リアクティブプログラミングを用いた ソフトウェア保守性の向上

大飛 尚登　　（指導教員　西村 俊二）

平成31年1月25日

Improvement of Software Maintainability using Reactive Programming

NAOTO OOTOBI　(ACADEMIC ADVISOR　SHUNJI NISHIMURA)

概要：近年，モバイル機器の普及に伴い，通信処理やGUIの更新といった非同期の処理が必要とされる場面が増えている．　しかし，プログラミングにおいて非同期の処理を記述するのは，想定するべき様々なシチュエーションを把握し各場合においての処理を記述するという多大な手間がかかる作業であり，効率化が求められている．そこで本研究では，リアクティブプログラミングと呼ばれるプログラミングパラダイムを用いた際の，非同期処理の可読性や保守性を測定する．その後，従来の方法での記述と比較した結果，非同期処理としてインターネット通信を多用するアプリケーションにおいては，リアクティブプログラミングを使うことにより生産性，保守性を向上できることが明らかになった.

キーワード: 非同期処理，リアクティブプログラミング，サイクロマティック複雑度

1.　緒言

現在，ハードウェア技術の進歩や IoT の利用拡大に伴い，スマートフォンをはじめとしたモバイル機器が普及している．これに伴って，ユーザの入力に反応し GUI の更新やバックグラウンドでの通信を行うといった機器とユーザ間の対話性を求められるようになった.

そうした要求に応えるためにソフトウェア開発の現場においては，与えられた情報をリアルタイムで処理する手法の重要性が大きくなっている[1]．しかし，システムやアプリケーションで扱うデータ量は年々増加しているものの，限られた時間で処理できるデータ量には限界がある．そこで，複数の処理を独立して行える非同期処理の手法についても，その需要が高まっている.

また，モバイル機器ではインターネットの通信を介して，アプリケーションの機能の改善や追加を行うことが一般的である．運用中のソフトウェアの更新を行う機会が増えたため，ソフトウェアのメンテナンス性もシステム開発における課題の 1 つとなっている.

そこで本研究では，これらの問題に対するアプローチの 1 つとして，データと処理の関係を記述することで，リアルタイムな処理と非同期処理を実現できるリアクティブプログラミングの利用を提案する．新たなプログラミングパラダイムであるリアクティブプログラミングを用いて非同期処理を行う Android アプリケーションの開発を行う．そして，完成したソースコードのサイクロマティック複雑度と単語数を計測することで，ソフトウェアの保守性について検証し，考察を行った.

2.　事前知識

2.1.　非同期処理

プログラムは 1 つの処理を実行している間は，基本的に別の処理を実行することはできず，動作中の処理が終わった後に次の処理を行う．このように逐次的に実行される処理のことを同期処理という.

これに対して，非同期処理とは，呼び出し元の動作を中断せずに，実行される処理のことをいい，処理が終了すると結果を呼び出し元に通知する仕組みとなっている[2]．呼び出し元は，非同期処理が実行されている間も，ユーザ入力のようなイベント待ちや別の処理の実行を行うことができる.

代表的なものでは，インターネット通信が挙げられる．例として，モバイル機器のブラウザを用いて，サーバの画像をダウンロードするといった通信処理を考える．画像のダウンロードに 10 秒かかるとした場合，同期処理の場合では画像のダウンロード中には，GUI の更新やユーザ入力が一切できなくなる．これは，ダウンロードするための通信に処理の一切を占有され，その他の入力や処理をプログラムが受け付けないためである．しかし，画像のダウンロードを非同期処理とすると，ダウンロード中であっても，その他の処理は継続して行える．通信をメインの処理とは別の部分で行うことで，メインの処理を阻害せずに通信を行えるのである．これが非同期処理の大きな利点であり，PC

やスマートフォンをはじめとした様々なシステムで利用されている. 本研究では, この非同期処理に着目し, 研究を行う.

2.2. リアクティブプログラミング

リアクティブプログラミングとは，変数やイベントといったあらゆるデータは時間軸に沿って流れてくるように発生するという特徴に注目し，発生したデータに反応してプログラムが処理を行うという考え方のプログラミングパラダイムである[3].

