Heijo: 動的なコード実行可視化によるJava/Androidアプリケーションのリアルタイムプロファイラ

大神 勝也　中才 恵太朗　畑 秀明　松本 健一

アプリケーションのパフォーマンス改善において，実行時間を解析するプロファイラは有用と思われる．しかし，既存のプロファイラは特定の実行シナリオのもとプロファイリング時間を事前に設定する必要があり，実行シナリオなしでシステムのボトルネックを見つけることは難しい．また，各メソッドごとの実行時間を表示するインタフェースはソフトウェアの階層構造上における実行モジュールの位置などを把握することが難しい．これらの課題に対処するため，リアルタイムでパフォーマンス分析可能なソフトウェア都市可視化ツール Heijo を提案する．提案するプロファイラでは，アプリケーションの実行は 3 次元のソフトウェア都市として可視化され，アプリケーションのソフトウェア構造と実行のパフォーマンスが表現される．提案するプロファイラを使用して実際の Java アプリケーションおよび Android アプリケーションのプロファイリングを行い，提案するプロファイラの有用性と実用性を確認した．

For software performance improvement, a profiler allows developers to quickly search and identify bottlenecks and leaks that consume much execution time. However, existing profilers require specific execution scenarios to set profiling periods, which is difficult to find out the bottlenecks of the systems. In addition, a current interface showing a list of individual executed methods does not help developers easily understand hierarchical namespace positions of executed modules. To address these issues, we propose a real-time profiler for Java and Android applictions with city-like visualization of dynamic code executions. With our profiler, program executions are visualized as a three-dimensional software city representing the structure and performance of the application. In our case studies, we profile the real Java and Android applications with our profiler to confirm its practicality and effectiveness.

1　はじめに

アプリケーションのボトルネックやメモリリークなどのパフォーマンスの問題を早期に発見するためにプロファイラが用いられる．2015 年のオンラインレポート [16] によると，最もよく使用される Java アプリケーション用のプロファイラとして，VisualVM[†1] と Jprofiler[†2] が挙げられている．いずれのツールも，

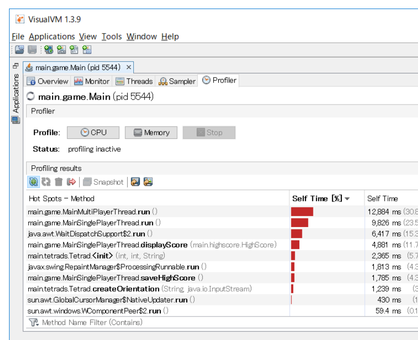

図 1　**Java 仮想マシン用プロファイラ VisualVM**

Java 仮想マシンに関する様々な統計情報を収集しながら，コード実行のプロファイリングやスレッドダンプおよびヒープダンプの収集とブラウズ機能をアプリケーションの開発者に提供する．

しかし，VisualVM や JProfiler などのプロファイ

Heijo: A Real-time Profiler for Java and Android Applications with City-like Visualization of Dynamic Code Executions

Katsuya Ogami, Keitaro Nakasai, Hideaki Hata, Kenichi Matsumoto, 奈良先端科学技術大学院大学, Nara Institute of Science and Technology.

コンピュータソフトウェア, Vol.26, No.0 (2009), pp.1–13.
[ソフトウェア論文] 2017 年 1 月 10 日受付.

†1　`https://visualvm.github.io/`

†2　`https://www.ej-technologies.com/products/jprofiler/overview.html`

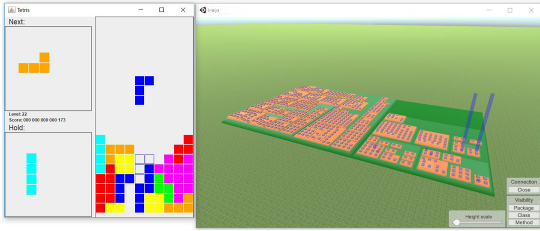

図 2　提案するプロファイラ：(左) プロファイリング対象のテトリスアプリケーション (右) Heijo

ラには次の 2 つの欠点が存在する．第 1 の欠点は，プロファイリングのために特定の実行シナリオを必要とする点である．これらのツールによるプロファイリングは，プロファイリングの開始と終了のタイミングを決める必要があり，プロファイリングの結果は，計測時間のうち各メソッドが占める実行時間の割合として出力される．そのため，実行シナリオなしでシステムのボトルネックを見つけたい場合には，これらのプロファイラを用いて適切にプロファイリングを行うことが難しくなる．

第 2 の欠点は，これらのツールによって提供されるユーザインタフェースにある．図 1 に示すように，これらのツールで表示される典型的な画面では，メソッド名と棒グラフのセットが表示される．これは直感的ではあるが，ソフトウェアの階層構造における（メソッドやクラスの）位置を把握することは容易でない．さらに，実行されるメソッドの数が多い場合，すべてのメソッドの情報を適切に表示することは難しく，ユーザが名前による情報のみから多数のメソッドを識別することは困難である．

本稿では，これらの欠点に対処するために，リアルタイムでパフォーマンスを示す 3 次元の都市可視化ツール Heijo を提案する．図 2 に示すように，提案するプロファイラはアプリケーションの実行中に可視化を瞬時に更新し，アプリケーションのソフトウェア構造をパフォーマンスと共に表示する．提案するプロファイラの実用性と有用性を評価するために，実際の Java アプリケーションと Android アプリケーションを対象にプロファイリングを行った．

本稿の構成を次に示す．第 2 節では，提案するプロファイラ Heijo の設計と詳細について述べる．第 3 節では，Heijo の実装について述べる．第 4 節では，Heijo の実用性と有用性を実証するためのケーススタディについて述べる．第 5 節では，いくつかの考察について述べる．第 6 節では，既存のプロファイラおよび関連研究について述べる．最後に，第 7 節で結論を述べる．

本稿は，Java のみを対象としたプロトタイプ発表 [12] と比べ，詳細な設計のもと本格的に実装し，規模の大きなアプリケーションでも適用した結果を報告するものである．Heijo のプログラムとソースコードは GitHub で公開している [†3]．

