修士論文 2008 年度（平成 20 年度）

携帯ゲームアプリケーションの開発を効率化する
オブジェクト指向を用いたフレームワーク

慶應義塾大学大学院 政策・メディア研究科
橋山 牧人

修士論文要旨 2008 年度（平成 20 年度）

携帯ゲームアプリケーションの開発を効率化するオブジェクト指向を用いたフレームワーク

論文要旨

本研究では，携帯電話で動作するゲームアプリケーション（以下，携帯ゲームと書く）開発の効率化を支援するためのアプリケーションフレームワーク Bridge を提案し，制作した．携帯ゲーム開発においては，携帯端末の高性能化に伴うアプリサイズや画面解像度の増加に比例して，開発の負担が増えている．また，携帯ゲームの動作構造が開発者に依存して属人的なため，再利用性が低く，他のプラットフォームに移植する場合に作業の障壁となる．

提案する Bridge では，これらの課題を 1) オブジェクト指向を用いた画面分割と状態遷移を用いた画面の管理，2) ゲームループ構造の外部化，3) ラッパークラスを用いた移植作業の自動化，によって解決する．Bridge を利用することで，煩雑な画面管理やゲーム構造の定義をフレームワーク側に一任することが可能となる．そのため，携帯ゲームの開発者はゲームのロジックやルールを考えたり，ゲームバランスを調整したりする作業に専念することができる．

制作した Bridge が，携帯ゲーム開発の効率化に有効であるかを検証するために，既存のゲームを Bridge を用いて作り直し，そのソースコードをメトリクスに従って評価した．その結果，規模・複雑度・凝集度の観点から Bridge を用いて作り直したゲームは保守性が高いことが確認できた．

また，Bridge を利用して 300 行ほどの簡単なシューティングテームを開発する試用実験を行った．その結果，平均実装時間が Bridge を利用しない場合は 109 分であったが，Bridge を利用した場合は 80 分に短縮された．さらに，メトリクス測定による定量的な評価とアンケートによる定性的な評価から，Bridge を利用することで携帯ゲームの開発効率と保守性が向上することが分かった．

キーワード: モバイルエンタテインメント, フレームワーク, オブジェクト指向, 状態遷移，ゲームループ

慶應義塾大学大学院 政策・メディア研究科
橋山 牧人

Abstract of Master's Thesis Academic Year 2008

Object-Oriented Framework for Improving Development Efficiency of Mobile Game Application

Summary

This thesis presents application framework named "Bridge" for improving development efficiency of mobile game application. Recently, as the application size gets larger and the screen resolution gets higher with the advance of mobile phones, the developing workload of mobile game application is increasing. Moreover, because a mobile game structure depends too much on a developer, the reuse ratio of source code is low and it is not easy to convert a mobile game into other versions which is executed on other platform.

We propose "Bridge" so as to solve these problems with the following approach: 1)managing a screen by state transition, 2)extacting "Game Loop" structure for reuse, 3)automating conversion by using wrapper class. Then, mobile game developer can leave the complicated screen management and game structure definition to the framework of "Bridge". Therefore, they can concentrate on consideration of the game logic, game rule, and adjustment of the game balance.

We rebuilt existing mobile games with "Bridge" and evaluated them by metrics in order to verify whether "Bridge" was effective in improving development efficiency of mobile game application or not. As a result, it was confirmed that the maintainability of mobile game was high from the point of view of the scale, the complexity degree, and the cohesion level when "Bridge" was used.

We also measured the performance of "Bridge" by developing simple shooting game of about 300 lines of code. It took 80 minutes on the average when "Bridge" was used, though it took 109 minutes on the average without "Bridge". Furthermore, it was observed from a quantitative evaluation by the metrics measurement and from a qualitative evaluation by the questionnaire that the development efficiency and the maintainability of mobile games were improved by using "Bridge."

Keywords: Mobile Entertainment, Framework, Object-Oriented
State Transition, Game Loop

Makito Hashiyama
Graduate School of Media and Governance
Keio University

目 次

図 目 次

表 目 次

第1章　はじめに

本章では，本研究を行う目的と論文全体の構成について述べる．

1.1　研究の目的

本研究の目的は，携帯ゲームに共通である動作構造を抽象化し，開発の効率化を支援することである．そのために，ゲーム全体の構造の分析を行い，それらを抽象化するためのフレームワークを提案し，その実装を行う．

ゲームアプリケーションの多くは，ゲーム内の進行状況に応じて，1) ユーザの入力に対する反応，2) 必要な計算，3) 画面表示という処理を切り替える必要がある．しかし，現在の携帯ゲームの開発では，これらの処理がゲームの進行状況と関連してまとまっておらず，ソースコード上での線引きが曖昧である．その結果，ゲーム全体の進行を管理する処理と，個々の画面で行う処理が密接に結合しているため拡張性に乏しい．また，画面の数が増加するに連れてソースコードが複雑になっていき，開発や保守の効率が悪化する．さらに，ゲーム全体の構造も属人性が強くなっているため，複数のゲームで同じ構造を共有したり，再利用したりすることが難しい．

本研究では，これらの問題を解決するために，オブジェクト指向と状態遷移を用いることでゲーム全体の構造を抽象化し，外部化する．また，煩雑なゲーム画面の管理と遷移を行うために，ゲーム内の個々の画面を分割して管理することができる仕組みを提案する．そして，それらの機能を持つアプリケーションフレームワーク Bridge の実装を行い，その有用性を検証する．

1.2　論文の構成

まず，2 章で本研究の対象である携帯ゲームの開発における問題点を明らかにする．次に，3 章で関連技術について述べる．続いて，4 章では，Bridge の理念と位置づけについて説明を行い，5 章では，Bridge の設計方針と実装について言及する．6 章では，Bridge の適用事例の分析と試用実験による評価を行う．最後に 7 章で今後の研究課題とまとめを述べる．

第2章　研究の背景

本章では，国内で最も携帯ゲームの供給数が多いNTTドコモのiアプリ [1] を例に，携帯ゲーム開発が抱える問題点を指摘する．2.1節で携帯ゲーム開発を取り巻く環境に着目し，現行の開発・移植作業の問題点を明らかにする．2.2節で携帯ゲームに適している実装方式について説明し，その構造が外部化されていない問題について述べる．

2.1　携帯ゲーム開発の現状

2.1.1　開発・移植作業における負担の増大

メガアプリの普及

図2.1は，iアプリサイズの上限の推移を表わしたものである．NTTドコモの携帯電話にiアプリが初めて搭載された2001年には，アプリケーションとスクラッチパッド（iアプリがデータ保存に利用できるアプリ外の領域）がそれぞれ10KB（キロバイト）しか用意されていなかった．しかし，5年後の2006年にはアプリケーションとスクラッチパッドを合わせたサイズが1MB（メガバイト）を超え，メガアプリと呼ばれるものが普及するようになった．携帯ゲームも例外ではなく，これまではいくつかのiアプリに分割されていた本格的なゲームが，1つのアプリとしてリリースできるようになった．そのため，これまでは容量の制限により移植できなかった家庭用ゲーム機向け大作RPG[1]などがメガアプリに移植されるようになった．さらに，2008年12月にはアプリサイズの上限は2MBを超え，より規模の大きなゲームを作ることができるようになり，開発・移植作業における負担が増大した．

画面サイズの拡大

表2.1は，iアプリの画面描画領域サイズの推移をまとめたものである．2008年現在，画面解像度は2001年に比べるとおよそ25倍となり，画質が飛躍的に向上した．そのため，ユーザが携帯ゲームのグラフィックに要求する品質は上がり，必要な開発作業が増加した．

[1] ラングリッサー (メサイア, 1993)，幻想水滸伝 (コナミ, 1995) など

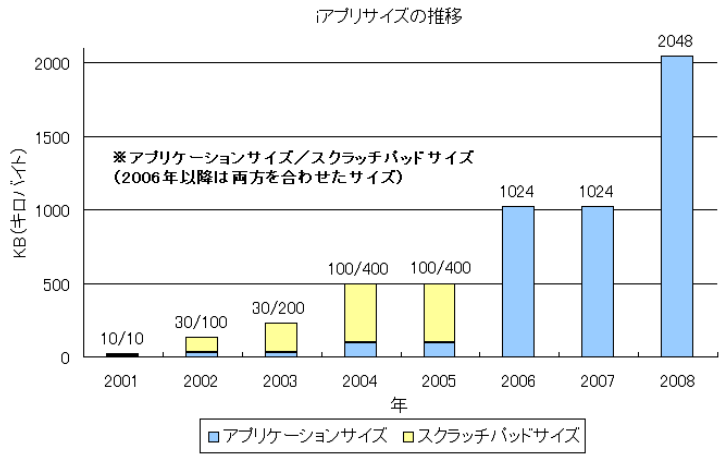


図 2.1: i アプリサイズの上限の推移

表 2.1: i アプリ画面領域サイズの推移

年	2001	2002	2003	2004	2005	2006	2007	2008
幅 (dot)	120	132	240	240	240	240	480	480
高さ (dot)	130	144	240	240	240	320	864	864

2.1.2 開発効率を考慮しないコーディング手法

i アプリを搭載できるようになった携帯電話が日本国内に普及しはじめた 2001 年頃は，アプリケーションのサイズが 10KB（キロバイト）に制限されており，アプリケーション開発者はいかにサイズを小さく押さえて開発するかということに主眼を置いていた．画像や音声などの品質を落とすことによってデータの容量を抑えるだけでは不十分で，ソースコードも軽量化しなければならなかった．そのため，クラスの数は必要最低限である 1 つないしは 2 つにして，さらに片方のクラスに機能を集中させる，という開発効率よりも軽量化を重視したコーディングスタイルをとる必要があった [2]．

だが，現在では，図 2.1 のように i アプリのサイズが増えたことで，携帯ゲーム全体の容量においてソースコードが占める割合は小さくなり，コーディングスタイルによってソースコードの軽量化を行なう必要はなくなった．また，開発の規模が大きくなったことで，過去には 1 人で可能であった開発や移植の作業を複数人で行なうことも多くなった．それでも，携帯ゲームの開発現場では過去の習慣が抜けておらず，オブジェクト指向による再利用性を意識したコーディングとは無縁であり，構造化も行わず，ソフトウェア工学的な視点から携帯ゲームの開発効率を向上させる試みを行うことは少ない．さらに，携帯ゲームは習慣的に曖昧な仕様書や口頭でのやり取りのみで開発することも多い．そのため，ソースコードのみが唯一の仕様書であるといったケー

スもあり，可読性の低いソースコードは保守や移植作業を妨げる原因となっている．

2.1.3 自動化されていない移植作業

国内の携帯アプリは表2.2にあげるように，携帯ブランドごとに採用しているプラットフォーム仕様が異なる．そのため，あるキャリアで開発したアプリを別のキャリアで動作させるには，表2.3のような移植作業が発生する．移植作業には，ソースコードの書き換えとそれ以外（設定ファイルの書き換え，リソース形式の変換など）がある．

このうち，ソースコードの移植作業はその大半は共通化することができる．実際に，部分的に見ればこれらを共通化しているケースもいくつか存在する [3][4][5]．しかし，そのいずれも部分的な共通化作業のみを扱っており，ゲーム全体の構造（起動，メインループ，画面遷移など）が共通化されておらず，移植作業を完全に自動化することができていない．

表 2.2: 日本の主な携帯ブランドと採用しているプラットフォーム仕様

ブランド名	アプリ名	プラットフォーム仕様	言語
NTT Docomo	i アプリ	J2ME(DoJa[1])	Java
Softbank	S!アプリ	J2ME(MIDP[2]) + MEXA[3]	Java
au	オープンアプリ	J2ME(MIDP)	Java
	BREW アプリ	BREW[4]	C/C++
WILLCOM	WILLCOM アプリ	J2ME(MIDP)	Java

[1]DoJa . . . NTT ドコモグループの携帯電話に搭載される Java 実行環境の仕様
[2]Mobile Information Device Profile . . . 携帯電話や PDA のような組み込み機器での Java の利用について記述した仕様
[3]Mobile Entertainment eXtension API . . . S!アプリを開発するための拡張 API
[4]Binary Runtime Environment for Wireless . . . CDMA 携帯電話向けのプラットフォーム

表 2.3: i アプリと S!アプリで異なる仕様の例

項目	i アプリの仕様	S!アプリの仕様
文字・画像の描画	座標のみを指定	座標とアンカーの指定
色の設定	色の名前，RGB を取得して指定	RGB を直接指定
キーイベントの取得	processEvent	keypressed, keyreleased
ソフトキーの取得	通常のキーと同様	コマンドによる通知
ユーザ入力の取得	IME を利用	TextBox を利用
外部データ領域	スクラッチパッド	レコードストア
画像形式	GIF, JPEG	PNG, JPEG
音声形式	MLD（i メロディ）	SPF（フレーズ）

2.2　携帯ゲームで採用されている実装方式

2.2.1　ゲームループ方式とイベントドリブン方式

ゲームではイベントドリブン方式ではなく，ゲームループ方式を採用することが一般的である．これは，後述するコマ落ちの問題（2.2.2 参照）を回避するためである．本項では，コマ落ちの問題について述べる前に，ゲームループ方式とイベントドリブン方式について解説する．

ゲームループ方式

図 2.2 はゲームループ方式でゲームを開発した場合のプログラムの流れを表す．ゲームループ方式では，プログラムがハードウェアの状態を断続的に問い合わせ，特定のタイミングにおけるアクションの発生有無を確認しつつ条件分岐して実行する．ユーザの操作とは無関係に，ひとつひとつの処理が固定間隔で制御できるメリットがある反面，常に処理を行っているため CPU に大きな負担をかけることになる．また，次の問い合わせが行われるまでの間にアクションが発生した場合は，そのアクションを取りこぼしてしまう．

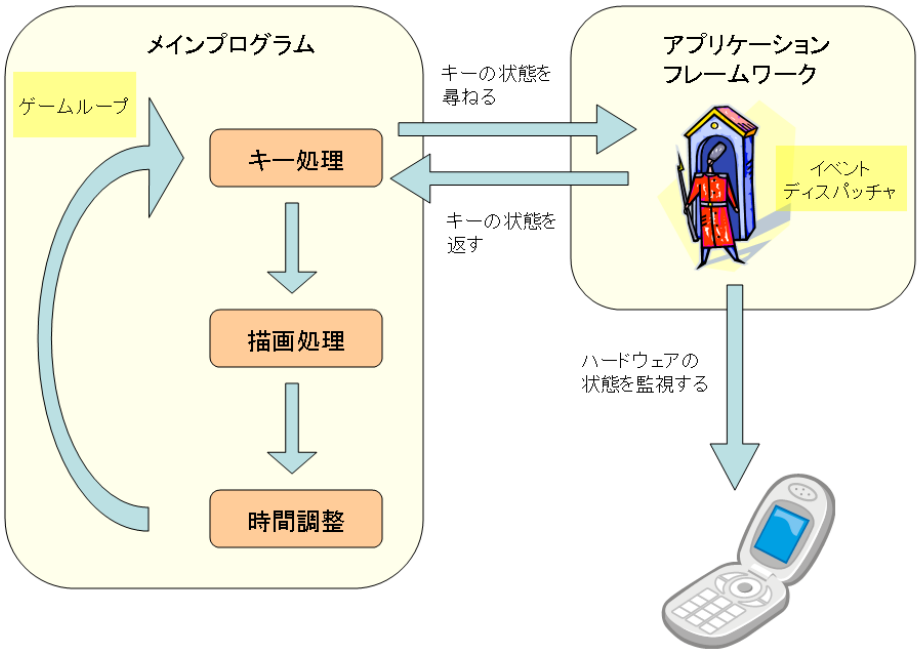


図 2.2: ゲームループ方式のプログラムの流れ

イベントドリブン方式

図 2.3 はイベントドリブン方式でゲームを開発した場合のプログラムの流れを表す．イベントドリブン方式では,「キーを押下した」「一定時間が経過した」などのイベントをイベントディスパッチャが監視して，イベントハンドラにイベントの通知を行うことで，処理を実行する．ユーザの操作に基づいた処理（GUI の制御など）を行う場合に最適であり，ゲームループ方式の問題であった CPU への負担も軽減できる．しかし，処理の発生するタイミングをイベントに委ねるため，プログラマが意図しない間隔でイベントを通知することがある．

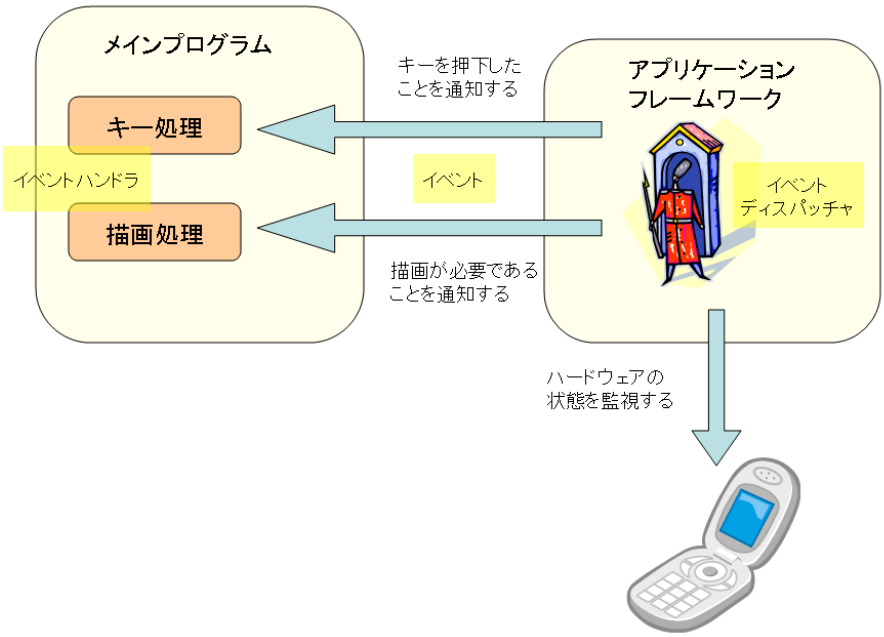


図 2.3: イベントドリブン方式のプログラムの流れ

2.2.2　イベントドリブン方式におけるコマ落ちの問題

イベントドリブン方式では，イベントを発生した順番に通知することが保障されていないために，コマ落ちの問題が発生する．図 2.4 は，キー操作によって星型の図形が移動するという簡単なアニメーションを行なったときの，イベントキュー（発生したイベントを一時的に保存するバッファ）と，画面の変化を表している．ここで，本文中の括弧内の数字は図中の番号と一致している．

ユーザが上キーを押下すると (1)，ハードウェアが上キーイベントを発行し，イベントキューに積む (2)．イベントディスパッチャがイベントキューに積まれている上キーイベントをイベントハンドラに通知すると，上キーの処理を行うイベントハンドラが星型の図形の座標を更新し (3)，画面を再描画するための描画イベントを発行してイベントキューに積む (4)．イベントディスパッチャがイベントキューに積んである描画イベントをイベントハンドラに通知すると，描画処理を行うイベントハンドラが画面を再描画する (5)．ユーザが右キーを押下するときも同様である (6-10)．

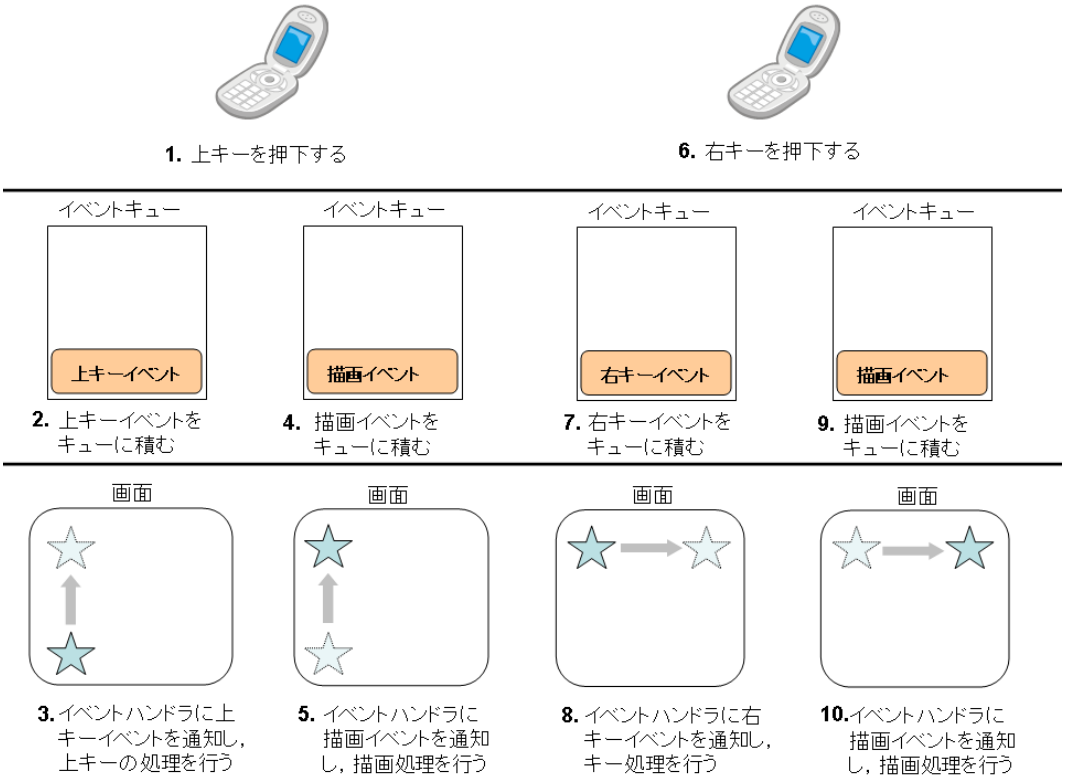


図 2.4: イベントドリブン方式プログラムの問題（正常な処理）

図 2.5 は，図 2.4 と同じプログラムにおいてコマ落ちが発生する場合を表している．ここでも，本文中の括弧内の数字は図中の番号と一致している．

ユーザが上キーを押してから上キーを処理するイベントハンドラが星型の図形の座標が更新するところ (1-3) までは正常な処理（図 2.4 参照）と同様である．ここで，上キーを処理するイベントハンドラが描画イベントを発行してイベントキューに積む前にユーザが右キーを押下すると (4)，ハードウェアが右キーイベントを描画イベントよりも先にイベントキューに積む (5)(6)．イベントディスパッチャはイベントハンドラに対して，描画イベントよりも先に右キーイベントが通知するため，右キーの処理

を行うイベントハンドラが再び星型の図形の座標が更新する (7)．その後，イベントディスパッチャはイベントキューに積んである描画イベントをイベントハンドラに通知し，描画処理を行うイベントハンドラが画面を再描画する．その結果，上キーを押した直後の描画が画面に反映されず，画面上には星型の図形が斜めに移動したかのように表示する (8)．この現象がコマ落ちである．

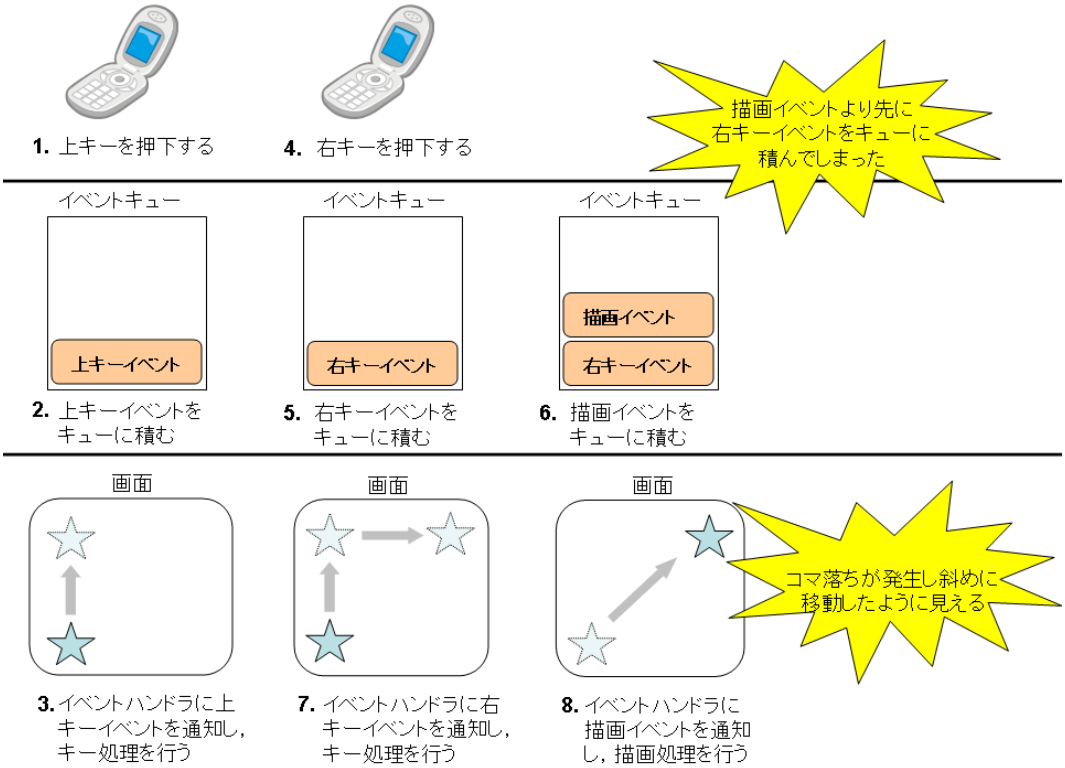


図 2.5: イベントドリブン方式プログラムの問題（コマ落ち発生）

ゲームループ方式では，処理の実行間隔をプログラマが制御できるので，描画とキー処理の順番が食い違うことで発生するコマ落ちの問題は解決できる．一般的に，ゲームループ方式は，常に描画しつづけているようなゲーム，ユーザのキー操作に敏感に反応する必要がある動的なゲーム（シューティングゲーム，アクションゲーム，音楽ゲーム etc...）に向いている．一方，イベントドリブン方式は，ユーザが入力を行って初めて動作するようなゲーム，ユーザの操作に対して反応速度を要求しない静的なゲーム（カードゲーム，パズルゲーム，タイピングゲーム etc...）に向いている．しかし，静的なゲームであっても，ゲームに与える影響は少ないとはいえコマ落ちの問題が発生するため，ほぼすべてのゲームがゲームループ方式を採用している．

2.2.3　携帯ゲームにおけるゲームループ構造の独自定義

i アプリや S!アプリの開発に利用するプラットフォーム仕様である DoJa プロファイルや MIDP は，イベントドリブン方式で設計してある．これは，リアルタイム性が要求されない業務用のアプリケーションや，待ち受け状態で常に起動し続けることができる待ち受けアプリなどを開発する上では，CPU にかかる負担を減らし電力消費を抑えることができるので，好都合である．しかし，携帯ゲームをこの設計に基づいて開発する場合，前述のコマ落ち問題が発生するため，都合が悪い．そこで，携帯ゲーム開発者はゲームループ方式が利用できるような独自構造を自前で定義する．しかし，この構造を作るだけで大きな手間になっている．また，ゲームループの構造は属人的で外部化しないことが多いため再利用性が低く，移植を考慮したときの効率も悪い．

2.3　携帯ゲーム開発が抱える問題に対する要求

本節では，これまでの議論から携帯ゲーム開発が抱える問題に対する次の要求を整理する．

- 開発効率，保守性を考慮したコーディング手法を導入すること．
- 他のプラットフォームへの移植作業をできる限り自動化すること．
- あらゆる携帯ゲームでゲームループ構造を再利用できるように外部化すること．

3 章で既存の技術だけではこれらの要求が解決できないことを検証し，4 章でこれを解決するアプリケーションフレームワーク Bridge を提案する．

第3章　関連技術

本章では，2.3 節で明らかになった問題に対する要求を満たすことができる可能性のある既存の関連技術について調査を行い，それらがすべての問題に対処可能であるかという視点から評価した結果について述べる．以下に，Caledonia，mokit，uk ゲームライブラリ，コンバータ，RPG ツクールを取り上げて詳しく紹介する．

