情報処理学会論文誌 Vol. 50 No. 10 2514–2519 (Oct. 2009)

テクニカルノート

複数のソフトウェアを横断した利用関係に基づくユーティリティクラスの自動検出

市井 誠[†1] 石尾 隆[†1] 井上克郎[†1]

ユーティリティクラスとは，オブジェクト指向プログラミングにおいて頻繁に用いられる基本的なデータ構造や定型処理を実装したクラスである．ユーティリティの自動的な識別は，プログラム理解や種々のプログラム解析技術の適用において，開発者がアプリケーション固有の機能の分析に注力するために重要である．本研究では，ソフトウェアの集合が与えられたとき，各クラスがユーティリティであるか，それともドメインあるいはアプリケーションに固有のクラスであるのかを測定する universality メトリックを提案する．従来手法は，ある 1 つのソフトウェア内部でのクラス間の利用関係に基づいてユーティリティを特定していたが，本研究では，アプリケーション間での利用関係を導入することで，従来手法では認識されなかったユーティリティクラスを特定することを可能とした．

Cross-application Use-relation Analysis for Finding Utility Classes

Makoto Ichii,[†1] Takashi Ishio[†1] and Katsuro Inoue[†1]

Utility classes are basic data structure and functional units that are frequently used in Object-Oriented Programming. Automatic detection of utility classes is important for program comprehension and program analysis techniques to focus on application-specific classes in a large-scale software system. In this paper, we propose a new metric named universality indicating whether a class is likely utility or not. While existing metrics measure use-relation among classes in a single application, we introduce cross-application use-relation that enables to detect utility classes that are not detected by the existing metrics.

†1 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University

1. まえがき

ユーティリティクラスとは，オブジェクト指向プログラミングにおいて頻繁に用いられる基本的なデータ構造や定型処理を実装したクラスである．ユーティリティの自動的な識別は，プログラム理解や種々のプログラム解析技術の適用において，開発者がアプリケーション固有の機能の分析に注力するために重要である．たとえば，統合開発環境 Eclipse 付属のデバッガは，ステップ実行を指示された場合でも Java 標準クラスの処理の実行途中では停止せず，開発者に実行経過を表示しないように設定することができる．これによって，開発者は，興味の対象外であるユーティリティの動作の分析を省略することができる．

開発者が多数のライブラリを熟知している場合には，分析不要なユーティリティクラスを指定してフィルタリングすることができるが，フィルタを注意深く設定しなければ，重要なクラスまで誤って分析対象から取り除いてしまう可能性がある．また，単純なパッケージ名やクラス名に頼らないフィルタリングが必要な場合として，たとえばライブラリの中でも開発者が知らない使用頻度の低いものだけは解析したいという要求や，アプリケーションフレームワークの内部動作は解析したいがフレームワーク付属のユーティリティクラスはフィルタリングしたいといった要求がある．

これに対して，本研究では，自動的にユーティリティを識別するためのメトリックとして *universality* を提案する．このメトリックは，ソフトウェアの集合が与えられたとき，各クラスがユーティリティであるか，それともドメインあるいはアプリケーションに固有のクラスであるのかを測定する指標である．このメトリックの定義は，あるクラスが多数のソフトウェアに出現するほど，また，各ソフトウェアの多数の場所で使用されるほど，そのクラスはユーティリティである可能性が高いという考え方に基づいている．

従来より，ユーティリティは多数の場所で使用されるという特徴に基づいて，様々なメトリックや解析手法が提案されている．たとえば，青木は，クラスがプログラム中で参照されている回数をクラスの人気度（Class Popularity）メトリックと定義し，多数の場所から参照される（人気のある）クラスを再利用すべきであると主張している[1)]．類似した指標としては，メソッド単位で計測される Utilityhood[2)] が提案されている．また，Marin らは，ロギングや同期処理など，システムの実行基盤となる機能を抽出するために，呼び出される回数の多いメソッドに着目している[6)]．

しかし，これら従来のメトリックは，単一のクラス集合に対して計測されるため，多数のソフトウェアで使用されるライブラリであっても，特定の機能の実装のために局所的に使用