2　Heijo の設計

2.1　可視化のデザイン

Heijo は，アプリケーションのソフトウェア構造と，単位時間あたりの実行時間を可視化する．実行状態を示す方法は，色や明度で示す方法も考えられる．しかし，本プロファイラでは，ブロックの種類（メソッド，クラス，パッケージ）を区別するために色情報を採用した. そのため，色や明度で実行状態を表現すると，情報が過多になると考えたため，ブロックの高さを実行状態を示す方法として採用した．ブロックの高さを実行状態の表現として利用するメリットは，プロファイルする対象が大きい場合であっても水平にカメラ操作を行えば，実行箇所が容易に特定可能であること，くわえて，容易にメソッドの実行時間の差を比較することが可能であることである．

ソフトウェア構造の表現

Heijo のソフトウェア構造可視化の例を図 3 に示す. メソッドは床面積一定の紫色のブロックで表され，クラスはオレンジ色のブロック，パッケージは緑色のブロックとする．実行時間は青色のブロックで表されている．図の例ではメソッドの実行時間が表されている．アプリケーションのソフトウェア構造は，これらのブロックの上下関係によって表現される．また，大規模なアプリケーションの可視化でも画面内に収めやすくするため，全体の面積が小さくなるようにブロッ

†3　https://github.com/k-ogami/Heijo

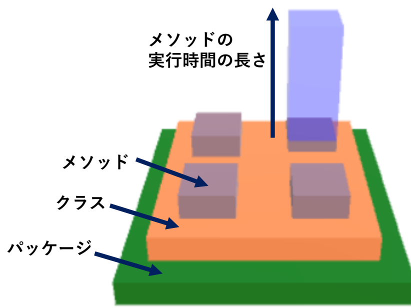

図 3 Heijo のソフトウェア構造可視化の例

クを配置する．本システムはソースコードを都市のメタファーで表現しているため，以下のようにブロックのカテゴリー色を選択した．メソッドと実行時間はビルを表現する紫色と青色とした．クラスはビルの間のレンガ道を表現するオレンジ色，パッケージは庭園を表現する緑色とした．また，ブロック選択時の色をビルに電気が付いたことを表現する黄色とした．図 6 はブロックの選択時の状態を示す．

単位時間あたりの実行時間の表現

図 4 に示すように，ブロックの高さは，固定時間フレーム L（本稿の実験では 1 秒に設定した）において各メソッドの実行時間が占める割合によって計算される．図 4 の場合，L の間に A メソッドが実行されていた時間は，t_1 から t_2 までの間と，t_3 から t_4 までの間である．したがって，A メソッドのブロックの高さは，$((t_2 - t_1) + (t_4 - t_3))/L$ となる．なお，マルチスレッドアプリケーションの場合はメソッドの実行時間をスレッドごとに別々に計算し，そのうち最も実行時間の長い値をブロックの高さとする．

2.2 ツールインタフェース

アプリケーションの選択

Java アプリケーションのプロファイリングの場合，コマンドライン上でアプリケーションを選択してプロファイリングを開始する．たとえば，“java -javaagent:profiler.jar -jar target.jar” のようなコマンドを実行する．profiler.jar は Heijo のプログラム，target.jar はプロファイリング対象として選択するアプリケーションのファイルパスである．

Android アプリケーションのプロファイリングの場合，図 5 に示す設定画面を操作してプロファイリングの設定を行う．この画面では，プロファイリング対象とするアプリケーションの選択（端末にインストール済みのアプリケーションの中から選択できる），可視化を行うコンピュータとの接続先の設定，プロファイリングの計測の対象から除外するパッケージの名前の入力を行うことができる．下部のボタンを押すと入力した設定内容でプロファイリングを開始する．

特定のパッケージの除外

Java アプリケーションのプロファイリングの場合は付属する設定ファイル，Android アプリケーションのプロファイリングの場合は図 5 の設定画面にパッケージ名を入力して，特定のパッケージをプロファイリングの計測の対象から除外することができる．除外されたパッケージのブロックは可視化画面で表示されず，除外されたパッケージのメソッドの実行時間は，そのメソッドを呼ぶメソッドの実行時間に含まれるようになる．この機能は，Android アプリケーションに標準で含まれる Android Support Library など，アプリケーションにとって主要ではない外部ライブラリを除外するために有用である．

可視化画面上の操作

可視化画面では主にマウスを用いて操作できる．可視化画面上で行える操作を以下に示す．

- カメラ操作：Heijo のカメラインタフェースは Unity[†4] などの既存の 3D 編集ツールのカメラインタフェースと基本的に同じであり，マウスのホイールと右ボタンを使用して，カメラの前後移動，水平移動，回転の操作ができる．
- 詳細な情報の表示：マウスカーソルをブロック上に合わせると，そのブロックが黄色となり，そのブロックの情報を示す詳細パネルが表示される（図 6）．詳細パネルには，選択されたブロックの名前（クラス名，メソッド名，パッケージ名），ブロックの高さの値，メソッドを同時に実行しているスレッドの数が表示される．クラスあるいはパッケージにマウスカーソルを合わせた場合，

