コード品質を追求:ソフトウェア設計者にとってのコード品質

結合メトリックを使用してシステム・アーキテクチャーをサポートする

Andrew Glover
CTO
Stelligent Incorporated

2006年 4月 25日

よく設計されたソフトウェア・アーキテクチャーのほとんどは、システムの拡張性、保守性、および信頼性をサポートすることを目的としています。残念ながら、品質問題を軽視すれば、ソフトウェア設計者の努力は水の泡になります。今回の「コード品質を追求する」では、品質のエキスパートであるAndrew Gloverが、ソフトウェア・アーキテクチャーの長期的存続に影響するコードの品質面を継続的に監視し、訂正する方法を説明します。

先月は、コード・メトリックを使用してコードの品質を評価する方法を説明しました。先月の記事で紹介した循環的複雑度（cyclomatic complexity）メトリックは、メソッドの実行パスの数など、低レベルの詳細に注目するものでしたが、他の種類のメトリックは、より高いレベルのコードの詳細に注目します。今月は、さまざまな結合メトリックを使用してソフトウェア・アーキテクチャーを分析し、サポートする方法を説明します。

まず、興味深い2つの結合メトリック、すなわち、求心性結合と遠心性結合から始めましょう。これら整数ベースのメトリックは、関連オブジェクト（すなわち、互いに組み合わされて動作を作り出すオブジェクト）の数を表します。いずれかのメトリックの数値が高い場合、アーキテクチャーに保守問題があることを示します。高い求心性結合は、オブジェクトの責任が重過ぎることを示し、高い遠心性結合は、オブジェクトの独立性が十分でないことを示唆します。今月は、これらの問題と対応策について説明します。

求心性結合

責任が重過ぎることは、必ずしも悪いことではありません。例えば、コンポーネント（すなわちパッケージ）は、アーキテクチャー全体を通じて利用されることを目的としている場合が多いため、求心性結合の値が高くなります。コア・フレームワーク（Strutsなど）、ロギング・パッケージなどのユーティリティー（log4jなど）、また、例外階層も、通常は高い求心性結合を持ちます。

図1のパッケージcom.acme.ascp.exceptionの求心性結合は4です。これは当然です。その理由は、web、dao、util、およびfrmwrkパッケージすべてが、共通の例外フレームワークを利用する前提になっているからです。

図1. 求心性結合の記号

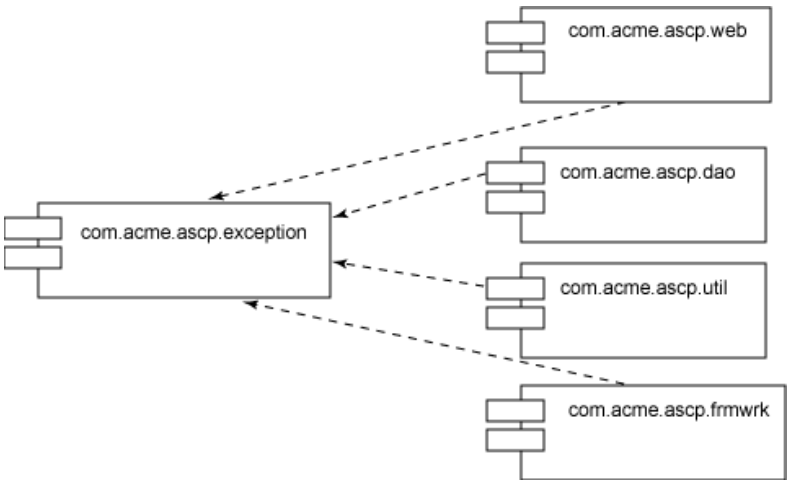


図1に示されているように、このexceptionパッケージの求心性結合（Ca）は4です。この場合、この数値はそれほど悪くはありません。例外階層が劇的に変化することはほとんどありません。ただし、exceptionパッケージの求心性結合を監視するのは、よい考えです。このパッケージの例外の動作またはコントラクトが劇的に変化した場合、4つの依存パッケージ全部に波及するからです。