されるクラスをユーティリティとして認識できない．たとえば，コマンドライン引数を処理するライブラリは，各プログラムの `main` メソッドなど特定の場所でのみ使用されるため，実際にはユーティリティであるにもかかわらず，人気度などの値は小さくなり，ユーティリティとして認識されない．これに対して，本研究で提案する *universality* メトリックの計算では，解析対象をアプリケーション集合とし，各クラスが所属するアプリケーションの情報を用いて，複数のソフトウェアで再利用されているユーティリティを識別する．

ケーススタディとして，Java 言語で書かれた約 13 万クラスのオープンソースソフトウェア群に対して *universality* とクラスの人気度という 2 つのメトリックの計算を行った．その結果，従来手法では識別が困難なユーティリティクラスを識別できることを示した．また，各クラスについてのメトリックの計算結果は，ウェブサイト上に公開した[⋆1]．なお，*universality* を利用する場合は，単純に多数のアプリケーションを集積することでユーティリティを識別するというだけでなく，ソフトウェア開発者が携わったことのあるアプリケーション集合を用いて *universality* メトリックを計算することで，開発者が熟知しているであろうユーティリティクラスのみを特定，フィルタリングするといった適用方法が考えられる．

以降，2 章ではメトリックの定義と計算方法を解説し，3 章ではケーススタディの結果について述べる．最後に，4 章でまとめと今後の課題について述べる．

2. 提 案 手 法

本研究では，あるアプリケーションソフトウェア集合が与えられたとき，それらのアプリケーションで共通して使用されるユーティリティクラスを抽出するための指標として，クラスの**一般性（*universality*）**メトリックを提案する．

あるクラスが一般性が高い，すなわち，一般的に用いられるほど，ユーティリティである可能性が高い．一般性の基準として，本研究では，以下の 2 点を用いる．

- 多数のアプリケーションで利用されるクラスは，少数のアプリケーションで利用されるクラスよりも一般性が高い．
- 多数のクラスから利用されるクラスは，少数のクラスから利用されるクラスよりも一般性が高い．

これらの基準に従い，クラス c を利用するクラス数 i_c とクラス c を利用するアプリケーション数 a_c を用いて，クラス c に対する一般性 $universality(c)$ を以下のように定義する．

$$universality(c) = \frac{\log(i_c + 1)}{\log(|C|)} \times \frac{\log(a_c + 1)}{\log(|A|)}$$

この式において，$|C|$ および $|A|$ は，それぞれ，解析対象のクラスの総数とアプリケーションの数である[⋆2]．式中で対数を用いているのは，非常に大きな i_c の値を持つ少数のクラスの影響を排除するためである[4)]．

なお，青木によるクラスの人気度（Class Popularity）[1)] は，$CP(c) = i_c$ と表現することができ，*universality* はクラスを利用するアプリケーション数を計算に含めたものだといえる．

$universality(c)$ の値を求めるには，クラス間の利用関係を求め，利用関係にあるクラス数，アプリケーション数を計算すればよい．そのために，複数のアプリケーションを横断した利用関係解析を行う．本解析の入力は，アプリケーション集合 A である．本研究では，対象として Java を想定し，A の各要素であるアプリケーションはクラスの集合であると考える．具体的な A の例としては，オープンソースソフトウェアの集合や，企業内で開発されたソフトウェアの集合などが考えられる．

利用関係の解析は，アプリケーション内の解析と，アプリケーション間の解析の 2 段階によって構成される．まず，A に含まれる各アプリケーションに対し，個別に利用関係グラフ（Use-relation Graph）を構築する．利用関係グラフは，有向グラフの一種であり，頂点はクラスあるいはインタフェースを意味する．また，クラスあるいはインタフェース c_1，c_2 間に以下のいずれかの利用関係が存在するとき，有向辺 $c_1 \rightarrow c_2$ を生成する[5)]．

- c_1 が c_2 を継承（extends）あるいは実装（implements）する．
- c_1 が c_2 型の変数を宣言する．
- c_1 が c_2 型のインスタンスを作成する．
- c_1 が c_2 のメソッドを呼び出す．このメソッド呼び出しはソースコード上の静的な型に従って解決する．ただし，ソースコード上でのメソッド呼び出し対象が c_2 のサブクラス c_3 であり，かつ c_3 が c_2 の該当メソッドをオーバライドしていない（実行されるメソッドが c_2 に定義されている）場合は，c_2 を呼び出しているものとして扱う．
- c_1 が c_2 のフィールドを参照あるいは操作する．