†4 https://unity3d.com/jp

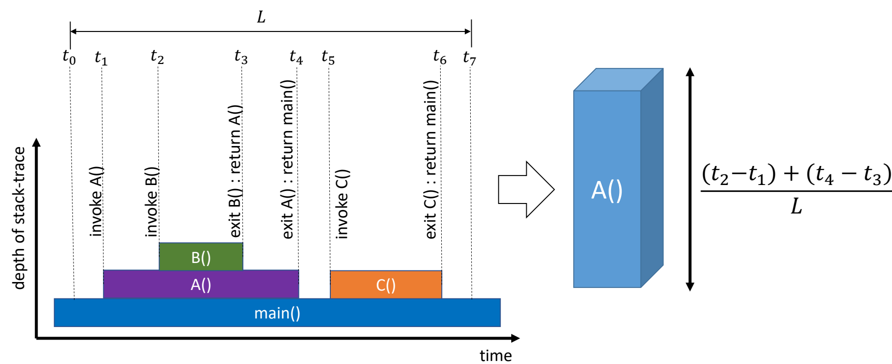

図 4　ブロックの高さの計算の例：(左）メソッド実行の時系列（右）時刻 t_7 における **A** メソッドの高さ

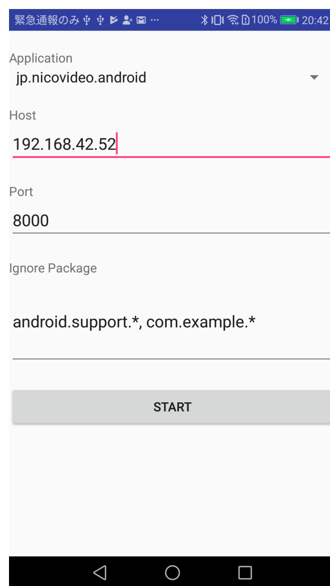

図 5　プロファイラの設定画面（**Android** 用）

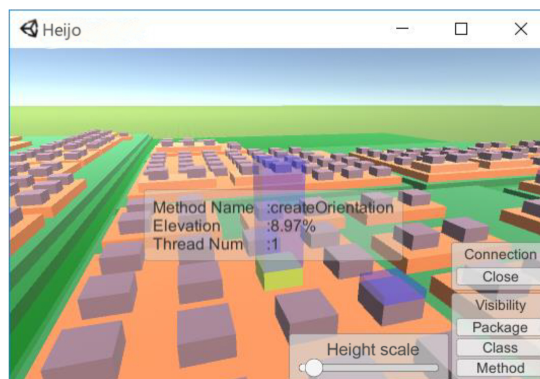

図 6　詳細な情報の表示

表示される高さの値は，それに属しているすべてのメソッドの実行時間をスレッド別に合算し，その中で最大となる値が計算に使用される．一方，スレッドの数は，属しているメソッドが実行されているスレッドの数である．

- ブロックの表示レベルの変更：図 7 に示すように，クラスやパッケージの「折り畳み」によってブロックの表示レベルを変更することができる．折り畳みは，ブロックをマウスの左ボタンでダブルクリックすることによりブロックを個別に折り畳むか，画面右下のボタンを操作してパッケージレベル，クラスレベル，メソッドレベルから選んでブロックを一括で折り畳みを行う．ただし，詳細パネルに表示される値と同様に，クラスやパッケージの高さの計算に使用される値はあくまでスレッド別に合算した実行時間の最大値であり，ブロックの幅は実行時間の長さに無関係であることに注意が必要である（すなわち，体積が大きいブロックほど実行に時間が掛かっているわけではない）．
- ブロックの高さ調整：可視化画面の下部にあるスライドバーを操作して，すべてのブロックの高さに定数倍の補正をかけることが可能である．この機能により，ユーザはカメラとブロックの距離やアプリケーションの規模に応じて見やすい高さになるように調整することができる．

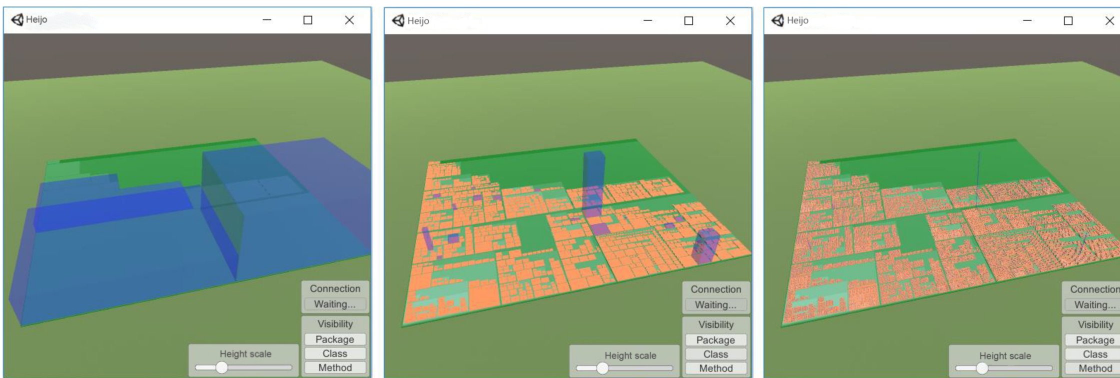

図 7 ブロックの表示レベルの変更：（左）パッケージレベル（中）クラスレベル（右）メソッドレベル

3 Heijo の実装

Heijo は，アプリケーションの実行を監視しプロファイリング情報の計測を行うためのプログラムと，得られたプロファイリング情報を可視化するためのプログラムによって構成される．ここでは，これらの 2 つのプログラムをそれぞれ計測プログラムおよび可視化プログラムと呼ぶ．これらのプログラムの実装について以下に示す．

3.1 計測プログラムの実装

計測プログラムは，Java アプリケーション用と Android アプリケーション用とで，それぞれ別々に実装した．いずれも Java で記述し，規模はそれぞれ 700 行程度である．

計測プログラムの実行方法

アプリケーションのプロセスにあるスレッドを監視してプロファイリング情報を取得するためには，計測プログラムはアプリケーションと同じプロセスで実行される必要がある．Heijo では，Java アプリケーションの場合と Android アプリケーションの場合で，それぞれ異なる方法を用いて計測プログラムを実行する．

Java アプリケーションの場合，java コマンドがサポートする javaagent オプションを利用する．javaagent オプションは，Java アプリケーションに対して任意の計測を行うために用意された機能である．Heijo における Java アプリケーションの計測では，javaagent オプションを利用して Java 仮想マシンの開始の直後に計測プログラムを実行する．

Android アプリケーションの場合，Xposed Framework[†5] を利用する．Xposed Framework は，Android アプリケーションの任意のメソッドに対して変更を行うことができるフレームワークである．Heijo における Android アプリケーションの計測では，Xposed Framework を利用してアプリケーションが実行される直前に計測プログラムを実行する．具体的には，android.app.Instrumentation クラスの newActivity メソッドの直後に割り込みを行う．

これらの方法は計測のためにアプリケーションのソースコードを必要とせず，元のアプリケーションに変更を加えることなく計測を行うことが可能であるという利点がある．

計測プログラムが行う処理

アプリケーションの実行の直前に挿入された計測プログラムは，以下の処理を順に行う．

1. アプリケーションのソフトウェア構造の解析：Heijo は，アプリケーションに含まれるメソッドやクラスの名前を使用する．そのために，計測プログラムは始めにメソッドやクラスの名前の収集を行う．Java アプリケーションの場合はクラスパス上にある class ファイル，Android アプリケーションの場合はアプリケーションの apk ファイルに含まれる dex ファイルを対象に収集を