3.1　Caledonia

Caledonia は PotRinStudent が開発した，携帯電話向け Java 低レベル API 実装のフレームワークである [6]．DoJa プロファイルが提供する API をラップして，i アプリの開発を補助する．Caledonia はフレームワークの構造が単純であり，そのほとんどが画面描画に対するライブラリである．そのため，簡単な画面表示や操作のみを行う小規模な携帯アプリであれば短い時間で開発できる．

図 3.1 に Caledonia のクラス図を示す．Caledonia は DoJa プロファイルの Graphics クラスをラップすることで，MIDP への移植作業を減らしている．また，アプリの起動からメインループまでの処理をフレームワークで処理し，シーン番号による画面遷移の管理を行っている．

しかし，Caledonia を携帯ゲーム開発に用いるとき，1) コマ落ち問題に対応していない，2) 画面管理の方法が不十分である，3) 移植が容易ではない，という大きな問題点が浮上する．

1 つ目の問題については，フレームワーク全体の構造にイベントドリブン方式を採用していることが原因である（2.2.2 参照）．

2 つ目の問題については，画面番号によって画面を管理していることが原因である．Caledonia で描画やキー処理を画面毎に分けるには，画面番号を条件として分岐するプログラムをキー処理や描画のメソッド内に書くことになる．そのようなコーディングを行うと，キー処理や描画のプログラムが複雑になり保守しにくくなる．また，画面番号では直前のシーンしか保存されないため，ゲームにありがちな複雑な画面遷移には対応できない上，画面の数を増やすたびに処理に関わるすべての分岐を見直す必要があるので，膨大な手間がかかる．

3 つ目の問題については，Caledonia では端末から呼び出されるスレッドと同一スレッド内で，メインループを実装していることが原因である．DoJa プロファイルでは，仕様上このような実装を行っても正常に動作するが，MIDP で同じ実装を行うと，メインループ中に端末が通知するイベントを処理できずにゲームが終了できなくなるなど，様々な問題が発生する．そのため，MIDP を利用しているキャリアへの移植を行う場合は，大幅な動作構造の書き換えが必要になる．また，描画と画面管理以外のサポートを行っておらず，その他の機能を使う場合は，すべて手作業で移植しなければならない．

したがって，Caledonia は携帯ゲーム開発における要求（2.3 節参照）を満たすには不十分であると判断できる．

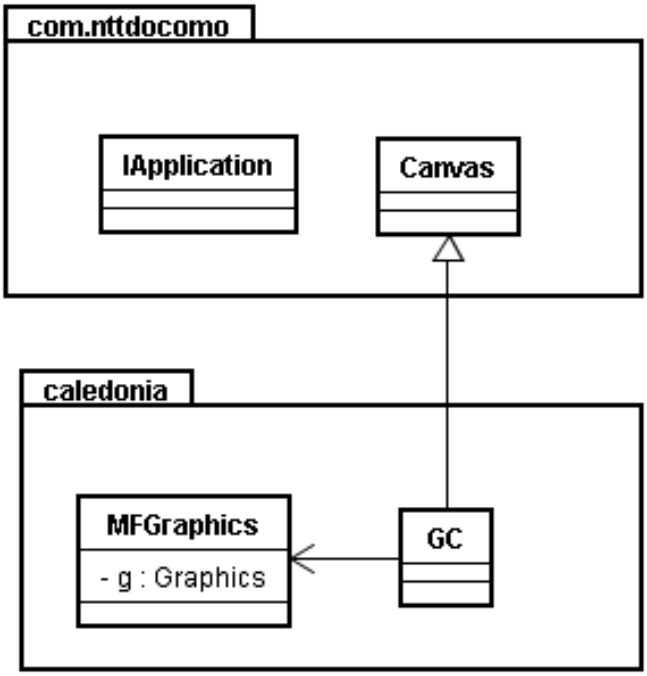


図 3.1: Caledonia クラス図

3.2 mokit

mokit は西岡拓人が個人で開発した，携帯アプリ開発フレームワークである [7]．DoJa プロファイルと MIDP に対応しており，それぞれ i アプリと S!アプリの開発を補助する．mokit は，携帯用業務アプリを開発することを主眼においており，通信や携帯端末のデバイス情報を取得する機能が充実している．また，描画やキーの処理を画面単位で分離できるため，保守を簡単に行うことができる．

図 3.2 に mokit のクラス図を示す．mokit は前述の Caledonia とは異なり，端末から呼び出されるスレッドとは別スレッドとしてメインループを定義している．また，多くの API が移植を意識して共通化しているため，MIDP を利用しているアプリへの移植の問題は少なくなる．また，処理を画面単位で分離しているので，Caledonia のように画面が増えるたびに関連するすべての分岐処理を見直すという事態にはならない．グラフィックや通信に関してはライブラリも充実しており，特に通信では，キューを用いた HTTP 通信の管理を行っている．

しかし，mokit を携帯ゲーム開発に用いるとき，1) コマ落ち問題に対応していない，2) 画面遷移の管理が不十分である，という大きな問題点が浮上する．

1 つ目の問題については，Caledonia と同様にイベントドリブン方式を採用していることが原因である（2.2.2 参照）．

2 つ目の問題については，画面の履歴や，次に遷移する画面の情報をアプリケーション側で管理しなければならないことが原因である．mokit では，画面数が増えるたびに，画面遷移に関連する分岐を修正しなければならない．そのため，例えば，どの画面からでも遷移でき，元の画面に戻ることができるメニュー画面などを実現するのは困難である．

したがって，mokit は携帯ゲーム開発における要求（2.3 節参照）を満たすには不十分であると判断できる．

3.3 uk ゲームライブラリ

uk ゲームライブラリとは，宍戸輝光が個人で開発したフレームワーククラスライブラリである [8]．MIDP と DoJa プロファイルの他，J2SE[1] にも対応している．uk ゲームライブラリはその名の通りゲームを開発するためのライブラリであり，ゲームに必要な機能を一通りサポートしており，簡単なゲームを作るならば十分に有用である．

図 3.3 に uk ゲームライブラリのクラス図を示す．uk ゲームライブラリは，mokit と同じくメインループを端末から呼び出される異なるスレッドで定義されている．グラフィック，サウンド，データ保存，通信の多くをサポートしており，さらに独自のウィンドウやダイアログボックスの雛型が利用できる．API も移植が意識されており，PC アプリ，i アプリ，S!アプリ間の移植を容易に行うことが可能である．

しかし，uk ゲームライブラリを携帯ゲームアプリ開発に用いるとき，1) コマ落ち問題に対応していない，2) 画面遷移が管理できない，という大きな問題点が浮上する．

1 つ目の問題については，ゲーム向けのライブラリでありながら，mokit，Caledonia 同様にイベントドリブン方式を採用していることが原因である（2.2.2 参照）．

[1] Java 2 の機能セットの 1 つでパソコンなどのネットワーククライアント環境，あるいはスタンドアローン用途向けの機能をまとめたもの

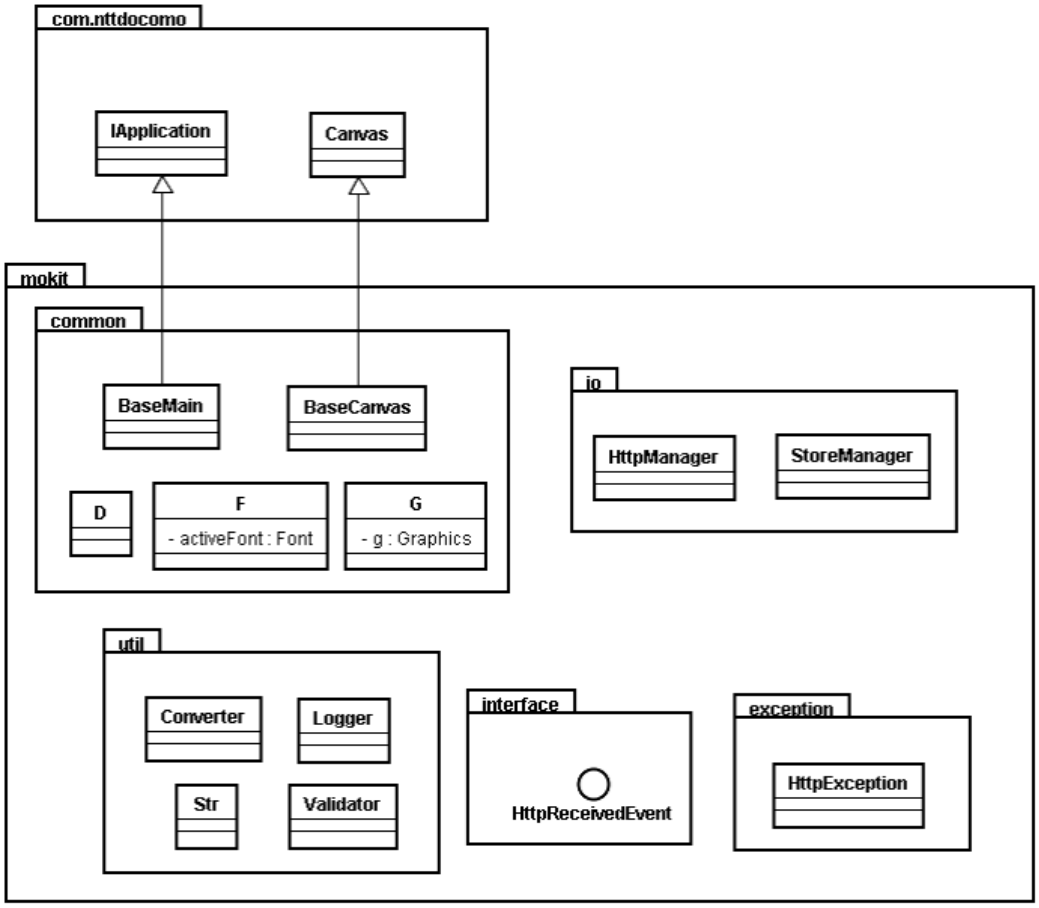


図 3.2: Mokit クラス図

2つ目の問題については，画面遷移の管理がまったく行われていないことである．画面遷移の機構を自前で用意する必要がある上，画面の数を増やすたびにmokit，Caledoniaと同様に画面遷移に関連するすべての分岐処理を見直すという問題が発生する．そのため，この uk ゲームライブラリを利用して規模が大きく画面数が多いゲームを開発することは困難である．

したがって，uk ゲームライブラリは携帯ゲーム開発における要求（2.3 節参照）を満たすには不十分であると判断できる．

3.4 コンバータ

コンバータとは東京工科大学の高橋武司が卒業研究として開発した，移植作業を簡略化するための変換プログラムである [9]．作者が名前をつけていないため，本稿では便宜的にコンバータという名前をつけておく．コンバータは，描画やキー処理の一部

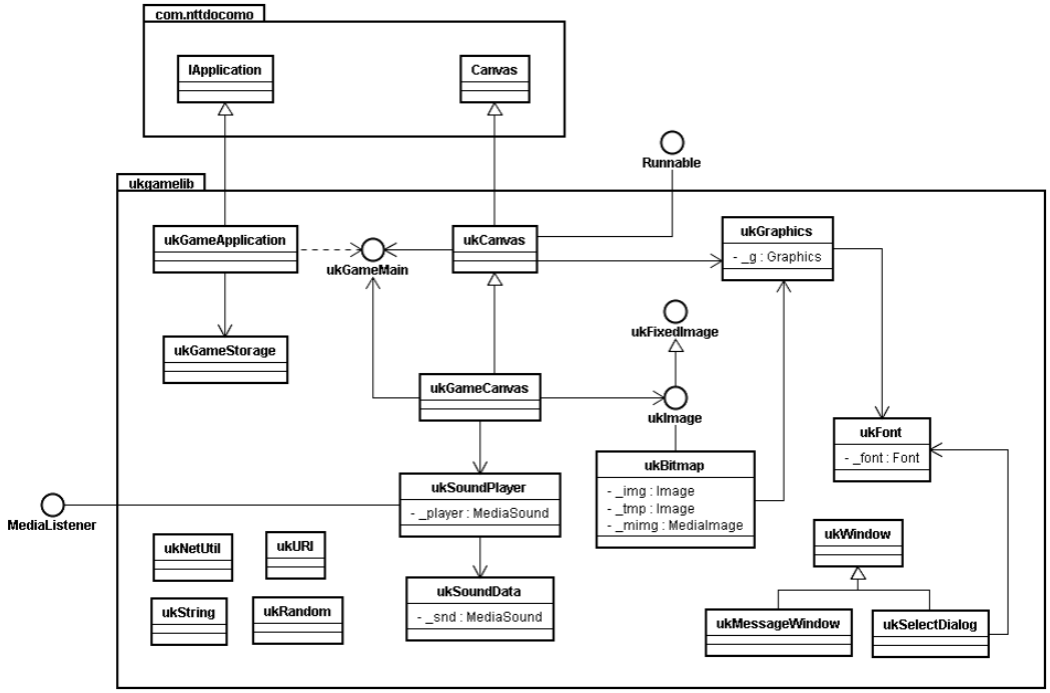


図 3.3: uk ゲームライブラリクラス図

を変換し，i アプリから S!アプリへの移植作業を軽減することができる．

コンバータを用いて携帯ゲームを移植しようとすると，1) 一般的なエディタが持つ編集機能と大差がない，2) 複雑な構造を持つ移植には対応できない，という大きな問題点が浮上する．

これらの問題については，コンバータがソースコード 1 行ずつ検索し，置換するという作業を行っていることが原因である．コンバータが対応している API が描画とキー処理の一部であり，エディタによる検索・置換を用いて個別に変換してもあまり効率が変わらない．また，検索・置換という方針を採用しているため，設計思想がまったく異なるような API を変換することができない．たとえば，i アプリにおける IME などのユーザ入力の受付をこの方針で変換するのは不可能である．

したがって，コンバータは携帯ゲーム開発における要求（2.3 節参照）を満たすには不十分であると判断できる．

3.5 RPG ツクールシリーズ

RPG ツクールシリーズは，株式会社エンターブレイズから発売されている，RPG（ロールプレイングゲーム）開発用ソフトウェアシリーズである [10]．シリーズのう

ち,「RPG ツクールα」「RPG ツクール for Mobile」では，PC 上で GUI による簡単な操作によって，i アプリ対応の RPG を開発することができる．

RPG ツクールは RPG を開発するために特化した機能（イベントの管理，戦闘シーンの設定，パラメータの調整など）を提供している．そのため，RPG 向けの様々なライブラリやツール群が用意されており，効率的な開発を行うことが可能である．しかし，シューティングゲームやアクションゲームなどジャンルが異なるゲームを開発することはできない．また，RPG ツクール上で直接コーディングを行うことができないので，RPG ツクールが提供する機能を組み合わせた限定的な RPG しか作成することができない．

したがって，RPG ツクールは携帯ゲーム開発における要求（2.3 節参照）を満たすには不十分であると判断できる．

第4章　Bridgeの理念

本章では，2.3 節で述べたすべての要求を満たすアプリケーションフレームワーク Bridge を提案し，その基本的な理念を示す．

4.1　Bridgeの位置づけ

図 4.1 は J2ME[1]プラットフォーム上で動作する携帯アプリケーションと Bridge の関係性を表したものである．Bridge は，DoJa プロファイルと MIDP 上（表 2.2 参照）に構築されており，i アプリ，S!アプリ，オープンアプリ，WILLCOM アプリに対応する．

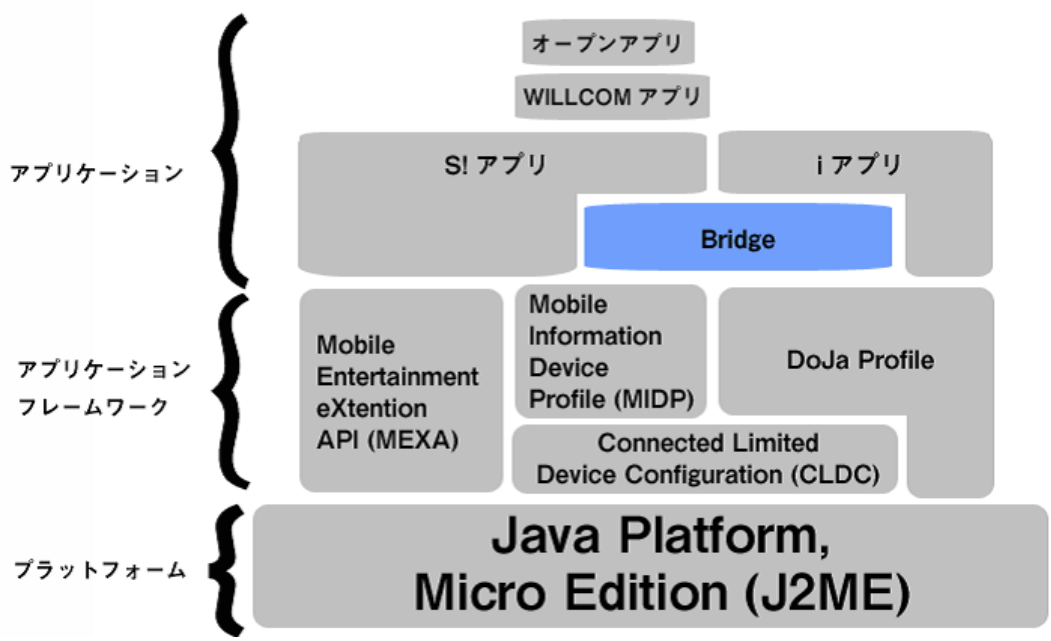


図 4.1: Bridge の位置づけ

[1]Java 2 の機能セットの 1 つで携帯電話などの組み込み機器向けの機能をまとめたもの

4.2 Bridge のアーキテクチャ

4.2.1 動的な処理の切り替えによる画面の分割管理

図 4.2 は Bridge のアーキテクチャを表している．Bridge では，ゲーム全体を画面単位で分割し，それぞれを別々に管理する方法を提供する．ゲーム本体は Bridge が提供するゲームループ構造の中で動作し，ゲームループ構造の中から現在の画面に対しての処理を一定時間ごとに呼び出す．各画面をオブジェクトとしてとらえてキー処理，計算処理，描画処理をカプセル化することで，画面ごとに必要な処理を簡潔に記述することができ，画面の管理の手間が大幅に省ける．ゲームが画面ごとに分かれているためソースコードも複雑になりにくく，開発効率や保守性も高まる．

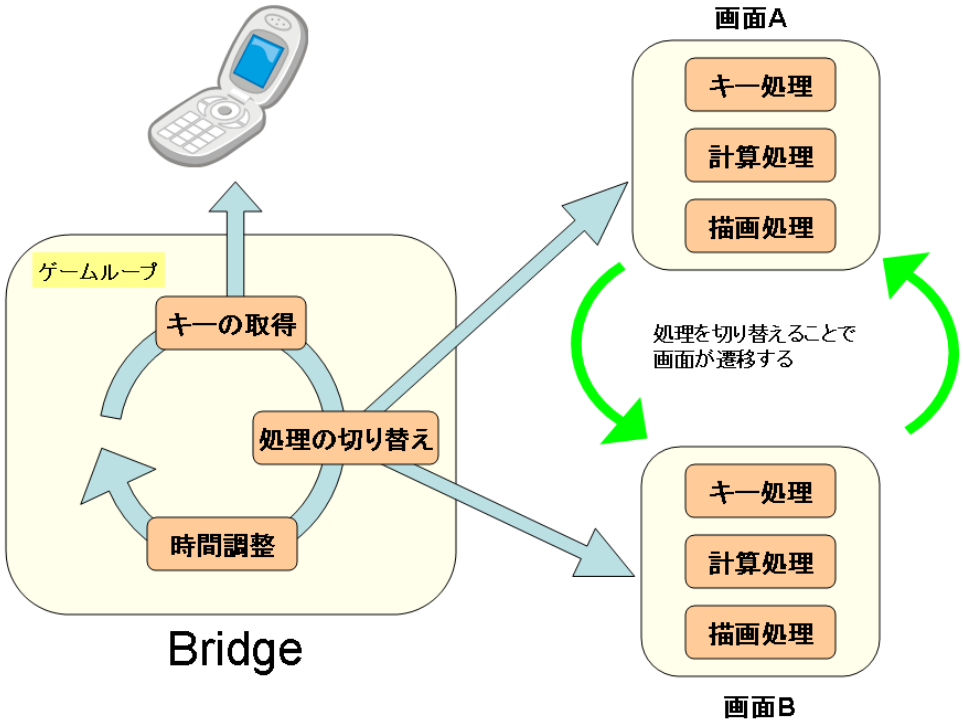


図 4.2: Bridge のアーキテクチャ

4.2.2　スタックによる画面遷移の管理

ゲームを開発する際に必ず行わなければならないのが画面遷移の設計である．どの画面からゲームが始まり，時間経過やキー操作によってどの画面からどの画面に遷移するかということを決定せずにゲームを開発すると，分岐が乱立することになり，画面数が多くなったときにプログラマはゲームの管理ができなくなってしまう．画面を管理する最も一般的な方法が，定数を用いた画面管理である．現在の画面を表す変数を用意し，その変数を書き換えることで画面遷移を実現できる．しかし，この方法では各画面で利用されている変数やオブジェクトの情報が画面内に隠蔽されないために，容易に外部から書き換えることができてしまう．そのため，遷移前の画面の全情報を確実に保存しているかどうかを保障することが難しい．

Bridge ではゲーム内の各画面をオブジェクトとして保持していることを利用して，画面遷移の管理にスタック（Stack）を利用する．画面遷移時に，遷移前の画面オブジェクトをスタックに push しておくと，遷移後にスタックから pop することで，遷移前の画面オブジェクトを取得することができる．この方法なら画面遷移時の履歴としてそのときの画面オブジェクトを保存することができるため，遷移前の画面を確実に保存していることを保障できる．

図 4.3 は，特定の画面遷移が行われたときのスタックの状態を表現したものである．ここで，本文中の数字は図中の番号と一致している．

1. 画面 A から画面 B に遷移し，スタックに画面 A のオブジェクトを push する．
2. 画面 B から画面 C に遷移し，スタックに画面 B のオブジェクトを push する．
3. 画面 C から画面 B に戻るために，スタックを pop して画面 B のオブジェクトを取得し，画面 C のオブジェクトを push する．
4. 画面 B から画面 D に遷移し，スタックに画面 B のオブジェクトを push する．
5. 画面 D から画面 A に戻るために，スタックを 3 回 pop して画面 A のオブジェクトを取得し，画面 D のオブジェクトをスタックに push する．

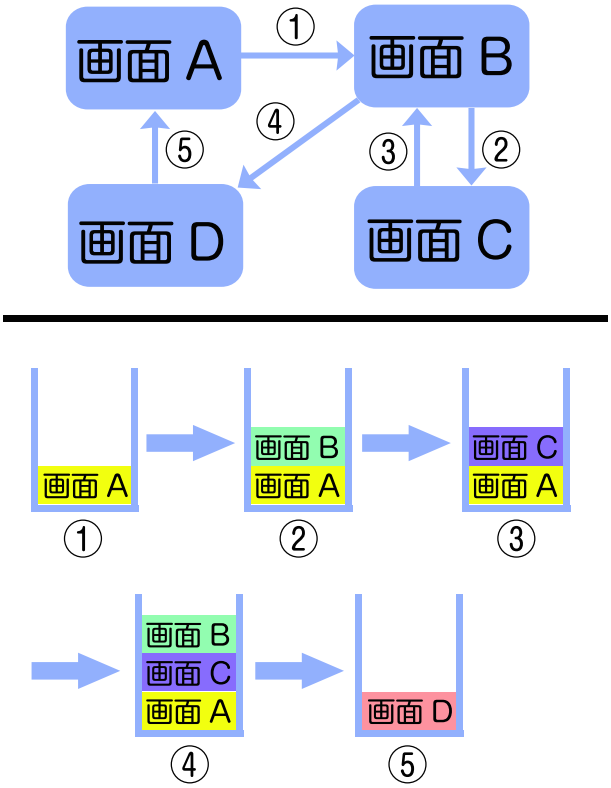


図 4.3: 画面遷移時のスタックの状態

4.3 Bridge を用いた携帯ゲーム開発のシナリオ

Bridge は図 4.4 のように eclipse[11] のプラグインとして動作しており，Bridge を使うことで，次のように効率良く携帯ゲームを開発することができる．なお，プラグインに関しては 5.5 節で述べる．

1. eclipse 上から Bridge を利用して，i アプリ用のスケルトンコードを生成する．
2. ゲームの仕様から必要な画面とその遷移を考える．
3. 画面クラス生成機能を用いて必要な画面を生成する．
4. 各画面クラスのキー処理，計算処理，描画処理を記述する．
5. そのほか，ゲームに必要なロジックを定義する．
6. コンバータを利用して，開発した i アプリを S!アプリに変換する．

このうち，携帯ゲーム開発者が最初から行う必要があるのは，2，4，5 の工程である．ゲームループ構造の定義，画面の管理と遷移，移植作業はすべて Bridge が担当し，携帯ゲーム開発者はゲームのロジックや各画面ごとの処理の記述に専念できる．

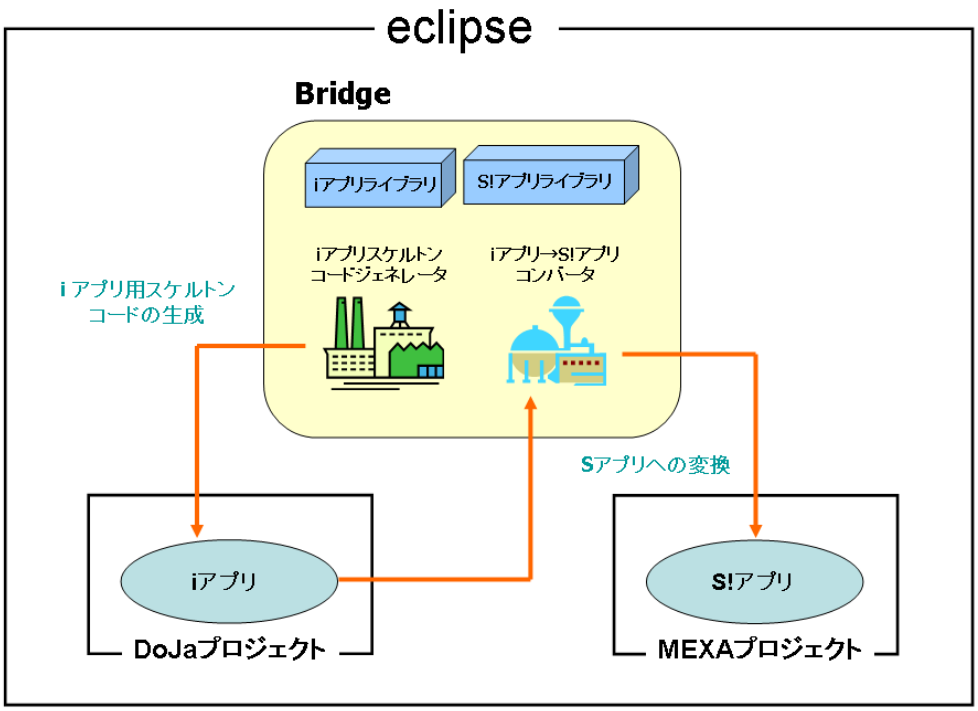


図 4.4: eclipse プラグインとしての Bridge の役割

第5章　Bridgeの設計と実装

本章では，4 章で述べた理念を実現するために行なった，Bridge の設計と実装について述べる．

5.1　Bridgeの全体設計

5.1.1　静的な設計

フレームワークとは,「抽象クラスの集合とそのインスタンス間の相互作用によって表現された，システムの全体または一部の再利用可能な設計 [12]」である．Bridge は大きく分けて 9 つのクラス（うち 2 つが抽象クラス）と 2 つのインターフェースから構成されており（図 5.1 参照），それぞれのクラス・インターフェースは以下のような役割をもつ．

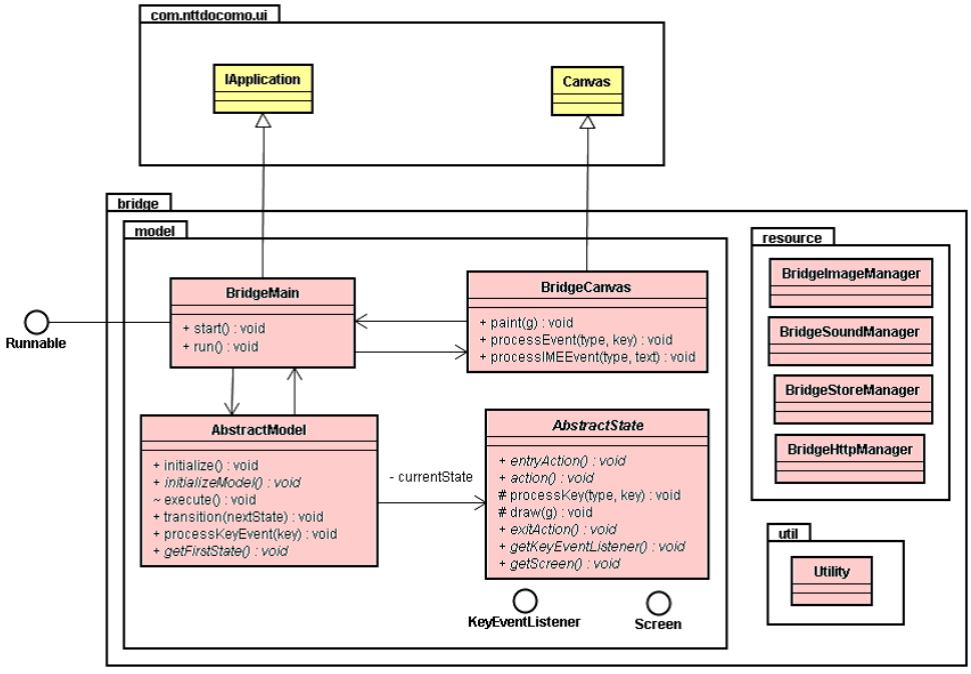


図 5.1: Bridge クラス図

BridgeMain クラス

i アプリの雛型となる com.nttdocomo.ui.IApplication クラスを継承し，ゲーム起動時に携帯電話の VM から呼ばれるクラスで，ゲームの起動，一時停止，終了などのライフサイクルを定義している．

また，コマ落ち問題（2.2.2 参照）に対処するために，次のような独自のゲームループ構造を定義している．

- ゲームループスレッドの作成
  Runnable インターフェースを実装し，携帯ゲームのライフサイクルを管理するスレッドとは別にゲームループ用のスレッドを用意し，スレッド内に無限ループを作り，一定時間ごとの計算処理，キー処理，描画処理を呼び出す．

- フレームレートの調整
  ゲームループの場合，前述のコマ落ちの問題以外にも，フレームレート（1 秒間に画面を更新する回数）が一定ではないとスムーズなアニメーションが行われないことがある．この問題を回避するために，ループ毎にかかった時間を測定し，ループの間隔ができるだけ一定になるようにフレームレートの調整を行う．

BridgeCanvas クラス

低レベル API で使用する com.nttdocomo.ui.Canvas クラスを継承し，描画やユーザ入力のイベントを処理するクラスである．ハードウェアが発行したキーイベントは，このクラスの processEvent メソッドと，BridgeMain クラスを介し，AbstractModel クラスが保持する．また，このクラスから携帯端末のキー情報や画面サイズなどを取得することができる．

AbstractModel クラス

ゲームのデータやロジックを管理するクラスで，各ゲームのモデルはこのクラスを継承して定義することを想定している．また，スタックを用いた画面遷移の管理もこのクラスが行っており，各画面で定義した描画処理やキー処理を動的に切り替えて呼び出している．5.2 節で後述するが，このクラスは，State パターンの Context に該当する [13].

