# Javaの理論と実践: 良き（イベント）リスナーであるために

## イベント・リスナーを書き、サポートする上での指針

Brian Goetz
Principal Consultant
Quiotix

2005年 7月 26日

Observerパターンが最もよく見られるのはSwing開発ですが、GUIアプリケーション以外の状況においても、Observerパターンはコンポーネント間を分離するために非常に有効です。ただし、リスナーを登録したり、呼び出したりする際に一般的な落とし穴が、幾つかあります。今回のJavaの理論と実践では、Javaの治療師であるBrian Goetzが、良きリスナーであるための、そして、正しくリスナーに接するための良き助言を提案します。

このシリーズの他の記事を見る

Swingフレームワークでは、イベント・リスナーの形でObserverパターン（publish-subscribeパターンとしても知られています）を頻繁に使います。ユーザーとの対話動作の対象であるSwingコンポーネントは、ユーザーがSwingコンポーネントとの対話動作を行うと、イベントを起動します。そしてデータ・モデル・クラスは、データが変化した時にイベントを起動します。このようにObserverを使うことによって、コントローラーがモデルから分離され、またモデルがビューから分離されるため、GUIアプリケーションの開発が単純になります。

「Gang of Four（4人組）」によるDesign Patternsという本（参考文献）では、Observerパターンを、次のように説明しています。「オブジェクト間の1対多の依存性を定義するものであり、1つのオブジェクトが状態を変更すると、そのオブジェクトに依存するものは全て自動的に通知を受け、更新される。」Observerパターンを使うと、コンポーネントを疎結合できるようになります。つまりコンポーネントは、必ずしもお互いの素性や内部に関する直接の知識を持たなくても、それぞれの状態を同期させることができ、そのためコンポーネントが再利用しやすくなるのです。

JButtonやJTableなどのAWTコンポーネントやSwingコンポーネントは、Observerパターンを使うことによって、アプリケーション内でのGUIイベントの生成と、イベントの意味体系を分離しています。同じように、TableModelやTreeModelのようなSwingモデル・クラスは、Observerを使うことによって、データ・モデルの表現をビュー生成から分離しています。そのため、同じデータに対して、複数の独立なビューを持てるようになります。Swingでは、Eventオブジェクトと EventListenerオブジェクトの階層構造を定義しており、JButton（ビジュアル・コンポーネント）やTableModel（データ・モデル）など、イベントを生成できるコンポーネントは、リスナーの登

録や登録解除のための、addXxxListener()メソッドやremoveXxxListener()メソッドを提供しています。こうしたクラスは、いつイベントを起動すべきかを決定し、起動する場合には、登録されているリスナー全てを呼ぶのです。

オブジェクトがリスナーをサポートするためには、オブジェクトは登録されたリスナーのリストを維持し、リスナーの登録、登録解除の手段を提供し、適当なイベントが発生した場合には、それぞれに対応したリスナーを呼ぶ必要があります。（GUIアプリケーションに限らず）リスナーを使い、サポートすることは簡単ですが、登録インターフェースの両側、つまりリスナーをサポートするコンポーネントにも、リスナーを登録するコンポーネントにも、注意すべき落とし穴が幾つかあるのです。

## スレッド・セーフの問題

リスナーが、自分が登録されているスレッドとは異なるスレッドから呼ばれることが頻繁にあります。別スレッドからのリスナーの登録をサポートするためには、どのような機構を使ってアクティブ・リスナーを保存、管理しているかによらず、その機構はスレッド・セーフである必要があります。Sunのドキュメンテーションに書かれている多くの例では、リスナー・リストの保存にVectorを使っています。これは問題の一部には対応していますが、問題全体に対する対応にはなっていません。あるコンポーネントがイベントを起動する時には、そのコンポーネントはリスナー・リストに対して繰り返しを行い、それぞれのリスナーを順次呼ぼうとします。ところが、リスナー・リストに対して繰り返しが行われている時に、あるスレッドがリスナーを追加、削除しようとすると、同時修正（concurrent modification）の危険性が出てきます。