†5 http://repo.xposed.info/

行う．class ファイルの解析には ASM[†6] を，dex ファイルの解析には dex2jar[†7] を利用する．収集されたメソッドにはそれぞれ一意の ID が計測プログラムによって割り振られる．

2. 可視化プログラムとの接続：計測プログラムと可視化プログラムの間では TCP を用いてリアルタイムに情報の送信が行われる．接続の後，計測プログラムは解析したソフトウェア構造の情報を可視化プログラムに送信する．
3. メソッドの実行時間の計測：アプリケーションの実行と並行して計測を行うために，計測用のスレッドを新たに追加する．計測用スレッドでは一定時間間隔（本稿の実験では 1 ミリ秒に設定した）ですべてのスレッドのスタックトレースのサンプリングを行い，それぞれのサンプルにおいて実行されているメソッドをスタックトレースから特定する．そして，実行中のメソッドが出現するサンプル数から，メソッドの実行時間を推定する．たとえば，1 秒間に 1,000 回スタックトレースをサンプリングし，そのうちあるメソッドが 500 個のサンプルにおいて実行されていた場合，メソッドの実行時間は 0.5 秒と推定できる．メソッドの実行時間はスレッドごとに分けて計測される．スレッドの区別は，Java 仮想マシンおよび Dalvik 仮想マシンが各スレッドに割り当てる一意の ID を用いて行う．なお，java.lang.Thread クラスの getAllStackTraces メソッドによって取得できるスタックトレースにはメソッドの引数の型情報は含まれないので，Heijo ではオーバーロードされた同名のメソッドは区別しない．
4. プロファイリング情報の送信：計測したプロファイリング情報は，一定時間間隔（本稿の実験では 100 ミリ秒に設定した）で可視化プログラムに送信される．送信されるプロファイリング情報の形式は，タプル {メソッド ID，スレッド ID，メソッドの実行時間の長さ} のリストおよび現在時刻である．また，送信にかかるオーバーヘッドを考慮し，送信データのシリアライズにはバイナリ形式のシリアライザである MessagePack[†8] を利用する．

3.2 可視化プログラムの実装

可視化プログラムは実装の容易さから，3D ゲームエンジンである Unity を利用して実装した．よって実装した可視化プログラムは，Windows や Mac など，Unity がサポートする多数の環境で実行が可能である．可視化プログラムは C#で記述し，規模は 700 行程度である．

可視化プログラムは，計測プログラムから受信したソフトウェア構造に関する情報を元に，ブロックの配置を行う．その後は，プロファイリング情報を受信するたびに各ブロックの高さを計算し画面に反映させる．

ソフトウェアの階層構造の表現としては，メソッド，クラス，パッケージの階層を一瞥可能な図 3 のような内包関係を表す表現が適切であると考えられる．さらに，限られたスペースで，なるべく沢山の情報を表現可能であることが望まれる．これを解決するため，提案プロファイラではビンパッキング問題の近似解を求めるアルゴリズムの一つである Binary tree bin packing アルゴリズム [†9] を採用した．Binary tree bin packing アルゴリズムはビンパッキング問題の近似解を求めるアルゴリズムの First-fit アルゴリズム [7] に長方形を挿入可能としたアルゴリズムである．ブロックを配置する際にはメソッド，クラス，パッケージの順番に Binary tree bin packing アルゴリズムを再帰的に実行している．

4 ケーススタディ

Heijo の実用性と有用性を，実際のアプリケーションのプロファイリング実験で評価した．実験は Java アプリケーションと Android アプリケーションを対象にしており，Java アプリケーションの実行環境は Java SE 1.8.0 Windows 64bit 版，Android アプリケーションの実行環境は Android 7.0 を搭載した HUAWEI nova lite である．

†6 http://asm.ow2.org/index.html
†7 https://github.com/pxb1988/dex2jar
†8 https://msgpack.org/
†9 https://codeincomplete.com/posts/bin-packing/

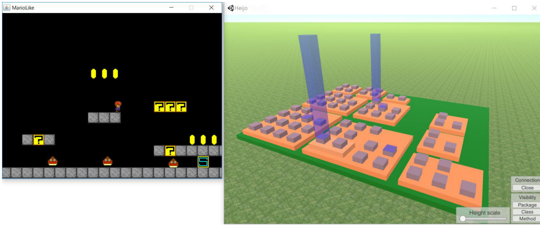

図 8 2D アクションゲームのプロファイリング

4.1 Java アプリケーションのプロファイリング

ユーザの操作が画面にインタラクティブに反映されること，操作中に様々な実行のケースが存在することから，ゲームアプリケーションをプロファイリング対象とする．ここでは，オープンソースソフトウェアの 2D アクションゲームである Java アプリケーション†10 をプロファイリングした．

2D アクションゲームのプロファイリングの様子を図 8 に示す．可視化に含まれるメソッドの数は 64 個である．このアプリケーションは画面中央のプレイヤーキャラクターをキーボード操作することで，移動，ジャンプを行い，画面にある黄色いコインを集めるなどして遊ぶことができるゲームである．図 8 の可視化画面のように，通常時には 2 つのメソッドが特に高くなっており，その値は 100%，すなわち一定時間フレームのうち常に実行されていることを示している．

このアプリケーションのプロファイリング中にスレッドリークの問題を発見した．図 8 で高くなっている 2 つのメソッドのうちの片方，Sprite$AnimationThread クラス（ドル記号は内部クラスを表す）の run メソッドは，いかなる操作を行っても高さは 100%のままで，実行中に低くなることはなかった．可視化画面の run メソッドにマウスカーソルを合わせて詳細パネルを表示させてみると，ゲームオーバー処理によってゲームのリセットが行われるたびに run メソッドを実行するスレッド数が 51 ずつ増加していることが分かった．スレッドリークの問題はアプリケーションのパフォーマンス低下の原因となる恐れがあり，既存のプロファイラを使用した場合，Java 仮想マシン上で実行されているスレッドの数を監視するなどして問題を発見することができる．ここでは，Heijo を使用した場合にも，既存のプロファイラで発見可能な問題を発見できるケースを確認できた．