AbstractState クラス

抽象的なゲーム画面を表すクラスで，このクラスのサブクラスがゲームにおける各画面となる．Screen インターフェースと KeyEventListener インターフェースを実装することで，画面ごとに AbstractModel クラスが呼び出すことを保障する．5.2 節で後述するが，このクラスは，State パターンの State に該当する [13].

ImageManager クラス

ゲーム内で利用する画像を Hash を使って管理しているクラスである．

SoundManager クラス

ゲーム内で利用するサウンドを Hash を使って管理しているクラスで，サウンドの再生，停止，一時停止，再開などの機能を提供する．

StoreManager クラス

読み込みストリームと書き込みストリームを保持し，外部データ領域に対するデータのアクセスをサポートするクラスである．

HttpManager クラス

通信処理を管理するクラスで，指定した URL にアクセスしてデータをダウンロードする機能を提供する．

Utility クラス

ゲーム開発上頻繁に利用される機能をまとめたクラスで，デバッグ情報の出力のほか，文字列操作や int 型，short 型，byte 型の相互変換などをサポートしている．

Screen インターフェース

このインターフェースを実装したクラスに描画処理を記述する．各画面に 1 つあることを想定している．

KeyEventListener インターフェース

このインターフェースを実装したクラスにキー操作に対する処理を記述する．各画面に 1 つあることを想定している．

5.1.2 動的な設計

図 5.2 は Bridge において，携帯ゲームが起動してから各画面の処理を行うまでの流れを表す．

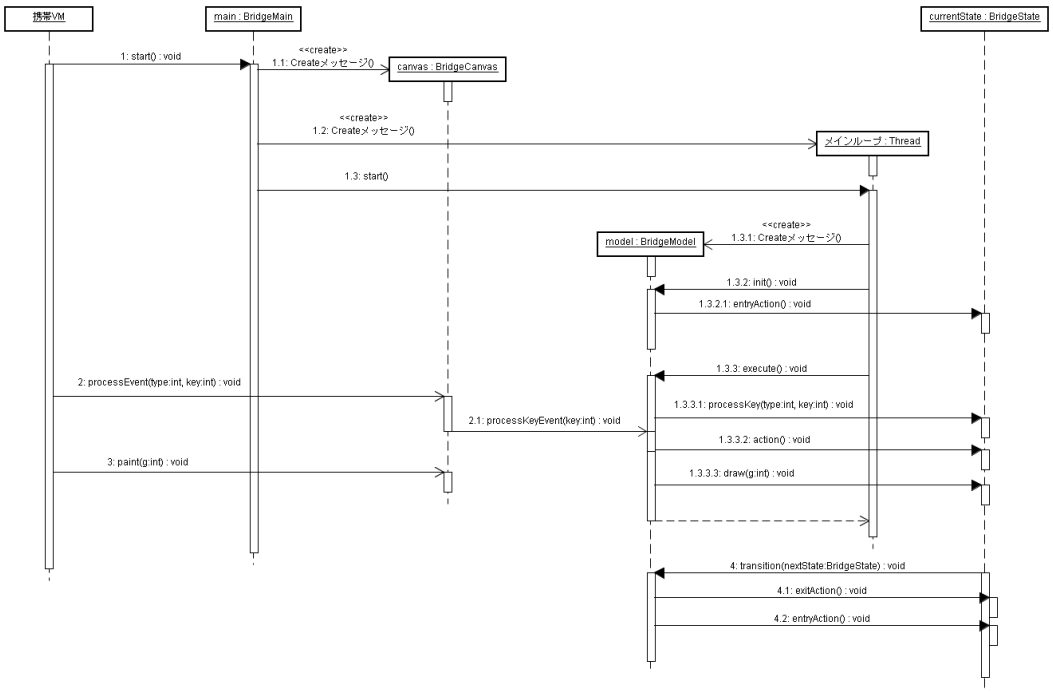


図 5.2: Bridge シーケンス図

1. 携帯ゲームが起動すると com.nttdocomo.ui.IApplication#start メソッドをオーバーライドした AbstractMain#start メソッドを呼び出す．
2. BridgeCanvas クラスのインスタンスを作成し，ゲームが利用するキャンバスに設定する．
3. ゲームループ用のスレッドを作成し，そのスレッドを開始する．
4. BridgeMain#run メソッドを呼び出し，AbstractModel のサブクラスのインスタンス（ゲーム固有のモデル）を作成する．
5. AbstractModel#execute メソッドを呼び出し，現在のキーの状態を取得する．

6. AbstractModel#execute メソッドで，現在の画面である AbstractState クラスのサブクラス（ConcreteState[13]）のインスタンスがもつ processKey メソッド, action メソッド, draw メソッド内の処理を順番に行う.
7. 6 の処理にかかった時間からループ時間の調整を行い，フレームレートを安定させる.
8. 以降は，一定時間毎に 5-7 を行う.

携帯端末がユーザの入力を感知したり，適切なタイミングでイベントを発行した場合は，必要なイベントのみ保存しておく（図 5.2 参照）.

- BridgeCanvas#paint メソッドに対して描画イベントが通知された場合，それは無視する.
- BridgeCanvas#processEvent メソッドに対して低レベル API イベントが通知されると，AbstractModel#processEvent メソッドを介して，直前のイベントのみを保存しておく.

画面遷移とそれに伴う処理の切り替えは次のように行う（図 5.2 参照）.

- 現在の画面で画面遷移のために AbstractModel#transition メソッドを呼ぶと，AbstractModel クラスは現在の画面（AbstractState クラスのサブクラス）のインスタンスを必要に応じてスタックに積む.
- AbstractModel#transition メソッドで前の画面の exitAction メソッドを実行する.
- 新しい画面（AbstractState クラスのサブクラス）のインスタンスを渡し，処理対象の画面がポリモーフィズムにより動的に切り替わり，新しい画面インスタンスがもつ entryAction メソッドを呼び出す.

5.2 State パターンによる画面管理の実現

ゲームは単一，または複数の画面で構成してある．そのため，それぞれの画面内で行う描画やキー操作の処理を管理する必要がある．複数の画面を管理する一番簡単な方法は，画面に対応した定数と現在の画面を表す変数を用意し，現在の画面変数に応じた分岐処理を if 文や switch 文で記述することである.（ソースコード 5.1 参照）.

この方法は実装も簡単でソースコードも直観的に分かりやすいように思える．しかし，各画面で行う処理や画面数が増大した場合に，1 つのクラス（ソースコード 5.1 では Main クラス）にすべての画面の処理が集中するため，著しく保守性が低下する.

また，画面を新しく追加したい場合や，既存の画面を削除したい場合には，画面変数に関わるすべての分岐を見直さなければならないので，拡張しにくい構造である．ソースコード 5.2 は，ソースコード 5.1 に説明画面（HOW_TO_PLAY）を追加する際の修正点をコメント（/*****/）で示したものである．この例では，画面固有の処理が書かれているメソッド 2 つ（processEvent, paint）を修正しなければならない．

ソースコード 5.1: 定数を用いた画面管理の例

```
1  public class Main {
2      public static final int TITLE = 1;
3      public static final int GAME_PLAYING = 2;
4      public static final int RESULT = 3;
5
6      private int gameState = TITLE; // 現在の画面を表す
7
8      public void processEvent(int type, int param) {
9          switch (gameState) {
10             case TITLE: // タイトル画面のキー処理
11                 break;
12
13             case GAME_PLAYING: // プレイ中画面のキー処理
14                 break;
15
16             case RESULT: // リザルト画面のキー処理
17                 break;
18
19             default: // その他の画面のキー処理
20                 break;
21         }
22     }
23
24     public void paint() {
25         switch (gameState) {
26             case TITLE: // タイトル画面の描画処理
27                 break;
28
29             case GAME_PLAYING: // プレイ中画面の描画処理
30                 break;
31
32             case RESULT: // リザルト画面の描画処理
33                 break;
34
35             default: // その他の画面の描画処理
36                 break;
37         }
38     }
39 }
```

ソースコード 5.2: 定数を用いた画面管理における画面追加時の修正点

```
1  public class Main {
2      public static final int TITLE = 1;
3  /*********************************************************/
4      public static final int HOW_TO_PLAY = 2;
5      public static final int GAME_PLAYING = 3;
6      public static final int RESULT = 4;
7  /*********************************************************/
8
9      private int gameState = TITLE; // 現在の画面を表す
10
11     public void processEvent(int type, int param) {
12         switch (gameState) {
13             case TITLE: // タイトル画面のキー処理
14                 break;
15
16 /*********************************************************/
17             case HOW_TO_PLAY: // 説明画面のキー処理
18                 break;
19 /*********************************************************/
20
21             case GAME_PLAYING: // プレイ中画面のキー処理
22                 break;
23
24             case RESULT: // リザルト画面のキー処理
25                 break;
26
27             default: // その他の画面のキー処理
28                 break;
29         }
30     }
31
32     public void paint() {
33         switch (gameState) {
34             case TITLE: // タイトル画面の描画処理
35                 break;
36
37 /*********************************************************/
38             case HOW_TO_PLAY: // 説明画面の描画処理
39                 break;
40 /*********************************************************/
41
42             case GAME_PLAYING: // プレイ中画面の描画処理
43                 break;
44
45             case RESULT: // リザルト画面の描画処理
46                 break;
47
48             default: // その他の画面の描画処理
49                 break;
50         }
51     }
52 }
```

Bridge では各画面を状態オブジェクトとして捉えて，State パターン [13] を用いることでこの問題を解決している．State パターンにより画面に依存する処理を画面クラス内に隠蔽することで，煩雑な switch 文や if 文から解放し，コードの可読性を向上させる．一般的に State パターンは図 5.3 のような構造になっている．Bridge では State パターンを表 5.1 のようにそれぞれのクラスに適用する．

表 5.1: State パターンを適用した Bridge のクラス一覧

State パターン	Bridge
Context クラス	AbstractModel クラス
State クラス	AbstractState クラス
ConcreteState クラス	各画面のクラス

ソースコード 5.3 では，State インターフェースで実装するべきメソッド（processEvent, paint）を定義している．各画面は State インターフェースを実装して，定義しているメソッドに画面固有の処理を記述する．このように実装することで，ソースコード 5.1 では 1 つのクラスに集中していた画面の処理を画面ごとに分散することができ，ポリモーフィズムによって画面を動的に切り替えることが可能である．

さらに，新しい画面を追加するときも既存のソースコードはほとんど変更せずに新しいクラスを追加するだけでよいため，拡張性が高い．ソースコード 5.4 は，ソースコード 5.3 に説明画面（HOW_TO_PLAY）を追加する際の修正点をコメント（/*****/）で示したものである．ソースコード 5.1 の場合とは異なり，既存のソースコードはまったく修正せず，説明画面に対応するクラスを追加するだけで画面を追加することができる．

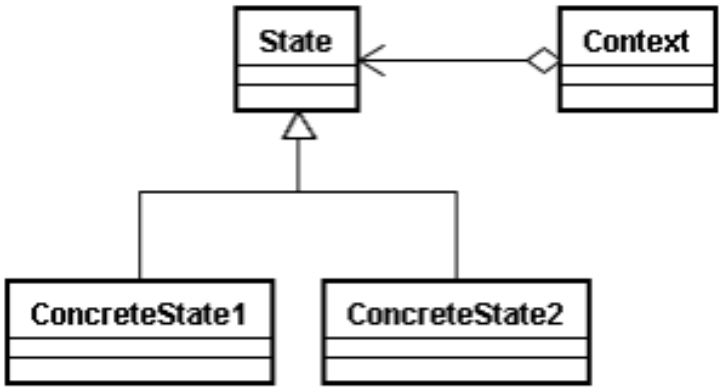


図 5.3: State パターンのクラス図

ソースコード 5.3: State パターンを用いた画面管理の例

```
public class Main {
    private State currentState = new Title();

    public void processEvent(int type, int param) {
        currentState.processKey(type, param);
    }
}

interface State {
    abstract public void processEvent(int type, int param);
    abstract public void paint();
}

class Title implements State {
    public void processEvent(int type, int param) {
        // タイトル画面のキー処理
    }

    public void paint() {
        // タイトル画面の描画処理
    }
}

class GamePlaying implements State {
    public void processEvent(int type, int param) {
        // プレイ中画面のキー処理
    }

    public void paint() {
        // プレイ中画面の描画処理
    }
}

class Result implements State {
    public void processEvent(int type, int param) {
        // リザルト画面のキー処理
    }

    public void paint() {
        // リザルト画面の描画処理
    }
}
```

ソースコード 5.4: State パターンを用いた画面管理における画面追加時の修正点

```
1  public class Main {
2      private State currentState = new Title();
3
4      public void processEvent(int type, int param) {
5          currentState.processKey(type, param);
6      }
7  }
8
9  interface State {
10     abstract public void processEvent(int type, int param);
11     abstract public void paint();
12 }
13
14 class Title implements State {
15     public void processEvent(int type, int param) {
16         // タイトル画面のキー処理
17     }
18
19     public void paint() {
20         // タイトル画面の描画処理
21     }
22 }
23
24 /*************************************************************/
25 class HowToPlay implements State {
26     public void processEvent(int type, int param) {
27         // 説明画面のキー処理
28     }
29
30     public void paint() {
31         // 説明画面の描画処理
32     }
33 }
34 /*************************************************************/
35
36 class GamePlaying implements State {
37     public void processEvent(int type, int param) {
38         // プレイ中画面のキー処理
39     }
40
41     public void paint() {
42         // プレイ中画面の描画処理
43     }
44 }
45
46 class Result implements State {
47     public void processEvent(int type, int param) {
48         // リザルト画面のキー処理
49     }
50
51     public void paint() {
52         // リザルト画面の描画処理
53     }
54 }
```

5.3 Singleton による再利用を用いた画面遷移

Bridge では画面遷移を行う際に遷移先の画面クラスのインスタンスを渡している．その際，毎回画面クラスのインスタンスを生成するのはメモリの無駄遣いである上，画面の過去の情報を残しておけないという問題がある．そのため，各画面クラスを Singleton[13] 化することで，インスタンスを 2 つ以上生成できないようにした上で再利用している（ソースコード 5.5 参照）．なお，Singleton パターンの適用に，static フィールドを用いているのは，コストパフォーマンスを上げるためである [14].

ただし，この方法では各画面のインスタンスがどこからでも参照可能になってしまうため，画面のインスタンスを取得するのは画面遷移時のみ，というルールをプログラマに課す必要がある．また，新しい画面を作成する際に画面クラスを Singleton 化する手間が発生するが，専用の eclipse プラグインを開発し，それを利用することで解決している（5.5 節参照）.

なお，画面の履歴管理にスタックを用いていることは 4.2.2 で説明したが，画面遷移が頻繁に行われた場合でもスタックオーバーフローが起こらないように画面履歴に制限をつけている．また，画面履歴を保存しないで遷移するメソッド（transition）と，画面履歴を保存して遷移するメソッド（transitionAndKeep）を用意し，必要に応じて使い分けられるようにしている．

ソースコード 5.5: Singleton 化したタイトル画面クラス

```
1  public class TitleState extends GameState {
2      /**
3       * この画面唯一のインスタンス
4       */
5      private static final TitleState instance = new TitleState();
6
7      /**
8       * Singleton を実現するために，コンストラクタは private
9       */
10     private TitleState() {
11     }
12
13     /**
14      * この画面の唯一のインスタンスを取得する
15      */
16     public static TitleState getInstance() {
17         return TitleState.instance;
18     }
19 }
```

5.4 ライブラリの共通化による移植作業の自動化

5.4.1 ラッパークラスの作成

図 4.2 で述べているように，Bridge では i アプリ，S!アプリ，オープンアプリ，Will-COM アプリ（表 2.2 参照）に対応することで，移植作業の自動化を行う．ただし，イメージ描画の一部とサウンドの演奏において S!アプリ独自の拡張 API である MEXA（2.2 参照）を利用しており，MIDP のみしか利用できないオープンアプリ，WILLCOM アプリには完全に対応できていない．

異なるプラットフォーム仕様をもつアプリの移植作業を自動化する方法として，MIDP 上のライブラリのラッパーを作成して，DoJa プロファイルと同じ API を提供することで，移植作業を行うようにしている．図 5.4 は，MIDP 上のライブラリと Bridge とラッパーの位置づけを表したクラス図である．それぞれのラッパークラスは以下のような機能を提供する．

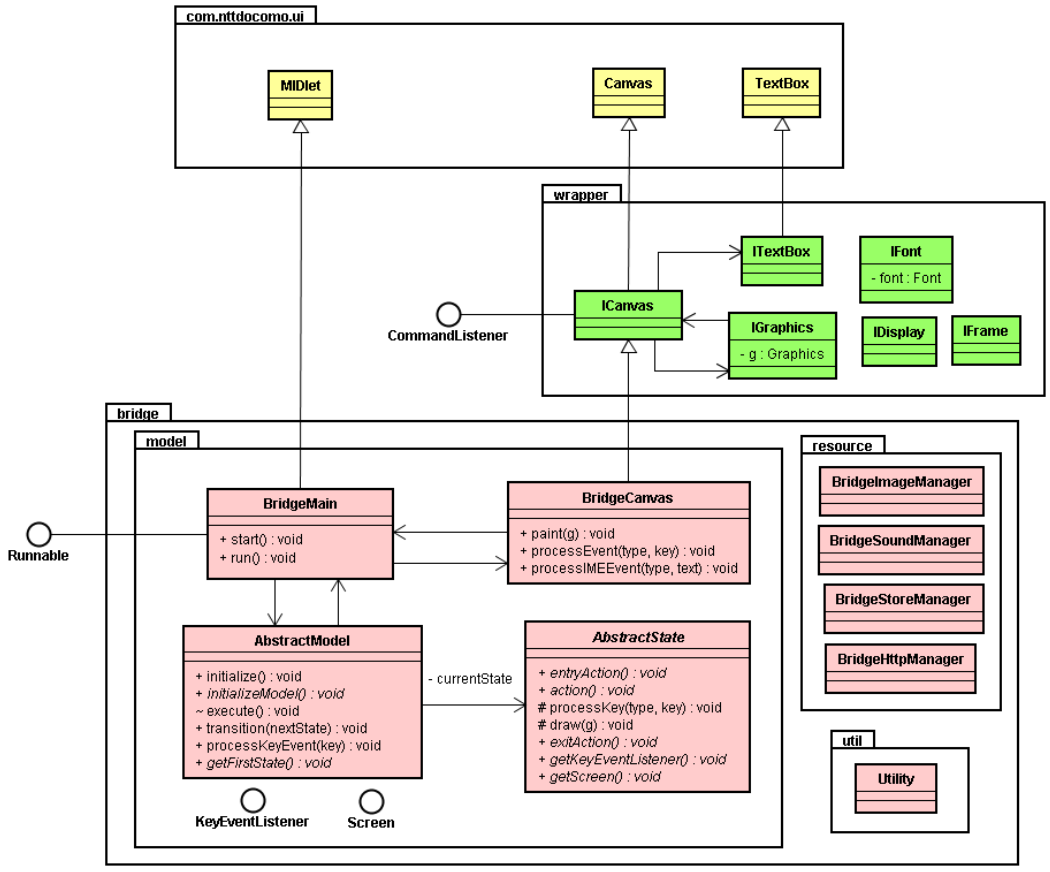


図 5.4: ラッパーによるライブラリの共通化を行った Bridge クラス図

ICanvas クラス

javax.microedition.lcdui.Canvas クラスを継承し，javax.microedition.lcdui.CommandListener インターフェースを実装している．IGraphics，ITextBox クラスのインスタンスを保持し，次の機能を提供する．

- 取得したキーイベント（ソフトキー含む）の保存，通知.
- オフスクリーンから実画面への描画.
- ソフトキーラベルの貼り替え.
- IME の起動，IME 終了の通知.

IGraphics クラス

javax.microedition.lcdui.Graphics クラスのインスタンスを保持しており，描画に関する次の機能を提供する．

- 色の設定，取得.
- 線，四角形，塗りつぶし四角形，円弧，塗りつぶし円の描画.
- 文字の描画，フォントの設定.
- イメージの描画，反転，切り抜き，変形.
- クリッピング領域の設定，解除.
- 描画原点座標の変更.

IFont クラス

javax.microedition.lcdui.Font インスタンスを保持しており，フォントに関する次の機能を提供する．

- フォント情報（フォントの高さ，アセント，ディセント）取得.
- 指定した文字列の幅の計算.
- 指定した文字列の改行位置の計算.