### リスナー・リストを管理する

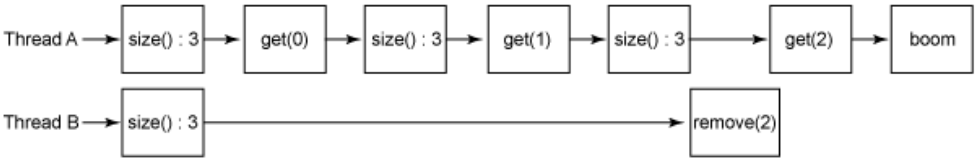
例えば、リスナー・リストの保存にVector<Listener>を使うとしましょう。確かにVectorクラスはスレッド・セーフです。つまり、追加的な同期化を行わなくても、Vectorのデータ構造を破損する危険性無しにVectorのメソッドを呼ぶことができます。しかし、コレクション（collection）に対して繰り返しを行うと、「チェックして、その後に行動（check-then-act）」というシーケンスになり、もし繰り返しの最中にコレクションが修正されると、フェールする可能性があります。例えば、繰り返しが開始した時に、リスナー・リストに3つのリスナーがあるとしましょう。Vectorに対して何度も繰り返しを行うと、検索すべき要素がなくなるまで、繰り返しsize()とget()を呼ぶことになります。これをリスト1に示します。

### リスト1. 安全でない、Vectorの繰り返し

```
Vector<Listener> v;
for (int i=0; i<v.size(); i++)
  v.get(i).eventHappened(event);
```

もしここで、Vector.size()を最後に呼んだ直後に、誰かがリスナーをリストから除いたら、どうなるでしょう。Vector.get()は、nullを返す（前回チェックした時からvectorの状態は変化しているので、これは正しいのです）ので、eventHappened()を呼ぼうとする時にはNullPointerExceptionを投げざるを得なくなります。これが、「チェックして、その後に行動」というシーケンスの例です。つまり、他に要素が無いかどうかをチェックし、無ければ次の要素に進むのですが、同時修正がある場合には、チェックした後で状態が変化している可能性があるのです。図1が、この問題を示しています。

## 図1. 繰り返しと修正の同時発生が、予期せぬフェールを生ずる




この問題を解決するための方法の1つとして、繰り返しの期間中、Vectorに対するロックを保持する方法があります。それ以外の方法としては、イベントが起きる度に、Vectorのクローンを作るか、あるいはtoArray()メソッドを呼び、その内容を検索するのです。どちらの方法も、パフォーマンス上の問題があります。最初の方法では、繰り返しの期間中にリスナー・リストにアクセスしようとする他のスレッドを、締め出してしまう危険性があります。2番目の方法では、イベントが起こる度に一時オブジェクトを作り、リストをコピーしなければなりません。

リスナー・リストをトラバースするのにイテレーター（Iterator）を使ったとしても、ちょっと様子が変わるだけで、同じ問題が生じます。繰り返しが開始された後でコレクションが修正されていることを検出すると、iterator()実装は、NullPointerExceptionを投げる代わりにConcurrentModificationExceptionを投げるのです。これも、繰り返しの期間中、コレクションをロックしておくことによって防止することができます。

この問題を防ぐために、java.util.concurrentのCopyOnWriteArrayListクラスが役に立ちます。このクラスはListを実装しており、スレッド・セーフなのですが、そのイテレーターはトラバースの期間中にもConcurrentModificationExceptionを投げず、追加のロックも必要としません。この組み合わせ機能は、リストが修正される度に、リスト内容を内部的に再度割り当て、コピーすることによって実現されています。ですからリスト内容に対して繰り返しを行うスレッドは、変更を考慮する必要がありません。そうしたスレッドから見ると、リスト内容は繰り返しの期間中、一定のままに見えるのです。これは効率が悪いように思えるかもしれませんが、Observerを使う状況の大部分では、各コンポーネントの持つリスナーの数は少なく、挿入や除去よりも、トラバースの方が圧倒的に多いのです。そのため、変異（mutation）が遅くても、繰り返しが高速なことで埋め合わせることができ、リストに対して複数のスレッドが同時に繰り返しを行えるため、並行性は向上するのです。