†10 `https://github.com/aidiary/javagame`

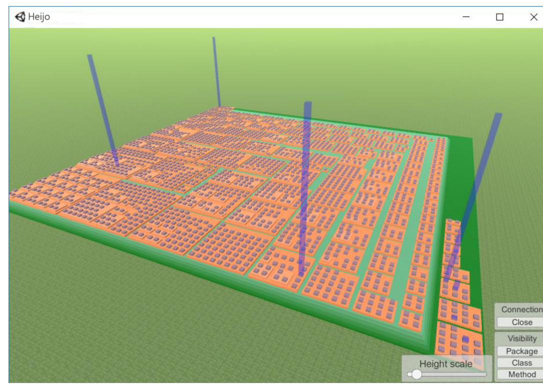

図 9 com.sun.media.sound パッケージを追加して可視化

その後，画面内のコインに接触して効果音が鳴った際に稀に発生する約 3 秒程度のフリーズに遭遇した．その際，可視化画面では Coin クラスの play メソッドの高さが一時的に 100%となり，フリーズから復帰すると 0%に戻った．このことから，Coin クラスの play メソッドが 3 秒程度のフリーズを引き起こす原因ではないかと推測した．実際に，Coin クラスの play メソッドを読んでみると，このメソッドは sun.applet.AppletAudioClip クラスの play メソッドのみを呼び出すことを確認した. 次に，sun.applet.AppletAudioClip クラスのソースコードを調べてみると，このクラスは Java クラスライブラリの一部であり，音声を鳴らすための主な実装は com.sun.media.sound パッケージに含まれていることを確認した．

フリーズの原因をより詳細に調べるため，com.sun.media.sound パッケージをプロファイリングの対象に含め，複数回フリーズのプロファイリングを行った．com.sun.*などの標準ライブラリは何も設定しなくてもクラスをロードできる．ただし，Heijo がプロファイリング対象とするのはクラスパス以下にあるクラスのみである．そのため，Java Runtime Environment から，com.sun.media.sound の jar ファ

イルを取り出し，アプリケーションのクラスパスにその jar ファイルを追加した．その後，再度プロファイリングを行なった．このときの可視化画面の様子を図 9 に示す（2D アクションゲームのクラスはデフォルトパッケージ直下にあるため，画面の右端や奥に追いやられている）．可視化に含まれるメソッドの数は 2,252 個である．フリーズが発生したときには，通常時に高さが 0%であった com.sun.media.sound.DirectAudioDevice$DirectClip クラスの implClose メソッドと com.sun.media.sound.AbstractDataLine クラスの start メソッドの高さが一時的に 100%となっているのが確認できた．くわえて com.sun.media.sound.AbstractDataLine クラスの stop メソッドの高さも同時に 100%となっているケースもあった．この実験結果は，Java クラスライブラリの中にフリーズの問題を発生させる可能性のあるコードが含まれているということを示している．

このフリーズのように一時的である現象を既存のプロファイラでプロファイリングしようとした場合，ユーザが狙って発生させることが難しいため，プロファイリングの開始と終了を適切に指定することは困難である．プロファイリングの時間をフリーズの時間のみに限定させず，長時間のプロファイリング中にこのフリーズが発生した場合，フリーズの発生時間は全体のプロファイリング時間に埋もれてしまい，プロファイリングの出力結果には影響しにくい．したがって，今回のような問題を既存のプロファイラでリアルタイムに発見するのは少し困難であるといえる．一方，Heijo の場合，プロファイリングの開始と終了の指定を必要とせず，常に直近の実行の様子が表されるため，今回の問題のプロファイリングに適している．また，より多くのメソッドの実行を同時に見渡せるため，しばらく実行されず目立つことのなかったメソッドが突然実行された場合にも，ユーザがすぐに発見することができるという利点がある．

4.2　Android アプリケーションのプロファイリ

図 10　Android 版 Google Maps のプロファイリング

ング

地図アプリケーションである Google Maps†11 のプロファイリングを行った．実験に使用したアプリケーションのバージョンは，2018 年 1 月 30 日の時点で Google Play Store で配信されていた最新のものである．使用される頻度の低さから Android Support Library（android.support パッケージ）をプロファイリングの対象外に設定している．

プロファイリングの様子を図 10 に示す．可視化に含まれるメソッドの数は 134,516 個である．図 10 はアプリケーションを操作せず放置したときの様子である．この実験では，異なる条件で同じ操作を行ったとき，メソッドの実行時間の差をどのように Heijo において確認できるか調べた．

Android 端末の画面をスクロールして地図の表示させる範囲を移動させたとき，実行時間で特に変化が見られたのは，図 11 に示す範囲にあるメソッドであった．可視化画面では，実行の頻度の高いクラスのみを表示させており，中央にある多数のクラスを含んでいるのが com.google.android.apps.gmm.map パッケージ，画面の左奥にあるより小さい物が com.google.android.apps.gmm.renderer パッケージである．

図 11 の右上は，地図を最大までズームさせた状態で画面をスクロールさせたときの実行の様子を表している．操作を行っていないときと比べ，いくつかのクラスが高くなっており，逆に可視化画面の中央にあ

†11　`https://play.google.com/store/apps/details?id=com.google.android.apps.maps`

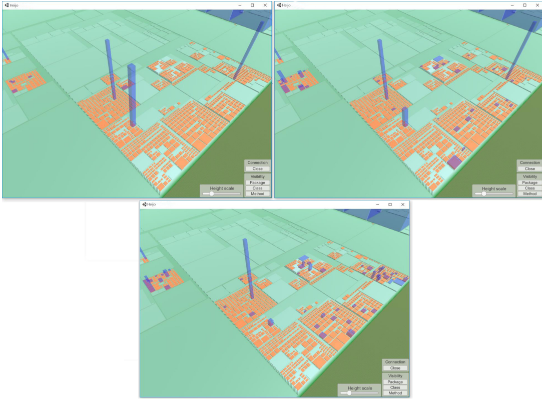

図 11　**Google Maps の実行時間の比較：(左上）無操作（右上）地図を狭い範囲で表示してスクロール（下）地図を広い範囲で表示してスクロール**

るクラスの高さが低くなっているのが分かる．続いて図 11 の下は，地図をズームアウトさせてより広範囲を表示した状態で画面をスクロールさせたときの実行の様子を表している．このとき，中央にあるクラスの高さの縮みは図 11 の右上より顕著であり，また可視化画面の最も右にある複数のクラスの高さの変化も見られる．