抽象度の測定

exceptionパッケージをさらに調べ、抽象クラスと具象クラスの割合を示すことにより、もう1つのメトリック、すなわち抽象度を引き出すことができます。この場合、exceptionパッケージのクラスはすべてが具象なので、抽象度はゼロです。これは筆者の以前の観察結果と関連があります。すなわち、exceptionパッケージの具象度が高いということは、例外に変更を加えると、すべての関連パッケージ（com.acme.ascp.frmwrk、com.acme.ascp.util、com.acme.ascp.dao、およびcom.acme.ascp.web）に影響することを意味します。

求心性結合はコンポーネントの責任を示すことを理解し、このメトリックを長期的に監視することによって、どんなによく設計されたシステムでも発生すると言われるエントロピーからソフトウェア・アーキテクチャーを保護できます。

設計の柔軟性をサポートする

多くのアーキテクチャーは、サード・パーティ製のパッケージを利用するときの柔軟性を念頭において設計されます。柔軟性は、理想的には、サード・パーティ製パッケージ内の変化からアーキテクチャーを保護するインターフェースを使用することによって得られます。例えば、システム設計者は、サード・パーティの請求コードを利用する内部インターフェース・パッケージを作成して、請求コードを使用するパッケージへのインターフェースだけを公開することができます。ところで、これはJDBCの機能と同様です。

図2. 設計による柔軟性

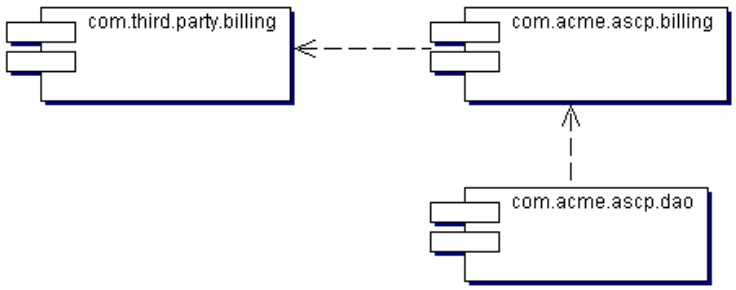


図2に示されているように、acme.ascpアプリケーションはcom.acme.ascp.billingパッケージを通じてサード・パーティの請求パッケージに結合されます。これにより、あるレベルの柔軟性が生まれます。サード・パーティの別の請求パッケージを利用した方がよいということになった場合、その変更によって影響を受けるのは1つのパッケージだけです。その上、com.acme.ascp.billingの抽象度は0.8であり、これはインターフェースと抽象クラスの変更から保護されることを意味します。

サード・パーティ実装を切り替える場合、リファクタリングが必要になるのは com.acme.ascp.billingパッケージだけです。さらによいことに、この柔軟性を設計に組み込み、変更の影響を理解することによって、開発者テスト中の変更による損害からあなた自身を守ることができます。

内部請求パッケージに変更を加える前に、コード・カバレッジ・レポートを分析して、何らかのテストによってパッケージが実際にテストされたかどうかを調べることができます。ある程度のカバレッジ（テスト対象範囲）が見つかった場合は、そのテスト・ケースを詳しく調べて、その妥当性を確認することができます。カバレッジが見つからなかった場合は、新しいライブラリーへの交換と挿入には、ある程度のリスクが伴い、時間がかかることがわかります。

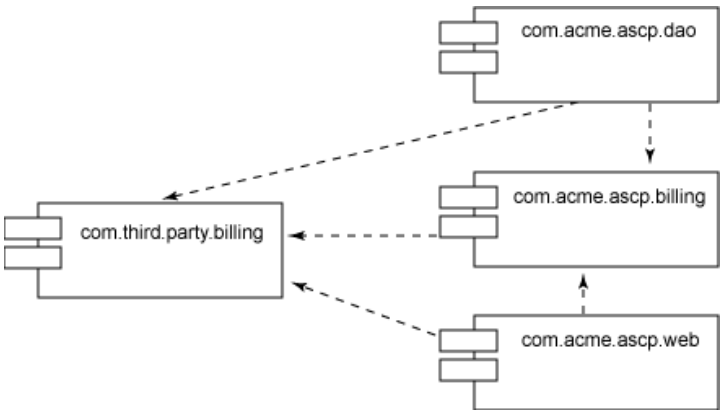
こうした瑣末な情報の収集は、コード・メトリックを使用すれば非常に簡単です。一方、テスト・カバレッジに関するパッケージの結合について何も知らなければ、サード・パーティ製ライブラリーの交換に要する時間は、せいぜい推測するしかありません。