⋆1 http://sel.ist.osaka-u.ac.jp/˜ishio/universality/

⋆2 i_c は最大で $|C|$，a_c は最大で $|A|$ となることから，この定義式で計算される *universality* メトリックは 1 以上の値をとることがありうる．0 から 1 までの値をとるよう正規化したい場合は，分母を $\log(|C| + 1)$，$\log(|A| + 1)$ と修正すればよい．分母の値は，解析アプリケーション集合によって決まる定数値なので，正規化の方法を変更しても，本論文のケーススタディで使用している *universality* 値の大小関係には影響しない．

アプリケーション単位での利用関係グラフを構築したのち，アプリケーションを横断する辺を接続する．本研究では，アプリケーション間でクラス名が一致するものを同一のクラスとして扱い，あるクラス c と同一のクラスが他の n 個のアプリケーションに $c_1, c_2, \cdots, c_n$ として存在するとき，アプリケーション内の参照辺 $v \to c$ に対応するアプリケーション間の参照辺 $v \to c_k$（$k = 1, \cdots, n$）を生成する．このようにして得られたアプリケーション間の利用関係グラフからクラス c を利用するクラス数 i_c を求めるには，クラス c へと利用辺を持つクラスを数え上げればよく，また，アプリケーション数 a_c は，それらのクラスが所属するアプリケーションを数え上げることによって求まる．

利用関係グラフの構築例を**図 1** に示す．`Liquor` というクラスが 2 つのソフトウェア `WarehouseApp` と `StoreApp` に含まれており，これらをそれぞれ Liquor1，Liquor2 と呼ぶことにする．`Warehouse` クラスは Liquor1 を参照しているため，同一のクラスである Liquor2 も参照しているものとして扱う．同様に，Liquor2 を参照する `Store`，`Shelf` は Liquor1 への辺を持つ．このグラフから，$i_{Liquor} = 3$（`Warehouse`，`Store`，`Shelf`），$a_{Liquor} = 2$（`WarehouseApp`，`StoreApp`）となり，$universality$(`Liquor`) $= 1.226$ が得られる[⋆1]．`Paper` クラスは，`StoreApp` の中では `Liquor` クラスと同じように 2 つのクラスから使用されているが，$universality$(`Paper`) $= 0.613$[⋆2]となり，`Liquor` クラスよりも一般性が低いと判定される．なお，Liquor1 と Liquor2 の利用関係は同一であるため，計算結果も同一となる．

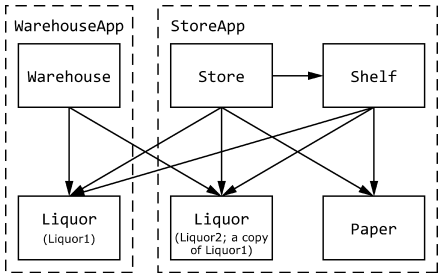

図 1　アプリケーションを横断した利用関係グラフの例
Fig. 1　An example of cross-application use-relation graph.

⋆1 $universality$ の定義式を変更し，値域が 0 から 1 の範囲となるよう正規化した場合は 0.712 となる．
⋆2 同様に正規化した場合は 0.356 となる．

3. ケーススタディ

提案した一般性メトリックの妥当性を確認するため，ケーススタディを行った．39 個のオープンソースソフトウェアを集めてアプリケーション集合とし，各クラスに対する $universality$ と，クラスの人気度 CP の値を計算した．合計クラス数は 131,328 であり，総行数は 18,778,821 である．アプリケーションは広い範囲のドメインを含められるように選定し，たとえば，統合開発環境 Eclipse やサーバソフトウェア Apache Tomcat，ドローツール JHotDraw などを含んでいる．完全なリストを**表 1** に示す．計算した 2 つのメトリックの実際の値は，1 章末尾に示したウェブサイト上に公開している．

表 2 に，$universality$ の値の上位 20 位に入ったクラスを示す．これらは Java プログラミングにおける基本的なクラス群であり，ほとんどすべての開発者があらゆる場面で使用することから，上位を占める結果となった．

表 3 は，$universality$ での順位（Univ. Rank）では 100 位以内に入るが，CP による順位（CP Rank）では 100 位より下になる 47 のクラスのうちの 5 つを示したものである．これら 47 個のクラスは，ユーティリティとしての役割を担う Java の標準クラスであり，多数のソフトウェアで使用されるが，1 つのソフトウェア中では頻繁に使用されるわけではないために，CP の値のみではユーティリティと判定されにくい．$universality$ メトリックは，使用されているアプリケーション数を数え上げることで，これらのクラスを識別している．

表 4 は，逆に，CP による順位では 100 位以内に入るが $universality$ による順

表 1　解析対象アプリケーション集合
Table 1　Target applications.