これらの変化は，地図の表示範囲を変化させたことによる地図の生成や描画の処理のために掛かる時間の増加が原因であると推測できる．特に，その名前から描画処理を行っていると思われる com.google.android.apps.gmm.renderer パッケージの高さの値を比較してみると，最大までズームさせた状態ではおよそ 50〜60% 程度の値であるのに対し，表示範囲を広くした状態ではおよそ 70〜80% 程度の値にまで増加した．

5　考察

- 可視化のスケーラビリティ：Heijo の可視化のサイズは，アプリケーションに含まれるメソッドの数に依存している．本稿のケーススタディでは，最大で 134,516 個のメソッドを含む可視化を行ったが，より大きい規模のアプリケーションを可視化する場合，すべてのメソッドやクラスを適切に表示することが困難となる可能性がある．しかし，ある 1 つの機能のために実際に実行されるメソッドの数は，アプリケーションに含まれるすべてのメソッドの数と比べると遥かに少なく，一般的に，実行時間のプロファイリングの際には使用されないメソッドに関してはそれほど興味を持たない．そのため，使用されないクラスやパッケージに関しては無視することができ，すべてのメソッドやクラスを表示する必要はない．我々が Google Maps のプロファイリングの際に行ったように，使用されないパッケージを非表示にしたり，実行されているメソッドのみに画面を注目させて Heijo を使用すれば，より規模の大きいアプリケーションに対しても対応できると考えられる．

  Heijo は，ソフトウェアの階層構造の表現を行うため，Binary Tree bin packing アルゴリズムを用いている．3.2 節でも示しているとおり，ビンパッキング問題の近似値を求めるアルゴリズムなので，これを用いることにより，ある程度は 2 次元のピクセルを有効活用することができると考えられる．しかしながら，このアルゴリズムは greedy アルゴリズムであるため，最適解から離れてしまい，2 次元のピクセルを有効活用できない場合がある．そのため，情報密度が低くなり，スケーラビリティに悪影響を及ぼす可能性がある．ただし，Binary Tree bin packing アルゴリズムは greedy アルゴリズムであるため，比較的実行時間が短いという利点があり，実行時間が短いという要素は，Heijo のようなリアルタイムプロファイラにおいては必要な要素であると言える．今後の課題として，実行時間を短く，情報密度を高める工夫が必要である．具体的には，Treemap [15] のような space filling のアルゴリズムを検討し，実装する予定である．
- 計測の精度とオーバーヘッド：メソッドの実行時間の計測において，精度とオーバーヘッドはトレードオフの関係にある．3.1 節で説明したように，Heijo はスタックトレースのサンプリングによって計測を行う．この方法は，計測によって生じるオーバーヘッドが非常に小さいという利点が

ある一方で，計測の精度が低く，サンプリングの間隔よりも短いメソッド実行の見逃しが発生する可能性がある．計測のために用いられるもう1つの主な方法では，メソッドの開始時刻と終了時刻を毎回取得し，その差によってメソッドの実行時間を計測する．この方法ではほぼ正確な計測を行うことができる反面，計測のオーバーヘッドは非常に大きい．特に，ただ値を返すだけのメソッドの呼び出しを繰り返すような場合には，アプリケーションの本来の処理よりも計測の処理のために時間が掛かってしまい，処理がほとんど停止してしまうような場合もある．計測を行うことがアプリケーションの実行に大きな影響を与えることは，ツールの有用性を損なう危険性があるのため，多少の精度よりも計測処理の軽量さを優先すべきであると我々は考えている．

- メソッドのオーバーロード問題 : Java API の仕様により，スタックトレースから取得できるメソッド情報には引数の型情報が含まれないため，実行されているメソッドをスタックトレースから判別する Heijo において，オーバーロードされた同名メソッドを区別することはできない．通常，同名のメソッドには同じ役割の処理を持たせるように設計されるため，これらのメソッドの実行時間が同一に扱われることには大きな問題はないと考えられる．しかし，難読化によってメソッド名が変更された場合は別であり，異なる役割を持つメソッドが共に同じ名前に変更される可能性がある（たとえば，同じクラスにある 2 つのメソッド get(int) と set(char) が，共に a(int) と a(char) に置き換えられる）．したがって，本稿のケーススタディで行った難読化された Android アプリケーションの可視化で表示されるメソッドの数は，難読化される前の本来のメソッドの数よりも少ない．現在の Java API の仕様ではこの問題は解決できないため，Heijo を用いて難読化されたアプリケーションのプロファイリングを行う場合には留意しておく必要がある．
- システムの有効性の範囲: Heijo は実行シナリオなしで，システムのボトルネックを見つけることを目的として開発されている．本システムが適用できるソフトウェアは Java または Android で開発されたソフトウェアである．ただし，Android はルート化が必要である．本システムは単一のアプリケーションで呼び出されるメソッドレベルの実行時間を監視の対象としており、ネットワークで接続された他のシステムや OS のシステム関数に起因するボトルネックを見つけることはできない．本システムが活かせるケースはユーザが操作する時間が多いアプリケーションである．特に，本稿でも取り上げた，2D アクションゲームや，Google map のようなユーザが常に操作をしているアプリケーションは本プロファイラに適していると考えられる．しかしながら，本システムでボトルネックを探すことが可能である範囲は，ユーザがシステムを目視で捉えることができる範囲のボトルネックや，ある程度は頻繁に起きる事象である．ごく短時間だけに起こる不具合や，再現性が非常に低い不具合には対応できない．
- 本システムの可視化表現の弱点: リスト表示を用いたシンプルなプロファイラと比較すると，Heijo は 3 次元表現によって生じる表示の複雑さが認められる．3 次元表現によって生じる問題は， 2 次元のディスプレイから 3 次元情報を読み取る人の能力が限定的であること，奥行の表現のために大きさや尺度が崩れること，手前のオブジェクトによって後方を隠してしまうことが知られている [11]．これらの問題は Heijo についても当てはまる．これらの問題点を考慮しても，3 次元表現を用いたソフトウェア構造の表示自体は，ユーザにソフトウェア構造を意識させつつプロファイリングを行うことによって原因の特定を促すことができると考えられるため有用だと考えられる．ただし，表示の複雑さやカメラの操作の煩わしさからその有用性が失われる可能性がある．この問題を解決するためには，Heijo が表示する情報の中から，ユーザが求める情報をシンプルに示唆する機能が必要である．例えば，Heijo の 3 次元表現に加えてリストによる表示を同時に行うことにより，ユーザは実行時間でソートされたリスト

