Vol.2014-SE-186 No.18
2014/11/14

複数クラスを横断する情報隠蔽を実現するリファクタリングフレームワーク

高橋 祐介[1,a)]　新田 直也[1,b)]

概要：情報隠蔽に関係するリファクタリングはいくつか存在しているが, 実際のソフトウェア開発において, いつどのリファクタリングを適用すればよいのかは自明ではない. 特に対象となるクラスを複数のクラスから隠蔽する場合には, これらのリファクタリングを適切に選択し, 適切な順序で適切な位置に適用しなければ隠蔽は正常に完結しない. そこで本研究では, 複数のクラスを横断するような情報隠蔽を支援するためのリファクタリングフレームワークを提案する. また, 提案フレームワークの構成要素として, いくつかの新しいリファクタリングを導入する. 本稿では, 提案フレームワークを実際の設計変更事例に適用した結果その有効性を確認することができたので, その報告を行う.

1. はじめに

一般に, ソフトウェアの内部構造は開発の進行に伴って劣化していく. そのため, 開発の途上でソースコードの内部構造を安全に改善する手段として, **リファクタリング** (文献 [1]) が広く注目されている. リファクタリングとは, ソフトウェアの外的振る舞いに影響を与えることなく, ソースコードの内部構造を改善する枠組みである. 近年の統合開発環境においては, 多くのリファクタリングが自動化されて組み込まれており, リファクタリングを利用する基本的な環境は整えられているといえる.

しかしながら, 実際のソフトウェア開発においては, 特に複雑な変更作業において自動リファクタリングが用いられない傾向にあることが指摘されている (文献 [2], [3] 参照). また, 連続する複数のリファクタリングの適用が, 特定のパターンに属することが多いことも指摘されている (文献 [2], [3], [4] 参照). そのため, 複合的なリファクタリングを自動化するツールがいくつか提案されている (文献 [5] 参照).

本研究では, 複数のクラスを横断するような情報隠蔽を支援するためのリファクタリングフレームワークを提案する. 情報隠蔽に関係するリファクタリングとしては, クラス抽出, 委譲の隠蔽, 引数オブジェクトの導入, オブジェクトそのものの受け渡しなどが挙げられるが, これらを用いる文脈はそれぞれ異なっている. そのため, これらのうちのどのリファクタリングをいつ適用すればよいのかは自明ではない. 特に対象となるクラスを複数のクラスから隠蔽する場合には, これらのリファクタリングを適切に選択し, 適切な順序で適切な位置に適用しなければ隠蔽は正常に完結しない. そこで, 隠蔽対象となるクラス, そのクラスを隠蔽するための新規クラス, 対象クラスをどのクラス (複数クラス) から隠蔽するかを入力として受け取って, 文脈に応じたリファクタリングの適用戦略を提示できるツールの開発を目指す. 本稿では, 適切なリファクタリングを導出するための系統的な手続きを定めたリファクタリングフレームワークの提案を行う. また目的となる情報隠蔽を完結するために必要なリファクタリングとして, **戻り値オブジェクトの導入**, **オブジェクトそのものの返却**, **隠蔽による型の置き換え**の 3 つのリファクタリングを新たに導入する.

本稿では, 提案フレームワークの有効性を確認するため, 我々の研究室で実際に行われた設計変更の事例を対象に, 本フレームワークによって変更作業の一部を再現できるかの確認を行った. その結果, 変更作業中の十分に大きな部分を提案フレームワークによって再現できることが判明した. 今後, 文脈の判別とリファクタリングの推奨部分を自動化し, さらに新たに定義したリファクタリングを含めて自動化されていないリファクタリングを自動化してツールの完成を目指す予定である.

2. 複数クラスを横断する設計変更

我々の研究室で開発された Radish(文献 [6] 参照) の設計変更事例を対象に, 複数クラスを横断する複雑な変更作業においてリファクタリングを適用する際の問題点について

[1] 甲南大学大学院 自然科学研究科
Graduate School of Natural Science, Konan University
[a)] m1424006@center.konan-u.ac.jp
[b)] n-nitta@konan-u.ac.jp