Fig.1 はリアクティブプログラミングを図で表したものである．矢印はデータの流れを示す時間軸であり，図上部の様々な形をした図形が入力データ，図形の右側にある縦線がデータを受け取った際の処理を始めるタイミングである．入力データは矢印の方向に移動していると想像してほしい．リアクティブプログラミングでは，プログラムは流れてくるデータに対して，様々なアプローチをとることができ，Fig.1 では，丸という形を満たすデータを抜き出し，形状を変更している．これにより，実際には 6 つあった入力データは，処理する側の解釈では，図下部の矢印にある 2 つのひし形が入力データということになる．データの選別と加工を処理開始前に行うことで，処理内容が洗練され，プログラム作成者でなくとも理解しやすい記述となるというメリットがある.

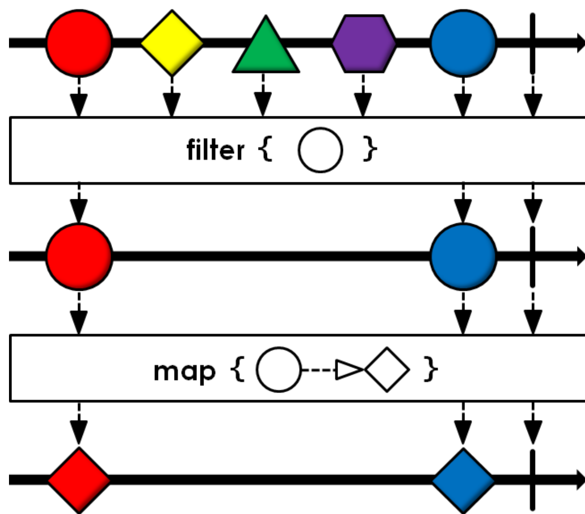

Fig.1 リアクティブプログラミングの考え方

リアクティブプログラミングは，ごく自然的な考え方であり, コンピュータ上のプログラム以外でも, 様々な場面や道具で用いられている考え方である．一例として，温度計が挙げられる．温度計は，外気温というデータを周囲の環境から受け取って，その結果を数値として出力する器具である．ここで重要なのは，外気温の表示という一連の処理が始まるきっかけが，温度計自身の働きかけによるものではなく，外気温という温度計外部からの入力であるという点である.

温度計が気温の変化を確認するのではなく，外気が温度計の温度を変化させるという観点で考えることがリアクティブプログラミングの大きな特徴である．温度計自身は，自ら気温を計測しに行くタイミングというものを気にする必要がなくなり，外気温という入力があったからその結果を出力するというシンプルな考え方で動作できる.

この考え方をプログラムに応用したものが，リアクティブプログラミングである．データの発信側は，自身のタイミングで，データを発信することができ，また受信側もデータの発信に合わせて処理を開始するという発信側のタイミングを気にする必要がなくなる．そのため，タイミングを見計らうための処理が減り，シンプルな記述を実現できる．アニメーションや GUI の入出力のような時間とともに状態が変化する処理や, 結果の出力に時間がかかる通信のような処理に対して有効に働くことが期待でき，本研究ではこのプログラミングパラダイムを用いて従来の方法との保守性の比較を行う.

2.3. サイクロマティック複雑度

THOMAS J. McCABE が提唱したプログラムの複雑度を示す指標である[4]．定義は，プログラムの制御フローを有効グラフとして描き，その経路から

グラフの辺数－グラフの頂点数
＋（2×連結成分の数）

という計算式に当てはめることで求められる値をサイクロマティック複雑度の評価値とする. 連結成分とは, 辺で結ばれた頂点の集合の数を表す.

例として，List1 のプログラムについて考えてみる．このプログラムの制御フローをグラフで表現すると，Fig.2 となる. このグラフからは, グラフの辺数が 9 本, 頂点数が 8 個，連結成分は 1 つということがわかる．それぞれの数を前述の計算式に代入すると，9－8＋(2×1)より，サイクロマティック複雑度は 3 という結果が得られる.