の表示を見て観測するメソッドを見定めた後に，Heijo の 3 次元表現の可視化を見て，階層構造において近しいメソッドを同時に監視するといったケースの利用も望めるようになると考えられる．このような機能の追加は，Heijo の今後の課題とする．

6 関連するツールと研究

6.1 既存のプロファイラとの比較

VisualVM は，Java アプリケーションのプロファイリングのために最も広く使用されているプロファイラである．VisualVM は軽量の監視ツールであるだけでなく，Java 仮想マシンに関する様々な統計情報を収集しながら，コード実行のプロファイリングや，スレッドダンプおよびヒープダンプの収集と参照を行うことができる．VisualVM の次に広く使用されていると報告されているもう 1 つのプロファイラは，JProfiler である．JProfiler は VisualVM と同様に，CPU とメモリのプロファイリング機能を提供する．

一方で，提案するプロファイラの可視化は，図 2 に示すように，CodeCity の可視化を用いてアプリケーションのソフトウェア構造を表す．Heijo と既存のプロファイラとの主な違いは 2 つあり，1 つは，提案するプロファイラはプロファイリングの開始と終了のタイミングを必要としない点である．これによりユーザは，特定の実行シナリオを持たない場合や，タイミングを適切に指定するのが困難な場合であっても，Heijo を用いてプロファイリングを行うことが可能である．

提案するプロファイラと既存のプロファイラとの 2 つ目の違いは，プロファイラの可視化にアプリケーションのソフトウェア構造を含める点である．これは，ユーザにアプリケーションの設計的側面についての理解を補助するほか，実行される個々のメソッドの識別を容易にする可能性がある．図 1 に示すように，VisualVM のユーザインタフェースでは実行されたメソッド間の関係は表されない．一方，JProfiler ではコールグラフビューを表示でき，実行に時間の掛かるコードがメソッド呼び出しのフローにおいてどこに存在するかを視覚的に表すことにより，ボトルネックを発見しやすくすることができる．しかし，何百ものメソッドを含む大規模なアプリケーションの場合，複雑過ぎるためにユーザが理解できないコールグラフが生成されることがある．

6.2 アプリケーションのプロファイリングに関する研究

最も関連した研究は，Alcocer らの “Performance Evolution Blueprint” [1] と，Bergel らの “Visualizing dynamic metrics with profiling blueprints” [3] である．これらの研究は，パフォーマンスのボトルネックを特定し，除去することを目的とした可視化として，プロファイリングのブループリント（ノードとリンクからなる図）を提案している．具体的には，メソッド間の呼び出し関係からブループリントを作成し，実行時間を四角形の高さで表現している．さらに，Evolution Blueprint はブループリントの表現を使用して，コードの変更の際に発生する，パフォーマンスの低下を特定し，除去することを目的とした可視化を提案した．同様に，Bezemer らの “Differential Frame Graphs” [4] は，異なるソフトウェアバージョンのパフォーマンスの差を理解することを目的とした可視化が提案しており，2 つのバージョン間のプロファイリング結果の差異を可視化している．これらの可視化は，実行シナリオが必要であり，コード実行時の呼び出し関係を可視化しているところが我々の可視化とは異なる．我々の可視化は，単一のバージョンの実行におけるパフォーマンスの変動のみに焦点を当てている．Altman ら [2] は，サーバーアプリケーションのパフォーマンスボトルネックを特定することを目的とした可視化を提案している．この可視化は，多層アーキテクチャにおけるボトルネックを特定することを目的としているが，我々の可視化は，単一のアプリケーションで呼び出されるメソッドレベルにおけるボトルネックを特定することを目的としていることが異なる．

多くのプロファイラは実行中のアプリケーションのスナップショットのプロファイリングに基づいている [5,13] が，フェーズ検出のアプローチを用いてよりリアルタイムなプロファイリングに近づけようと

する研究も存在する．これらの研究では，実行の特徴に基づいて実行トレースが小さなフェーズへと分割される．たとえば，Watanabe ら [18] は，アクティブなオブジェクトに基づいてフェーズを検出する技術を提案している．Voigt ら [17] は，実行トレース内のアクティブなオブジェクトをプロットして可視化する手法を提案している．Medini ら [10] は，検出されたフェーズについて有意なラベルを抽出するために有効なアルゴリズムを提案している．リアルタイムプロファイラの場合，Reiss ら [14] は，アクティブなメソッドを連続するタイムスロットのペアの間で比較する手法を提案している．フェーズの検出と分割の技術によって，ユーザは実行トレースの一部に集中することが可能となる．我々が提案するプロファイラでは，アプリケーションの実行中に瞬間的なフィードバックを提示するために，固定時間フレームを使用している．

6.3 コード都市の可視化に関する研究

ソースコードを都市のメタファーを用いて，3D で可視化をした初期の研究は Kight らによる Software World [9] が知られている．Wettle らの CodeCity [19] による都市可視化は，ソフトウェア構造をインタラクティブに探索することを可能としている．ソースコード以外を対象とする都市形式の可視化については，Santos ら [6] のネットワークモニタリングのための都市の可視化や，伊藤・小山田による研究 [8,20] がよく知られている．

一般的な都市可視化では，ファイルやクラスのサイズがブロックの高さによって表現される．一方で，我々の可視化においてブロックの高さはパフォーマンスの表現のために使用され，アプリケーションの実行に対応して動的に変更されるという特徴がある．我々の可視化では，ブロックの伸び縮みによってパフォーマンスのピークと変動をユーザが容易に認識することを可能にしている．

7 おわりに

本稿において我々は，アプリケーションのパフォーマンスをリアルタイムにプロファイリングするコード都市可視化ツール Heijo を開発した．ケーススタディでは，提案するプロファイラが実際のアプリケーションのプロファイリングのために有用であることを確認した．特に提案するプロファイラは，実行シナリオなしでボトルネックを見つけることや，多数のメソッド実行の同時監視に有用であることが示せた．今後の課題として，再現性の乏しい事象についてプロファイリングするため，常にプロファイル情報を保存し，実行状況を再生/逆再生を行うことができる機能を実装する必要がある．現在は可視化パッケージの絞り込みをするため，実行前に設定として与える必要があるが，今後は実行中にインタラクティブに可視化パッケージの絞り込みを行う機能の実装を行う予定である．