ITextBox クラス

javax.microedition.lcdui.TextBox インスタンスを保持しており，DoJa プロファイル同様の IME（ユーザ入力の受け取り）のインターフェースと処理を提供する．IME の起動やユーザ入力終了時の処理は，ICanvas で行う．

IDisplay クラス

DoJa プロファイルのキーコード，キーイベントを MIDP で利用できる形で再定義している．

IFrame クラス

DoJa プロファイルのソフトキー情報を MIDP で利用できる形で再定義している．

5.4.2 共通化が実現した機能

ラッパークラスと Bridge のライブラリを組み合わせることで，DoJa プロファイルと MIDP において以下の共通化を行った．

起動からゲームループまで

DoJa プロファイルでは，com.nttdocomo.ui.IApplication クラスの start メソッドを，MIDP では javax.microedition.midlet.MIDlet クラスの startApp メソッドをアプリの開始時に呼ぶ．Bridge では，これらのメソッドを呼び出してからゲームループ用のスレッドを作成し，現在の画面のキー処理，計算処理，描画処理を呼び出すという構造を共通化している．

描画機能

IGraphics が提供する描画機能について，MIDP での開発でも DoJa プロファイルと同じ API が利用できる．

キー，ソフトキーの処理

ICanvas，IDisplay，IFrame が提供するキーコードの取得，キーイベントの通知，ソフトキーラベルの変更について，MIDP での開発でも DoJa プロファイルと同じ API が利用できる．

サウンドの演奏

サウンドを演奏には，DoJa プロファイルでは com.nttdocomo.ui.AudioPresenter クラス，MIDP では javax.microedition.media.Player インターフェースを利用する．Bridge では，SoundManager クラスがこれらの差異を吸収し，演奏するために必要な API を提供する．ただし，MIDP ではサウンドの一時停止や再開をサポートしておらず，サウンドの読み込み速度が遅いという問題がある．そこで，S!アプリの独自 API である MEXA が定義している com.jblend.media.smaf.phrase.PhrasePlayer クラスを利用することでこの問題に対応している．しかし，S!アプリの独自 API を利用しているため，サウンドの演奏はオープンアプリ，WILLCOM アプリで動作させることはできず，改善の余地を残している．

外部データへのアクセス

DoJa プロファイルではスクラッチパッド，MIDP ではレコードストアというそれぞれ異なる構造をもった外部記憶領域にデータを保存している．Bridge では，StoreManager クラスが外部記憶領域に保存してあるデータをストリーム（java.io.DataInputStream, java.io.DataOutputStream）として扱い，そのストリームに対して読み込みや書き込みを行う API を提供することで，外部記憶領域の構造の違いを吸収している．

ユーザ入力の受け取り

DoJa プロファイルでユーザから入力を受け取る方法は次のとおりである．

1. com.nttdocomo.ui.Canvas クラスの imeOn メソッドにより IME を起動する．
2. IME においてユーザが入力を行う．
3. 入力が終わると，com.nttdocomo.ui.Canvas クラスの processIMEEvent メソッドを呼び出して，ユーザの入力したデータを渡す．

Bridge では，ICanvas クラスと ITextBox クラスを利用して MIDP 上で DoJa プロファイルの IME 機能を再現している．

通信機能

Bridge では，HttpManager クラスが javax.microedition.io.HttpConnection インターフェースを利用し，指定された URL に対して HTTP 通信を行ってデータを取得する部分を，DoJa プロファイルと MIDP で共通化している．

5.5 Bridge eclipse プラグイン

eclipse は，eclipse プロジェクト [11] が開発した統合開発環境（IDE）であり，Java 開発で利用する IDE のデファクトスタンダードとなりつつある [15]．eclipse のもつリファクタリング機能やコードアシスト機能はもとより，各キャリアが公式に配布している i アプリ開発用，S!アプリ開発用のプラグイン [1][16] を利用することで，エミュレータと eclipse のデバッガを連携させることが可能となる．したがって，eclipse を利用することで，携帯アプリの開発効率は格段に向上する．そこで，eclipse で携帯ゲームを開発することを前提として，Bridge の機能を eclipse で利用できるように，Bridge プラグインの開発を行っている（図 5.5 参照）．なお，プラグインの利用方法については付録 A に含めておく．

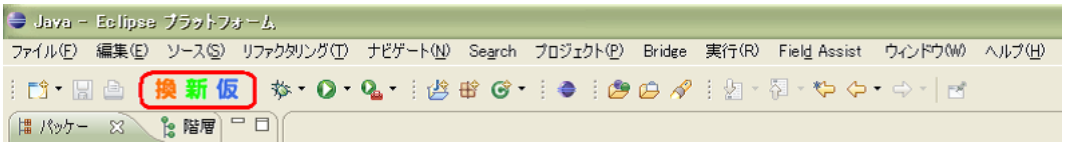


図 5.5: eclipse 上の Bridge プラグイン

i アプリスケルトンコード生成機能

i アプリスケルトンコード作成ウィザード（図 5.6 参照）でパッケージを指定すると，そのパッケージ以下に，i アプリの起動からアプリの制御までを記述したスケルトンコード，および必要なパッケージ構造と設定ファイルを自動生成する．図 5.7 は生成したスケルトンコードと Bridge の関係性を表すクラス図である．スケルトンコードはゲームのモデルを表すクラス（ソースコード 5.6 参照），各画面の元となるクラス（ソースコード 5.7 参照），サンプルとして最初に遷移するタイトル画面クラス（ソースコード 5.8 参照）の 3 つからなる．プログラマはこれらのスケルトンコードに沿って指定したところに必要な処理を記述するだけで，簡単に携帯ゲームを作成することができる．

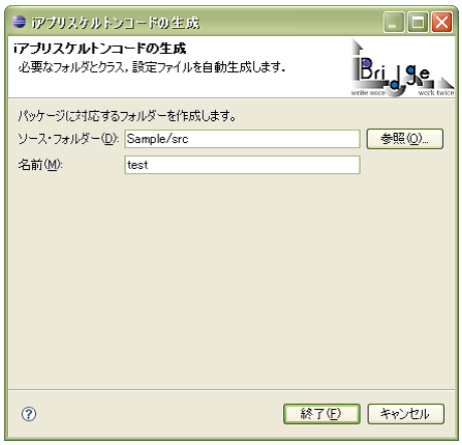


図 5.6: Bridge プラグインにおける新規スケルトンコード生成ウィザード

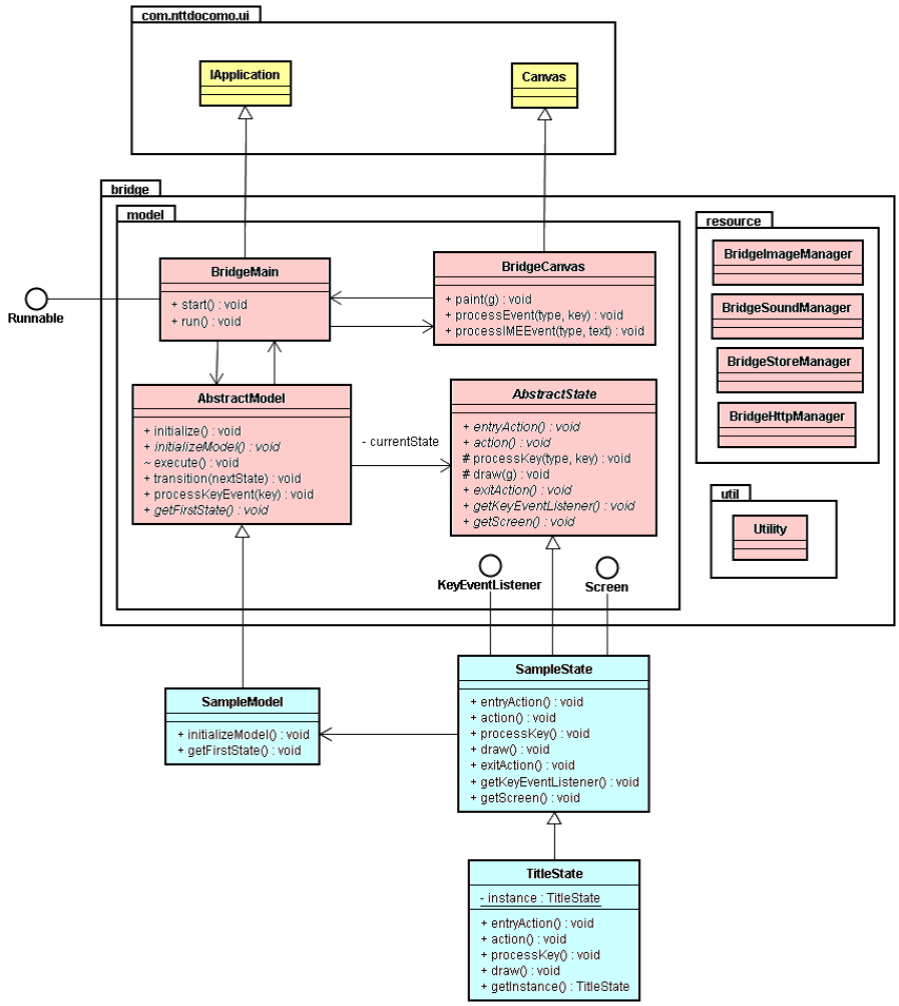


図 5.7: スケルトンコードのクラス図

ソースコード 5.6: 生成されたスケルトンコード（ゲームモデル）

```
1  package test.model;
2
3  import bridge.model.AbstractState;
4  import bridge.model.AbstractModel;
5  import test.state.TitleState;
6
7  /**
8   * ゲームのモデルを表すクラスです.
9   *
10  * @author BridgeFrameWork
11  * @version 1.0 Sat Nov 29 16:30:46 JST 2008
12  */
13 public class SampleModel extends AbstractModel {
14     protected void initializeModel() {
15         // TODO ここで各種リソース(画像・サウンド等)を読み込んで下さい
16     }
17
18     /**
19      * 最初に遷移する画面のインスタンスを返します.
20      */
21     protected AbstractState getFirstState() {
22         return TitleState.getInstance();
23     }
24 }
```

ソースコード 5.7: 生成されたスケルトンコード（抽象画面）

```
1  package test.model;
2
3  import test.model.SampleModel;
4
5  import bridge.model.AbstractModel;
6  import bridge.model.AbstractState;
7  import bridge.model.KeyEventListener;
8  import bridge.model.Screen;
9
10 import com.nttdocomo.ui.Graphics;
11
12 /**
13  * ゲームの状態を表すクラスです.
14  * ゲームにおける各状態はこのクラスを継承して下さい.
15  *
16  * @author BridgeFrameWork
17  * @version 1.0 Sat Nov 29 16:30:46 JST 2008
18  */
19 public class SampleState extends AbstractState implements
        KeyEventListener, Screen {
20     protected SampleModel model;
21
22     protected void action() throws Exception {
23     }
24
25     protected void entryAction() throws Exception {
26     }
27
```

```
28      public void processKey() throws Exception {
29      }
30
31      public void draw(Graphics g) throws Exception {
32      }
33
34      protected void exitAction() throws Exception {
35      }
36
37      protected KeyEventListener getKeyEventListener() {
38          return this;
39      }
40
41      protected Screen getScreen() {
42          return this;
43      }
44
45      protected void setModel(AbstractModel model) {
46          this.model = (SampleModel) model;
47      }
48  }
```

ソースコード 5.8: 生成されたスケルトンコード（タイトル画面）

```
1   package test.state;
2
3   import test.state.TitleState;
4   import test.model.SampleState;
5
6   import com.nttdocomo.ui.Graphics;
7
8   /**
9    * TitleState.java
10   *
11   * @author yourname
12   * @version 1.0 Sat Nov 29 16:30:46 JST 2008
13   */
14  public class TitleState extends SampleState {
15      private static final TitleState instance = new TitleState();
16
17      /**
18       * このコンストラクタはSingleton実現のために定義されているので,
19       * 初期化処理はこの中で行わず,entryAction() 内に記述して下さい.
20       */
21      private TitleState() {
22      }
23
24      /**
25       * この画面に遷移したときの処理を行います
26       */
27      protected void entryAction() throws Exception {
28      }
29
30      /**
31       * 毎ループごとの処理を行います
32       */
33      protected void action() throws Exception {
```

```
34      }
35
36      /**
37       * キーが押された場合の処理を行います
38       */
39      public void processKey() throws Exception {
40          // ソフトキー 1が押された場合
41          if (model.isKeyPressed(KEY_SOFT1)) {
42          }
43
44          // ソフトキー 2が押された場合
45          if (model.isKeyPressed(KEY_SOFT2)) {
46          }
47
48          // 決定キーが押された場合
49          if (model.isKeyPressed(KEY_SELECT)) {
50          }
51
52          // 上キーが押された場合
53          if (model.isKeyPressed(KEY_UP)) {
54          }
55
56          // 下キーが押された場合
57          if (model.isKeyPressed(KEY_DOWN)) {
58          }
59
60          // 左キーが押された場合
61          if (model.isKeyPressed(KEY_LEFT)) {
62          }
63
64          // 右キーが押された場合
65          if (model.isKeyPressed(KEY_RIGHT)) {
66          }
67      }
68
69      /**
70       * 描画の処理を行います
71       */
72      public void draw(Graphics g) throws Exception {
73          g.drawString("HELLO, BRIDGE WORLD!", 50, 100);
74      }
75
76      /**
77       * 他の画面に遷移する直前の処理を行います
78       */
79      protected void exitAction() throws Exception {
80      }
81
82      /**
83       * この画面の唯一のインスタンスを返します
84       */
85      public static TitleState getInstance() {
86          return TitleState.instance;
87      }
88  }
```

新規画面クラス作成機能

新規画面クラス作成ウィザード（図 5.8 参照）で画面の名前を指定すると，画面クラス（ソースコード 5.8 参照）を作成することができる．

図 5.8: Bridge プラグインにおける新規画面クラス作成ウィザード

i アプリから S!アプリに変換する機能

S!アプリ変換ウィザード（図 5.8 参照）で指定した i アプリプロジェクト（DoJa プロジェクト）を S!アプリプロジェクト（MEXA プロジェクト）に変換することができる．具体的には，i アプリのソースコードと S!アプリライブラリの間に 5.4.1 で解説したラッパークラスを入れることで，両者の異なる API の差を吸収する．そこで，吸収できなかったインポート文やクラス名を，変換辞書ファイル（5.10 参照）に従って変換する．変換辞書ファイルはコロン 2 つ（::）を区切り文字として，変換前の文と変換後の文を記述できる．

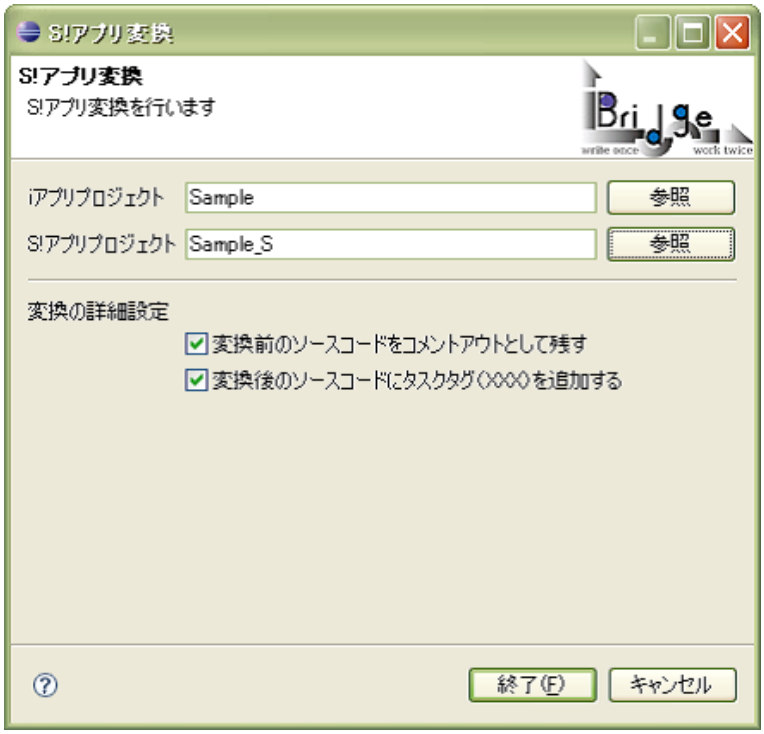


図 5.9: Bridge プラグインにおける S!アプリ変換ウィザード

```
%% 置換前::置換後
com.nttdocomo.ui.Graphics::bridge.wrapper.IGraphics
com.nttdocomo.ui.Canvas::bridge.wrapper.ICanvas
com.nttdocomo.ui.Font::bridge.wrapper.IFont
com.nttdocomo.ui.TextBox::bridge.wrapper.ITextBox
com.nttdocomo.ui.Image::javax.microedition.lcdui.Image
com.nttdocomo.ui.MediaResource::com.jblend.media.smaf.phrase.Phrase
MediaSound::Phrase
Graphics::IGraphics
Canvas::ICanvas
Display::IDisplay
Font::IFont
Frame::IFrame
TextBox::ITextBox
BGM_PORT::BGM_MASTER_PORT
```

図 5.10: 変換辞書ファイルの例

第6章　Bridgeの評価と考察

本章では，既存ゲームへの適用と試用実験による Bridge の定量的・定性的な評価について述べる．

6.1 節では，既存の携帯ゲームであるリバーシ，7 並べ，クロンダイクと既存の PC 向けゲームであるピクロスを，Bridge を用いて作り直して，ソフトウェアメトリクスに従って評価する．ソフトウェアメトリクスの定義は「ソフトウェア開発で成果物の何かを測定し何かのために使うこと [17]」である．ここでは，Bridge を用いて作り直したゲームのソースコードを成果物と考え，それを測定することで Bridge の有用性に対する定量的な評価を行う．測定には，オブジェクト指向を用いたソフトウェアのソースコードを評価するために用いられる CK メトリクス [18] を中心に規模，複雑度，凝集度を評価するためのメトリクスを用いる [19]．なお，メトリクスの計測ツールとして，eclipse のプラグインである Eclipse Metrics Plugin（Frank Sauer）[20] を利用している．

6.2 節では，Bridge の試用実験について述べる．仕様書に沿った簡単なシューティングゲームを実装してもらうことで，Bridge の利用によって携帯ゲームアプリの開発効率がどの程度変化するかということを評価する．実験では，仕様書に沿った各機能を実装するまでの所要時間を計測し，実装後のソースコードを分析する．また，実験終了後に開発に対する感想を自由に記述してもらう．

6.1　メトリクスによる既存ゲームの評価

6.1.1　Bridge を適用したゲームの仕様

リバーシ

リバーシとはオセロによく似たボードゲームである．プレイヤーは互いに盤面へ石を打ち，相手の石を挟んで自分の色の石に返すことで，盤面上の石を取り合う．図 6.1 は既存の携帯ゲームであるリバーシゲームの画面遷移図である．ゲームを起動すると，イメージやサウンドデータを読み込むロード中画面に遷移した後，タイトル画面へ遷移する．タイトル画面からはゲームの設定（サウンドの音量や難易度）を変更できる設定画面と，ゲーム開始の準備画面であるキャラクター選択画面へ遷移することがで

きる．ゲーム中は，プレーヤーとコンピュータ（以下 CPU と書く）が交互に石を置いていく．双方置ける石が無くなるか，プレーヤーが降参するとゲームが終了し，勝敗決定画面を経てゲーム終了画面へ遷移する．

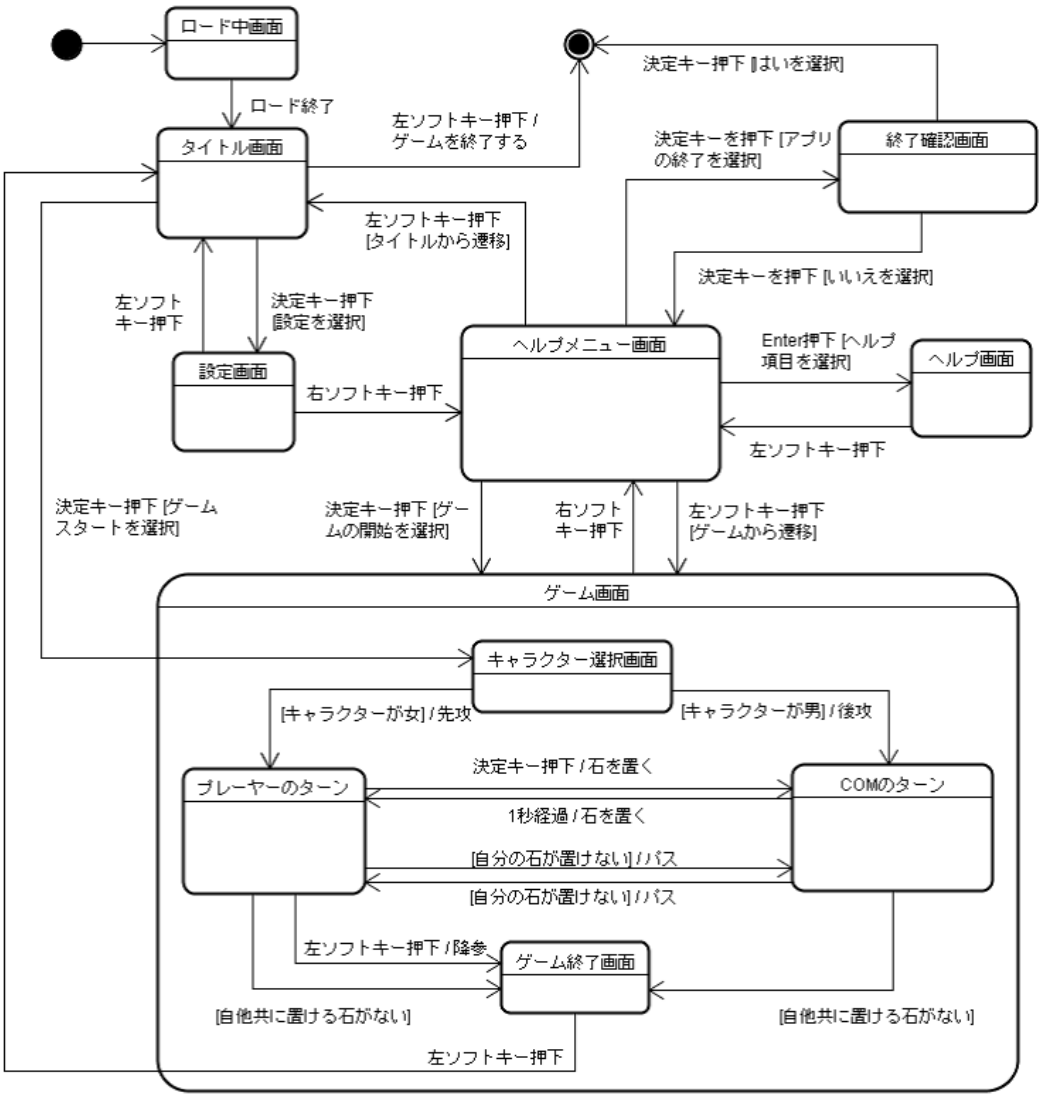


図 6.1: リバーシ画面遷移図

7 並べ

7 並べとは，ルールに従って場に手札を出し，早く手札をなくすことを競うトランプゲームである．図 6.2 は既存の携帯ゲームである 7 並べゲームの画面遷移図である．ゲーム中以外の画面構成や画面遷移は，リバーシとほぼ同様である（図 6.1 参照）．

ゲーム中は，手番決定画面で決まる手番に応じてプレーヤーと複数の CPU が順番にカードを置いていく．カードが置けない場合は一定回数以内でパスができる．全員が上がるか誰かがパスできなくなった時点で，その時のカードの残り具合から勝敗が決まり，順位決定画面に遷移する．プレーヤーが最下位でなければスコア集計画面を経て手番決定画面に戻り次のゲームが始まる．プレーヤーが最下位の場合はゲームオーバーとなり，タイトル画面へ遷移する．

クロンダイク

クロンダイクとは，トランプを用いた代表的な一人遊びゲーム（ソリティア）である．階段状に並べられたジョーカーを除く一組のトランプ 52 枚を，決められたルールに基づき移動していき，すべてを指定された場所に移動することができれば上がりである．図 6.3 は既存の携帯ゲームであるクロンダイクゲームの画面遷移図である．リバーシや 7 並べに比べるとかなり簡素な画面遷移だが，これはクロンダイクが対戦型ゲームではなく一人で行うタイプのゲームのため，必然的にゲームプレイ中の画面遷移が少なくなるからである．なお，チュートリアル画面は厳密には別画面だが，実際のゲーム画面とほぼ同等の遷移を行うため，画面遷移図上では同じ画面として扱っている．

ピクロス

ピクロスとはイラストロジックとも呼ばれるペンシルパズルである．縦と横の数字をヒントに塗り潰すマス目を割り出し，そのとおりに塗り潰していくと，最終的に絵が浮かび上がる．図 6.4 は既存の PC 向けゲームであるピクロスゲームの画面遷移図である．ゲームが起動すると，タイトル画面へ遷移し，ゲームの開始，ステージの作成，ゲームの説明画面を選択することができる．ゲーム開始後はステージ選択画面に遷移し，そこで選択したステージの難易度に応じてゲームが開始する．ステージ作成中とゲーム中画面からはメニューに遷移することができ，タイトルに戻ることができる．また，ステージ作成中であればデータの保存ができる．ゲーム中は，ヒントをもとにマスを開いていくが，開けないマスを開こうとしてしまった場合はおてつきとなり，制限時間が減る．制限時間以内に絵を完成させるとゲームクリアとなり，制限時間がなくなるとゲームオーバーとなる．

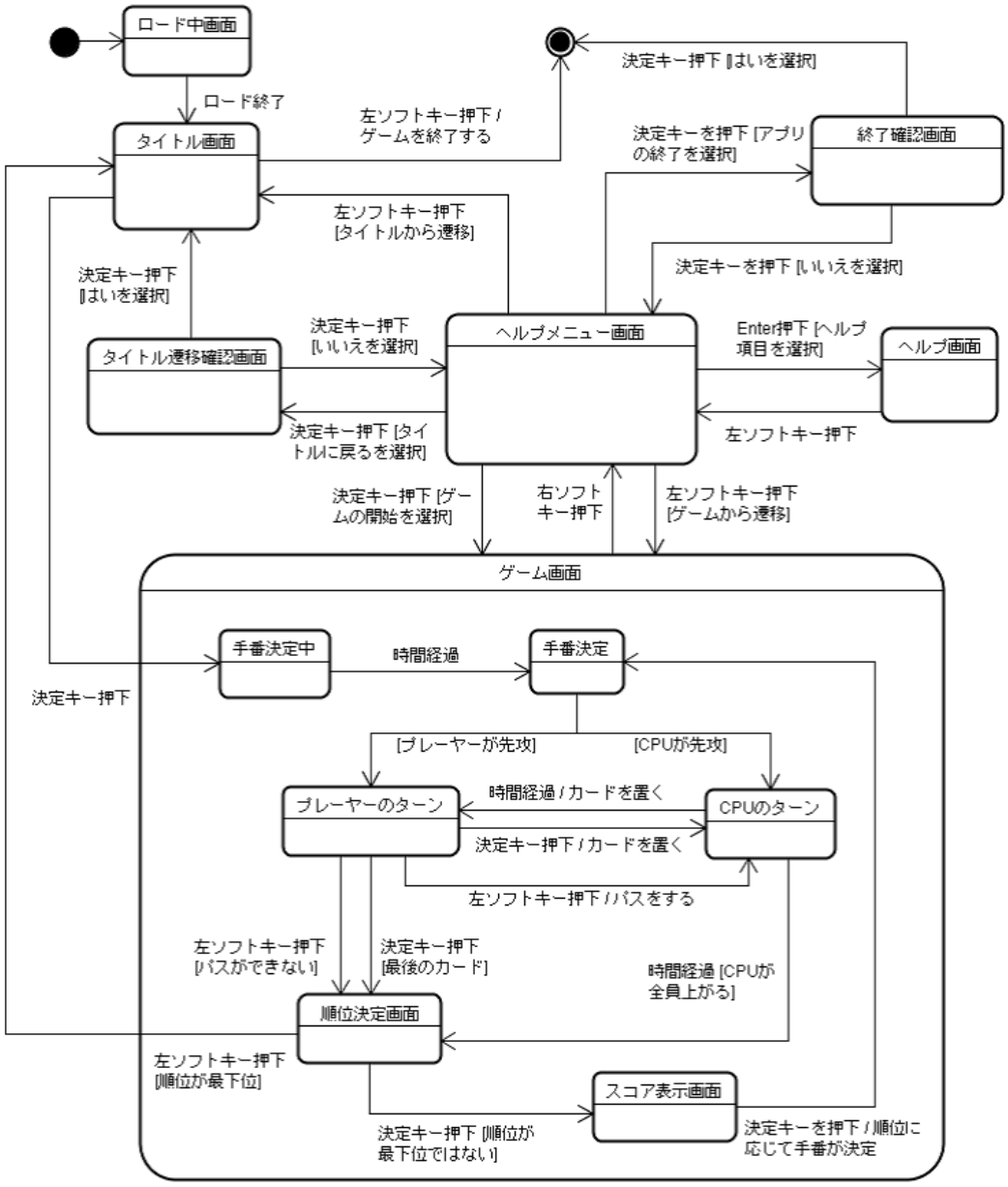


図 6.2: 7 並べ画面遷移図

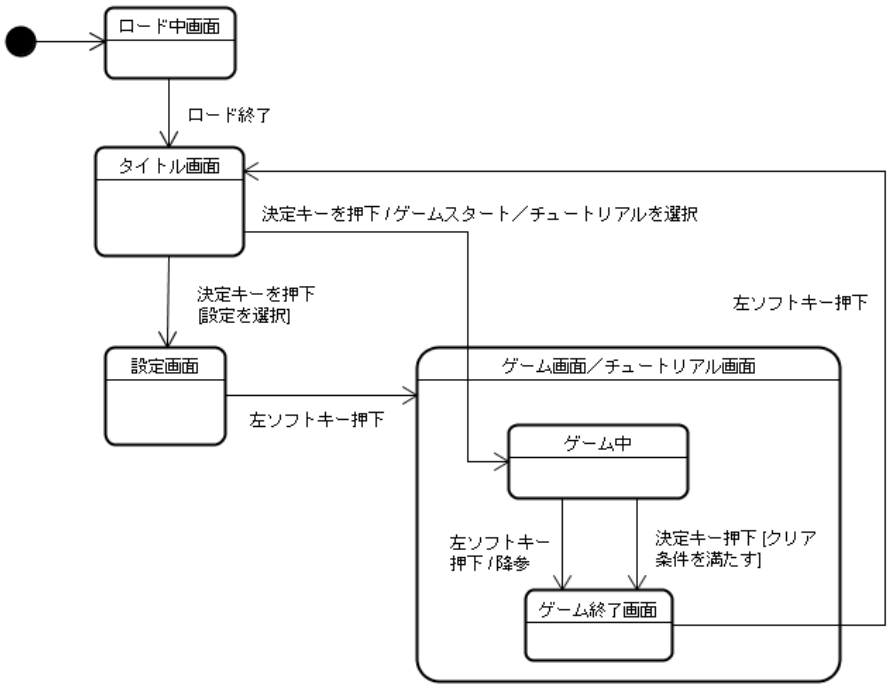


図 6.3: クロンダイク画面遷移図

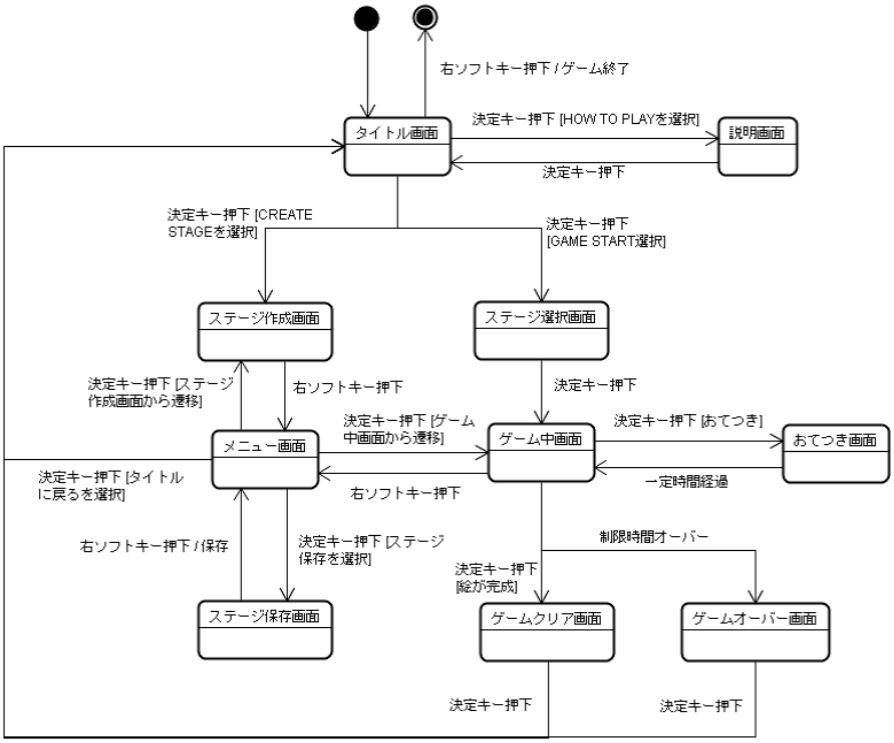


図 6.4: ピクロス画面遷移図

6.1.2 規模から見た評価

表 6.1 は，Bridge を用いて作り直した各ゲームのソースコードに対して，規模の側面からメトリクスの測定を行った結果をまとめたものである．7 並べが他のゲームに比べて TLOC（コード行数）と NOC（クラス数）がやや多めになっているが，他の 3 つゲームはほぼ同様の規模である．