しかし，制御フローをグラフで表す作業は，プログラムの規模が増大するほど図が複雑になり，手間と時間がかかるため，実用的とは言い難い．そこで，プログラムのコードメトリクス計測では，より単純な方法として，if や while，switch のような分岐処理を行う関数の使用回数に 1 を足した数値がサイクロマティック複雑度の評価値として用いられる．List1 においても，if 関数の数は 2 つであり，この数に 1 を足すと 3 という結果が出る．これは，辺数や頂点数を用いた計算式による求め方と同一の結果となっている.

サイクロマティック複雑度は，数値が高いほど，ソースコードが複雑であることを示しており，転じて保

守性が低いといえる．

List1 プログラム例

```
void cyclomaticComplexity(){
  if(c1){
    if(c2){
      f5();
      f7();
    }
    else{
      f6();
    }
  }
  else{
    f2();
    f4();
    f7();
  }
  f8();
}
```

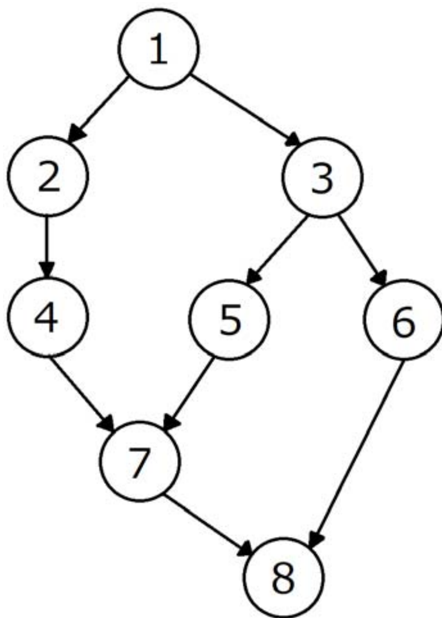

Fig.2 サイクロマティック複雑度 3 のグラフ

3. 実験内容

本実験は，オープンソース IDE である Android Studio，Java でリアクティブプログラミングを行うためのライブラリである RxJava，作成したアプリケーションの実行環境として Android 端末である Fire HD 8 タブレットを用いて行う．モバイル機器のアプリケーションを想定し，Android OS を実行環境とする． Fire HD 8 タブレットは，Amazon が開発した独自の OS である Fire OS が採用されているが，この OS は Android OS をベースに開発されているため，Android 用アプリケーションも問題なく動作する．

実験手順は，Android Studio を用いて同一の動作をする Android アプリケーションを，従来の Java のみで作成したものと，RxJava を利用して作成したものの 2 つを用意する．用意するアプリケーションの機能は，代表的な非同期処理の 1 つとしてインターネット通信を行うものにした．そして，完成したそれぞれのプログラムを実行環境の Android 端末上で動作確認を行い，同一の動作を行っていることを確認する．確認が終わったら，サイクロマティック複雑度と単語数を計測し，その結果を確認，考察する．

ここで，RxJava を用いた非同期処理の記述例を List2 に示す．

List2 RxJava の非同期処理

```
Observable.just(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
  .filter(i -> i%2==0)
  .map(i -> i * 10)
  .subscribe(observer);

Observer<Integer> observer = new Observer<Integer>() {

  //データが流れてくる前に行う処理
  @Override
  public void onSubscribe(Disposable d) {
    System.out.print("Start¥n");
  }

  //データが流れてきた際の処理
  @Override
  public void onNext(Integer i) {
    System.out.print(i+"¥n");
  }

  //エラー発生時の処理
  @Override
  public void onError(Throwable e) {
    System.out.print(e.toString());
  }

  //データが流れ終わった際の処理
  @Override
  public void onComplete() {
    System.out.print("Complete");
  }
};
```

Observable はストリームに流すデータを決め，流れてくるデータに処理を施す関数である．List2 の例では，1 から 10 までの整数をデータとして流し，filter 関数を

用いて偶数のみを選別し，map 関数で流れてきた値を 10 倍するという処理を施している．Observable 末尾の subscribe 関数は，流れてきたデータの処理を開始する関数であり，Observer で指定した処理が始まる．Observer には，4 つのインターフェースが用意されており，ストリームの状況に応じてそれぞれの処理が自動で適用される．List2 のプログラムを実行した結果を List3 に示す．