## 初期化の安全性のリスク

リスナーを、そのコンストラクターから登録したい、という誘惑にかられますが、この誘惑に負けてはいけません。これは「失効リスナー（lapsed listener）・・・（この後すぐに触れます）」の問題を伴うだけではなく、スレッド・セーフに関する問題も幾つか生ずるのです。リスト2は、リスナーの構築と登録を同時に行おうとする、一見無害に見える試みを示しています。実はこれには問題があり、オブジェクトに対する「this」参照が、完全に構築される前に漏れてしまうのです。まさかコンストラクターが登録することはない、と一見無害に見えるのですが、見えるままを信じては危険です。

## リスト2. 「this」参照を漏らしてしまい問題を引き起こすイベント・リスナー

```
public class EventListener {

  public EventListener(EventSource eventSource) {
    // do our initialization
    ...

    // register ourselves with the event source
    eventSource.registerListener(this);
  }

  public onEvent(Event e) {
    // handle the event
  }
}
```

この手法の危険性が顕在化するのは、イベント・リスナーがサブクラス化された時です。EventListenerコンストラクターが実行すると、その後には、サブクラス・コンストラクターが行うことは何でも起きます。従って、EventListenerが公開されると、競合条件が起こります。リスト3で、リスト・フィールドが初期化される前に、ある運の悪いタイミングでonEventメソッドが呼ばれると、最終フィールドを参照解除する時に、非常にまぎらわしいNullPointerExceptionが起きます。

## リスト3. リスト2のEventListenerクラスをサブクラス化することによって起きる問題

```
public class RecordingEventListener extends EventListener {
  private final ArrayList<Event> list;

  public RecordingEventListener(EventSource eventSource) {
    super(eventSource);
    list = Collections.synchronizedList(new ArrayList<Event>());
  }

  public onEvent(Event e) {
    list.add(e);
    super.onEvent(e);
  }
}
```

たとえそのリスナー・クラスが最終であり、従ってサブクラス化できない場合であっても、やはりコンストラクターから「this」参照が漏れるのを許すべきでありません。許してしまうと、Javaのメモリー・モデルで提供されている安全性の保証が無効になってしまいます。また、プログラムの中に「this」という言葉が現れなくても、「this」参照が漏れてしまうことがあります。つまり、非静的な内部クラス・インスタンスを公開しても、同じことが起こるのです。内部クラスは、自分を取り囲むオブジェクトの「this」参照を保持しているからです。しかし、誤って「this」参照の漏洩を許してしまう原因として最も一般的なのは、リスナーを登録することなのです。これをリスト4に示します。**イベント・リスナーは、絶対にコンストラクターから登録してはなりません！**

### リスト4. 内部クラス・インスタンスを公開することによって、暗黙的に「this」参照の漏洩を許してしまう

```
public class EventListener2 {
  public EventListener2(EventSource eventSource) {

    eventSource.registerListener(
      new EventListener() {
        public void onEvent(Event e) {
          eventReceived(e);
        }
      });
  }

  public void eventReceived(Event e) {
  }
}
```

## リスナーのスレッド・セーフについて

リスナーを使うことによって起こるスレッド・セーフの問題の3番目は、リスナーがアプリケーション・データにアクセスしようとする場合がある、という事実から、そしてリスナーが呼ばれるのは通常、アプリケーションが直接制御していないスレッドである、という事実から生ずるものです。リスナーをJButtonなどのSwingコンポーネントに登録すると、そのリスナーはEDTから呼ばれます。リスナー・コードは、Swingコンポーネント上のメソッドを安全にEDTから呼ぶことができます。しかしリスナーからアプリケーション・オブジェクトにアクセスするとなると、こうしたオブジェクトが既にスレッド・セーフでない場合には、プログラムには新たなスレッド・セーフ要求が追加して課されることになります。