まず，NOM（メソッドの数）に着目する．再利用性が上がるという視点から見れば，メソッドの数は多くなることが望ましい．しかし，その数が多すぎるとクラスは複雑になり拡張性が低下する．また，多くの責任が 1 つのクラスに集中してしまうため，各クラスにおける NOM は 20 以下であることが推奨されている [21]．各ゲームにおける NOM の平均を見るといずれもこの値を下回っている．しかし，リバーシは標準偏差が大きく，メソッドの数にバラツキがあることが分かる．さらに NOM の最大値が 68 と大きくなっている．これは，リバーシにおけるすべてのデータを，ゲームモデルを表すクラスで定義しており，約 9 割にあたるメソッド（68 個中 62 個）がアクセサメソッドであることが原因となっている．同様の問題はピクロスでも発生しており，Bridge の新たな検討課題としてゲームモデルの抽象化を進めていく必要があることが分かった．

次に，MLOC（コメント・空行を除いたメソッドの行数）に着目する．メソッドの行数が大きくなりすぎるということは，1 つのメソッドが複数の機能を有している可能性が高い．このようなメソッドはテストを行うことが難しく，再利用性も低いため，MLOC は 24 行以下にすることが推奨されている [21]．リバーシについては，全メソッドに対するアクセサメソッド（大抵は 1 行）の割合が多いため，他のゲームに比べて標準偏差が大きくなっている．それ以外のゲームでは MLOC の平均が 24 行を大きく下回っていることが分かる．このことから，Bridge を適用したゲームは各々のメソッドの行数が少なく，再利用性や保守の効率が高くなっていると言える．

表 6.1: Brige を適用した携帯ゲームのメトリクス（規模）測定結果

メトリクス		リバーシ	7 並べ	クロンダイク	ピクロス
TLOC[1]		2120	3537	2139	2648
NOC[2]		17	44	32	27
NOM[3]	総数	176	331	207	260
	平均	10.35	7.52	6.47	9.63
	標準偏差	14.87	6.12	5.09	8.98
	最大	68	26	27	44
MLOC[4]	総数	1442	2130	1380	1855
	平均	8.1	6.28	6.54	6.40
	標準偏差	17.62	8.65	10.31	8.30
	最大	114	85	61	53

[1]TLOC(Total Lines of Code)... 全コード行数（コメント，空行は含まない）
[2]NOC(Number of Classes)... クラス数
[3]NOM(Number of Methods)... 各クラスのメソッド数
[4]MLOC(Method Lines of Code)... メソッドのコード行数（コメント，空行は含まない）

6.1.3 複雑度から見た評価

表 6.2 は，Bridge を用いて作り直した各ゲームのソースコードに対して，複雑度の側面からメトリクスの測定を行った結果をまとめたものである．

まず，MMC に着目する．MMC は Thomas McCabe が開発したソフトウェア測定尺度の一種である [22]．ソースコードにおける線形的に独立した経路の数を直接数えて，プログラムの複雑度を測定する方法である．「MMC が 5 以下だとプログラムは単純であり理解しやすく，10 以下であれば，それほど理解困難とは思われない．MMC が 20 以上で複雑さは高いとみなすことができ，50 以上になるとテスト不可能となり，実用に供せなくなる [23]」と言われている．各ゲームの MMC を見ると平均は 5 を下回っており，標準偏差によるバラツキを考慮しても，ほとんどは 10 以下に収まる．

次に，WMC に着目する．WMC はクラスにおけるメソッドの複雑度の総和であり，この複雑度は MMC によって計算する．MMC が 1 である場合は，WMC はクラス内のメソッド数と等しくなる．WMC が大きくなると複雑度が上がり，メンテナンスのコストがかかるため，30 以下であることが推奨されている [19]．各携帯ゲームの WMC を見ると平均は 30 を下回っているが，いくつかのクラスでは 30 を超えている．WMC が 30 を超えているクラスはすべてゲームモデルに関連するクラスであり，コンピュータの思考ルーチンやゲームのルールに関するロジックが記述されている．

最後に，NBD に着目する．NBD はメソッドのネスト構造の深さを表しており，一般的にネスト構造が深くなるとソースコードの可読性が下がり，保守することが難し

くなる．各ゲームアプリの NBD を見ると平均 1.5 程度であり，これはネスト構造がほとんどない状態である．また，どのゲームにおいても，ネスト構造の最大値は WMC 同様にゲーム固有のロジックを記述している部分に現れている．

以上の結果より，Bridge を適用したゲームは，ゲーム特有のロジック部分を除けば，複雑性が低く，保守しやすい構造になっていると言える．

表 6.2: Brige を適用した携帯ゲームのメトリクス（複雑度）測定結果

メトリクス		リバーシ	7 並べ	クロンダイク	ピクロス
MCC[5]	平均	2.71	1.98	2.91	2.09
	標準偏差	4.98	2.38	2.58	2.42
	最大	37	30	16	22
WMC[6]	総数	483	670	463	606
	平均	28.41	15.23	14.47	22.44
	標準偏差	33.89	14.09	13.04	21.92
	最大	134	66	59	105
NBD[7]	平均	1.47	1.51	1.51	1.53
	標準偏差	1.00	0.90	0.94	0.94
	最大	7	5	6	5

[5]MCC(McCabe's Cyclomatic Complexity)...線形的に独立した経路の数
[6]WNC(Weighted methods per Class)...メソッドの複雑度の総和
[7]NBD(Nested Block Depth)...メソッド中の最大ネスト数

6.1.4 凝集度から見た評価

表 6.3 は，Bridge を用いて作り直した各ゲームのソースコードに対して，凝集度の側面からメトリクスの測定を行った結果をまとめたものである．

凝集度とは，モジュール内のソースコードが特定の機能を提供する際に，どれだけ協調しているかを表す度合いである．あるクラスの凝集度を測定するには，そのクラス内のメソッドがどれだけクラス内の変数との関連が強いかを調べる．LCOM（凝集度の欠如）が 0 に近づくほど凝集度が高く，まとまりのあるよい設計である．逆に，LCOM が 1 に近づくほど凝集度が低く，メソッド間で変数を共有している部分が多いことを表し，メンテナンス性が低い [19]．ただし，クラスの特性によっては凝集度が低いことが必ずしも悪いとはいい切れない場合もある．

各ゲームの LCOM を見ると，平均は 0.5 を下回っており，凝集度は高い．このことから，Bridge を適用したゲームは，凝集度が高く，関連性の高い機能が局所化されているよい設計であると言える．

表 6.3: Brige を適用した携帯ゲームのメトリクス（凝集度）測定結果

メトリクス		リバーシ	7 並べ	クロンダイク	ピクロス
LCOM[8]	平均	0.20	0.26	0.16	0.48
	標準偏差	0.36	0.36	0.30	0.38
	最大	1.00	0.98	0.87	1.00

[8]LCOM(Lack of Cohesion of Methods)...クラス中のメソッドの凝集性欠如の度合い

6.2 試用実験による評価

6.2.1 実験の内容

本項では試用実験における被験者と，実験時の注意，実験手順について述べる．

被験者の情報

被験者は 9 人で，大学生 6 名，大学院生 2 名，社会人 1 名である．いずれの被験者も授業，アルバイト，業務を通じて半年以上 Java を学習しており，Java を用いたゲームの開発を経験したことがある．また，9 名のうち 2 名は携帯ゲームを開発した経験がある．

実験時の注意

この実験は，Bridge を利用することでゲームの開発効率が上がることを実証することが目的である．そのため，同じ仕様書に従った携帯ゲームを Bridge を利用する場合と利用しない場合で 2 度開発してもらう．しかし，先の開発で実装したゲームロジック（アルゴリズムやデータ設計など）を流用することで，後の開発に要する時間が短くなるという結果が出てしまうことが想定できる．

そこで，この問題を回避するために，実験では必要なゲーム特有のロジックや画像などのリソースデータをすべての被験者に提供する．さらに，被験者を 2 つのグループに分けて，片方のグループには Bridge を利用しない開発を先に行なってもらい，もう一方のグループには Bridge を利用した開発を先に行なってもらう．こうすることで，後者のグループで Bridge を利用した場合に開発時間が短縮すれば，時間の短縮はゲームロジックの流用とは無関係であることが言える．

実験の手順

実験では，まず，それぞれの被験者を 2 つのグループ（先に Bridge を利用しない開発を行う方をグループ 1，もう一方をグループ 2 とする）に分け，双方のグループに同じ実験仕様書（付録 B 参照）と参考資料 [24]，Bridge マニュアル（付録 A 参照）を配布した．その上で，仕様書に従って実装を行ってもらい，表 6.4 で示す段階を達成するまでにかかった時間をそれぞれ測定した．

なお，実装時間が 3 時間を超えた場合は，それ以上時間を使っても実装が終わらない可能性があるため実装を打ち切り，実装不可とした．最後に，Bridge を利用した場合と利用しなかった場合についての感想を記述してもらった（付録 C 参照）．

表 6.4: 仕様の実装段階と内容

実装段階	実装内容
第 1 段階	タイトル画面が表示できる
第 2 段階	自機の描画・操作ができる
第 3 段階	敵機の描画・動作と，クリア画面ができる
第 4 段階	敵機との衝突判定が実装でき，メニュー画面を除いたゲームが作れる
第 5 段階	仕様書通りのゲームが作れる

6.2.2　開発効率に対する評価

実装時間から見た分析

表 6.5 は被験者のプログラミング経験と，試用実験における実装にかかった時間を表したものである．表 6.6 と表 6.7 は，それぞれ Bridge を利用しなかった場合と利用した場合で，各実装段階（表 6.4 参照）に達するまでにかかった実装時間の詳細を表したものである．

まず，この結果から全体としては Bridge を利用しない場合に比べ，Bridge を利用した場合の実装時間が平均 30 分程度短縮されていることが分かる．実際には，Bridge を利用しない場合に 2 名が時間切れによる実装不可となっており，これを考慮するとこの差はさらに大きくなると思われる．

次に，各段階ごとの開発効率の差に着目する．第 1 段階，第 2 段階では両グループとも Bridge を利用した時のほうが開発時間が短くなっている．第 1 段階，第 2 段階はゲームが起動するまでの構造や，基本的な画面遷移，描画，キー操作を実装する段階であり，これらの段階においては，Bridge によって開発効率が向上していると言える．被験者の感想でも「Bridge を利用すると必要なフレームワークが全て用意されているので，適切な場所に適切なコードを差し挟むだけでよかった」「Bridge を利用した場合は利用していない場合に比べて，何がどう遷移しているのかとか，どの部分でどう実装されたのがどう反映されるのかがわかったため，すんなり作れた」「Bridge を使用した場合，ゲームの骨組み（キーイベントの処理，イメージリソースの読み込み処理等）があるため，まずその点の実装の時間が大幅に短縮できた」などの意見を得た．

また，第 3 段階，第 4 段階では Bridge を利用することでグループ 1 の被験者は大幅に時間が短縮しているものの，グループ 2 の被験者は逆に Bridge を利用しない方が実装時間が少なくなっている．これは，第 3 段階，第 4 段階が敵機の出現・動作や衝突判定のアルゴリズムといったゲームロジックが中心となっている段階であり，両グループとも最初の実装で作り上げたゲームロジックを流用したために，後に行った実装の時間が短くなったと考えられる．グループ 1 の被験者の感想として「Bridge 未利用版を作ったことで必要なコード断片を用意してあり，それを貼り付けることが大幅な時間短縮につながった」「先に Bridge を用いないシューティングゲームを作成していたため，仕様書の把握，当り判定モデルの構想にかかる時間を省略できた」とあり，グループ 2 の被験者の感想として「Bridge 利用時にアルゴリズムを考えていた時間があり，Bridge 未利用時にはその時間はなかったので，実質は Bridge 利用時のほうが実装時間が 10 分ほど短いのではないか，と思った」とあるため，これを裏付けている．ゲームロジックによる時間差をできるだけ少なくするために予め必要なアルゴリズムをサンプルコードとして提供してあったが，サンプルコードの解読や，実際

にそれをゲームロジックとして応用するところに時間がかかったため，このような結果となったと考えられる．

最後に，第 5 段階の実装時間について分析を行う．Bridge を利用することでグループ 1 では実装時間が伸び，グループ 2 では実装時間が短縮しているが，いずれも大きな差は出ていない．グループ 1 で実装時間が伸びたひとつの要因として，2 名が Bridge 未利用時に画面遷移を実装するところまで辿り着かなかったことがあげられる．第 5 段階は画面遷移を主とした段階であるが，今回の仕様はそれほど複雑な画面遷移もなく，Bridge でサポートしている画面遷移を利用しなくても，実装時間に大きな差は出なかった．これは被験者の感想の「Bridge の使い方も分からない状態では，今回のシューティングゲーム程度の規模のゲームを開発する効率は上がらない．むしろ，Bridge の使い方を覚える時間が生じたため，かえって時間を費やすことになった」からも分かる．

表 6.5: 被験者の情報と試用実験における実装時間

被験者		Java の経験年数	携帯ゲーム開発経験	Bridge 未利用時の実装時間 (分)	Bridge 利用時の実装時間 (分)
グループ 1	A	11 年	なし	81	37
	B	1 年	あり	135	88
	C	1 年	なし	-（実装不可）	116
	D	1 年	なし	-（実装不可）	70
グループ 2	E	3 年	なし	70	63
	F	1.5 年	なし	163	101
	G	2 年	なし	110	67
	H	1.5 年	なし	89	87
	I	1 年	あり	114	90

表 6.6: 試用実験における Bridge 未利用時の実装時間（分）詳細

被験者		第 1 段階	第 2 段階	第 3 段階	第 4 段階	第 5 段階	合計
グループ 1	A	11	30	20	14	6	81
	B	47	46	14	11	17	135
	C	33	80	64	-	-	-
	D	24	60	35	61	-	-
グループ 2	E	25	21	6	8	10	70
	F	28	69	12	32	22	163
	G	9	56	13	6	26	110
	H	23	21	16	16	13	89
	I	14	66	4	6	24	114
グループ 1 平均		28.8	54	33.3	28.7	11.5	108
グループ 2 平均		19.8	46.4	10.2	13.6	19	109.2
総合平均		23.8	49.9	20.4	19.3	16.9	108.9

表 6.7: 試用実験における Bridge 利用時の実装時間（分）詳細

被験者		第 1 段階	第 2 段階	第 3 段階	第 4 段階	第 5 段階	合計
グループ 1	A	10	11	5	5	6	37
	B	8	37	8	14	21	88
	C	3	30	22	33	28	116
	D	3	22	16	6	23	70
グループ 2	E	3	18	15	23	4	63
	F	4	21	39	29	8	101
	G	3	20	10	12	22	67
	H	3	31	19	8	26	87
	I	5	37	13	18	17	90
グループ 1 平均		6	25	12.8	14.5	19.5	77.8
グループ 2 平均		3.6	25.4	19.2	18	15.4	81.6
総合平均		4.7	25.2	16.3	16.4	17.2	79.9

ソフトウェアメトリクスから見た分析

Bridgeの有用性を定量的な観点から調べるために，試用実験のソースコードを回収し，Bridgeを利用していない場合と利用した場合のそれぞれにおいて，6.1節と同様のソフトウェアメトリクスを使って測定を行った．

表6.8は，TLOC（コメント・空行を除いた全コード行数）の測定結果をまとめたものである．Bridge利用時の右列には，新規実装行数（TLOCからBridgeが自動生成したスケルトンコード行数[1]を差し引いた値）を示す．Bridgeを利用したことで，新規に実装した行数の平均がBridgeを利用しない場合のおよそ40%となっており，コーディングの分量が減ったことが伺える．

表6.9は，MLOC（コメント・空行を除いたメソッドのコード行数）の測定結果をまとめたものである．Bridgeを利用したことで，MLOCの平均は20%以下に減少し，最大値もMLOCの推奨値である24行を下回る結果となっている．このことから，Bridgeによって機能指向によるコーディングが少なくなり，保守性が上がっていると言える．

表6.10は，MCC（線形的に独立した経路の数）の測定結果をまとめたものである．Bridgeを利用したことで，MCCの平均は6.11から2まで減少し，最大値も26.78から9.22まで減少した．MCCが5以下であればプログラムは単純であり理解しやすく，MCCが10以下であればプログラムは理解困難ではないとしており[23]，Bridgeを利用したことでプログラムが理解可能となり，保守がしやすくなったと言える．

表6.11は，WMC（メソッドの複雑度の総和）の測定結果をまとめたものである．Bridgeを利用したことで，WMCの平均は20.5から13.2まで減少し，最大値も43.3から23.9まで減少した．WMCは30以下が推奨されており[19]，Bridgeを利用したことで，クラスあたりのメソッドの複雑度が下がったことが分かる．

表6.12は，NBD（メソッド中の最大ネスト数）の測定結果をまとめたものである．Bridgeを利用したことで，NBDの平均，最大値ともにおよそ70%まで減少しており，メソッドのネスト構造が複雑ではなくなったことが分かる．なお，他のメトリクスに比べてBridgeの利用前後で変化が小さいのは，今回の試用実験の仕様の規模がそれほど大きくなかったからである．

表6.13は，LCOM（クラス中のメソッドの凝集性欠如の度合い）の測定結果をまとめたものである．Bridgeを利用したことで，LCOMの平均は0.42から0.13に減少し，最大値も0.76から0.46まで減少した．LCOMが0に近づくほど凝集度が高く，まとまりのあるよい設計であるので[19]，Bridgeを利用したことで関連のある機能が局所化し，設計が改善したと言える．

以上のことより，Bridgeを利用したことで開発効率と保守性が向上したことをメトリクスによって定量的に示すことができた．

[1]ソースコード5.6，ソースコード5.7，ソースコード5.8の総和（試用実験では215行）

表 6.8: 試用実験におけるメトリクス（TLOC）測定結果

被験者		未利用時 TLOC	利用時 TLOC	利用時 新規実装行数
グループ 1	A	240	300	85
	B	345	377	162
	C	107	281	66
	D	137	281	66
グループ 2	E	159	284	69
	F	174	285	70
	G	179	279	64
	H	224	304	89
	I	192	285	70
平均		195.22	297.33	82.33

*TLOC(Total Lines of Code)...全コード行数（コメント，空行は含まない）

表 6.9: 試用実験におけるメトリクス（MLOC）測定結果

被験者		平均 未利用時	平均 利用時	標準偏差 未利用時	標準偏差 利用時	最大 未利用時	最大 利用時
グループ 1	A	5.1	4.3	13.3	6.2	158	19
	B	13.6	4.5	19.1	6.3	62	24
	C	9.3	3.9	10.4	6.0	30	19
	D	20.2	3.9	34.0	5.8	88	19
グループ 2	E	23.0	3.9	41.0	6.7	105	25
	F	33.5	3.9	54.0	6.2	127	24
	G	14.2	3.8	22.5	6.1	77	20
	H	19.0	3.9	49.2	6.1	158	19
	I	20.7	3.7	22.6	6.0	70	20
平均		18.36	3.27	29.39	5.65	85.00	21.00

*MLOC(Method Lines of Code)...メソッドのコード行数（コメント，空行は含まない）

表 6.10: 試用実験におけるメトリクス（MCC）測定結果

被験者		平均		標準偏差		最大	
		未利用時	利用時	未利用時	利用時	未利用時	利用時
グループ 1	A	3.07	1.91	3.11	2.25	12	8
	B	3.74	1.86	4.67	2.26	19	11
	C	3.29	1.96	2.71	2.38	8	8
	D	7.8	1.98	12.61	2.37	33	8
グループ 2	E	7.8	2.02	13.1	2.66	34	12
	F	11.5	2.04	17.05	2.61	41	12
	G	5.33	1.98	7.97	2.37	27	8
	H	5.89	2.26	13.48	2.62	44	8
	I	6.57	1.96	7.27	2.24	23	8
平均		6.11	2.00	9.11	2.42	26.78	9.22

*MCC(McCabe's Cyclomatic Complexity)...線形的に独立した経路の数

表 6.11: 試用実験におけるメトリクス（WMC）測定結果

被験者		平均		標準偏差		最大	
		未利用時	利用時	未利用時	利用時	未利用時	利用時
グループ 1	A	23	12.57	22	5.73	45	22
	B	23.67	13.5	28.55	6.82	64	27
	C	11.5	12.57	10.5	5.73	22	22
	D	19.5	12.71	18.5	5.97	38	23
グループ 2	E	19.5	13	18.5	6.48	38	25
	F	23	13.14	22	6.75	45	26
	G	24	12.71	23	5.97	47	23
	H	17.67	15.8	20.81	3.6	47	23
	I	23	12.86	22	6.2	45	24
平均		20.5	13.2	20.7	5.9	43.3	23.9

*WMC(Weighted methods per Class)...メソッドの複雑度の総和

表 6.12: 試用実験におけるメトリクス（NBD）測定結果

被験者		平均		標準偏差		最大	
		未利用時	利用時	未利用時	利用時	未利用時	利用時
グループ 1	A	1.67	1.17	0.79	0.43	4	3
	B	1.58	1.19	0.67	0.47	3	3
	C	1.71	1.18	0.7	0.44	3	3
	D	1.8	1.18	1.17	0.38	4	2
グループ 2	E	1.8	1.16	1.6	0.42	5	3
	F	2	1.2	1.73	0.45	5	3
	G	1.89	1.2	1.29	0.45	5	3
	H	1.44	1.26	1.26	0.5	5	3
	I	2.14	1.22	0.64	0.46	3	3
平均		1.78	1.20	1.09	0.44	4.11	2.89

*NBD(Nested Block Depth)...メソッド中の最大ネスト数

表 6.13: 試用実験におけるメトリクス（LCOM）測定結果

被験者		平均		標準偏差		最大	
		未利用時	利用時	未利用時	利用時	未利用時	利用時
グループ 1	A	0.41	0.08	0.41	0.18	0.83	0.53
	B	0.62	0.43	0.17	0.28	0.86	0.8
	C	0.38	0.04	0.38	0.09	0.75	0.27
	D	0.4	0.04	0.4	0.09	0.81	0.27
グループ 2	E	0.33	0.03	0.33	0.07	0.67	0.2
	F	0.46	0.04	0.46	0.1	0.92	0.28
	G	0.42	0.06	0.42	0.16	0.83	0.44
	H	0.38	0.1	0.38	0.2	0.9	0.5
	I	0.35	0.35	0.7	0.12	0.29	0.82
平均		0.42	0.13	0.41	0.14	0.76	0.46

*LCOM(Lack of Cohesion of Methods)...クラス中のメソッドの凝集性欠如の度合い

6.2.3 Bridge の理念に対する評価

実験の結果，以下のような感想が得られた．このことから，Bridge の設計の理念である画面管理機能と画面遷移機能によって開発効率や保守性が向上していること分かり，Bridge の有効性を定性的に示すことができた．

画面管理に対する評価

- 分けるべき構造がきっかりと分かれていて，個別で具体的な処理の記述に集中することができた．
- 規模の大きいゲームとなると画面の管理が煩雑になるため，クラスによって画面の状態を管理するという考えは非常に考えやすく，また，扱いやすかった．
- 意識していないで書くとごちゃ混ぜのプログラムになってしまいがちなので，遷移時の処理，ループごとの処理，キーの処理，描画の処理などが予め分けられている点がわかりやすい．
- Bridge を利用しないと，ソースコードをきれいに書くことを考えることに，ものすごく時間がかかる．また，開発以外のところというか，アルゴリズム以外の部分に手間がかかる．
- Bridge を使うことで時間が大幅に短縮できただけでなく，プログラム自体もかなり見やすくきれいになったと思うし，変更したい部分が出来た時も容易にできると思う．
- switch 文でゲーム状態を変更するよりもソースが見やすくなり，各画面中の処理が分かりやすく書けるようになっている．

画面遷移に対する評価

- 画面遷移の為のメソッド，保存しつつ遷移のメソッドは自分で書くと非常に煩雑になってしまうので，一番便利だと思った．
- 「画面に移ったときの処理」「画面から遷移するときの最終的な処理」があるので，いつ，どんな処理をするのか，イメージとして脳内に描きやすい．
- Bridge を使わない版をやったのだが，やはり画面遷移やキーイベントの処理が非常に面倒で，時間もかかってしまった．
- ゲーム開発で画面の状態遷移はありがちな処理であり，1 つのファイルにこれをすべて書いていくと可読性が悪い．

第7章　おわりに

本章では，評価や試用実験から分かったBridgeにおける今後の課題を明らかにし，最後に本研究のまとめを述べる．

7.1　今後の課題

7.1.1　ゲームモデルの抽象化

Bridgeは画面の分割管理やゲームループ構造の外部化という視点から，携帯ゲーム開発の効率化を行った．しかし，ゲーム特有のロジックやルールについてはほとんどサポートしていない．テーブルゲームであれば盤面やトランプ，アクションやシューティングであれば画面に表示されるキャラクターオブジェクトなど様々なゲームにおいて再利用可能なモデルが考えられる．また，サウンドのボリュームやゲームの得点，読み込み時の進捗を表すステータスバーなど，ゲームシステムにおいて共通化できる部分も数多く存在する．これらの機能をBridgeで抽象化して提供することで，より開発効率を高めることができる．

7.1.2　画面遷移の外部化

Bridgeでは，画面を遷移するときに元の画面からBridgeのメソッドを呼び出して，遷移先の画面クラスのインスタンスを渡すことで遷移を行っている．これらの画面遷移を予め外部に定義しておき，画面遷移を動的に変更することができれば，Bridgeは画面遷移を受理して動作するプッシュダウンオートマトン[25]であると考えることができる．その結果，状態遷移図を描くだけで，Bridgeがそれを解釈して画面遷移を実行することができれば，画面仕様が明確になり，さらに開発効率を上げることができる．

7.1.3　他のプラットフォームへの対応

本研究では，J2ME上で動作する携帯ゲームのみを対象範囲とした．しかし，auのBREWをはじめSymbian OS，iPhone，Google Androidなど携帯のプラットフォー

ムは多様化している．これらのプラットフォームに対応するためには，より柔軟なフレームワークの設計が必要であり，また，Java 以外の言語（C, C++, Objective C など）にも対応しなければならない．

7.2 まとめ

本研究では，携帯ゲーム開発を画面管理や動作構造の抽象化という観点から効率化するフレームワーク Bridge を提案して実装し，既存のゲームへの適用と試用実験によって，Bridge が携帯ゲーム開発を効率化ができることを明らかにした．本研究の成果は，2009 年 3 月に開催される情報処理学会第 71 回全国大会で発表予定である [26]．

本研究の試みは，携帯電話のハードの進化に伴うゲームの高品質化による携帯ゲーム開発者の負担を減らすことを目指すものである．携帯ゲーム開発に利用できるフレームワークは外部に公開されているものがほとんどないため，開発者の多くは書籍や Web を参考にしたり，携帯ゲームの開発経験がある人から携帯ゲーム独自の開発手法を学ぶ必要がある．Bridge の試用実験を通じて，携帯ゲームの開発経験が皆無であっても，Bridge を利用することで小規模の携帯ゲームを短時間で開発できたため，携帯ゲーム開発の敷居を下げることにも十分な効果があることが分かった．Bridge が普及することで，誰にでも高品質のゲームが効率良く開発できるようになれば，携帯ゲーム市場の更なる活性化が期待できる．今後は，プログラミングの経験がない人でも，簡単に効率良く携帯ゲームを開発できるようなフレームワークの可能性についても模索していきたい．

携帯電話のハードの進化に伴い，携帯ゲームは 10 年前の家庭用ゲームと同等の品質を持つようになってきた．このことから，10 年後には現在の家庭用ゲームと同等の規模のゲームが携帯ゲームとして登場することも考えられる．現在の家庭用ゲームでは，プログラムに対してグラフィックやサウンドといったリソースの割合が極端に大きくなっており，プログラムをリソースからうまく分離し，いかにリソースを効率良く制御するかということが重要となっている．Bridge がその役割を担うことができるように，現在の家庭用ゲームにも対応できるような柔軟なフレームワークを考案し，さらなる発展を目指していきたい．

謝辞

本研究を遂行し修士論文としてまとめるにあたり，多くの方にお世話になりました．この場を借りて感謝の意を述べさせていただききます．

まず，指導教官であり主査である有澤誠先生には，毎週のミーティングで論文執筆のスケジュール管理を行なっていただきました．何事も早め早めに行なう有澤先生の指導で，1ヶ月前にはドラフト版が完成し，余裕を持って推敲を行なうことができたことを感謝いたします．

修士論文副査である大岩元先生には，学部時代から多くのご指導をいただきました．コンピュータサイエンスやソフトウェア工学をはじめとして，技術の本質を勉強する大切さを教えていただいたことは，学生生活の中で最も有益であったと思っています．

産業技術大学院大学の中鉢欣秀さんには，技術的な内容を中心に本研究と論文執筆を指導していただきました．研究の独自性と評価について相談に乗っていただけたことで考えが整理され，研究成果を論文としてまとめる上でとても役に立ちました．

株式会社インテムの方々には長い間修士論文執筆の支援をしていただきました．金澤徹さんには本研究の問題提起をしていただきました．熊谷哲孝さん，岡田健さんには設計・実装を進める上で技術的な相談に乗っていただきました．今村慶裕さんには試用実験に協力していただきました．

また，大岩研究室の皆さんにもお世話になりました．海保研さんには，本研究の礎となる状態遷移モデルを考えていただきました．松澤芳昭さんには，オブジェクト指向の基礎的な概念など技術的なことをはじめ，仕事の仕方など様々なことを教えていただきました．杉浦学さんには事務手続きをはじめとして，色々な形でご指導ご支援をいただきました．学部時代からの同期である荒木恵さんには，研究を行なう上で様々な協力していただき，公私ともにお世話になりました．藤原育美さん，篠崎友識さんには学部時代に本研究の前進であるフレームワークの実装を手伝っていただきました．

最後に，体の心配をしてくれた母 真子と，家事と子育てに専念して研究活動を支えてくれた妻 沙織に心から感謝します．

橋山 牧人

参考文献

[1] NTT DOCOMO. i アプリ. http://www.nttdocomo.co.jp/service/imode/make/content/iappli/, 2008.