JDK5	ArgoUML 0.24	JGraph 5.12.1.1
FreeMind 0.8.1	GanttProject 2.0.7	JFreeChart 1.0.9
JHotDraw 7.0.9	Batik 1.6	PDFRenderer（2008-09-01 時点最新版）
iText 2.1.3	NetBeans 5.5.1	POI 3.1-FINAL-20080629
Eclipse 3.3	JEdit 4.3pre11	Commons Logging 1.1.1
Log4J 1.2.15	SVNKit 1.1.8	Maven 2.0.9
BioJava 1.5-beta2	Cocoon 2.1.11	Commons Collections 3.2.1
Trove 2.0.4	ASM 3.1	Antlr 3.0.1
Ant 1.7.0	SableCC 3.2	Soot 2.2.4
HSQLDB 1.8.0.10	Darby 10.4.1.3	BerkeleyDB Java Edition 3.2.76
H2 Database 2008-08-28	Lucene 2.3.2	JBoss 4.2.3 GA
Tomcat 6.0.14	Struts 1.2.7	MyFaces 1.2.4
Spring Framework 2.5.5	Azureus 3.0.3.4	BitTyrant（2008-09-04 時点最新版）

表 2　Universality の上位 20 クラス
Table 2　Top 20 classes in the universality.

Class name	Univ.	CP
java.lang.String	0.933	69,324
java.lang.Object	0.915	55,628
java.util.List	0.793	12,981
java.lang.System	0.780	11,191
java.lang.Class	0.776	10,590
java.lang.Throwable	0.775	10,467
java.util.Iterator	0.773	10,191
java.util.ArrayList	0.772	10,135
java.lang.Exception	0.761	8,840
java.util.Map	0.757	8,476
java.lang.Integer	0.748	7,568
java.util.Set	0.741	6,954
java.io.File	0.736	6,554
java.lang.StringBuffer	0.735	6,907
java.io.PrintStream	0.730	6,132
java.util.HashMap	0.730	6,129
java.io.IOException	0.725	6,115
java.util.Collection	0.724	5,690
java.lang.Illegal-ArgumentException	0.714	5,057
java.lang.Runnable	0.699	6,790

表 3　Universality が高く CP が低いクラス
Table 3　Classes with high universality but low CP.

Class name	Univ. Rank	CP Rank
java.lang.Character	39	104
java.util.LinkedList	41	105
java.io.FileOutputStream	56	177
java.lang.Comparable	78	240
java.util.Stack	95	354

表 4　CP が高く Universality が低いクラス
Table 4　Classes with high CP but low universality.

Class name	Univ. Rank	CP Rank
org.eclipse.swt.widgets.Control	213	25
org.eclipse.swt.SWT	221	34
org.eclipse.core.resources.IResource	564	69
org.openide.util.NbBundle	1,398	24
org.openide.ErrorManager	1,496	54

表 5　一般性メトリックの分布
Table 5　Distribution of universality.

Univ.	#classes	packages
1.0–0.9	2	java.lang
0.9–0.8	0	(none)
0.8–0.7	17	java.util, java.lang, java.io
0.7–0.6	18	java.lang, java.util, java.io, java.net, java.awt
0.6–0.5	49	java.util, java.lang, java.io, javax.swing, java.awt,...
0.5–0.4	80	java.io, java.lang, javax.swing, javax.swing, java.awt,...
0.4–0.3	196	org.eclipse.swt.widgets, javax.swing, java.lang, java.awt.event, ...
0.3–0.2	348	org.eclipse.swt.graphics, javax.swing, javax.management, ...
0.2–0.1	1,385	org.eclipse.swt.dnd, org.gudy.azureus2.core3.util, javax.management, org.eclipse.swt.widgets, org.bouncycastle.asn1, ...
0.1–0	129,233	soot.jimple.parser.node, org.apache.poi.....functions, test, ...