Swingコンポーネントは、ユーザーとの対話動作の結果としてイベントを生成しますが、Swingモデル・クラスは、fireXxxEvent()メソッドが呼ばれた時にイベントを生成します。そうするとfireXxxEvent()メソッドは、どのスレッドで呼ばれたとしても、リスナーを呼びます。Swingモデル・クラスはスレッド・セーフではなく、EDTの中に収まっているべきものなので、fireXxxEvent()へのコールも、すべてEDTから実行すべきなのです。ですから、別のスレッドからイベントをトリガーしたい場合には、SwingのinvokeLater()機構を使って、そのメソッドがEDTの中から呼ばれるようにする必要があります。一般的に言って、どのスレッドからイベント・リスナーが呼ばれるのかに注意する必要がある、ということです。また、そのイベント・リスナーがアクセスするオブジェクトが全てスレッド・セーフであるかどうか、そしてオブジェクトにアクセスする場所では、必ず適当な同期（Swingモデル・クラスの場合ではスレッド制限（thread-confinement））によって保護されていることを確認する必要があります。

## 失効リスナー（Lapsed listener）

Observerパターンを使う場合には、必ず2つの別々なコンポーネント、つまり観察側（observer）と観察対象側（observed）を結合していることになります。この2つは一般的に、明確に異なるライフサイクルを持っています。リスナーを登録することによる1つの結果として、観察対象オブジェクトからリスナーへの強い参照が作られます。これによって、リスナー（と、リスナーが参照する任意のオブジェクト）は、登録が解除されるまで、ガーベジ・コレクションを受けずに済むのです。多くの場合、リスナーのライフサイクルは、少なくとも観察対象コンポーネントと同じだけの長さを持つようになっています。実際、多くのリスナーは、アプリケーションが持続す

る限り存在を続けます。しかし場合によると、短命であるはずのリスナーが、永遠に存在してしまうことがあります。そのことが、アプリケーションのパフォーマンス低下や、必要以上のメモリー使用によって、ようやく分かるのです。

「失効リスナー」の問題は、設計レベルでの不注意、つまり関係するオブジェクトの存続時間を適切に考慮しないとか、あるいは単純に、コーディングが雑なために起こります。リスナーの登録と登録解除は、常に、一対として行う必要があります。しかし、例えそのように実行している場合であっても、実際に登録解除が適切なタイミングで行われるかどうかについても、確認の必要があるのです。リスト5は、失効リスナーの危険性をはらむコーディング・イディオムの例を示しています。このコードでは、コンポーネントにリスナーを登録し、あるアクションを行い、そしてリスナーを登録解除しています。

### リスト5. 失効リスナーの危険性をはらむコード

```
public void processFile(String filename) throws IOException {
    cancelButton.registerListener(this);
    // open file, read it, process it
    // might throw IOException
    cancelButton.unregisterListener(this);
  }
```

リスト5の問題は、もしファイル処理のコードがIOExceptionを投げると（現実に起こりうることです）、リスナーは登録解除されないという点です。つまり、このリスナーはガーベジ・コレクションを受けません。登録解除の操作は、processFile()メソッドからの全パスで登録解除が実行されるように、finallyブロックで行うべきなのです。

失効リスナーの問題に対応するために、弱い参照を使うように提案されることが時々あります。確かにこの方法は可能ですが、実装はかなり面倒です。これがうまく働くためには、そのリスナーのライフサイクルと全く同じライフサイクルを持つ別のオブジェクトを見つけ、そのオブジェクトにリスナーへの強い参照を保持させる必要があるのですが、これは必ずしも容易ではありません。

隠れた失効リスナーを見つけるために時々使われる別の方法として、与えられたリスナー・オブジェクトが、与えられたイベント・ソースに2度登録されないようにする方法があります。2重登録されているという状況は、一般的にバグの証です。つまり、リスナーが登録されていて登録解除されておらず、しかも再度登録されているのです。必ずしも問題を検出するのではなく、この影響を減らす方法として、リスナーの保存にListではなく、Setを使う方法があります。あるいは、リスナーを登録する前に、そのリスナーが登録されていないかをListでチェックし、登録されている場合には例外を投げ（あるいはエラー・ログをとり）、コーディング・エラーの証拠を集めて対策が打てるようにするのです。