エントロピーを監視する

すでに説明したように、エントロピーは、どれほど入念にプランニングされたアーキテクチャーにも忍び込んできます。チームの縮小や、目的が明確に文書化されていないことなどによって、新人開発者が便利そうなパッケージをうっかりインポートしてしまうと、やがて、システムの求心性結合値が大きくなり始めます。

例えば、図2と図3を比べてください。アーキテクチャーの脆弱性が増したことがわかりますか。daoパッケージが、サード・パーティ製請求パッケージを直接利用するだけでなく、請求コードを使用するつもりがなかった別のパッケージも両方の請求パッケージを直接参照しています。

図3. コードに忍び寄るエントロピー



com.third.party.billingパッケージを別のパッケージに切り替えようと思うと、実に大変なことになります。テスト現場で、欠陥につながるリスクや、システムのさまざまな局面での動作が損なわれるリスクを緩和する必要がある場合を想像してみてください。実際、このようなアーキテクチャーは修正をサポートできないため、変更されることはまれです。悪いことに、既存のコンポーネントのアップグレードなど、重要な修正でも、コード・ベース全体を損なう原因となることがあります。

遠心性結合

求心性結合が特定のコンポーネントに依存するコンポーネントの数だとすると、遠心性結合は、特定のコンポーネントが依存するコンポーネントの数です。遠心性結合は、求心性結合の逆と考えてください。

遠心性結合の意味合いは、変更がコードにどのように影響するかという点で、求心性結合と同様です。例えば、図4に示されているcom.acme.ascp.daoパッケージの遠心性結合（Ce）は3です。

図4. daoパッケージの遠心性結合

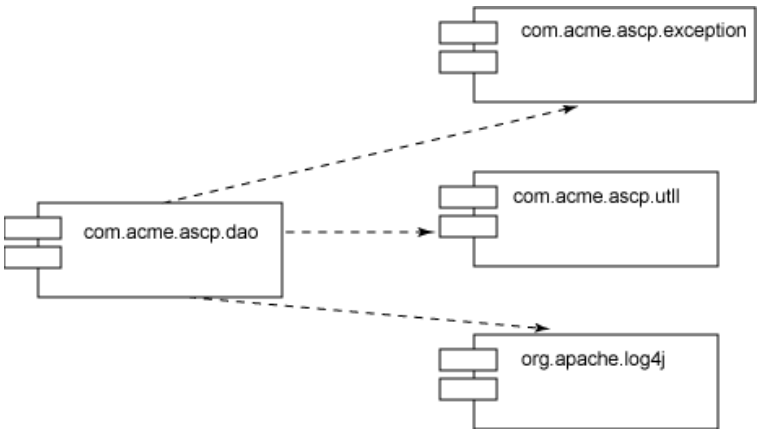


図4に示されているように、com.acme.ascp.daoパッケージはorg.apache.log4j、com.acme.ascp.util、およびcom.acme.ascp.exceptionコンポーネントに依存して、動作のコントラクトを果たします。求心性結合でもそうであったように、依存性のレベルは、それ自体では悪いことではありません。それは結合に関する知識であり、問題となっている関連コンポーネントへの変更にどのように影響するかという知識です。

求心性結合と同様、遠心性結合でも抽象度メトリックが作用します。図4では、com.acme.ascp.daoパッケージは完全に具象です。したがって、抽象度は0です。これは、遠心性結合にcom.acme.ascp.daoを含むコンポーネントは、それら自体が脆弱になる恐れがあることを意味します。com.acme.ascp.daoは、他の3つのパッケージに対して遠心性結合を持つからです。そのうちの1つ（例えば、com.acme.ascp.util）に変更が加えられた場合、com.acme.ascp.dao内で波及効果が発生します。daoはインターフェースまたは抽象クラスを通じて移植の詳細を隠すことができないため、変更が加えられると依存コンポーネントに影響します。