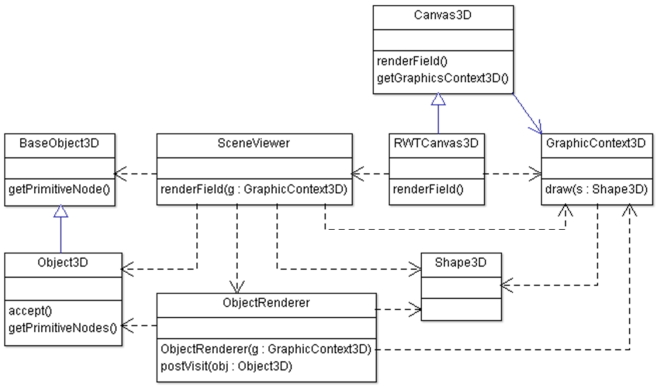

図 1 情報隠蔽前のクラス図

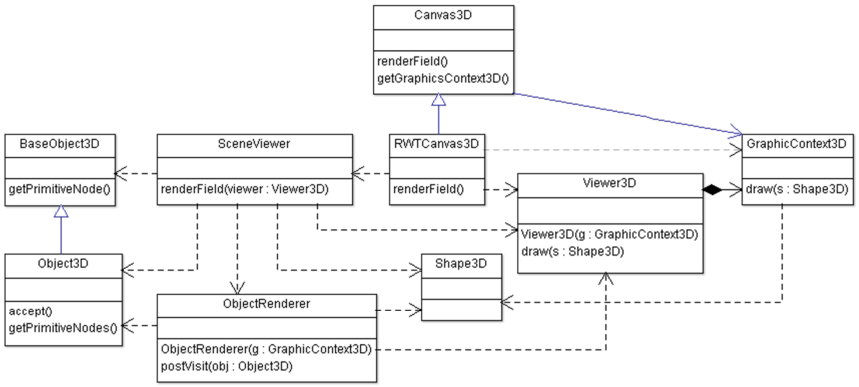

図 2 情報隠蔽後のクラス図

考える. 本節では, Radish の設計変更事例の紹介を行う.

Radish は, 我々の研究室で Java を用いて開発された約 1 万行規模の 3D ゲームフレームワークで, グラフィックス API として Java3D を利用している. 対象とする設計変更事例は, 2009 年 9 月 19 日から 9 月 29 日にかけて行なわれたもので, 目的に応じて 9 月 19 日から 24 日までの前半と 9 月 25 日から 29 日までの後半に分けることができる. ソースコード上では前半 647 行, 後半 289 行の計 936 行の変更が行われた. 本稿ではリファクタリングの適用対象として, 前半の作業の一部を取り上げる. 前半の設計変更は, 影付けやバンプマッピングなどのレンダリング機能の追加を容易にすることを目的として行われた. 対象となる変更作業を行う前と行った後の該当部分のクラス図をそれぞれ図 1, 2 に示す. 各図において, `Canvas3D`, `GraphicsContext3D`, `Shape3D` は Java3D によって提供されたクラスであり, それ以外はすべて Radish 特有のクラスである. Java3D において, `Shape3D` は基本図形を表すクラスである. `Shape3D` の各インスタンスは, `GraphicsContext3D#draw()` の呼び出しの引数として渡されることによってレンダリングされる. この変更作業においては, `GraphicsContext3D` を `SceneViewer` と `ObjectRenderer` から隠蔽するために, 新規クラスとして `Viewer3D` を作成している. なお, レンダリング機能の追加はこの `Viewer3D` に対して行われる.

3. リファクタリングの設計変更例への適用可能性

前節で紹介した変更作業に対するリファクタリングの適用可能性について検討する. この変更作業の目的は, `SceneViewer` と `ObjectRenderer` から `GraphicsContext3D` を `Viewer3D` という新しいクラスの背後に隠蔽することにある. この変更作業を手作業で行う手順を以下に説明する. ただし, リファクタリング本来の動機を大幅に逸脱しない範囲で, 既存のリファクタリングをできる限り用いる.

(1) 図 3 に適用前, 図 4 に適用後のソースコードを示す. `SceneViewer#renderField()` の呼び出しに [引数オブジェクトの導入] を適用する. このとき新規クラスとして `Viewer3D` が生成される.

(2) 図 4 に適用前, 図 5 に適用後のソースコードを示す. `SceneViewer` クラス内で `GraphicsContext3D` クラスのインタフェースを利用している箇所を [委譲の隠蔽] を適用することによって `Viewer3D` のインタフェースを利用するように置き換える.

