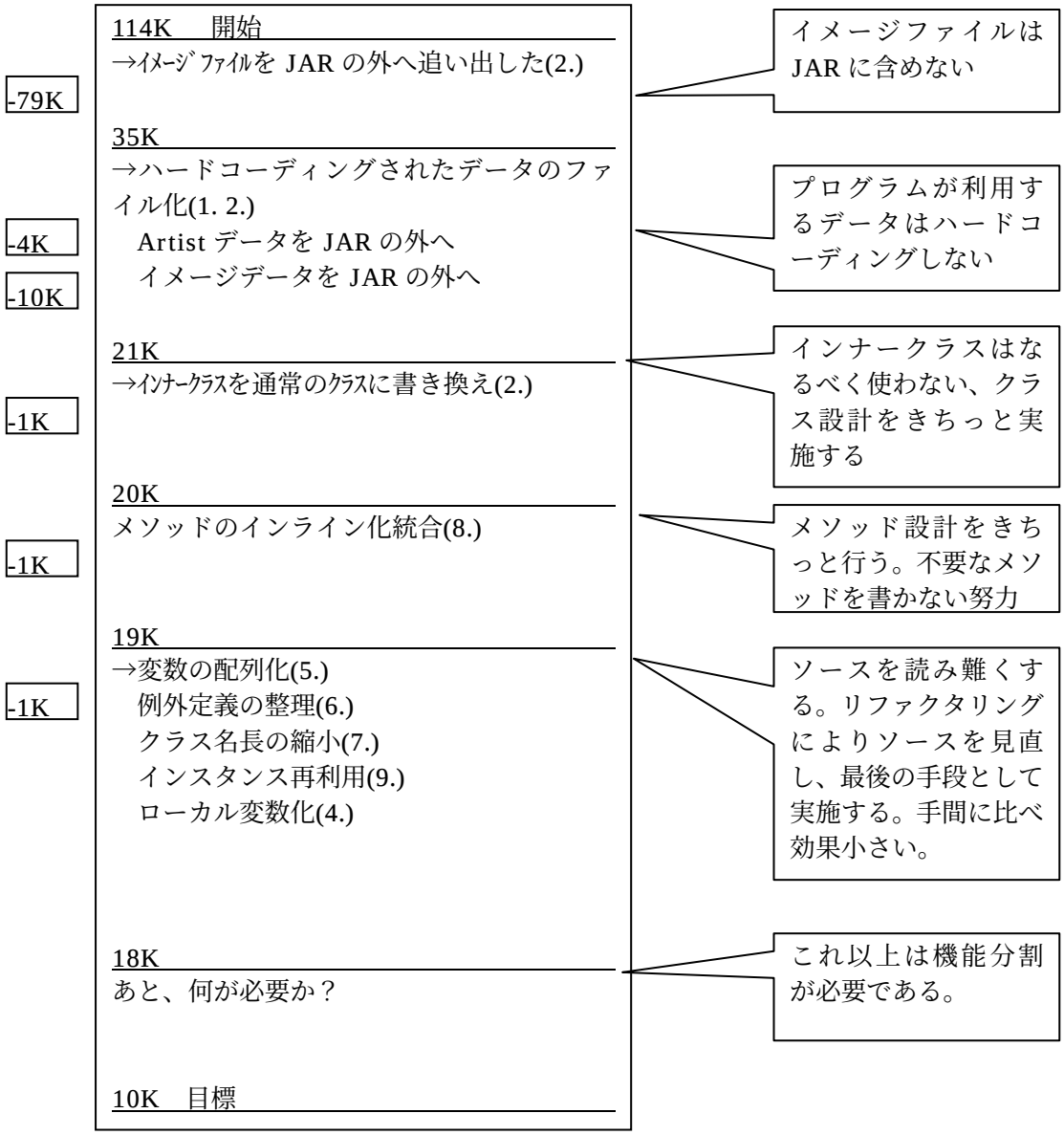

図 13 ダウンサイズの実施結果と教訓

表 11 手法別削減結果

手法	実施内容	削減結果
クラスファイルを少なくする	Artist クラスを JAR の外付けに	-4KB
インナークラスの利用	インナークラスを普通のクラスに（５クラス）	-1KB
イメージファイルの持ち方	JAR 内のイメージを外に置く	-79KB
	ハードコーディングされていたイメージデータをファイル化	-10KB
ローカル変数の利用	クラス内変数４２個中３個をローカル化	-45Bytes
配列の利用	Artist 変数の配列化	-94Bytes
例外処理の工夫	未実施	―
クラス名を短くする	OrderTicket 以外のクラス名を２バイト化	-241Bytes
メソッドのインライン化	３７メソッドを１７メソッドに統合	-1093Bytes
クラスインスタンスの再利用	ソース中の１５箇所で実施	-285Bytes