結合 + カバレッジ =

遠心性結合の関係データを調べ、コード・カバレッジに関連付けることで、より賢明な意思決定が可能になります。例えば、新しい要件が開発チームに手渡されたと想像してください。この要件に関連する変更は図4に示されているcom.acme.ascp.utilパッケージに加えればよいことがわかりました。また、過去のいくつかのリリースで、daoパッケージ（utilに依存し、抽象度はゼロ）には、優先度の高い多数の欠陥が見つかっています（おそらく、このパッケージに対する開発者テストが限られていたためであり、コードの複雑度が高いため）。

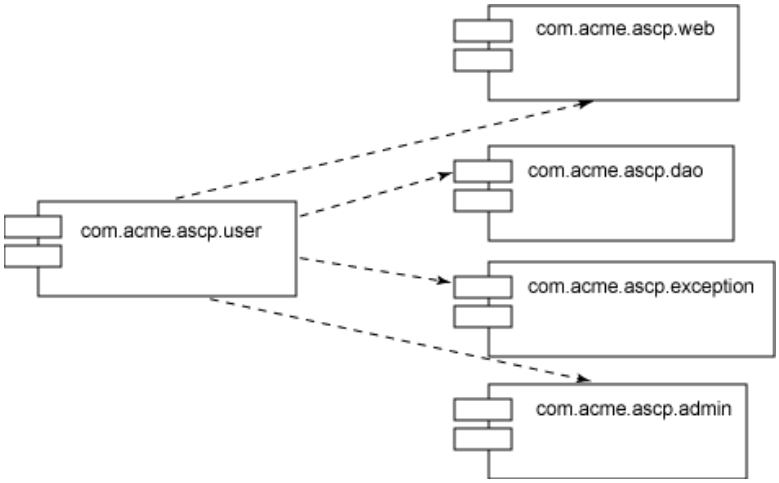
この状況には、プラスの面もあります。com.acme.ascp.utilとcom.acme.ascp.daoの関係を理解しているからです。daoパッケージはutilに依存していることがわかっているので、新しい要件をサポートするための修正をutilに加えると、厄介なdaoパッケージに悪影響を及ぼす恐れがあります。

この関連を見ることは、リスク評価だけでなく、労力分析にもある程度役立ちます。この関連に注目しなかった場合、新しい要件をサポートするには短時間のコーディング作業しか必要ではないと推測したかもしれません。関連を把握していれば、daoパッケージに発生する付帯的損害を緩和するために、適切な時間またはリソースを割り当てることができます。

依存性を監視する

求心性結合を継続的に監視することによって、アーキテクチャー設計のエントロピーを発見できるのと同様に、遠心性結合の監視は望ましくない依存性の特定に役立ちます。例えば、図5では、ある時点で誰かがcom.acme.ascp.webパッケージにはcom.acme.ascp.userに提供すべきものがあると判断したようです。userパッケージのどこかで、1つ以上のオブジェクトが実際にwebパッケージからオブジェクトをインポートしています。

図5. userパッケージの遠心性結合



明らかに、これはアーキテクチャー設計の当初の意図ではありませんでした。しかし、システムの遠心性結合を定期的に監視しているので、この不協和音を容易にリファクタリングして是正することができます。おそらく、webパッケージの便利なユーティリティー・オブジェクトをユーティリティー・パッケージに移動して、望ましくない依存性を招くことなく、他のパッケージも利用できるようにすべきでしょう。

不安定性の計測

システムの遠心性結合と求心性結合の数値を結び付けて、もう1つのメトリックを作ることができます。すなわち、不安定性です。遠心性結合を、遠心性結合と求心性結合の和で割ると（ Ce / (Ca + Ce)）、安定したパッケージ（ 0に近い値 ）か、不安定なパッケージ（ 1に近い値 ）かを測る比になります。この式が示すように、遠心性結合はパッケージの安定性にとってマイナスに作用します。パッケージが他のパッケージに依存する度合いが大きいほど、変更に際して波及効果を受けやすくなります。逆に、依存される度合いが大きいパッケージほど、変更の可能性は少なくなります。