位では 100 位より下になる 47 のクラスのうち 5 つを示したものである．これらのクラスのうち，`org.eclipse.swt` に属するクラスは少数のアプリケーションでのみ使用される GUI ライブラリであり，また，`org.eclipse.core.resources.IResource` や `org.openide.ErrorManager` はそれぞれアプリケーション固有のリソース管理や例外処理に用いられているクラスである．いずれも，使用されているアプリケーションの規模が大きいために，CP の値のみでは汎用的なユーティリティクラスと区別が付かなかった．なお，本ケーススタディでは，従来の CP の定義をそのまま使用しているが，CP の値をアプリケーションごとに個別に計算し，それぞれアプリケーションの規模によって正規化することも可能である．その場合は，1 つのクラスがアプリケーションごとに異なる CP の値を持つことになり，最終的に，あるアプリケーション集合に対して定まる単一の数値として使用することができない．複数のアプリケーションを横断したときにどのように値を計算すべきか，という点を考慮したことが，*universality* メトリックの独自性となっている．

表 5 は，*universality* の値の分布である．それぞれの値の区間に対して，クラスが 1 つでもその区間に含まれるようなパッケージ名を並べている．クラスの分布を分析したところ，1.0–0.5 の範囲には汎用的なクラスが存在し，GUI やネットワーク，ロギングなどの特定ドメインに属するクラスは 0.5–0.2 の範囲に存在する．そして，アプリケーション固有のクラスは，0.2 以下の *universality* を持つ．この結果から，*universality* は，一般的なクラス，ドメイン固有のクラス，アプリケーション固有のクラスを分類するために使用できる．たとえばソフトウェア部品検索においては，汎用的なクラスほど好ましい，あるいは汎用的なクラスは既知なので不要であるというように，開発者の検索目的に応じたフィルタリングに適用できると考えられる．

表 5 からは，*universality* が高いクラスの数が全体のごく一部に限られていることが分かる．この分布は，多数のクラスから利用されるクラスはごく少数であるという，CP が持っている特徴と同様である[4]．*universality* の値を使うことで，多数のアプリケーションで再利用されているこれら少数のクラスを解析対象から取り除くことになり，たとえばメソッド呼び出し関係の解析においては，ユーティリティに関する多くの呼び出し関係を解析

対象から取り除くことが可能である．その他の利用方法としては，ソフトウェア開発者が携わったことのあるアプリケーション集合を用いて *universality* メトリックを計算し，開発者が熟知しているであろうユーティリティクラスのみをフィルタリングするといった方法も考えられる．

3.1 妥当性への脅威

ケーススタディの妥当性への脅威としては，まず，アプリケーション集合による結果への影響が考えられる．本実験の適用対象がオープンソースソフトウェアであることから，限られた著名なライブラリが積極的に再利用される一方，互いに類似したプロジェクトが少なく，アプリケーション間でのクラスの再利用が行われていない可能性がある．企業で行われているプロジェクト群において，類似プロジェクト間でのソースコード再利用が行われている状況で適用した場合には，異なる結果が得られると予測される．

本研究における解析対象は，オープンソースソフトウェア集合から，小規模な検索エンジンに相当する約 13 万クラスを解析した[3]．しかし，Koders[⋆1]が蓄積する 60 万ファイルや，Google Code Search[⋆2]の 250 万ファイルなどであれば，異なる結果となる可能性がある．たとえば，表 4 には，GUI ライブラリの 1 つである SWT のクラスがあがっているが，SWT を用いる GUI アプリケーション数が増加することで，これらのクラスの *universality* の値がより大きくなる可能性がある．

universality の計算方法の妥当性への脅威として，パッケージ名，クラス名が一致していれば，アプリケーション間で同一のクラスを利用していると判断している点があげられる．しかし，同じ名称のクラスであっても，アプリケーションごとにカスタマイズされた異なるバージョンを利用している状況がありうる．このような場合に対しては，アプリケーション間での利用関係を計算するとき，クラスの同一性をより厳密に，ソースコードの比較などによって判定することで対応可能であると考えられるが，*universality* の計算結果への影響の評価は今後の課題である．