[2] 鷲見豊. プログラミング i モード Java. オライリー・ジャパン, 2001.

[3] アプリゲット. アプリ開発講座. http://appget.com/contest/au2007/lecture/index.html, 2007.

[4] ソフィア・クレイドル. Doja と midp の相違点&アプリサイズ. http://www.s-cradle.com/developer/java/difference_carrier.html, 2007.

[5] MIDP2.0 メモ. i アプリを s!アプリ・オープンアプリに移植する. http://www.saturn.dti.ne.jp/~npaka/kvm/midp2/ActionGame/index.html, 2007.

[6] Mobile JavaApplication Development Forum From PotRinStudent. 携帯アプリフレームワーク caledonia. http://mjdf.potrin.com/, 2005.

[7] 西岡拓人. 携帯アプリ開発フレームワーク mokit. https://sourceforge.jp/projects/mokit, 2008.

[8] 宍戸輝光. uk ゲームライブラリ. http://sfkonu.vni.jp/sbcsoft/ukglib/, 2007.

[9] 高橋武司. プログラム変換による携帯電話用アプリケーションの移植作業の簡略化について. 東京工科大学 メディア学部ゲームサイエンスプロジェクト 卒業論文, 2006.

[10] 株式会社エンターブレイン. R p g ツクール. http://tkool.jp/, 2007.

[11] eclipse Project. eclipse. http://www.eclipse.org/, 2008.

[12] Ralph E. Johnson, et al. パターンとフレームワーク. 共立出版, 1999.

[13] Erich Gamma, et al. オブジェクト指向における再利用のためのデザインパターン. ソフトバンククリエイティブ, 2006.

[14] Peter Haggar. double-checked locking と singleton パターン. http://www.ibm.com/developerworks/jp/java/library/j-dcl/, 2002.

[15] 小柴豊. Java solution 第 11 回 読者調査結果 -2004 年 java 開発環境に関する意識調査-. http://www.atmarkit.co.jp/fjava/survey/survey0406/java0406.html, 2004.

[16] SOFTBANK MOBILE. S!アプリ. http://creation.mb.softbank.jp/sapp/index.html, 2008.

[17] Lawrence H. Putnam, et al. 初めて学ぶソフトウエアメトリクスプロジェクト見積もりのためのデータの導き方. 日経 BP 社, 2005.

[18] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design,. *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, pp. 476–493, 1994.

[19] Brian Henderson-Sellers. *Object-Oriented Metrics, measures of Complexity.* Prentice Hall, 1996.

[20] Software metrics in eclipse. http://open.ncsu.edu/se/tutorials/metrics/, 2006.

[21] Stephen H. Kan, et al. ソフトウェア品質工学の尺度とモデル. 共立出版, 2004.

[22] THOMAS J. McCave. A complexity measure. *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, pp. 308–320, 1976.

[23] Capers Jones, et al. ソフトウェア開発の定量化手法. 構造計画研究所, 1998.

[24] 布留川英一. i アプリゲーム開発テキストブック 901i/900i/700i/506i/505i 対応. 毎日コミュニケーションズ, 2004.

[25] J.E. ホップクロフト, J.D. ウルマン. 言語理論とオートマトン. サイエンス社, 1999.

[26] 橋山牧人, 中鉢欣秀, 大岩元. 携帯ゲームアプリケーション開発を支援するオブジェクト指向を用いたフレームワークの開発. 情報処理学会 第 71 回全国大会（発表予定）, 2009.

付 録A　Bridgeマニュアル

Bridgeマニュアル

2008.11.26 (ver 1.1)
Makito Hashiyama

目次

1. Bridgeとは？

1.1 Bridgeの概要

Bridgeとは，携帯ゲームアプリを簡単に開発・保守・移植するためのアプリケーションフレームワークです．オブジェクト指向による画面管理機能始はじめ，ゲーム開発に役立つ様々な機能やライブラリを提供します．携帯アプリの規模やプラットフォームが拡大した現在，プログラマはBridgeを用いることでより効率的に ゲームアプリを開発することができるようになります．

1.2 Bridgeの機能

Bridgeには以下のような機能があります．

- スケルトンコード生成機能

  ボタン1つでDocomo用アプリ(iアプリ)の起動からアプリの制御までが記述されたスケルトンコードが作成されます．プログラマはこのソースコードに沿って，指定されたところに必要な処理を記述するだけで，簡単にゲームアプリを作成することができます．

- 画面管理機能

  Bridgeではゲーム画面を1つの状態(オブジェクト)として捉えて，スタック(Stack)を用いて管理します．またStateパターンを用いることで，状態が変化したとき(画面が遷移したとき)がクラス内に振る舞いの変化を記述する必要なく，振る舞いを変えることができます．つまり，画面を追加するたびに関連するすべてのif文の分岐を見直すというゲームにおける典型的な作業から解放されます．

- Softbank用アプリ(S!アプリ)への変換機能

  Bridgeで提供されているライブラリはDoJa(Docomoが準拠している携帯プログラミング仕様)とMIDP(Softbankが準拠している携帯プログラミング仕様)の双方をできるだけ共通化するように記述されています．そのため，Bridgeを用いて作成したDocomo用アプリ(iアプリ)は，Bridgeに用意されている変換機能を用いれば，簡単な設定ファイルの記述やリソースのデータ形式を変更することで，Softbank用のアプリ(S!アプリ)として動作します．

- ゲームアプリに必須な機能のサポート

  ゲームでは必須である，イメージやサウンド，データ保存領域へのアクセスなどの機能を独自のライブラリによってサポートします．

2. Bridgeの導入

2.1 事前準備

Bridgeはeclipseプラグインの形で提供されるため，利用するためにはeclipseが必要です．また，Docomo用アプリ(iアプリ)，Softbank用アプリ(iアプリ)を開発するために各キャリアが提供しているツールも必要となります．ここでは，それらのインストール方法を順を追って説明します．

2.1.1 eclipseのインストール

Bridgeは現在eclipse 3.3または3.4で動作することを確認しています．それ以前のバージョンではエラーが発生します．Eclipse Wikiを参考に，上記バージョンのいずれかをダウンロード・インストールして下さい．インストールパッケージがいくつかありますが，以下のものがおすすめです．

- MergeDoc Project(Pleiades All in One 日本語ディストリビューション)

2.1.2 J2ME Wireless Toolkitのインストール

携帯アプリを開発するために必要なツール群をインストールします．以下のサイトからツール群をダウンロードして下さい(無料の会員登録が必要です)．

- Java 2, Micro Edition (J2ME) Wireless Toolkit 2.2

ダウンロードしたら，インストーラをダブルクリックしてインストールして下さい．

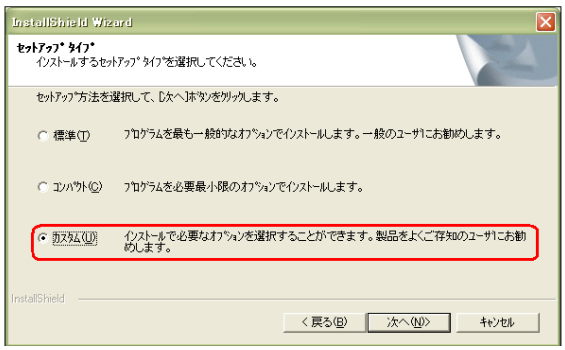
※依存関係の問題で，各キャリアの開発ツールより先にインストールする必要があります

2.1.3 iアプリ開発ツールのインストール

iアプリを開発するためのツールをインストールします．以下のサイトから開発ツールをダウンロードして下さい．

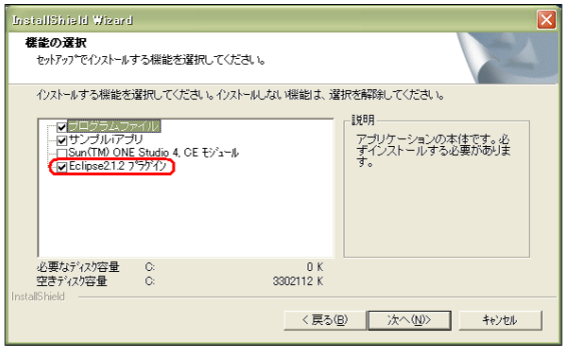
- DoJa-3.5プロファイル向けiアプリ開発ツール

ダウンロードして解凍したら，DISK1フォルダにあるsetup.exeをダブルクリックするとインストーラが起動します．使用許諾，インストール先を選択した後，セットアップ方法で「カスタム」を選びます．

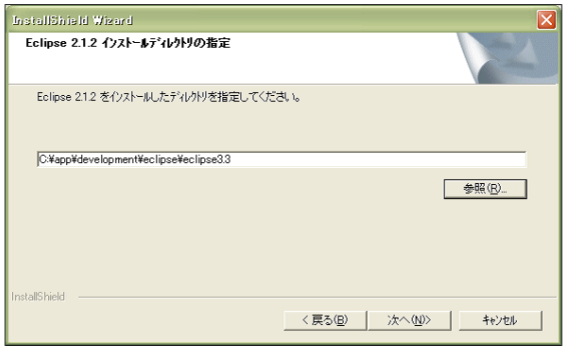


続いて，「機能の選択」にて eclipse 2.1.2プラグイン(※)を忘れずにチェックしてください．SunのOne Studioは利用しないので，チェックを外します．

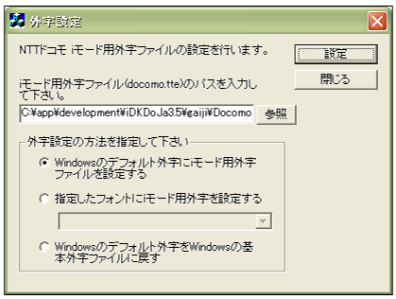
※eclipseの対応バージョンが異なりますが，3.3以降でも動作することを確認しています．



続いて，eclipseをインストールしたフォルダの指定を行います．



インストールが完了すると，iアプリの外字設定ダイアログが開きます．これは，DoCoMo携帯の外字をどのように登録するかを指定するものです．デフォルトで外字ファイルと外字設定の方法(Windowsのデフォルト外字にiモード用外字ファイルを設定する)のまま設定してからダイアログを閉じてください．



2.1.4 S!アプリ開発ツールのインストール

S!アプリを開発するためのツールをインストールします．以下のサイトから開発ツールをダウンロードして下さい（無料の会員登録が必要です）．

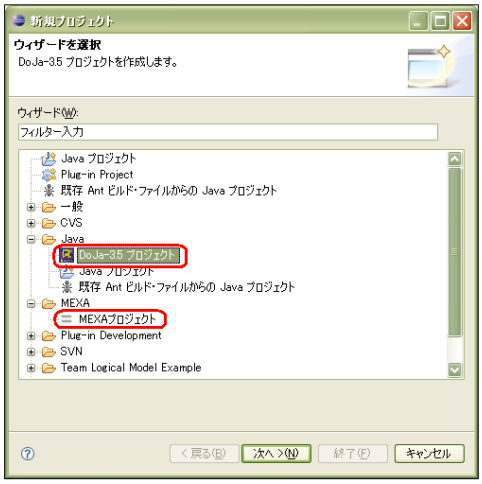
- S!アプリ開発ツール MEXA SDK
- Eclipse向けプラグイン for MEXA SDK

ダウンロードして解凍したら，インストーラをダブルクリックしてインストールして下さい．Eclipse向けプラグイン for MEXA SDKをインストールする際に，eclipseをインストールしたフォルダを指定してください．

2.1.5 インストールの確認

インストールが終わったら，確認のためにeclipseを起動します．「ファイル」→「新規」→「プロジェクト」を選択して，新規プロジェクトウィザードを開きます．下図のように「Java」の下に「DoJa-3.5 プロジェクト」，「MEXA」の下に「MEXAプロジェクト」があれば，インストールは成功です．

アイコンが表示されなかった場合は，インストールに失敗した可能性があるので，インストールしたプラグインを一度アンインストールして下さい．



2.2 Bridgeのインストール

eclipseとアプリ開発環境の準備が整ったら，Bridge本体のインストールを行います．下記より，Bridgeをダウンロードして下さい．

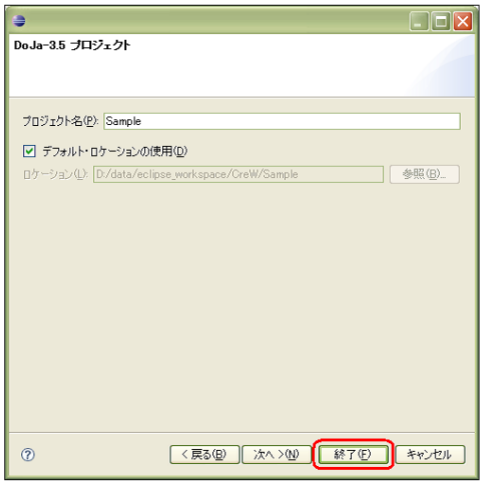
- Bridge プラグイン 1.0.8（2008/11/27 現在の最新版）

ダウンロードして解凍したら，中に入っているjarファイルをeclipseのpluginフォルダに入れてeclipseを再起動してください．下図のようにeclipseのツールバーにBridgeのアイコンが表示されたらインストール完了です．

3. チュートリアル

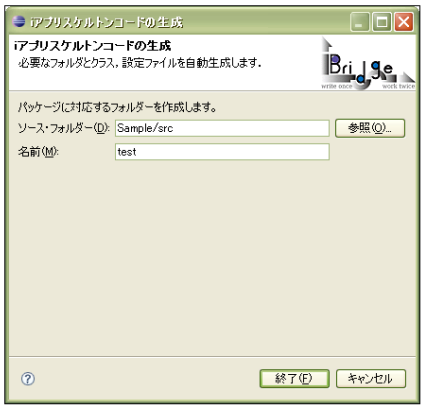
3.1 iアプリのスケルトンコードを生成する

eclipseを起動し，メニューから「ファイル」→「新規」→「プロジェクト」を選択し，新規プロジェクトウィザードを開きます．「Java」の下にある「DoJa-3.5 プロジェクト」を選択し，『次へ』を押下します．下図のように適当なプロジェクト名を入力して『終了』を押下すると，iアプリプロジェクトが作成されます．この時点では，まだスケルトンコードは生成されていません．

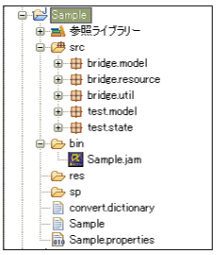


次に，実際にスケルトンコードの生成を行います．先ほど作成したiアプリプロジェクトを選択してから，ツールバーにある「仮」アイコンをクリックします．すると，下図のようなウィザードが開くので，作成するアプリのパッケージ名(※)をつけて，『終了』を押下して下さい．

※パッケージ名はライブラリ(bridge)と名前の衝突を避けるためにつける必要があります．プロジェクト名と同じでも構いません．



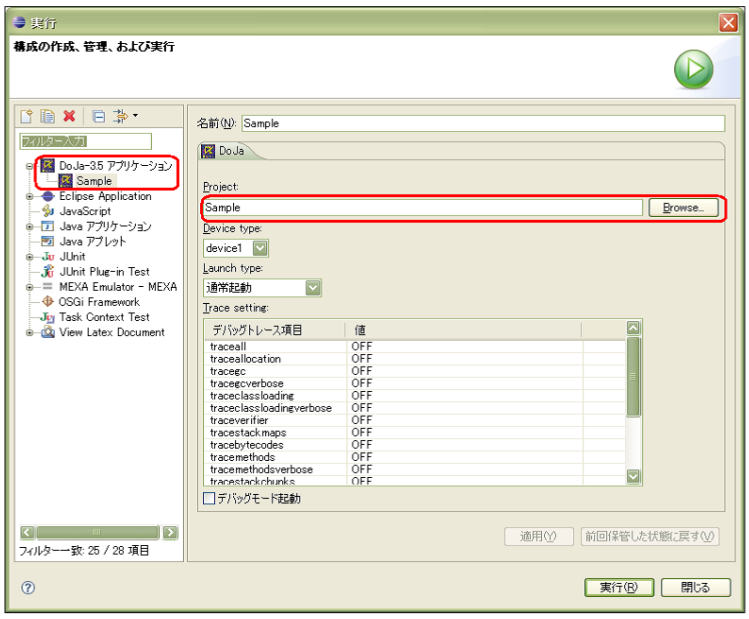
スケルトンコードの生成に成功すると，下図のようなパッケージ構造とクラス，必要なファイルが作成されます．



- srcフォルダ
  - bridge.modelパッケージ
    iアプリの制御を行うクラスが置かれています．詳しくはAPI Referenceをご覧下さい．
  - bridge.resourceパッケージ
    リソースにアクセスを補助するためのクラスが置かれています．詳しくはAPI Referenceをご覧下さい．
  - bridge.utilパッケージ
    iアプリの作成する上で，よく利用される機能をまとめたクラスが置かれています．詳しくはAPI Referenceをご覧下さい．
  - test.modelパッケージ
    これから作成するアプリのゲームのモデルやロジックを表すクラスが置かれます．
    初期状態ではゲームのモデルを表すクラス(SampleModel.java)と，基礎となる画面クラス(SampleState.java)が置かれます．
  - test.stateパッケージ
    これから作成するアプリの実際の画面クラスが置かれます．
    初期状態ではタイトル画面を表現するサンプルクラス(TitleState.java)が生成されています．
- binフォルダ
  - Download.htmlファイル
    iアプリを実機にダウンロードするときにアクセスするファイルです．
  - jamファイル
    iアプリ実行時の設定情報が記述されたファイルです．
- resフォルダ
  イメージやサウンドが置かれるフォルダです．
- spフォルダ
  iアプリの外部に保存したいデータが置かれます．
- convert.dictionaryファイル
  S!アプリに変換を行う際の変換ルールが記述されたファイルです．
- その他のファイル
  iアプリプロジェクトの情報が記述されています．

3.2 iアプリを動かす

先ほど生成したスケルトンコードを実際に動かしてみます．eclipseのメニューから「実行」→「実行ダイアログを開く」を選択し，実行ダイアログを開きます．下図のように「DoJa3.5アプリケーション」をダブルクリックし，先ほど作成したプロジェクト（ここではSample）を選択し，『実行』を押下します．

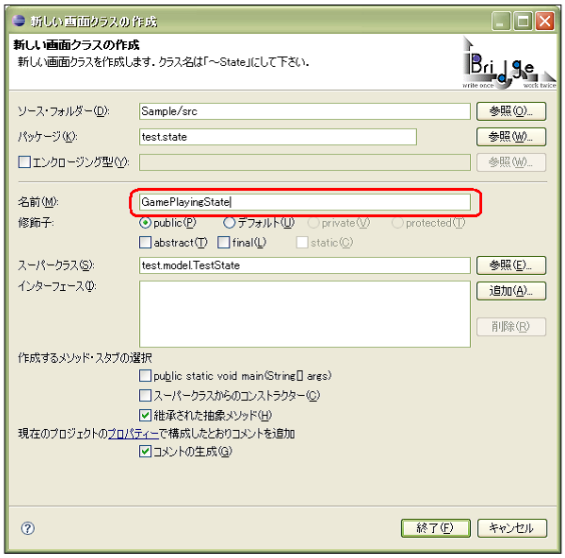


問題が無ければ，eclipse上からiアプリのエミュレータが起動し，以下のような画面が表示されます．ここでは，srcフォルダ以下の test.state.TitleState クラスの draw() メソッド内に記述してある文字列が描画されています．

3.3 新しい画面を作成する

無事にタイトル画面が動いたところで，実際に自分で定義した新しい画面を作成してみます．srcフォルダ以下の test.state パッケージを選択した状態でツールバーにある「新」アイコンをクリックして，新規画面クラス作成ウィザードを開きます．下図のように作成する画面の名前(※)を入力して，『終了』を押下して下さい．

※クラス名は状態(画面)であることが分かるように末尾に「State」をつけるのが好ましいですが，つけなくても問題ありません．



成功すると，選択したパッケージ(ここでは test.state)に新しい画面クラス(ここでは GamePlayingState)が作成されます．予め定義されているそれぞれのメソッドにその画面の処理を記述していきます．

- コンストラクタ

  画面クラスのコンストラクタは private で宣言されていて，外部からインスタンス化できないようになっています．こうすることで，画面遷移するときに常に同じ画面オブジェクトが渡されることが保障されます．初期化処理は entryAction() に記述して，コンストラクタ内には何も記述しないで下さい．

- entryActionメソッド

  ここには画面に遷移してきたとき最初に行うべき処理を記述します．パラメータや座標などのデータの初期化や，BGMの開始などはここに記述して下さい．原則的に必ず呼び出されますが，画面遷移時に呼ばないようにすることも可能です．

- actionメソッド

  ここには一定時間(毎フレーム)ごとに行われる処理を記述します．画面のスクロール，コンピュータキャラの移動などユーザのキー入力と無関係に行われる処理はここに記述して下さい．

- processKeyメソッド

  ここにはユーザがキー入力を行った場合の処理を記述します．AbstractModelに定義されている isKeyPressed, isKeyPressing メソッドを使ってキーの状態を調べることができます．

- drawメソッド

  ここには画面の描画処理を記述します．引数として渡される Graphics オブジェクトの描画メソッドを呼び出すことで，描画を行うことができます．

- exitActionメソッド

  ここには他の画面に遷移する直前に行うべき処理を記述します．データの後処理や，BGMの停止などはここに記述して下さい．

- getInstanceメソッド

  画面遷移をするときには遷移先の画面オブジェクトを渡す必要がありますが，画面クラスはインスタンスを外部で持つことが出来ません．そこで，getInstance メソッドを用いることでその画面のインスタンスを取得することができます．

3.4 画面を遷移させる

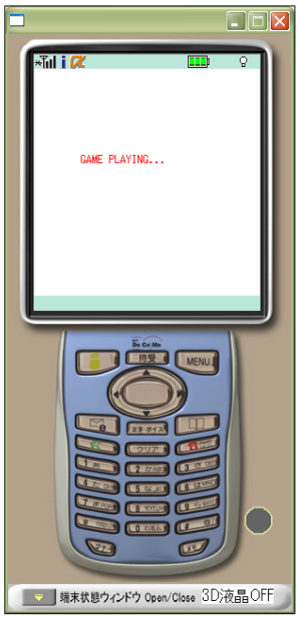
せっかく新しい画面(GamePlayingState)を作成したので，タイトル画面(TitleState)で決定キーを押下したら，新しい画面に遷移するようにプログラムを改良してみます．画面を遷移させるには transition メソッドを利用します．以下のように，TitleState.java の processKey メソッド内に遷移処理を記述します．

```
// 決定キーが押された場合
if (model.isKeyPressed(KEY_SELECT)) {
	model.transition(GamePlayingState.getInstance());
}
```

画面クラスが必ず持っているゲームモデルクラスへの参照(model)を介して，transition メソッドを呼び出します．transition メソッドの引数は遷移先の画面オブジェクトです．画面オブジェクトは，画面クラスで定義されている getInstance メソッドを静的(static)に呼び出すことで取得することができます．次に，画面が遷移されたことが分かるように，新しい画面クラスの描画メソッド draw を以下のように書き換えます．

```
/**
 * 描画の処理を行います
 */
public void draw(Graphics g) throws Exception {
	model.clearDisplay(Graphics.getColorOfName(Graphics.WHITE));

	g.setColor(Graphics.getColorOfName(Graphics.RED));
	g.drawString("GAME PLAYING...", 50, 100);
}
```

clearDisplay メソッドを使って一度画面を白く塗りつぶしてから，別の色で新しい画面特有の文字を描画します．もう一度実行ダイアログからiアプリを起動して，エミュレータの中央にある丸いキーを押下してみましょう．画面遷移の記述に問題がなければ，画面が切り替わり，下図のように新しい文字が表示されます．