例えば、図5では、userパッケージの不安定性の値は1であり、遠心性結合が4、求心性結合が0であることを意味します。com.acme.ascp.daoのようなパッケージ内の変更は、userパッケージに影響を与えます。

アーキテクチャーを設計・実装するときには、安定したパッケージに依存する方が有利です。安定したパッケージは変更の可能性が少ないからです。同様に、不安定なパッケージへの依存は、変更時にアーキテクチャーに被害が及ぶリスクを高めます。

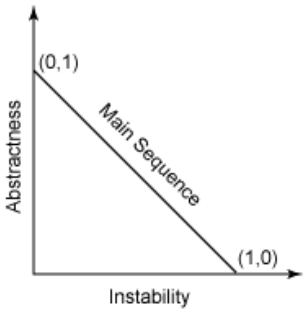
メインからの距離

ここまで、パッケージに変更を加えた場合の影響の推測に役立つ求心性結合と、外部の変更がパッケージに及ぼす影響を評価するのに役立つ遠心性結合を説明しました。また、パッケージの変更が容易かどうかを理解する上で役立つ抽象度メトリックと、パッケージの依存性が特定のパッケージに及ぼす影響を理解する上で役立つ不安定性メトリックについても説明しました。

さらにもう1つのメトリックを使用して、ソフトウェア・アーキテクチャーに影響するファクターについて知ることができます。このメトリックは、X軸とY軸を結ぶ直線によって、抽象度と不安

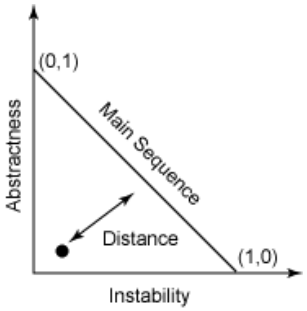
定性のバランスをとります。メイン・シーケンスは、図6に示されているように、デカルト座標X=0、Y=1とX=1、Y=0を結ぶ直線です。

図6. メイン・シーケンス



この直線に沿ってパッケージを点で表し、メイン・シーケンスからの距離を測定することによって、パッケージのバランスを推測できます。パッケージは、図7に示されているように、抽象度と不安定性のバランスが取れている（距離が0に近い）か、バランスが取れていずメイン・シーケンスからの距離が1に近いかのいずれかです。

図7. メイン・シーケンスからの距離



メイン・シーケンス・メトリックからの距離を調べると、興味深い結果が得られます。例えば、上記のuserパッケージの値は0になります。このパッケージは、実装パッケージであるという意味でバランスが取れていますが、非常に不安定です。

一般に、メイン・シーケンス・メトリックからの距離は、実世界での実装を補償しようとします。どんなコード・ベースも、含んでいるすべてのパッケージの抽象度と不安定性の値が1または0ということはありません。ほとんどのパッケージは、両者の中間です。メイン・シーケンス・メトリックからの距離を監視することによって、パッケージがアンバランスになりつつあるかどうかを推測することができます。値が1に最も近いパッケージなど（可能な限りメイン・シーケンスから遠いことを意味します）、中心から離れた値を探すと、特定のアンバランスがアーキテクチャーの保守性にどのように影響するか（例えば、脆弱性）を理解する助けになります。

まとめ

今月は、長期的に監視できるアーキテクチャー・メトリックをいくつか学びました。求心性結合と遠心性結合、不安定性、抽象度、およびメイン・シーケンスからの距離はすべて、JDepend、JarAnalyzer、およびMetrics plug-in for Eclipseなどのコード分析ツールによって報告されます（「参考文献」を参照）。システムのコード結合メトリックを監視すると、アーキテ

クチャーを徐々に弱体化させる一般的な傾向、すなわち、設計の硬直性、パッケージのエントロピー、および望ましくない依存性を掌握しやすくなります。また、システムの抽象度と不安定性のバランスを測定することで、長期的な保守性の概要を知ることができます。

著者について

Andrew Glover

Andrew Gloverは合衆国ワシントン特別区にある、Vanward TechnologiesのCTO（最高技術責任者）です。Vanward Technologiesは自動化テスト・フレームワークの構築を専門としており、ソフトウェアのバグ発生数や統合時間やテスト時間の減少、また全体的なコード安定性改善に貢献しています。