(3) 図 6 に適用前, 図 7 に適用後のソースコードを示す. `ObjectRenderer` クラスのコンストラクタ呼び出しに対して [オブジェクトそのものの受け渡し] を適用する.

(4) 図 8 に適用前, 図 9 に適用後のソースコードを示す.

この作業に該当する既存リファクタリングは存在しない. `ObjectRenderer` を以下のように改変する. フィールド定義 '`GraphicsContext3D graphisContext3D;`' を '`Viewer3D viewer;`' に置き換える. コンストラクタ中の代入文を引数 `viewer` そのものを代入するようにする. `graphicsContext3D` をレシーバとして利用していた部分は `viewer` フィールドから取得するよう置き換える.

(5) 図 9 に適用前, 図 10 に適用後のソースコードを示す. `ObjectRenderer` クラス内で `GraphicsContext3D` クラスのインタフェースを利用している箇所を [委譲の隠蔽] を適用することによって `Viewer3D` のインタフェースを利用するように置き換える.

ここで, この変更作業の過程において, 文脈に応じて引数オブジェクトの導入, 委譲の隠蔽, オブジェクトそのものの受け渡しの 3 種類のリファクタリングを切り替えて適用する必要があることがわかる. また, 手順 4 には既存のリファクタリングを適用できないことがわかる. この変更作業全体に対して他のリファクタリングの組み合せによってアプローチすることも可能であるが, いずれにせよ, 既存のリファクタリングだけではすべての作業を完結できないことがわかっている.

```
public class RWTCanvas3D extends Canvas3D {
  public void renderField(int fieldDesc) {
    Enumeration<BranchGroup> bgs = universe.getLocale().getAllBranchGraphs();
    for (; bgs.hasMoreElements();) {
      BranchGroup bg = bgs.nextElement();
      if (bg instanceof SceneViewer) {
        ((SceneViewer)bg).renderField(this.getGraphicsContext3D());
      }
    }
  }
}
```

```
public class SceneViewer extends BranchGroup {

  public void renderField(GraphicsContext3D graphicsContext3D) {
    if (graphicsContext3D.numLights() == 0) {
      for (int n = 0; n < lights.size(); n++) {
        graphicsContext3D.addLight(lights.get(n));
      }
    }

    Transform3D trans = new Transform3D();
    for (int n = 0; n < shadows.size(); n++) {
      BaseObject3D obj = shadows.get(n);
      if (obj instanceof Object3D) {
        ((Object3D)obj).accept(new ObjectRenderer(graphicsContext3D));
      } else {
        obj.center.getTransform(trans);
        graphicsContext3D.setModelTransform(trans);
        Node primitive = obj.getPrimitiveNode();
        if (primitive instanceof Shape3D) {
          graphicsContext3D.draw((Shape3D)primitive);
        }
      }
    }
    :
    for (int n = 0; n < occluders.size(); n++) {
      Object3D occuluder = occluders.get(n);
      occuluder.accept(new ObjectRenderer(graphicsContext3D));
    }
  }
}
```

図 3 手順 1 適用前

4. リファクタリングフレームワークの提案

前節で述べたように複数クラスを横断する設計変更において

```
public class RWTCanvas3D extends Canvas3D {
  public void renderField(int fieldDesc) {
    Enumeration<BranchGroup> bgs = universe.getLocale().getAllBranchGraphs();
    for (; bgs.hasMoreElements();) {
      BranchGroup bg = bgs.nextElement();
      if (bg instanceof SceneViewer) {
        ((SceneViewer)bg).renderField(new Viewer3D(this.getGraphicsContext3D()));
      }
    }
  }
}
```

```
public class SceneViewer extends BranchGroup {

  public void renderField(Viewer3D viewer) {
    if (viewer.getGraphicsContext3D().numLights() == 0) {
      for (int n = 0; n < lights.size(); n++) {
        viewer.getGraphicsContext3D().addLight(lights.get(n));
      }
    }

    Transform3D trans = new Transform3D();
    for (int n = 0; n < shadows.size(); n++) {
      BaseObject3D obj = shadows.get(n);
      if (obj instanceof Object3D) {
        ((Object3D)obj).accept(new ObjectRenderer(viewer.getGraphicsContext3D()));
      } else {
        obj.center.getTransform(trans);
        viewer.getGraphicsContext3D().setModelTransform(trans);
        Node primitive = obj.getPrimitiveNode();
        if (primitive instanceof Shape3D) {
          viewer.getGraphicsContext3D().draw((Shape3D)primitive);
        }
      }
    }
    :
    for (int n = 0; n < occluders.size(); n++) {
      Object3D occuluder = occluders.get(n);
      occuluder.accept(new ObjectRenderer(viewer.getGraphicsContext3D()));
    }
  }
}
```