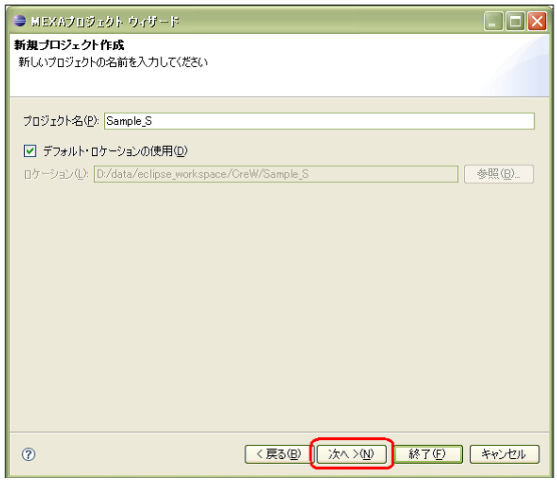


3.5 iアプリをS!アプリに変換する

ここでは，作成したiアプリをS!アプリに変換してみましょう．iアプリをS!アプリに変換するためには，まず変換先のS!アプリプロジェクトを作成して，それから変換を行います．

3.5.1 S!アプリプロジェクトを作成する

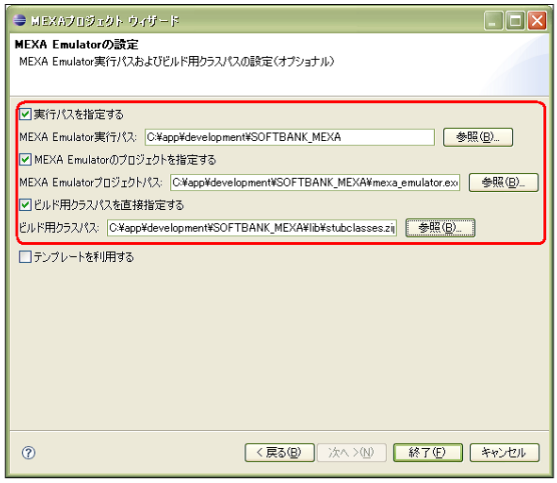
メニューから「ファイル」→「新規」→「プロジェクト」を選択し，新規プロジェクトウィザードを開きます．「Java」の下にある「MEXA プロジェクト」を選択し，『次へ』を押下します．下図のように適当なプロジェクト名を入力して『次へ』を押下します．



エミュレータの設定画面で，下図のように上位3項目のチェックボックスにチェックを入れて，MEXAエミュレータの実行パスやビルドのクラスパスを指定します．入力が終わったら最後に『終了』を押下すると，S!アプリプロジェクトが生成されます．

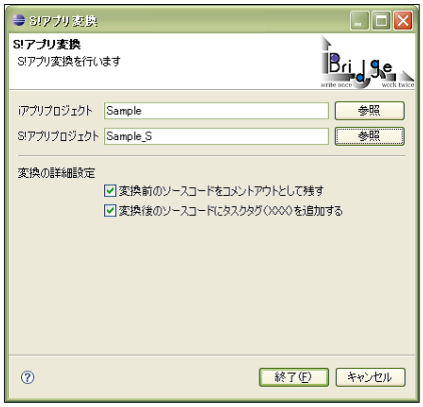
- MEXA Emulator実行パス
  MEXAエミュレータが保存されているフォルダまでのパスを入力します(※)．
- MEXA Emulatorプロジェクトパス
  MEXAエミュレータの実行保存されているフォルダまでのパスを入力します(※)．
- ビルド用クラスパス　MEXAのライブラリまでのパスを指定します．

※インストール時のデフォルトの設定では「C:¥Program Files¥SOFTBANK_MEXA_EMULATOR**¥」に設定されています(**はバージョン番号)．似たような名前のフォルダがCドライブ直下に生成されるので間違えないで下さい．正しいフォルダの下にはmexa_emulatorの実行ファイル置かれています．

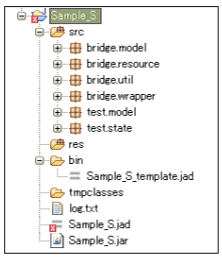


3.5.2 変換を行う

S!アプリプロジェクトを作成したら，いよいよ変換を行います．変換元のiアプリプロジェクトを選択した状態で，ツールバーにある「換」アイコンをクリックして，S!アプリ変換ウィザードを開きます．下図のように，変換先のSアプリプロジェクト選択して，『終了』を押下して下さい．



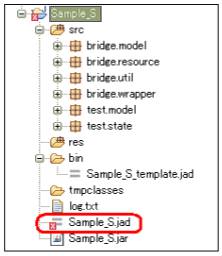
変換に成功すると，選択したS!アプリプロジェクト以下に，下図のようなパッケージ構造とクラス，必要なファイルが作成されます．



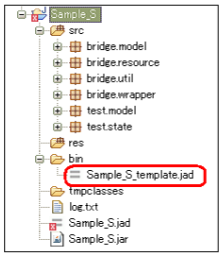
- srcフォルダ
  - bridge.wapperパッケージ
    iアプリとS!アプリのAPIの差異を吸収するためのライブラリです．
  - その他のパッケージ
    bridgeパッケージに入っていたライブラリは変換時にS!アプリ用ライブラリと挿げ替えられます．
    また，プログラマが作成したパッケージ(ここではtest)は変換辞書(convert.dictionary)に従って，変換が行われたものが置かれます．
- binフォルダ
  - jadファイル
    S!アプリ実行時の設定情報が記述されたマニフェストファイルのテンプレートです．
- resフォルダ
  イメージやサウンドが置かれるフォルダです．
  iアプリとS!アプリのリソースには互換性がない可能性が高いので，手作業で変換する必要があります．
- log.txtファイル
  変換結果のログが出力されています．
- jadファイル
  S!アプリ実行時の設定情報を記述するマニフェストファイルです．
  プロジェクト作成直後は設定ファイルの記述が不十分なためエラーマークが表示されていますが，次節で行う設定が終わればマークは消えます．
- jarファイル
  クラスファイルやリソースを圧縮した実行ファイルです．

3.6 S!アプリを動かす

最後にMEXAエミュレータを使って，S!アプリを動かしてみます．S!アプリを実行するには，マニフェストファイルを記述する必要があります．プロジェクト名と同名の jad ファイル（_templateがついていないもの）をダブルクリックすると，JADエディタで開くことができます．

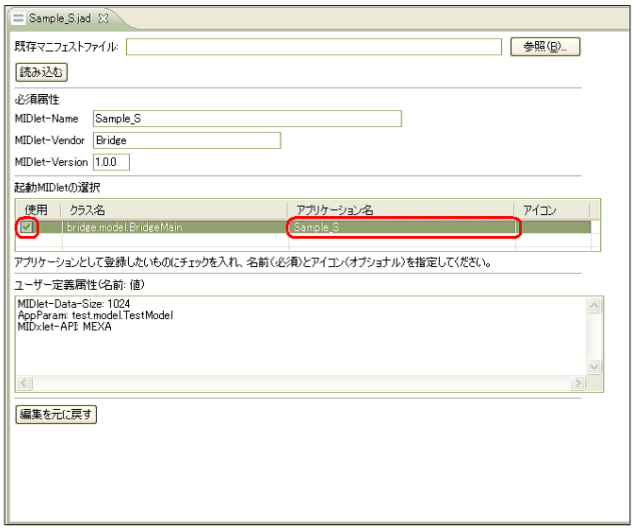


既存マニフェストファイルとして，bin フォルダ以下に生成された「プロジェクト名_template.jad」ファイルを読み込むと，設定が反映されます．



あとは，下図のように，「起動MIDletの選択」でチェックボックスにチェックを入れて，アプリケーション名に適当な名前をつけてEnterを押下します．JAD ファイルの保存を行い，ファイルに出ていたエラーマークが消えれば実行準備完了です．

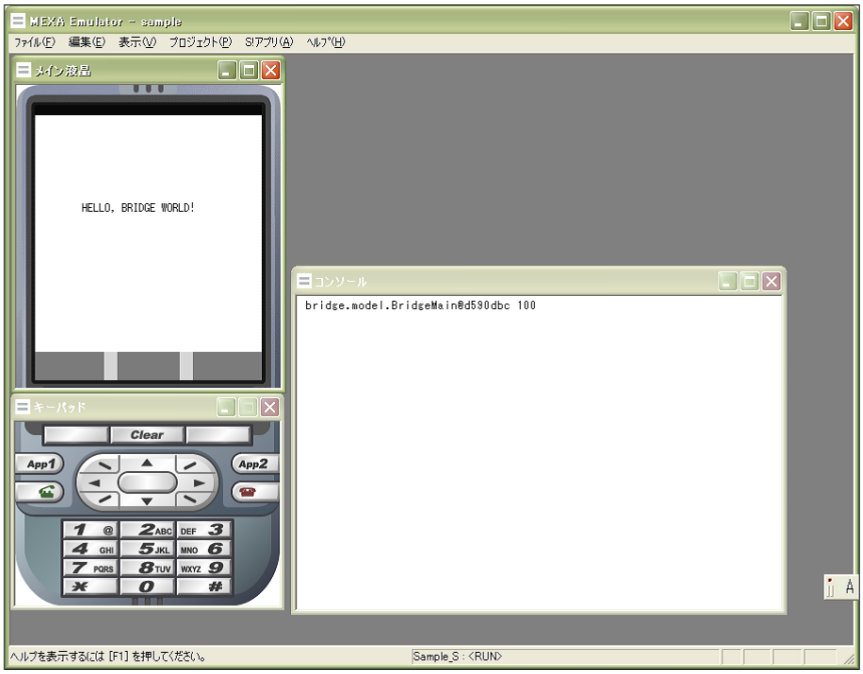
※エラーマークが消えない場合は，プロジェクトを選択して，メニューの「プロジェクト」→「クリーン」を実行してください



後は，JAD ファイルを右クリックして「実行」→「MEXA Emulator」を選べば，エミュレータが起動して，下図のような画面が表示されます．

※Windowsのファイアウォールが原因でエミュレータが起動した後，何も表示されない場合があります．その場合はコチラを参照して下さい．

※エミュレータ起動時にインストール方法を質問された場合「Trusted (3rd Party Domain)でインストール」を選択して下さい．



<< Bridgeマニュアル/3. チュートリアル >>

4. 逆引きリファレンス

4.1 文字・図形の描画

ここでは，文字・図形の描画方法と，フォント・色の変更方法について解説します．

4.1.1 文字を描画したい

文字を描画するためには，Graphicsクラスの drawString メソッドを利用します． 第1引数に描画する文字列，第2引数にX座標，第3引数にY座標を指定します．

```
public void draw(Graphics g) throws Exception {
	g.drawString("描画をします．", 0, 0);
}
```

4.1.2 フォントサイズを変更したい

フォントサイズを変更するためには，以下の手順を行います．

1. Fontクラスのget Fontメソッドを使い，フォントを取得する
2. Graphicsクラスのset Fontメソッドを使い，フォントを設定する

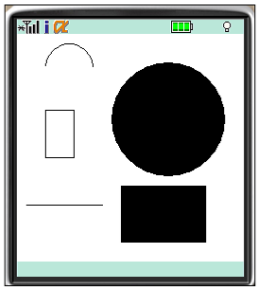
```
import com.nttdocomo.ui.Font;

public void draw(Graphics g) throws Exception {
        Font font = Font.getFont(Font.SIZE_TINY);
        g.setFont(font);
        g.drawString("TINYサイズです．", 10, 40);

        font = Font.getFont(Font.SIZE_SMALL);
        g.setFont(font);
        g.drawString("SMALLサイズです．", 10, 80);

        font = Font.getFont(Font.SIZE_MEDIUM);
        g.setFont(font);
        g.drawString("MEDIUMサイズです．", 10, 120);

        font = Font.getFont(Font.SIZE_LARGE);
        g.setFont(font);
        g.drawString("LARGEサイズです．", 10, 160);
}
```

なお，現在のBridgeでは，フォントサイズ以外の変更をサポートしていません(※)．

4.1.3 図形を描画したい

図形を描画するには，Graphicsクラスで定義されている各種メソッドを利用します．

```
public void draw(Graphics g) throws Exception {
        // 円弧を描画します
        g.drawArc(30, 10, 50, 50, 0, 180);

        // 長方形を描画します
        g.drawRect(30, 80, 30, 50);

        // 直線を描画します
        g.drawLine(10, 180, 90, 180);

        // 塗りつぶしの円弧を描画します
        g.fillArc(100, 30, 120, 120, 0, 360);

        // 塗りつぶしの長方形を描画します
        g.fillRect(110, 160, 90, 60);
}
```



なお，現在のBridgeでは，多角形の描画をサポートしていません(※)．

4.1.4 文字・図形の色を変更したい

図形を描画するには，Graphicsクラスで定義されている setColor メソッドを利用します．setColor メソッドの引数として指定する色は，次のどちらかの方法で取得します．

- Graphicsクラスのget Color Of Nameメソッドを使い，Graphicsクラスで定義されている色を指定する
- Graphicsクラスのget Color Of RGBメソッドを使い，赤・緑・青の輝度を個別に指定する

```
public void draw(Graphics g) throws Exception {
	g.setColor(Graphics.getColorOfName(Graphics.RED));
	g.drawString("赤い文字です．", 10, 20);

	g.setColor(Graphics.getColorOfRGB(0xff, 0, 0));
	g.drawString("これも赤い文字です．", 10, 40);
	g.fillRect(10, 60, 20, 20);

	g.setColor(Graphics.getColorOfName(Graphics.BLUE));
	g.fillRect(10, 100, 40, 40);
}
```

4.2 イメージの描画

ここでは，ファイルから読み込んだイメージを描画する方法について解説します．読み込むイメージファイルは res フォルダ以下に置いてください．

iアプリで読み込むことができるイメージ形式は「.gif」「.jpeg」のみです．
SIアプリで読み込むことができるイメージ形式は「.png」「.jpeg」のみです．

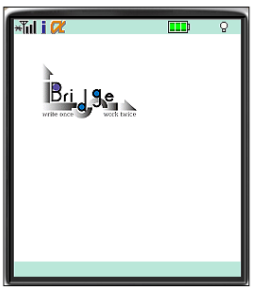
4.2.1 イメージを描画したい

イメージを描画するためには，以下の手順を行います．

1. ImageManagerクラスのloadImageメソッドを使ってイメージファイルを読み込む
2. Graphicsクラスのdrawimageを使ってイメージを描画する

Utilityクラスの getImage メソッドを使うと，読み込み済みのイメージを名前で取得することができます．なお，イメージを描画のたびに読み込むのはメモリの無駄使いなので，ゲームモデルが提供する initializeModel メソッド内で利用するすべてのイメージを読み込むべきです．

```
// ゲームモデルでイメージを読み込んでおきます
protected void initializeModel() {
	ImageManager manager = ImageManager.getInstance();
	manager.loadImage("sample.gif");
}

// ------------------------------------------------------------------ //

// 実際にイメージを利用する画面で，イメージを取得して描画します
public void draw(Graphics g) throws Exception {
	g.drawImage(Utility.getImage("sample.gif"), 30, 30);
}
```

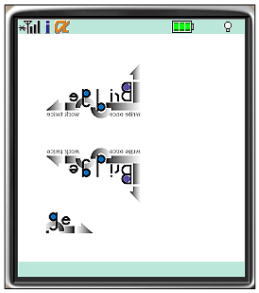


4.2.2 イメージを変形したい

イメージは描画時に以下のような変形ができます。

- Graphicsクラスの setFlipModeメソッドを使って画像を回転させる
- Graphicsクラスの drawImageを使ってイメージの一部を描画する

```
// ゲームモデルでイメージを読み込んでおきます
protected void initializeModel() {
	ImageManager manager = ImageManager.getInstance();
	manager.loadImage("sample.gif");
}

// ------------------------------------------------------------------ //

// 実際にイメージを利用する画面で，イメージを取得して描画します
public void draw(Graphics g) throws Exception {
	// 左右を反転します
	g.setFlipMode(Graphics.FLIP_HORIZONTAL);
	g.drawImage(Utility.getImage("sample.gif"), 30, 30);

	// 180度回転させます
	g.setFlipMode(Graphics.FLIP_ROTATE);
	g.drawImage(Utility.getImage("sample.gif"), 30, 120);

	// 画像を切り抜きます
	g.setFlipMode(Graphics.FLIP_NONE);
	g.drawImage(Utility.getImage("sample.gif"), 30, 180, 50, 20, 50, 30);
}
```



4.3 サウンドの再生・停止

ここでは，ファイルから読み込んだサウンドを再生・停止する方法について解説します．読み込むサウンドファイルは res フォルダ以下に置いてください．

iアプリで読み込むことができるサウンド形式は「.mld」のみです．S!アプリで読み込むことができるサウンド形式は「.spf」のみです．

4.3.1 サウンドを再生・停止したい

サウンドを再生・停止するためには，以下の手順を行います．各再生・停止メソッドの仕様についてはAPI Referenceをご覧下さい．

1. SoundManagerクラスのloadSoundメソッドを使ってサウンドファイルを読み込む
2. SoundManagerクラス，またはUtilityクラスの各種再生メソッドを使って再生する
3. UtilityクラスのstopやstopAllメソッドを使って停止する

サウンドを再生するためにはポートを指定する必要がありますが，BridgeではSoundManager.BGM_PORTとSoundManager.SE_PORTの2つを用意してあります．それぞれのポートはBGM（画面内で長時間再生される音楽），SE（短時間しか再生されない効果音）を想定していますが，プログラム上2つのポートに差はないので，どちらを利用しても問題ありません．

サウンドを再生するたびに読み込むのはメモリの無駄使いなので，ゲームモデルが提供する initializeModel メソッド内で利用するすべてのサウンドを読み込むべきです．

```
// ゲームモデルでサウンドを読み込んでおきます
protected void initializeModel() {
	SoundManager manager = SoundManager.getInstance();
	manager.loadSound("sample.mld");
}

// ------------------------------------------------------------------ //

// 実際にサウンドを利用する画面で，サウンドを再生します
public void processKey() {
	if (model.isKeyPressed(KEY_UP)) {
		// とりあえず簡易的に再生します
		Utility.playBGM("sample.mld");
	}

	if (model.isKeyPressed(KEY_DOWN)) {
		// 再生条件を細かく決めて再生します
		SoundManager manager = SoundManager.getInstance();
		manager.play("sample.mld", SoundManager.BGM_PORT, 5, true);
	}

	if (model.isKeyPressed(KEY_LEFT)) {
		// ポートを指定して停止します
		Utility.stop(SoundManager.BGM_PORT);
	}

	if (model.isKeyPressed(KEY_RIGHT)) {
		// すべてのポートの再生を停止します
		Utility.stopAll();
	}
}
```

4.3.2 サウンドを一時停止・再開したい

サウンドを一時停止するにはSoundManagerクラスの pause メソッドを，一時停止したサウンドを再開するにはSoundManagerクラスの resutart メソッドを利用します．

```
public void processKey() throws Exception {
	if (model.isKeyPressed(KEY_UP)) {
		// 一時停止します
		SoundManager manager = SoundManager.getInstance();
		manager.pause(SoundManager.BGM_PORT);
	}
	if (model.isKeyPressed(KEY_DOWN)) {
		// 再開します
		SoundManager manager = SoundManager.getInstance();
		manager.restart(SoundManager.BGM_PORT);
	}
}
```

4.3.3 ボリュームを変更したい

ボリュームを変更するためには，SoundManagerクラスの setVolume メソッドを利用します．

```
public void processKey() {
	if (model.isKeyPressed(KEY_SOFT1)) {
		SoundManager manager = SoundManager.getInstance();
		manager.setVolume(0); // ミュートにする
	}
}
```

4.3.4 同期再生を行いたい(S!アプリ限定)

S!アプリでは1つのサウンドファイルで4和音までしか再生できないため，それ以上の音数を再生するために複数のサウンドファイルを再生する必要があります．同期再生を行うにはサウンドファイルを2つ用意し，SoundManagerクラスの play メソッドを利用します．

```
// ゲームモデルでサウンドを読み込んでおきます
protected void initializeModel() {
	SoundManager manager = SoundManager.getInstance();
	manager.loadSound("sample1.spf");
	manager.loadSound("sample2.spf");
}

// ------------------------------------------------------------------ //

// 実際にサウンドを利用する画面で，サウンドを再生します
public void processKey() {
	if (model.isKeyPressed(KEY_SELECT)) {
		// とりあえず簡易的に再生します
		Utility.playBGM("sample1.spf", "sample2.spf");

		// 再生条件を細かく決めて再生します
		SoundManager manager = SoundManager.getInstance();
		manager.play("sample1.spf", "sample2.spf", 5, true);
	}
}
```

なお，iアプリでは1つのサウンドファイルに利用できる音数が制限されていないため，同期再生を行う必要はありません．また，S!アプリでの同期再生は自動変換できないので，手作業で実装する必要があります．

4.4 データ入出力

ここでは，アプリが終了しても残しておきたいデータの保存方法や，外部からのデータの入力方法について解説します．

保存されたデータは，iアプリはスクラッチパッドと呼ばれるデータ保存領域(Windows上ではspフォルダ)に，S!アプリはレコードストアと呼ばれるデータ保存領域にバイナリデータとして置かれます．

4.4.1 データを保存したい

データを外部に保存するには，以下の手順を行います．

1. StoreManagerクラスのopenToWriteメソッドを使って，データ領域にアクセスして書き込みストリームを開く
2. StoreManagerクラスの各書き込みメソッドを使って，書き込みストリームにデータを書き込む
3. StoreManagerクラスのcloseToWriteメソッドを使って，書き込みストリームを閉じ，データを保存する

openToRead メソッドでは，引数として今から読み込みを行うデータ領域のアドレスを指定します．データ領域の大きさを超えてアクセスしようとすると，エラーが発生するので注意して下さい．データ領域の大きさを変更するには，節[4.4.3 データ領域の大きさを変更したい]を参照して下さい．

```
private int data = 10; // 書き込むデータ

protected void initializeModel() {
	StoreManager manager = StoreManager.getInstance();
	try {
		manager.openToWrite(0);
		manager.writeInt(data);
		manager.closeToWrite();
	} catch (Exception e) {
		e.printStackTrace();
	}
}
```

4.4.2 データを読み込みたい

データを外部から読み込むには，以下の手順を行います．

1. StoreManagerクラスのopenToReadメソッドを使って，データ領域にアクセスして読み込みストリームを開く
2. StoreManagerクラスの各読み込みメソッドを使って，読み込みストリームからデータを読み込む
3. StoreManagerクラスのcloseToReadメソッドを使って，読み込みストリームを閉じる

openToWrite メソッドでは，引数として今から書き込みを行うデータ領域のアドレスを指定します．データ領域の大きさを超えてアクセスしようとすると，エラーが発生するので注意して下さい．データ領域の大きさを変更するには，節[4.4.3 データ領域の大きさを変更したい]を参照して下さい．

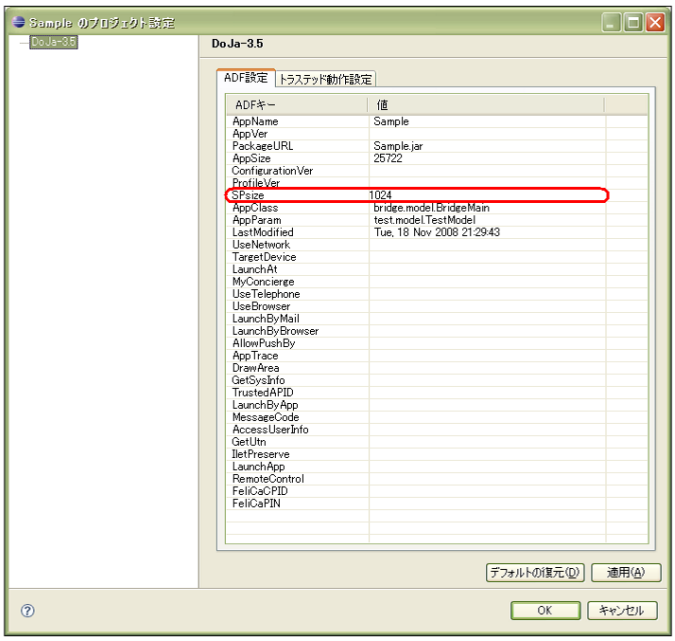
```
private int data; // 読み込んだデータ

protected void initializeModel() {
	StoreManager manager = StoreManager.getInstance();
	try {
		manager.openToRead(0);
		data = manager.readInt();
		manager.closeToRead();
	} catch (Exception e) {
		e.printStackTrace();
	}
}
```

4.4.3 データ領域の大きさを変更したい

iアプリではプロジェクトを選択し，メニューから「プロジェクト」→「DoJa-3.5 設定」を選択し，プロジェクト設定ダイアログを開きます．下図のようにSP-Sizeの項目に変更したい値(バイト)を設定します．Bridgeでは，デフォルトとして1024バイトに設定されています．

※SP-Sizeに設定できる値の上限は，400KB(409600バイト)です．

※スクラッチパッドを複数に分割することができます．その場合は，SP-Sizeを(1024, 1024, 1024)のようにカンマ区切りで記載します．分割したスクラッチパッドにアクセスする場合は，StoreManagerクラスの openToRead, openToWrite メソッドで，スクラッチパッド番号(0から始まります)を指定します．



S!アプリでは，JADファイルを開き，下図のようにMIDlet-Data-Sizeの項目に変更したい値(バイト)を設定します．Bridgeでは，デフォルトとして1024バイトに設定されています．

※MIDlet-Data-Sizeに設定できる値の上限は，512KB(524288バイト)です．

※レコードストアは複数に作成することができます．iアプリのスクラッチパッドとは異なり，StoreManagerクラスの openToRead, openToWrite メソッドで，レコードストアIDを指定すれば，自動的に別のレコードストアが作成され，データが保存されます．

4.4.4 ユーザからの入力を受け取りたい

ユーザから入力を受け取るためには，以下のような手順を行います．

1. AbstractModelクラスのimeOnメソッドを使ってユーザ入力画面を起動する
2. ユーザが「確定」または「取消」を選択すると，processIMEEventが呼ばれるので，メソッドをオーバーライドして処理を行う

入力の種類や文字数に制限を加えたい場合は，processIMEEventメソッド内に記述して下さい．BridgeではAPIを統一するために，ユーザ入力中にこれらを処理する機能をサポートしていません(※)．

```
private String name = "";

// ソフトキー2が押されたらIMEを起動します．
public void processKey() {
	if (model.isKeyPressed(KEY_SOFT2)) {
		model.imeOn(name, TextBox.DISPLAY_ANY, TextBox.KANA);
	}
}

// 名前の入力を受け取って変数に格納します．
public void processIMEEvent(int type, String text) {
	if (type == Canvas.IME_COMMITTED) {	// ユーザが「確定」を選択した場合
		input = text;
	}
}

// 名前が入力されていたら表示します．
public void draw(Graphics g) throws Exception {
	model.clearDisplay(Graphics.getColorOfName(Graphics.WHITE));

	if (!name.equal("")) {
		g.setColor(Graphics.getColorOfName(Graphics.BLACK));
		g.drawString("私の名前は「" + input + "」です", 10, 10);
	}
}
```

4.5 キー操作

ここではキー操作を判定する方法について解説します．取得できるキーの種類については，KeyEventListenerインターフェースを参照して下さい．

通常のキーの処理はiアプリもSIアプリも同じですが，ソフトキーの処理が若干ことなります．iアプリは通常のキーと同様に処理できますが，SIアプリはソフトキーラベルを貼るときにソフトキーの初期化を行うため，ラベルが貼られていない場合はそのソフトキーの処理が行われません．ソフトキーのラベルを貼る方法は，節[4.5.2 ソフトキーにラベルをつけたい]を参考にして下さい．