謝辞 本研究は JSPS 科研費 16H05857 の助成を受けた．

参考文献

[1] Alcocer, J. P. S., Bergel, A., Ducasse, S., and Denker, M.: Performance evolution blueprint: Understanding the impact of software evolution on performance, *Proc. of 1st IEEE Working Conference on Software Visualization*, VISSOFT '13, IEEE, 2013, pp. 1–9.
[2] Altman, E., Arnold, M., Fink, S., and Mitchell, N.: Performance Analysis of Idle Programs, *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, New York, NY, USA, ACM, 2010, pp. 739–753.
[3] Bergel, A., Robbes, R., and Binder, W.: Visualizing dynamic metrics with profiling blueprints, *Proc. of International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, Springer, 2010, pp. 291–309.
[4] Bezemer, C.-P., Pouwelse, J., and Gregg, B.: Understanding software performance regressions using differential flame graphs, *Proc. of 22nd IEEE International Conference on Software Analysis, Evolution and Reengineering*, SANER '15, IEEE, 2015, pp. 535–539.
[5] Blanton, E., Lessa, D., Arora, P., Ziarek, L., and Jayaraman, B.: JI. FI: Visual test and debug queries for hard real-time, *Concurrency and Computation: Practice and Experience*, Vol. 26, No. 14(2014), pp. 2456–2487.
[6] Dos Santos, C. R., Gros, P., Abel, P., Loisel, D., Trichaud, N., and Paris, J.-P.: Mapping information onto 3D virtual worlds, *Proc. of IEEE International Conference on Information Visualization*, IEEE, 2000, pp. 379–386.

[7] Eilon, S. and Christofides, N.: The loading problem, *Management Science*, Vol. 17, No. 5(1971), pp. 259–268.
[8] Itoh, T., Takakura, H., Sawada, A., and Koyamada, K.: Hierarchical visualization of network intrusion detection data, *IEEE Computer Graphics and Applications*, Vol. 26, No. 2(2006), pp. 40–47.
[9] Knight, C. and Munro, M.: Virtual but visible software, *Information Visualization, 2000. Proceedings. IEEE International Conference on*, IEEE, 2000, pp. 198–205.
[10] Medini, S., Antoniol, G., Guhneuc, Y. G., Penta, M. D., and Tonella, P.: SCAN: an approach to label and relate execution trace segments, *Proc. of 19th Working Conference on Reverse Engineering*, WCRE '12, pp. 135–144.
[11] Munzner, T.: *Visualization: Analysis and Design*, AK Peters Visualization Series, A K Peters/CRC Press, November 2014.
[12] Ogami, K., Kula, R. G., Hata, H., Ishio, T., and Matsumoto, K.: Using High-Rising Cities to Visualize Performance in Real-Time, *Proc. of 5th IEEE Working Conference on Software Visualization*, VISSOFT '17, Sept 2017, pp. 33–42.
[13] Pauw, W. D., Jensen, E., Mitchell, N., Sevitsky, G., Vlissides, J. M., and Yang, J.: Visualizing the Execution of Java Programs, *Revised Lectures on Software Visualization, International Seminar*, London, UK, UK, Springer-Verlag, 2002, pp. 151–162.
[14] Reiss, S. P.: Dynamic Detection and Visualization of Software Phases, *Proc. of the 3rd International Workshop on Dynamic Analysis*, WODA '05, New York, NY, USA, ACM, 2005, pp. 1–6.
[15] Shneiderman, B.: Tree visualization with treemaps: 2-d space-filling approach, *ACM Transactions on graphics (TOG)*, Vol. 11, No. 1(1992), pp. 92–99.
[16] Simon Maple: Top 5 Java profilers revealed: Real world data with VisualVM, JProfiler, Java Mission Control, YourKit and Custom tooling, `https://zeroturnaround.com/rebellabs/top-5-java-profilers-revealed-real-world-data-with-visualvm-jprofiler-java-mission-control-yourkit-and-custom-tooling/`.
[17] Voigt, S., Bohnet, J., and Dollner, J.: Object aware execution trace exploration, *Proc. of 25th IEEE International Conference on Software Maintenance*, ICSM '09, IEEE, 2009, pp. 201–210.
[18] Watanabe, Y., Ishio, T., and Inoue, K.: Feature-level phase detection for execution trace using object cache, *Proc. of International Workshop on Dynamic Analysis*, ACM, 2008, pp. 8–14.
[19] Wettel, R., Lanza, M., and Robbes, R.: Software Systems As Cities: A Controlled Experiment, *Proc. of 33rd International Conference on Software Engineering*, ICSE '11, New York, NY, USA, ACM, 2011, pp. 551–560.
[20] 伊藤 貴之 小山田耕二: 平安京ビュー 〜 階層型データを碁盤状に配置する視覚化手法, 可視化情報学会第 *9* 回ビジュアリゼーションカンファレンス, 2003.

大 神 勝 也

平成 28 年大阪市立大学工学部情報工学科卒業．平成 30 年奈良先端科学技術大学院大学情報科学研究科博士前期課程修了．修士（工学）．ソフトウェアの可視化に関する研究に従事.

中才恵太朗

平成 28 年近畿大学理工学部情報学科卒業．平成 30 年奈良先端科学技術大学院大学情報科学研究科博士前期課程修了．修士（工学）．現在，同大学院博士後期課程に在学．ソフトウェア工学，特にソフトウェアリポジトリマイニングに関する研究に従事.

畑 秀 明

平成 19 年大阪大学工学部電子情報エネルギー工学科卒業．平成 24 年同大学大学院博士後期課程修了．博士（情報科学）．平成 25 年奈良先端科学技術大学院大学助教．ソフトウェアエコシステムデザインの研究に従事.

松 本 健 一

昭和 60 年大阪大学基礎工学部情報工学科卒業．平成元年同大学大学院博士課程中退．同年同大学基礎工学部情報工学科助手．平成 5 年奈良先端科学技術大学院大学助教授．平成 13 年同大学教授．工学博士．エンピリカルソフトウェア工学，特に，プロジェクトデータ収集／利用支援の研究に従事．電子情報通信学会, 日本ソフトウェア科学会，プロジェクトマネジメント学会各会員,IEEE Senior Member.