図 4 手順 1 適用後, 手順 2 適用前

```
public class SceneViewer extends BranchGroup {
  public void renderField(Viewer3D viewer) {
    if (viewer.numLights() == 0) {
      for (int n = 0; n < lights.size(); n++) {
        viewer.addLight(lights.get(n));
      }
    }

    Transform3D trans = new Transform3D();
    for (int n = 0; n < shadows.size(); n++) {
      BaseObject3D obj = shadows.get(n);
      if (obj instanceof Object3D) {
        ((Object3D)obj).accept(new ObjectRenderer(viewer.getGraphicsContext3D()));
      } else {
        obj.center.getTransform(trans);
        viewer.setModelTransform(trans);
        Node primitive = obj.getPrimitiveNode();
        if (primitive instanceof Shape3D) {
          viewer.draw((Shape3D)primitive);
        }
      }
    }
    :
    for (int n = 0; n < occluders.size(); n++) {
      Object3D occuluder = occluders.get(n);
      occuluder.accept(new ObjectRenderer(viewer.getGraphicsContext3D()));
    }
  }
}
```

図 5 手順 2 適用後

いてリファクタリングを用いるためには, 文脈に応じてリファクタリングを切り替えて適用しなければならない. しかしながら, どのような文脈においてどのリファクタリングを適用すればよいかは, 作業者にとって自明ではない. また, 既存のリファクタリングでは対処できない箇所が存在する. そこで本節では, 新たなリファクタリング及びリファクタリングフレームワークの提案を行う.

4.1 新しいリファクタリング

提案フレームワークの構成要素として以下の新しいリ

```
public class SceneViewer extends BranchGroup {

  public void renderField(Viewer3D viewer) {
    :
    Transform3D trans = new Transform3D();
    for (int n = 0; n < shadows.size(); n++) {
      BaseObject3D obj = shadows.get(n);
      if (obj instanceof Object3D) {
        ((Object3D)obj).accept(new ObjectRenderer(viewer.getGraphicsContext3D()));
      } else {
        :
      }
    }
    :
    for (int n = 0; n < occluders.size(); n++) {
      Object3D occuluder = occluders.get(n);
      occuluder.accept(new ObjectRenderer(viewer.getGraphicsContext3D()));
    }
  }
}
```

```
private class ObjectRenderer extends ObjectVisitor {
  GraphicsContext3D graphicsContext3D = null;

  public ObjectRenderer(GraphicsContext3D graphicsContext3D) {
    :
    this.graphicsContext3D = graphicsContext3D;
  }
}
```

図 6 手順 3 適用前

```
public class SceneViewer extends BranchGroup {

  public void renderField(Viewer3D viewer) {
    :
    Transform3D trans = new Transform3D();
    for (int n = 0; n < shadows.size(); n++) {
      BaseObject3D obj = shadows.get(n);
      if (obj instanceof Object3D) {
        ((Object3D)obj).accept(new ObjectRenderer(viewer));
      } else {
        :
      }
    }
    :
    for (int n = 0; n < occluders.size(); n++) {
      Object3D occuluder = occluders.get(n);
      occuluder.accept(new ObjectRenderer(viewer));
    }
  }
}
```

```
private class ObjectRenderer extends ObjectVisitor {
  GraphicsContext3D graphicsContext3D = null;

  public ObjectRenderer(Viewer3D viewer) {
    :
    this.graphicsContext3D = viewer.getGraphicsContext3D();
  }
}
```

図 7 手順 3 適用後

```
private class ObjectRenderer extends ObjectVisitor {
  GraphicsContext3D graphicsContext3D = null;

  public ObjectRenderer(Viewer3D viewer) {
   :
    this.graphicsContext3D = viewer.getGraphicsContext3D();
  }

  @Override
  public void postVisit(Object3D obj) {
    Transform3D t = transforms.pop();
    if (!obj.hasChildren()) {
      graphicsContext3D.setModelTransform(t);
      :
    }
  }
}
```

図 8 手順 4 適用前

ファクタリングを導入する.