結果的に効果的であった手法としては、
　－クラスファイル数を少なくする
　－インナークラス利用を控える
　－イメージデータを JAR の外に持つ、また、データをハードコーディングしない
　－メソッドインライン化
であった。これらの手法を実施することで、大きな効果が得られる。また、これらの手法は携帯 Java アプリ開発に限ったものではなく一般的な手法である。
他の手法、
　－変数の配列化
　－クラス名長の縮小
　－インスタンス再利用
　－例外処理見直し
　－ローカル変数化
　については、ソースが読みにくくなり、メンテナンス性、品質を落とすことが予測されるため、i モードアプリの 10K を目標とする特殊な手法と考えたい。例えば、１０Ｋの境界付近で、あと１００バイト削減すれば目標をクリアできるとかいった状況で利用すれば有効である。
　実施した手法を見てわかるとおり、まず、大切なことは、クラス設計およびメソッド設計をあらかじめきちっと実施することである。そして、完成したソースのリファクタリングによってソースの見直しを徹底的に行うことが重要である。すなわち、オブジェクト指向による手法によって設計されたプログラムであれば携帯電話への移植は行いやすいと考えられる。

5.2 評価

　結局、ｉモードで実行可能なサイズ 10K をクリアすることはできなかった。その原因としては以下のことが考えられる。
　１）機能削減をしないことが条件のため、アプリケーション機能設計の見直しを見送った
　２）不要コードの削除を徹底して行わなかった

　しかし、今回、様々な手法を駆使した結果、題材として取り上げたアプリケーションが３０Ｋの範囲であれば動作できるようになった。30K とは、FORMA の制限サイズである。裏返せば、３０Ｋあれば多くの機能を実装したアプリケーションの実行が可能であるという想像も

成り立つ。また、１００Ｋあれば本格的アプリケーションの実行も可能ではないかと推測できる。

次に、各手法に対する評価をまとめておく。

表 12 評価

手法	評価	カテゴリ	コメント
クラスファイルを少なくする	効果大	クラス設計	
インナークラスの利用	効果中	クラス設計	
イメージファイルの持ち方	効果大	アプリ設計	外部アクセスのオーバヘッドとのバランスで考える。
ローカル変数の利用	効果小	クラス設計	
配列の利用	効果小	コーディング技術	i モード対応
例外処理の工夫	効果小	コーディング技術	i モード対応
クラス名を短くする	効果小	クラス設計	i モード対応
メソッドのインライン化	効果中	クラス・メソッド設計	
クラスインスタンスの再利用	効果小	コーディング技術	i モード対応

各手法の評価、実行環境の変化などより、今後、携帯電話アプリケーションは、メンテナンス性を犠牲にしてまでもサイズ縮小を行う必要はないと考える。機能設計／クラス設計を充分に行うことで、互換性も高まり、移植も容易になる。Java 携帯電話アプリケーションはオブジェクト指向を逸脱していると言われることがあるが、今回の経験からすれば、しっかりオブジェクト設計することこそがやはり大切であることがわかる。

最後に、OrderTicket プログラムの再設計を行ったのでクラス間関連図を付録に添付する。今回、新設計のもとで開発することができなかったが、きちっと設計されたプログラムであれば携帯移植も容易であるということを確認するには、オブジェクト指向設計に基づくプログラムの開発実験が必要ではないだろうか。

参考資料：

[文献]
1）「使えるｉアプリの作り方」　Ｊａｖａ　Ｗｏｒｌｄ　２００１年８月号／９月号
2）平成１２年度　ＪＡＶＡ部会作成論文
3）「ｉモード対応Ｊａｖａコンテンツ開発ガイド」第１．１版　ＮＴＴドコモ

[Ｗｅｂサイト]
1）　http://www.atmarkit.co.jp/fmobile/rensai/doja07/doja07.html
2）　http://www.kajas.com/faq/faq5.html
3）　http://gigahz.net/ml/java/
4）　http://java.sun.com/

付録１：OrderTicket プログラム新クラス間関連図