List3　List2 の実行結果

```
Start       //onSubscribe()の処理
20          //onNext()の処理
40          //onNext()の処理
60          //onNext()の処理
80          //onNext()の処理
100         //onNext()の処理
Complete    //onComplete()の処理
```

データが流れてくる前に，onSubscribe()が実行され，「Start」が表示される．その後，Observable によって選別，加工された 20，40，60，80，100 という整数が流れ，それぞれに onNext()が実行される．ストリームを流れるデータがなくなると，onComplete()が実行され「Complete」を表示し，一連の処理を終了する．これが，RxJava を用いたリアクティブプログラミングの一例である．

単語数とは，メソッド中に存在する意味のある文字列を数えたものである．例えば「printf(“hello world”);」のような文は，「printf / (/ ” / hello / world / ” /) / ;」といった 7 つの単語で構成されている．括弧やセミコロンも単語の 1 つとして数えられる．単語数は，プログラムの構成要素の数を表しているため，この値が低いほど可読性の高い効率的なプログラムであるといえる．また，機能追加に伴った単語数の変化量は，コードの変更点の数を表すため，この変化量が大きいほど機能更新にかかるコストやリスクも大きくなることを示す．そのため，本実験における保守性の指標の一つとして計測を行う．
サイクロマティック複雑度の計測においては，オープンソースソフトウェアツールの「lizard」を用いる．このツールは，Github で公開されており [5]，サイト上にあるテキストボックスにプログラムコードを打ち込むことで，サイクロマティック複雑度や有効ステップ数といった様々な評価値をメソッド単位で計測することができる．ツール実行時の一例を Fig.3 に示す．Hoge クラスに定義されている main，piyo といった 5 つのメソッドのそれぞれの評価値が表示されている．NLOC が有効ステップ数，Complexity がサイクロマティック複雑度，Token #が単語の数，Parameter #がパラメータ数を表している．

今回の実験では，全国の高等専門学校のホームページをアクセス対象とした．アクセス対象の数は 57 個であるため，非同期処理で行うインターネット通信の回数を 1 回，5 回，10 回，20 回，30 回，40 回，50 回，57 回に変更した場合の複雑度を計測する．

File Type .java　Token Count 213　NLOC 39

Function Name	NLOC	Complexity	Token #	Parameter #
Hoge::main	3	1	18	
Hoge::piyo	4	1	17	
Hoge::hoge	5	2	31	
Hoge::bar	9	3	57	
Hoge::foo	16	4	70	

Fig.3　lizard 実行結果

4.　実験結果

4.1.　作成したアプリケーション

本実験の検証のために作成したアプリケーションの解説を行う．アプリケーションの実行画面を Fig.4 に示す．

Fig.4　作成したアプリケーション

アプリを起動し，左上の「HTML GET」のボタンをタップすると，バックグラウンドでインターネット通信が開始され，各高等専門学校のホームページに順次アクセスされていく．このときのホームページへのアクセスには，アプリケーション側に各高等専門学校の

URL を文字列型配列として設定しておいたものを使用する．そして，トップページの HTML に設定されているタイトルの文字列を取得し，画面中央のテキストボックスに表示していく．すべての高等専門学校へのアクセスとタイトルの画面表示が終わると，「HTML GET」ボタンをタップした瞬間から，最後のタイトル表示の完了までにかかった時間をインスタントテキストで表示する．画面右上の「CLEAR」と書かれたボタンをタップすると，テキストボックスの内容がリセットされる. 以上の機能を従来の Java のみで作ったものと，RxJava を用いたものの 2 パターンで作成し，検証を行った．

今回作成したプログラムは，大きく分けて 3 つの機能で実現されている．ボタンタップを起点としてイベントを起こす機能，インターネット通信のための手続きを行い, 各高等専門学校ホームページにアクセスし, HTML からタイトルに指定されている文字列を取り出す機能，取得した文字列を画面に反映させ，次の通信を行うための非同期処理の機能の 3 つである．