- **隠蔽による型の置き換え**:
  前節で説明した手順 4 に相当する. 型 c_1 を持つ変数の型を c_1 を隠蔽している型 c_2 に置き換える.
  (1) c_1 型の局所変数またはフィールドの型を c_2 に置き換える.
  (2) 代入文については右辺の型が c_1 そのものだった場合, c_1 をパラメータとして c_2 のインスタンスを生成して右辺を置き換える. c_2 オブジェクトから c_1 オブジェクトを取得している場合は c_2 オブジェクトそのものを代入するように置き換える.
  (3) c_1 のメソッドを呼び出している箇所は, c_1 オブジェクトを適切な c_2 オブジェクトから取得するようにする.

```
private class ObjectRenderer extends ObjectVisitor {
  Viewer3D viewer = null;

  public ObjectRenderer(Viewer3D viewer) {
   :
    this.viewer = viewer;
  }

  @Override
  public void postVisit(Object3D obj) {
    Transform3D t = transforms.pop();
    if (!obj.hasChildren()) {
      viewer.getGraphicsContext3D().setModelTransform(t);
      :
    }
  }
}
```

図 9 手順 4 適用後, 手順 5 適用前

```
private class ObjectRenderer extends ObjectVisitor {
  Viewer3D viewer = null;

  public ObjectRenderer(Viewer3D viewer) {
   :
    this.viewer = viewer;
  }

  @Override
  public void postVisit(Object3D obj) {
    Transform3D t = transforms.pop();
    if (!obj.hasChildren()) {
      viewer.setModelTransform(t);
      :
    }
  }
}
```

図 10 手順 5 適用後

- **戻り値オブジェクトの導入**:
  引数オブジェクトの導入と対をなすものである.
  (1) 置き換えようとしている戻り値を表現する新しいクラスを作成する.
  (2) 新しいクラスのインスタンスを返すように修正する.
  (3) 戻り値の型を新しいクラス名に変更する.
  (4) 呼び出し側の元の戻り値を戻り値オブジェクトから取得するようにする.
- **オブジェクトそのものの返却**:
  オブジェクトそのものの受け渡しと対をなすものである.
  (1) データを保持しているオブジェクトそのものを, 戻り値として返すようにする.
  (2) 戻り値の型をオブジェクトそのものに変更する.

(3) 呼び出し側の元の戻り値を戻り値オブジェクトから取得するようにする.

4.2 リファクタリングフレームワーク

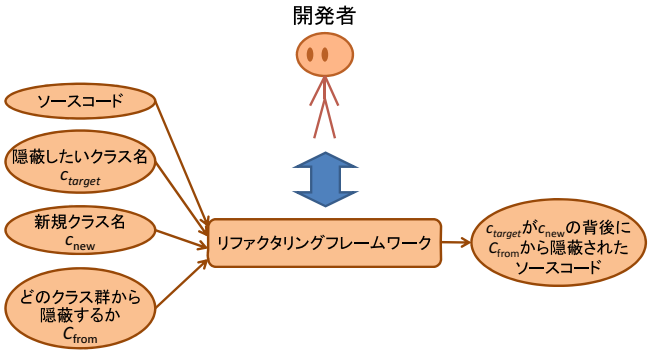

図 11 リファクタリングフレームワーク

本リファクタリングフレームワークではまず, 隠蔽したいクラス c_{target}, c_{target} を隠蔽する新規クラス c_{new}, どのクラス群 C_{from} から隠蔽するかを入力として与える. 次に c_{target} の C_{from} 上での出現位置に応じて, リファクタリングを切り替えて適用する. C_{from} が複数存在する場合は呼び出し関係において最も呼び出し元のクラスから順に適用する. 出現位置と適用するリファクタリングの対応関係を表 1 に示す. 複数の位置で同時に出現している場合は, 表 1 の (a), (b), (c), (d), (e), (f) の優先順位で適用する.