4.5.1 キー処理を判定したい

Bridgeではキーが押されたときの処理を各画面クラスの processKey に記述しますが，キーの処理方法として2種類あります．

- AbstractModelクラスのisKeyPressedメソッドを利用して，キーがその瞬間に押されたかを判定する
- AbstractModelクラスのisKeyPressingメソッドを利用して，キーが継続的に押されているかを判定する

```
int y = 200;

public void processKey() throws Exception {
	// 上キーが押されたら
	if (model.isKeyPressed(KEY_UP)) {
		y--;
	}

	// 上キーが押しっぱなしだったら
	if (model.isKeyPressing(KEY_UP)) {
		y--;
	}
}

public void draw(Graphics g) throws Exception {
	model.clearDisplay(Graphics.getColorOfName(Graphics.WHITE));

	g.setColor(Graphics.getColorOfName(Graphics.BLACK));
	g.drawRect(100, y, 10, 10);
}
```

4.5.2 ソフトキーにラベルをつけたい

ソフトキーにラベルを貼る場合は，AbstractModelクラスの setSoftLabel メソッドを利用します．引数として左右のソフトキーラベルに設定する文字列を渡しますが，空文字を渡すとラベルを消すことが出来ます．もし，片方のラベルだけを変更したい場合には，変更したくない方のラベル文字列にnullを渡してください．

ラベルに利用できる文字列は半角4文字(全角2文字)以内にして下さい．それ以上だと，端末によっては表示できない可能性があります．

```
protected void entryAction() throws Exception {
	model.setSoftLabel("終了", "ﾒﾆｭｰ");
}
```

4.6 デバッグ機能を使いたい

ここでは，Bridgeが持つデバッグ機能と，各キャリアのエミュレータが持つデバッガについて解説します．

4.6.1 Bridgeのデバッグ機能を使いたい

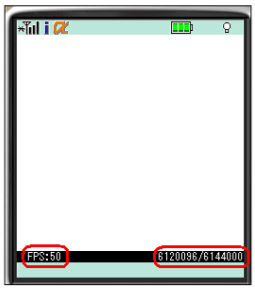
Utilityクラスの setDebugMode でデバッグ機能を有効にすると，以下のような機能が使えるようになります．

- コンソールに出力を行う専用のメソッドを使えます
- 画面上に，現在のフレームレートとメモリ使用量が表示されます

コンソールへの出力は System.out.println メソッドを使っても実現しますが，専用のメソッドを使うことによって デバッグ機能を無効にするだけで，すべてのコンソール出力を切ることが容易に行えます．

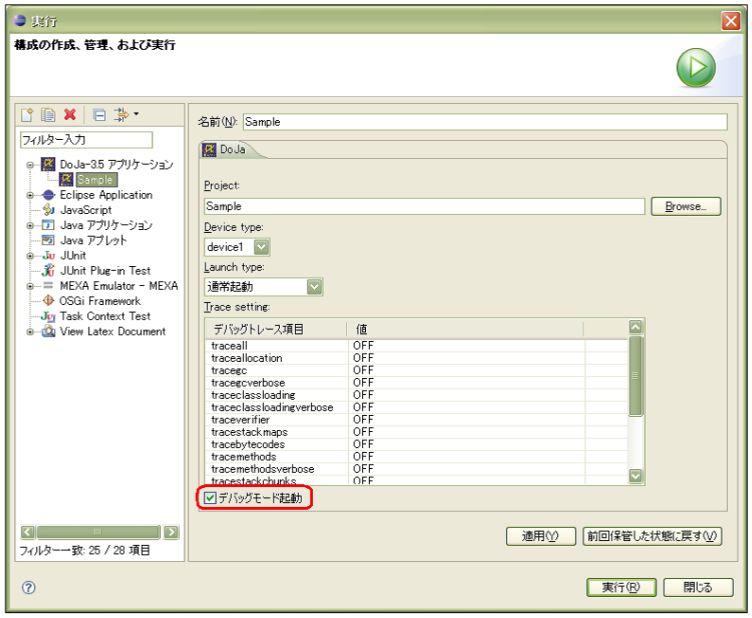
```
protected void initializeModel() {
        // System.out.printlnの出力結果だけが表示されます．
        System.out.println("1. デバッグ機能：無効");
        Utility.println("2. デバッグ機能：無効");

        Utility.setDebugMode(true);
        System.out.println("------------------------");

        // 両方の出力結果が表示されます．
        System.out.println("3. デバッグ機能：有効");
        Utility.println("4. デバッグ機能：有効");
}
```

```
1. デバッグ機能：無効
------------------------
3. デバッグ機能：有効
4. デバッグ機能：有効
```



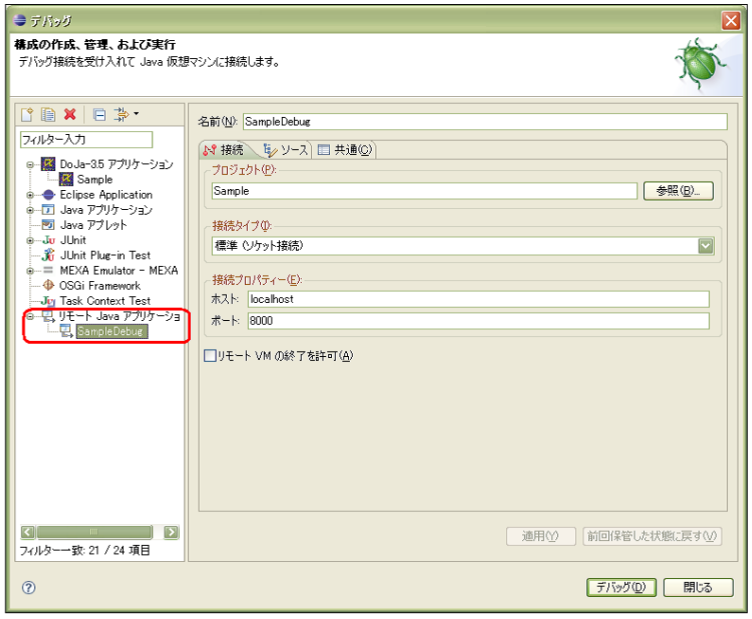
4.6.2 iアプリのデバッガを使いたい

iアプリのデバッガを使うには，メニューの「実行」→「実行ダイアログを開く」を選択し，実行ダイアログを開きます．そこで，下図のように「Doja-3.5アプリケーション」を選択し，「デバッグモード起動」にチェックをして『実行』を押下します．

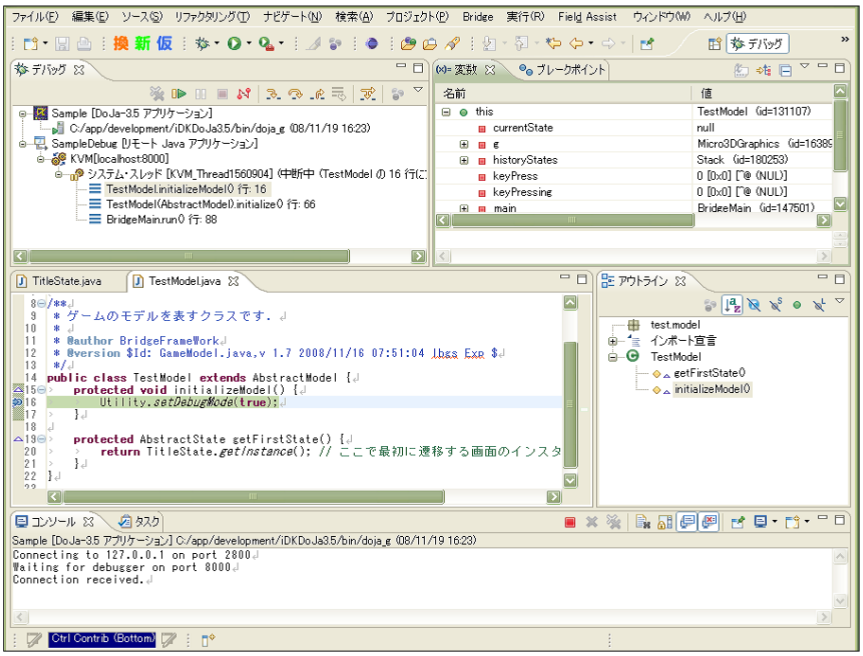


デバッガとの通信準備ができると，コンソールに以下のような文字が表示されます．ファイアウォールソフトなどでポートが空いていないとデバッガとの通信に失敗にすることがあります．

上記表示を確認したら，再びメニューから「実行」→「デバッグダイアログ」を選択し，デバッグダイアログを開きます．そこで，下図のように「リモートJavaアプリケーション」を選択し，『デバッグ』を押下します．



あとは，実行時と同様にエミュレータが実行されます．eclipseで通常のデバッグを行うように，ブレークポイントを設定して，その状態でプログラムを止めて，変数の状態などを確認することができます．



4.6.3 S!アプリのデバッガを使いたい

S!アプリのデバッガを使うには，jadファイルを右クリックして「デバッグ」→「MEXA Emulator」を選択するだけです．後は，iアプリのデバッガと同じように，利用することができます．

※S!アプリのデバッガは安定しておらず，デバッガが起動しなかったり，突然強制終了する場合があります．その場合は，再起動して下さい．

※MEXAエミュレータのメイン液晶ウィンドウに「Launching...」と表示されたままになる，または「リモートVMに接続できませんでした。接続がタイムアウトしました。」と出る場合は コチラを参照して下さい．

4.7 実機で動かしたい

4.7.1 iアプリを実機で動かしたい

iアプリを実機で動かすには，binフォルダ以下のすべてのファイル（Download.html, jamファイル, jarファイル）をサーバの同じ階層にアップロードします．あとは，実機からDownload.htmlファイルにアクセスすればアプリがダウンロードされます．

4.7.2 S!アプリを実機で動かしたい

iアプリと異なり，S!アプリを実機で動かすには，専用のサイトを経由してダウンロードする必要があります．有名どころだとアプリ☆ゲットがあります．ただし，アプリ☆ゲットは公開・配布する意思があることを前提とおり，登録も必要です．実機で動作させたい場合は，規約を良く読んで利用してください．

4.8 その他

ここでは，上記以外でアプリ開発に役立つことを解説します．

4.8.1 乱数を取得したい

Utilityクラスの nextInt メソッドを利用します．

```
protected void initializeModel() {
	// 0以上，100未満の整数を取得する
	int randomNumber = Utility.nextInt(100);
}
```

4.8.2 前の画面に戻りたい

AbstractModelクラスの transitionAndKeep メソッドを利用すると画面遷移を行うと同時に，現在の画面を履歴として保存することができます．履歴はスタックで保存され，transitionToHistory メソッドを利用することで，前の画面に戻ることができます．遷移の詳細に関しては，API Refereneceをご覧下さい．

```
// 戻ってくる可能性のある画面は，遷移時に履歴を保存するようにする
public void processKey() throws Exception {
	if (model.isKeyPressed(KEY_SELECT)) {
		model.transitionAndKeep(GamePlayingState.getInstance());
	}
}

//----------------------------------------------------------------------//

// 履歴に保存されている画面に戻るには，transitionToHistoryを利用する
public void processKey() throws Exception {
	if (model.isKeyPressed(KEY_SELECT)) {
		model.transitionToHistory(false);
	}
}
```

4.8.3 アプリを終了させたい

AbstractModelクラスの exit メソッドを利用します．

```
public void processKey() throws Exception {
	if (model.isKeyPressed(KEY_SOFT1)) {
		model.exit();
	}
}
```

※各キャリアのAPIを参考に実装することは可能ですが，変換機能が対応していないため，独自に変換する必要があります．

5. その他の情報

5.1 リンク集

- Bridge API Reference
- DoJa 3.5(iアプリ) API Reference
- MIDP(S!アプリ) API Reference

付 録B　Bridge実験仕様書

シューティングゲーム仕様書

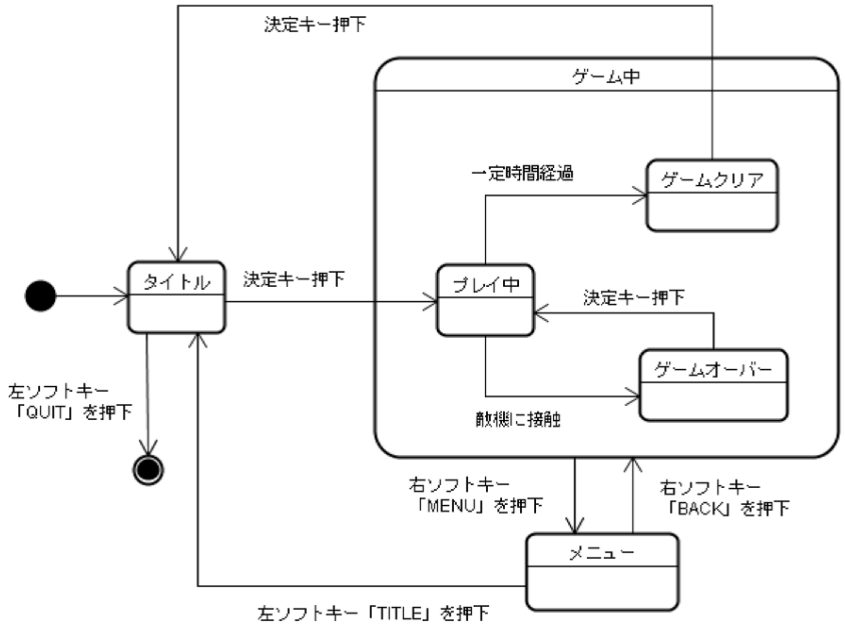
1　概要

・横スクロールシューティングゲーム（避けゲー）
・画面は以下の画面遷移図のように遷移する（メニューの文字列などは変更して良い）
・各画面の仕様を参考に実装を行う
・画面仕様に書かれていないことは自由に仕様を決めて，実装して良い

2　実装手順（目標）

2.1　タイトル画面を表示する
2.2　自機の描画・操作ができる
2.3　敵機の描画・動作と，クリア画面ができる
2.4　敵機との衝突判定が実装されて，メニュー画面を除いたゲームが作れる
2.5　仕様書通りのゲームが作れる

3　画面遷移



4　画面仕様

4.1　タイトル画面例

■表示
・タイトルが画面中央に表示されている

■キー操作
・決定キーで，ゲームが開始する
・左ソフトキー「QUIT」で，ゲームが終了する

4.2　プレイ中画面例

■表示
・画面左側に自機が表示される
・画面右側から敵機がランダムに出現し，自動的に直進する

■キー操作
・上下左右キーで，自機を操作できる
・右ソフトキーで，メニュー画面へ

■その他
・一定時間敵機を避け続けるとゲームクリアとなる
・敵機に接触するとゲームオーバーとなる

4.3　ゲームクリア画面例

■表示
・ゲームをクリアしたことが画面中央に表示される

■キー操作
・決定キーで，タイトル画面へ
・右ソフトキー，メニュー画面へ

4.4　ゲームオーバー画面例

■表示
・ゲームオーバーしたことが画面中央に表示される

■キー操作
・決定キーで，ゲーム画面へ（リトライ）
・右ソフトキー，メニュー画面へ

4.5　メニュー画面例

■表示
・メニューであることが画面中央に表示される

■キー操作
・右ソフトキーで，一つ前の画面に戻る
・左ソフトキーで，タイトル画面へ

5　その他の設定

Bridge を使わない場合には，i アプリをエミュレータで実行するために，以下の設定をする必要がある．

5.1　コンパイルバージョンの設定

i アプリのエミュレータは Java 1.3 以下でコンパイルしないと動作しない．そのため，プロジェクトのコンパイルバージョンの設定を変更する必要がある．

（1）　プロジェクトを選択して右クリックし，「プロパティー」→「Java コンパイラ」を選ぶ
（2）　「プロジェクト固有の設定を可能にする」にチェックをする
（3）　コンパイラ準拠レベルを 1.3 に変更する
（4）　生成された.class ファイルの互換性が 1.1，ソースの互換性が 1.3 であることを確認する
（5）　OK を押下する

5.2　i アプリプロジェクトの設定

i アプリを実行する際に，i アプリの情報を設定ファイルに記述する必要がある．

（1）　プロジェクトを選択して，メニューの「プロジェクト」→「DoJa3.5 設定」を選ぶ
（2）　ADF 設定タブで以下の項目を入力する
・AppName：アプリの名前（何でも良い）
・AppClass：IApplication を継承しているクラス名

付 録C　試用実験の感想

ここでは，本文中 6 章の 6.2 節で述べた試用実験における，被験者たちの感想の全文を紹介する．アルファベットで表記されている被験者の名前は，本文中の表と対応している．なお，感想はほぼ原文のままであるが，被験者が特定できるような記述，本文中と違った意味で使われている用語などについては加筆修正を加えてある．

被験者 A

- Bridge を利用した方が時間が短縮されたけれど、理由は Bridge 未利用版を作ったことで、必要なコード断片が用意されており、それを貼り付けることが、大幅な時間短縮につながったからである。

- ただし Bridge を利用することによる「気持ちよさ」「快感」は存在していた。これは、Bridge 未利用版の場合は自分でフレームワークを考案・実装し、そこに必要な処理を記述した。Bridge を利用すると必要なフレームワークが全て用意されているので、適切な場所に適切なコードを差し挟むだけで良かったのが「安心」につながった。

- コメント記述の必要性が少なかったのも「気持ちよさ」「安心」につながった。Bridge 未利用版は自作フレームワークなので、フレームワークの各部の意味を詳細にコメントとして記述する必要がある。もちろん今回のテストではコメントは書かなかったが、「いつかコメントとして書かなくちゃ...」というある種のやましさのようなものが残る。Bridge 利用版は、そういったやましさがほとんど残らなかった。分けるべき構造がきっかりと分かれていて、特に entryAction、processKey、action、draw の各メソッドは、私が Bridge 未利用版で自力で分割していたブロックだった。その分割と処理をフレームワークに委譲できるのはとても安心できて、私は個別具体的な処理の記述に集中することが出来た。

- transition の代わりに transitionAndKeep() と transitionFromHistory() を使う案だが、アイデア自体は面白い。ただし transitionFromHistory() をしたときに、entryAction() を行うかどうかを引数の boolean 値でやるのは、結構微妙

である。理由は、transitionAndKeep() が生じることで、２種類の entry が生じている。つまり通常の遷移による entry と、transitionFromHistory() による entry。この２種類の entry に対応したメソッドを作るべきかもしれない...。が、難しい判断だと思う。transitionAndKeep() が多用されるのであれば、entry は２種類あった方が良い。だがそんなに使わないのであれば...、現在の実装のままで良いのかもしれない。

- package をクリックして「新」ボタンを押すのは、インターフェースとしては分かりにくい。eclipse 文化に従うなら、右クリックから「新規作成」で、BridgeState を作る、というのが常道ではないだろうか。

被験者 B

- Bridge を用いないシューティングゲームの作成
  ゲームを制作する上で、画面によって処理をわけるようにしている。まずは、画面の管理やキー入力などの基礎作りを 30 分かけて作成した。
  - 仕様書の把握に 5 分かけた。
  - 当り判定の構想に 5 分かけた。

  setSoftLabel() の設定では、表示させるソフトキーは問題ないのだが、表示させないソフトキーの場合に設定忘れがあった。Bridge では一度に両方のソフトキーのラベルを設定するため、このようなヒューマンエラーは起こらない。画面の状態を switch() で管理していたため、さらに画面が増えたり処理が増えてしまうとメソッドが長くなり、状態の把握が困難になると考えられる。

- Bridge を用いたシューティングゲームの作成

  - 先に「Bridge を用いないシューティングゲーム」を作成していたため、仕様書の把握、当り判定モデルの構想にかかる時間が省略された。
  - Bridge の使い方に 5 分、Bridge のモデル概要の把握に 5 分から 10 分を費やした。

  Bridge マニュアルは要所要所で読む時間はあったが、やりたいことを行うには Bridge API リファレンスを読む必要があった。しかし、読む時間がほとんどなく、「おそらくこういう動きをするだろう」という形でコピペでゲームを制作していたため、悪い開発の仕方を取っていた（ただし、テスト制作ということ、時間が限られているということで、この開発スタイルを変更する必要があ

る大きなデメリットはなかったと考えられ、マニュアルを読めば基礎的なことができた)。
processKey() に主要なキー押下処理が予め設定されているので入力の手間が省け、かつ draw() 内に書いてしまうといった間違いを抑制できる。

- 問題に感じたこと
  グラフィックスの取得方法が分からなかった。また、今回、機体をユニットとして扱うクラスを作成したが、どこのパッケージ内に置けばよいのかが分からなかった。

- 総評
  Bridge の使い方も分からない状態では、今回のシューティングゲーム程度の規模のゲームを開発する効率は上がらない。むしろ、Bridge の使い方を覚える時間が生じたため、かえって時間を費やすことになった。しかし、さらに規模の大きいゲームとなると画面の管理が煩雑になるため、クラスによって画面の状態を管理するという考えは非常に考えやすく、また、扱いやすかった。
  また、Bridge の重要な機能の 1 つに Softbank 版にそのまま移植できることにより、SoftBank 版への移植の時間が短縮されるだろう。もちろん、Bridge の使い方を覚えれば問題は解消されると考えている。
  したがって、Bridge を用いると利点がある場面は、
  - ステージが複数あるなど、より大きな規模での開発
  - SoftBank 版への移植がある場合

  と考えられ、また、これらこそが重要であるため、今回のテストで判断できないのが残念である。

- 要望
  - グラフィックスの取得方法をマニュアルに記述して欲しい
  - model.getLastState() の使い方をマニュアルに記述して欲しい
  - javadoc ロケーションに Bridge API リファレンスを自動で追加してほしい

被験者 C

フレームワークを使ったら、作業時間が大幅に縮み、かなり楽に作業ができました。画面遷移が特に楽でした。画像の表示と、キー操作の設定も、地味に楽になりました。

あらかじめ、どのメソッドに何を書いたらいいかがくっきり分かれていて、しっかり説明もされているので、ソースの読みやすさ（書き込みやすさ）が上がったと思います。

ただ、メソッド数が多いので、各メソッドの説明をしっかり読まないと、どこに何を書いたらいいのかがわかりにくいのが、ちょっと大変なところかなと思いました。一目でパッとわかれば、もっと便利だったと思います。今回は橋山[1]さんに聞きまくってしまったのでそんなにそこで時間を取られることはありませんでしたが、ひとりで実装するとなると、慣れるまでは時間を取られるかも。

逆引きリファレンスはとても心強かったです。コピペができるということはありがたいことですね。リファレンスに載っているサンプルプログラムは、仕切り線の上と下で違うクラスに書くものだというのは、おそらく橋山さんに言われなければ気づかなかったかも...と思いました。

被験者 D

- 良い点
    - 遷移時の処理、ループごとの処理、キーの処理、描画の処理などが予め分けられている点（意識していないで書くとごちゃ混ぜのプログラムになってしまいがちなので、わかりやすい）
    - キーの処理が、予め書かれているので、いちいち調べなくて良い点
    - 画面遷移の為のメソッド、保存しつつ遷移のメソッド（→これも自分で書くと非常に煩雑になってしまいます）
    - 一番便利だと思ったのは、画面遷移のためのメソッド 3 つです。
- 悪い点
    - 画像読み出しの仕方はマニュアルのみだと躓く恐れがあります
    - ソフトキーと、ラベルの関係が分かりづらかった、というか知らなかったという部分が大きいのかもしれません

被験者 E

- Bridge 利用版と Bridge 未利用版の比較について
    - 意外だったこと
      所要時間があまり変わらなかったのが意外だった。しかし、Bridge 利用版でアルゴリズムを考えていた時間があり、Bridge 未利用版のほうでは

[1] Bridge 開発者である著者のこと.

その時間はなかったので、実質は Bridge 利用版のほうが十分ほど短いのではないか、と思った。

- 嫌だったこと
  ひたすら長いメソッドを書くのが嫌だった（Bridge 未利用版）。このコードをメンテしろと言われたら断りたい。
- オブジェクト指向
  オブプロ[2]での Shooting ゲームと似ていたので、PlayerAircraft なんかを作りたくなりましたが、今いちこのフレームワークのなかでどう活かしていいのか分からず...。サンプルが欲しいです。せっかく「きれいに書く」フレームワークを目指すなら、ベタ書きではもったいないような気が。
- Bridge への疑問
  今回は１面だけでしたが、同じ機体が１，２，３面と出てきたり...。という場合どう書けばいいのかな...。やはりオブジェクト作るべきでしょうか。画面間でのオブジェクトの受け渡しはどうやるんでしょうか。
- Bridge への疑問（キー操作）キー押しっぱなしのメソッドがテンプレに出てこないのはなぜでしょう。

- Bridge の教育効果について
  Bridge 利用版で、画面が状態であり、画面ごとに描画、処理のループを持つ、というモデルを学んで、それを Bridge 未利用版でのプログラミングに活かしていた気がする。

被験者 F

１は Bridge 未利用、２は Bridge 利用です。

- １では一つ一つの動作が何をしているのかわからなかった。どこにどんな処理を書けばどう動くのかがわからなかった。２は１に比べて、何がどう遷移しているのかとか、どの部分でどう実装されたのがどう反映されるのかがわかったため、すんなり作れた。
- ２は画面の管理がわかりやすく、処理が大変わかりやすい。
- １はきれいにかけない。きれいに書くのがものすごく時間掛かると思う。というか、きれいに書くことを考えるのにものすごく時間がかかる。開発以外のと

[2]慶應 SFC で実施されているオブジェクト指向プログラミングという授業のことで，松澤芳昭氏が担当している．

ころというか、アルゴリズム以外の部分に手間がかかる。ということは、重複とか無駄な処理とかとても無駄なことを沢山やっていると思う。

- １のコードはもう見たくない。本当に見たくないです。２を先にやったから１をやる気になった。（内部処理がどうなっているのか自分の中におとせたため、やれるかなという気になった）
- １は、サンプルがないとできない。２はサンプルがなくてもできる
- ２に「画面に移ったときの処理」「画面から遷移するときの最終的な処理」があるのがよかったイメージとして脳内に描きやすい。いつ、どんな処理をするのか
- １はコメント書いておかないとどこでどんな処理してるのかわからなくなってくる。あれってどこ書いたっけ...ってなる。
- ２ではどの画面を書いているのかわかりやすい。よし、ここ書くぞという気になる

被験者 G

Bridge を使わないときは、「どの処理をどこに書けばいいのか」わからず、時間がかかっていたが Bridge を使うとそれがすごくわかりやすかった。「○○の時」という記述が用意されていることが多かったので、処理を行いたいところに「○○する」という処理を書けばよい、という感じで進められた。またステージごとにクラスが別に作れ、それ同士の遷移も簡単にできるのが非常にわかりやすくて良かった。私は一度 Bridge を使ってゲームを完成させた後で、Bridge を使わない版をやったのだが、やはり画面遷移やキーイベントの処理が非常に面倒で、時間もかかってしまった。また Bridge を使うことで時間が大幅に短縮できただけでなく、プログラム自体もかなり見やすく、きれいになったと思うし、変更したい部分が出来た時も容易にできると思う。

被験者 H

- Brige の良い点
  ゲーム開発で画面の状態遷移はありがちな処理であり、１つのファイルにこれをすべて書いていくと可読性が悪い。Bridge では一つの状態に対して一つのファイル（クラス）という設計なのでソースがきれいになってよい。

- Brige の改善点
  キーイベントの if 文はすべて書かれていなくてもいいかも。無駄になるとこが多いから、ソースがきれいにならない。

被験者 I

- メリットを感じた点
  Bridge を使用した場合、ゲームの骨組み（キーイベントの処理、イメージリソースの読み込み処理等）があるため、まずその点の実装の時間が大幅に短縮出来た。また、あらかじめゲーム作成に必要となるメソッドが用意されているため、あまりプログラミング経験のない人でも容易にゲームアプリが作成できると思う。次にループ中の処理に関して、遷移する画面を自動で生成してくれるため、switch 文でゲーム状態を変更するよりもソースが見やすくなり、各画面中の処理が分かりやすく書けるようになっている。

- デメリット
  今回のテスト中では特にデメリットを感じた点はなかった。

業務用として Bridge を使う場合実機の場合機種依存の問題や、Bridge で対応してない機能（HTTP 接続やカメラ機能）があるので、そのための拡張方法などについてマニュアル or 仕様書があると嬉しいです。また、i アプリから S!アプリへの変換は普通に業務で使いたい機能だと思いました。