まず，ボタンタップを起点としたイベントの開始はメインの処理にあたる部分であり，非同期処理ではない．ホームページにアクセスし，タイトルを取得する機能は，非同期処理の一部として扱うことになるが，文字列を取得する手段については本実験においては重要でなく，得られた文字列を非同期でどのように扱うのかが最も重視すべきことである．これらの機能で結果に差が生じることは実験目的にそぐわないため，プログラムでは同一のメソッドを使用している. しかし，取得したテキストを画面に反映させる非同期処理は，プログラミングパラダイムによる処理方法の差が生じるため, 従来の方法と RxJava を用いたものでは記述を異なるものにし，違いを確かめる必要がある．

Java のみで作成する従来の方法では，AsyncTask クラスと呼ばれる非同期処理を行うための機能を用いる必要がある．これは，メインスレッドとは別のスレッドで処理を行うための機能であり，非同期処理と画面更新処理ができるようになるものである．

実験に使用したプログラムの一部を List4，List5 に示す．AsyncTask を用いたプログラムでは，AsyncTask クラスに非同期の処理を定義し，メインスレッド上で AsyncTask クラスのインスタンスを生成し，特定の関数を使うことで非同期処理が実行される．それに加え今回のプログラムでは，現在実行中の非同期処理が終わったら，次の非同期処理を始めるという手順を繰り返す必要がある．今回は，AsyncTask クラスにコールバック用のインターフェースを用意し，処理終了時の処理をインスタンス生成時に記述する方法をとることにした．

また，AsyncTask クラスから生成されたインスタンスは，非同期処理を 1 度完了すると破棄されてしまうため，1 つのインスタンスを何度も使いまわすといった方法が使えない．そのため，アクセスするホームページの数だけ，コールバック内部でインスタンスを新たに生成する必要が生じ，さらに同数のコールバックを定義しなくてはならないため，ネストが深く読みづらいコードになってしまう．List4，List5 では，取得するタイトル数を 2 つに制限しているが，これがさらに増えていくと，比例してネストやステップが増えていく．

List4　AsyncTask を用いた非同期処理（1）

```
//AsyncTask クラスの定義
class MyAsyncTask extends AsyncTask<String, Void,
String> {
    private AsyncTaskCallbacks callback;
    MyAsyncTask(AsyncTaskCallbacks callback) {
        super();
        this.callback = callback;
    }

    @Override
    protected void onPreExecute() {
        super.onPreExecute();
    }

    @Override
    protected String doInBackground(String... params) {
        String sb = HtmlGet.htmlGet(params[0]);
        return HtmlGet.titleGet(sb);
    }

    @Override
    protected void onPostExecute(String result) {
        super.onPostExecute(result);
        this.callback.onTaskFinished(result);
    }
}
//非同期処理の定義
public void asyncProcess(Context context) {
  long start = System.currentTimeMillis();

  try{
  MyAsyncTask task = new MyAsyncTask(new
AsyncTaskCallbacks() {
    @Override
    public void onTaskFinished(String result) {
      textView.append(result + "¥n¥n");
      scrollView.scrollTo(0, textView.getBottom());
      try{
      MyAsyncTask task = new MyAsyncTask(new
AsyncTaskCallbacks() {
        @Override
```

List5 AsyncTask を用いた非同期処理（2）

```
public void onTaskFinished(String result) {
textView.append(result + "¥n¥n");
scrollView.scrollTo(0, textView.getBottom());
                long end = System.currentTimeMillis();
                Toast.makeText(context, (end - start) + "ms",
Toast.LENGTH_LONG).show();
            }
        });
        task.execute(urls[1]);
        }catch(Exception e){
            textView.append("ERROR");
            e.printStackTrace();
        }
    });
    task.execute(urls[0]);
    }catch(Exception e){
        textView.append("ERROR");
        e.printStackTrace();
    }
 }
}
```

対して，RxJava を用いたプログラムでは，リアクティブプログラミング特有の，データをストリームに流すという仕組みを使う．List6 は本実験に使用したプログラムの一部である．今回は，各高等専門学校ホームページの URL を格納した文字型配列を順に流し，別スレッド上でホームページへのアクセスとタイトルの取得を行ったのち，メインスレッド上で画面に反映するという手順を行っている．Observer は流れてきたデータをどのように扱うのかを定義している．Observable はデータを流すストリームを表し，流すデータの個数制限や選別，加工方法を決めることができる．今回のプログラムでは，初めは URL の文字列を流しているが，map 関数の適用後，observer にデータが渡されるときには，タイトルの文字列という別のデータに変化している．