表 1 出現位置毎の適用リファクタリング

C_{from} における c_{target} の出現位置	リファクタリング
(a) 仮引数	引数オブジェクトの導入
(b) 戻り値かつレシーバの型が c_{new} でない	戻り値オブジェクトの導入
(c) フィールド, 局所変数	隠蔽による型の置き換え
(d) レシーバ	委譲の隠蔽
(e) 実引数	オブジェクトそのものの受け渡し
(f) return 文で返す値	オブジェクトそのものの返却

5. 事例研究

提案手法により複数クラスからの情報隠蔽を実現できることを確認するため, 2 節で紹介した変更作業を本フレームワークによって再現することができるかの確認を行った. その過程のソースコードの変更内容は図 3〜図 10 と同様である. 提案手法によってこの変更作業を正しく再現できることが分かった. 各手順の内容を表 2 に示す. 適用したリファクタリングは表 1 の (a), (d), (e), (c), (d) の計 5 回で, 変更行数は全体で 60 行であった.

6. 考察と今後の課題

実際の設計変更例の中の小さくない部分に対して本リファクタリングフレームワークを適用することで, 正しく変更が完了することが確認できた. この事例では, リファクタリングを 5 回適用しなければ一連の変更作業を完結させることができなかった. リファクタリングは実際にソースコードに適用してみなければ, その正確な効果を把握しにくい部分があるため, 複数クラスを横断する複雑な変更において意図した効果を得るには, どのようにリファクタリングを組み合わせて適用すればよいかについての膨大な試行錯誤が必要となる. 本フレームワークは横断するクラスが増えた場合でも, どの部分にどのリファクタリングをどのような順番で適用すればよいかが一意に定まるため, 情報隠蔽に関わる設計変更においては, 大規模になればなる程有用になると考えられる. 今回の事例では表 1 の (b), (f) のパターンは出現しなかった. 今後, 他の事例にも適用することによって, 本リファクタリングフレームワークの一般性について確認していく予定である. また, 今回の提案ではツールによる自動化は行われていないが, 今後文脈の判断, リファクタリングの推奨, 自動化されていないリファクタリングなどについて自動化し, ツールとしての完成を目指す予定である.

7. おわりに

複数クラスを横断するような情報隠蔽を行う際のリファクタリングの適用を支援するリファクタリングフレームワークの提案を行った. また, 提案フレームワークの有効性を確認するために, 我々の研究室で実際に行われた設計変更事例に対してフレームワークの適用を試みた. その結果, 提案フレームワークによって事例中の一定規模の変更作業を正しく完了できることを確認した. 今後, 他の事例への適用を試みると同時に, 現状ではすべて手作業に頼っている改変手続きの一部を自動化するツールの開発を行う予定である.

参考文献

[1] マーチン・ファウラー著, 児玉 公信ら訳: リファクタリング プログラミングの体質改善テクニック, ピアソン・エディケーション (2004).
[2] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson and Danny Dig: A comparative study of manual and automated refactorings, *Proc. The European Conference on Object-Oriented Programming (ECOOP)* (2013).
[3] Emerson Murphy–Hill, Chris Parnin and Andrew P. Black: How we refactor, and how we know it, *IEEE Trans. on Software Engineering (TSE)*, Vol. 38, No. 1, pp. 5–18 (2012).
[4] Tom Mens and Tom Tourwé: A survey of software refactoring, *IEEE Trans. on Software Engineering*

表 2 変更手順

	適用リファクタリング	変更行数	変更クラス
手順 1	引数オブジェクトの導入	25 行	RWTCanvas3D SceneViewer Viewer3D
手順 2	委譲の隠蔽	18 行	SceneViewer Viewer3D
手順 3	オブジェクトそのものの受け渡し	5 行	SceneViewer ObjectRenderer
手順 4	隠蔽による型の置き換え	17 行	ObjectRenderer
手順 5	委譲の隠蔽	21 行	ObjectRenderer Viewer3D
全体		60 行	RWTCanvas3D SceneViewer ObjectRenderer Viewer3D

(TSE), Vol. 30, No. 2, pp. 126–139 (2004).

[5] Mohsen Vakilian, Nicholas Chen, Roshanak Zilouchian Moghaddam, Stas Negara and Ralph E. Johnson: A compositional paradigm of automating refactorings, *Proc. The European Conference on Object-Oriented Programming (ECOOP)* (2013).

[6] 新田 直也, 久野 剛司, 久米 出, 武村 泰宏: 3D ゲームエンジン Radish の開発とそのアーキテクチャ比較への応用, 日本デジタルゲーム学会, デジタルゲーム学研究, Vol. 4, No. 1, pp.1–12 (2010).