本ケーススタディでは，*universality* メトリックの値を計算し，従来手法である CP との比較を行っている．ユーティリティクラスを自動的に識別する機能については，様々なプログラム解析手法の適用状況によって要求が変化するため，本ケーススタディのみをもって十分な評価ということはできない．今後，様々な手法に本メトリックを組み込み，従来手法との比較実験を行っていくことが必要となる．

⋆1 Koders. http://www.koders.com/
⋆2 Google Code Search. http://www.google.com/codesearch

4. ま と め

本研究では，多数のソフトウェアで頻繁に使用されるユーティリティクラスを抽出する一般性メトリック *universality* を定義し，ケーススタディによって，アプリケーション固有のクラス，ドメイン固有のクラスと汎用的なクラスを分類できることを示した．メトリックの実際の値については，ウェブサイト上に公開している．

今後の課題としては，異なるアプリケーション集合へと提案手法を適用し，*universality* の妥当性の評価を行うことがあげられる．また，*universality* メトリックをソフトウェア部品検索へと応用する予定である．メトリック値による単純なフィルタリングだけでなく，開発者が知識を持っているプロジェクトを事前に指定しておくことで，開発者が利用したことのないユーティリティクラスの再利用を促進することを検討している．

謝辞　本研究は，文部科学省科学研究費補助金若手研究（B）（課題番号：21700030）の助成を得た．

参 考 文 献

1) 青木　淳：オブジェクト指向システム分析設計入門，ソフトリサーチセンター (1993).
2) Hamou-Lhadj, A. and Lethbridge, T.: Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System, *Proc. 14th International Conference on Program Comprehension*, pp.181–190 (2006).
3) Hummel, O., Janjic, W. and Atkinson, C.: Code Conjurer: Pulling Reusable Software out of Thin Air, *IEEE Software*, Vol.25, No.5, pp.45–52 (2008).
4) Ichii, M., Matsushita, M. and Inoue, K.: An Exploration of Power-law in Use-relation of Java Software Systems, *Proc. 19th Australian Software Engineering Conference*, pp.422–431 (2008).
5) Inoue, K., Yokomori, R., Yamamoto, T., Matsushita, M. and Kusumoto, S.: Ranking Significance of Software Components Based on Use Relations, *IEEE Trans. Softw. Eng.*, Vol.31, No.3, pp.213–225 (2005).
6) Marin, M., van Deursen, A. and Moonen, L.: Identifying Crosscutting Concerns Using Fan-in Analysis, *ACM Trans. Softw. Eng. Methodology*, Vol.17, No.1, p.3 (2007).

(平成 21 年 2 月 20 日受付)
(平成 21 年 7 月 2 日採録)

市井　誠（正会員）

平成 16 年大阪大学基礎工学部情報科学科卒業．平成 18 年大阪大学大学院情報科学研究科博士前期課程修了．平成 21 年大阪大学大学院情報科学研究科博士後期課程修了．博士（情報科学）．大阪大学在学中，ソフトウェア部品検索の研究に従事．現在，日立製作所組込みシステム基盤研究所において組み込みソフトウェアの生産性向上技術の研究に従事．

石尾　隆（正会員）

平成 15 年大阪大学大学院基礎工学研究科博士前期課程修了．平成 18 年大阪大学大学院情報科学研究科博士後期課程修了．同年日本学術振興会特別研究員（PD）．同年ブリティッシュコロンビア大学ポストドクトラルフェロー．平成 19 年大阪大学大学院情報科学研究科コンピュータサイエンス専攻助教．博士（情報科学）．プログラム解析，アスペクト指向プログラミングに関する研究に従事．日本ソフトウェア科学会，ACM，IEEE 各会員．

井上　克郎（フェロー）

昭和 54 年大阪大学基礎工学部情報工学科卒業．昭和 59 年大阪大学大学院博士課程修了．同年大阪大学基礎工学部情報工学科助手．昭和 59〜61 年ハワイ大学マノア校情報工学科助教授．平成 1 年大阪大学基礎工学部情報工学科講師．平成 3 年同学科助教授．平成 7 年同学科教授．平成 14 年大阪大学情報科学研究科コンピュータサイエンス専攻教授．平成 20 年国立情報学研究所客員教授．同年情報処理学会フェロー．同年電子情報通信学会フェロー．工学博士．ソフトウェア工学の研究に従事．日本ソフトウェア科学会，電子情報通信学会，ACM，IEEE 各会員．