前述の AsyncTask を用いたものでは，インターネット通信の開始を，画面への反映が終わったタイミングで行うということを明示的に記述する必要があったが，RxJava を用いたものでは，このタイミングを意識することなく，データが流れてきたら処理を開始するというような記述ができる．

また，RxJava では，ラムダ式でメソッドチェーンを用いた記述をすることができるため，ネストが深くなることもなくシンプルな見た目となり，データにどのような加工を施しているのかが比較的理解しやすいとう点がある．

List6 RxJava を用いた非同期処理

```
public void asyncProcess(Context context) {
    long start = System.currentTimeMillis();

    //非同期処理の定義
    Observer<String> observer = new Observer<String>() {
        @Override
        public void onSubscribe(Disposable d) {}

        @Override
        public void onNext(String s) {
            textView.append(s + "¥n¥n");
            scrollView.scrollTo(0, textView.getBottom());
        }

        @Override
        public void onError(Throwable e) {
            textView.append("ERROR¥n");
            textView.append(e.toString() + "¥n");
        }

        @Override
        public void onComplete() {
            scrollView.scrollTo(0, textView.getBottom());
            long end = System.currentTimeMillis();
            Toast.makeText(context, (end - start) + "ms",
Toast.LENGTH_LONG).show();
        }
    };

    //ストリームの設定
    Observable.fromArray(urls)
                .subscribeOn(Schedulers.io())
                .take(2)
                .map(HtmlGet::htmlGet)
                .map(HtmlGet::titleGet)
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(observer);
}
```

4.2. 複雑度の計測

作成したプログラムのサイクロマティック複雑度を lizard により算出する．結果を表にまとめたものを Table1 に示す．また，通信回数の変化による複雑度の増加量を表したグラフを Fig.5 に示す．

計測の結果，AsyncTask を用いる方法ではインターネット通信を行った回数に比例して，サイクロマティック複雑度が上昇していることが分かった．これは，コードにある try-catch 文を複雑度としてカウントしているためと考えられる．try-catch 文も try の処理中にエ

ラーが起こるという条件を満たした場合，catch の処理を行うという分岐処理の 1 種なので，サイクロマティック複雑度の増加につながる．

Table1 複雑度の比較

	サイクロマティック複雑度	
通信回数	AsyncTask	RxJava
1回	2	1
5回	6	1
10回	11	1
20回	21	1
30回	31	1
40回	41	1
50回	51	1
57回	58	1

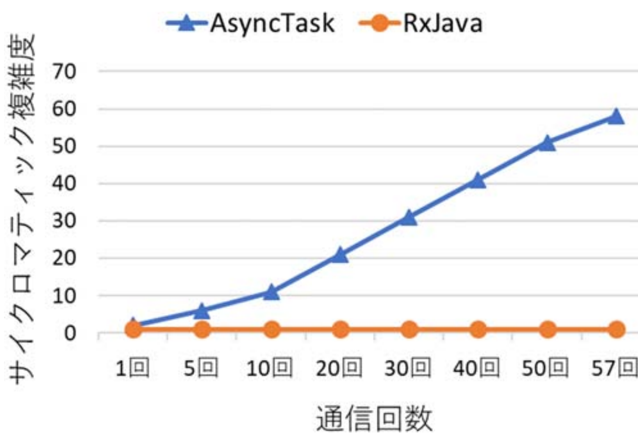

Fig.5 サイクロマティック複雑度の比較

インターネット通信は，端末側の不具合や電波状況といったエラーを起こしうる要因が多い．処理が正常に動作し終了する保証がないため，Java で提供されているインターネット通信の機能では，通信を開始する際に必ず try-catch で記述部を囲み，エラー時の対処を決めておく必要がある．そのため，従来の方法では，通信を行おうとする度に try-catch の記述をする必要があるため，通信の回数が増えるほど比例して複雑度も増えたと考えられる．