## リスナーが引き起こす可能性のある、他の問題

リスナーを書く際には、リスナーを実行する環境を常に意識する必要があります。スレッド・セーフの問題に注意するだけではなく、リスナーが、他の問題でも呼び出し側を混乱させる可能性があることを頭に置く必要があります。例えば、（いかなる時間であっても）他を妨げてしまうことは、リスナーが『すべきではない』ことです。リスナーは、すぐに制御が戻ると想定している実行コンテキストから呼ばれている可能性が高いのです。もしリスナーが、時間を食う可能

性のある操作（巨大な文書の処理など）や、他を妨害する可能性のある操作（ソケットI/Oの実行など）を行う場合には、そうした操作を別スレッドで行うようにし、呼び出し側にすばやく戻るべきなのです。

不注意なイベント・ソースに対してリスナーが問題を引き起こす、もう1つの場合が、未チェック例外を投げる場合です。私たちは通常、意図的に未チェック例外を投げようとはしませんが、時には投げてしまう状況が発生します。もし皆さんがリスト1のイディオムを使ってリスナーを呼び出し、リスト中の2番目のリスナーが未チェック例外を投げると、その後のリスナーが呼ばれない（アプリケーションの一貫性が失われた状態になる可能性があります）だけではなく、自分が実行しているスレッドを壊してしまい、部分的なアプリケーション・フェールさえ引き起こしてしまう可能性があります。

未知のコード（リスナーが正にそうです）を呼び出す時には、たちの悪いリスナーが必要以上の損害を与えないように、そのコードをtry-catchブロックの中で実行した方が賢明です。あるいはもっと進めて、未チェック例外を投げるリスナーを自動的に登録解除しても良いかもしれません。未チェック例外を投げるということは結局、そのリスナーは壊れている、ということの証明なのです。（また、これのログを取るとか、あるいは、プログラムがなぜ停止したのかユーザーが判断できるように、ユーザーの注意を促したほうが良いかもしれません。）リスト6は、こうした手法の例として、try-catchブロックを繰り返しループ内部にネストしています。

### リスト6. 堅牢なリスナー呼び出し

```
List<Listener> list;
for (Iterator<Listener> i=list.iterator; i.hasNext(); ) {
    Listener l = i.next();
    try {
        l.eventHappened(event);
    }
    catch (RuntimeException e) {
        log("Unexpected exception in listener", e);
        i.remove();
    }
}
```

## まとめ

疎結合のコンポーネントを作る場合や、コンポーネントを再利用しようとする場合には、Observerパターンは非常に有用ですが、リスナーを書く人もコンポーネントを書く人も、幾つかの危険性に注意する必要があります。例えば、リスナーを登録する場合には、常にリスナーのライフサイクルに注意する必要があります。リスナーの存続時間をアプリケーションよりも短くしたい場合には、リスナーが必ず登録解除され、ガーベジ・コレクションを受けるようにする必要があります。また、リスナーやコンポーネントを書く場合には、スレッド・セーフの問題が関係していることを意識する必要があります。リスナーがアクセスするオブジェクトは、全てスレッド・セーフである必要があります。あるいはSwingモデルのように、スレッドに閉じ込められるオブジェクトの場合では、リスナーは確実に正しいスレッドで実行している必要があります。

# 著者について

Brian Goetz

Brian Goetz は18 年間以上に渡って、専門的ソフトウェア開発者として働いています。彼はカリフォルニア州ロスアルトスにあるソフトウェア開発コンサルティング会社、Quiotixの主席コンサルタントであり、またいくつかのJCP Expert Groupの一員でもあります。2005年の末にはAddison-Wesleyから、Brianによる著、Java Concurrency In Practiceが出版される予定です。Brian著による有力業界紙に掲載済みおよび掲載予定の記事のリストを参照してください。