対して，RxJava を用いたプログラムでは，サイクロマティック複雑度が常に一定という結果となった．これは，AsyncTask では用いた try-catch 文を利用しなかったためであると考えられる．RxJava では，catch 文にあたるエラー時の処理を決める機能が元から存在するため，try-catch を用いる必要がない．またインターネット通信を開始するという記述が一つしかない点も複雑度を低く抑えている要因といえる．通信の開始を行うタイミングを製作者が都度指定する必要がないため，通信回数が増えたとしても，データが流れてきた時に通信を始めるという一度の記述で済むのである．以上の理由から RxJava を用いたプログラムコードではサイクロマティック複雑度が増えなかったと考えられる．

4.3. 単語数の計測

lizard を用いた単語数の結果をまとめたものを Table2 に示す．また，通信回数の変化による単語数の変化を Fig.6 に示す．

Table2 単語数の比較

	単語数	
通信回数	AsyncTask	RxJava
1回	128	210
5回	448	210
10回	848	210
20回	1648	210
30回	2448	210
40回	3248	210
50回	4048	210
57回	4848	210

Fig.6 単語数の比較（1 回～57 回）

計測結果から，従来の AsyncTask を用いる方法では，通信回数が増えるほど，単語数が増えていることがわかる．通信の回数だけ，コールバックを使用する必要があるため，記述量も増えていき，このような結果になったと考えられる．

対して RxJava は，一定数を保ち続けているという結果になった．これは，流れてくるデータの数を制限する関数を用いたためである．この関数の引数を変更するだけで，通信回数を変更できるため，単語数の変化

がなかったと考えられる．

以上の結果から，通信回数が 5 回目以降の結果では，AsyncTask の単語数が RxJava より多かったが，1 回目の場合は，RxJava の方が多いという結果が明らかになった．そこで，計測の範囲を小さくし，より詳細な記録をとることにした．調べる通信回数を 1 回目から 5 回目まで 1 回刻みで変更し，それぞれの場合における単語数を計測した．計測結果を Table3，グラフに表したものを Fig.7 に示す．

Table3　単語数の比較

	単語数	
通信回数	AsyncTask	RxJava
1回	128	210
2回	208	210
3回	288	210
4回	368	210
5回	448	210

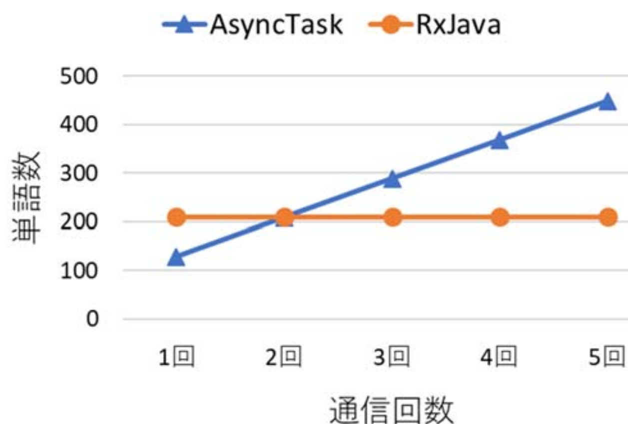

Fig.7　単語数の比較（1 回〜5 回）

計測の結果，通信回数が 2 回以下の場合，AsyncTask の単語数が RxJava を下回ることが分かった．この結果から，従来の方法でも RxJava より効率的に非同期処理を記述できる場合があると判明した．本実験では，最大 57 回のインターネット通信を行っているが，小規模のプログラムで 10 回以上の通信を 1 度の処理で行うことは稀であるため，一概に RxJava が優れているわけではないといえるだろう．

5.　考察

サイクロマティック複雑度を比較した結果では，リアクティブプログラミングを用いた方法の方が複雑度を低く抑えられることが分かった．複雑度が 10 を超えることはなく，処理内容が理解しやすいプログラムであるといえるだろう．サイクロマティック複雑度はコード中に存在する分岐処理の数を示すため，保守性に限らず，アプリケーションの開発段階におけるテストケースの数を減らすことにつながる．この結果から，生産性，保守性の面においてリアクティブプログラミングは有効であるといえるだろう．

単語数を比較した結果では，通信の回数が少ない場合に限り，従来の方法の方が少ない単語数でプログラムを作成できることが判明した．通信回数が 2 回の場合，その差は微々たるものだが，1 回の場合は 80 以上の差が生まれている．これは，リアクティブプログラミングのデータの扱い方であるストリームが関係している．ストリームの仕組みをプログラムで実現するための記述により，単語数が増えたものと考えられる．しかし，通信回数が増えるほど，従来の方法は単語数が増え続け，対してリアクティブプログラミングは常に一定をキープしている．一度，ストリームの仕組みを確立できれば，少ない変更量で対応できることがわかる．この結果から，リアクティブプログラミングは単純な記述量では従来の方法に劣る場合もあるが，機能の更新や追加への対応力は高いといえるだろう．

計測結果から，インターネット通信を多用する大規模なプログラムを作成する場合には，リアクティブプログラミングは有効な手段であるといえるだろう．

今回の実験に使用した RxJava はオープンソースのライブラリであり，他のライブラリに対する依存がないため，ライブラリの導入が容易に行える．そのため，運用中のアプリケーションでもリアクティブプログラミングを適用することが可能である．すでに世に存在している多くのアプリケーションにも保守性向上の可能性があるといえるだろう．しかし，本実験を経て，リアクティブプログラミングは学習コストが高いということが分かった．ストリームのとらえ方や，処理の方法などオブジェクト指向とは違う考え方を理解するのは，相応の労力が必要となる．半端な知識で活用した場合，プログラムが動作しなくなる原因にもなりかねないため，利用には注意が必要だろう．

6.　結言

本論文では，モバイル機器や IoT の普及により，需要の高まったインターネット通信を行うアプリケーションについて，リアクティブプログラミングを用いた際の保守性の変化に関する実験を行った．実際にアプリケーションを作成し，保守性の指標にサイクロマティック複雑度と単語数を用いて，プログラムコードを評価した結果，一部の条件下では，単語数において劣る場合もあったが，サイクロマティック複雑度は常に優れた数値が得られた．総じて，リアクティブプログラミングを用いた場合，従来の方法と比較して，より保守性が高まるという結果が得られたといえる．

今回の実験には，Java でリアクティブプログラミングを行うためのライブラリである RxJava を用いた．しかし，C#や JavaScript など，その他のプログラミング言語にもこういったライブラリは開発されている．これらのプログラミング言語でもリアクティブプログラミングを用いた際に，保守性の向上がみられるのか，未だ研究の余地は残されている．

また，本実験では，インターネット通信に焦点を当てた．近年，医療や農業などあらゆる分野において通信を利用する機会が多く，重要性が大きいと考えたからである．しかし，非同期処理には，負荷がかかるような計算処理を別スレッドで行うバックグラウンド処理や, 同じ計算を複数のコアで処理する並列計算など，様々な種類が存在する．インターネット通信は，バックグラウンド処理の一つであり，本研究の成果だけではリアクティブプログラミングがあらゆる非同期処理に対して有効であるとは言えない．そのため，別の処理においても研究をする必要があるだろう．

謝辞

本研究に際して，様々なご指導を頂きました西村修二講師に深謝いたします．また，この研究の機会をくださった情報工学科の先生方，そして多くの知識やご指摘を下さいました同研究室の先輩・同期の皆様に厚く御礼申し上げます.

参考文献

[1] G.Salvaneschi, and M.Mezini(2014). Technische Universitat Darmstadt: Towards Reactive Programming for Object-oriented Applications, Darmstadt, Germany

[2] 佐久間隆平，福田浩章（2018）「非同期処理の記述を支援するライブラリの提案と実装」,『研究報告ソフトウェア工学（SE)』，198（19）

[3] 須田智之 : RxJava リアクティブプログラミング，株式会社翔泳社，2017，p.2-8

[4] THOMAS J.McCabe(1976).A Complexity Measure,IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. SE-2, NO. 4.

[5] “GitHub”，<https://github.com/terryyin/lizard> 2018 年 12 月 21 日アクセス