ユースケースに基づく作業量の見積もり

John Smith

Rational Software ホワイト・ペーパー

TP 171, 10/99

Rational®
the software development company

目次

問題

直感的には、ユースケース・モデルの特徴に基づき、開発で必要になるサイズと作業量を見積もることが可能なように思えます。最終的にユースケース・モデルは機能要件を取り込むため、ユースケース・ベースでファンクション・ポイントと同等のものはないはずです。ここにはいくつかの問題点があります。

- ユースケース仕様のスタイルと正式さはさまざまであり、メトリックスを定義するのが困難です。例えば、ユースケースの長さを測定したい場合などです。
- ユースケースは外部アクターから見たシステムのビューを表現しなければならないので、SLOC (ソフトウェア・コードの行数) が 500,000 のシステムと 5,000 のサブシステムでは***レベル*** がかなり異なります (Cockburn97 はレベルと目標の概念を説明します)。
- ユースケースは、作成時に明示的に、要求された実装中に暗黙的に、さまざまな複雑さを持つ可能性があります。
- ユースケースはアクターの観点で動作を記述しなければなりませんが、これは特にシステムが状態を持つ場合 (ほとんどの場合があてはまります)、かなり複雑になることがあります。したがって、この動作を記述することにより、(どの実現も完了する前に) システムのモデルを要求する場合があります。動作の本質を把握しようとすると、非常に多くのレベルの機能分割と詳細が生じる場合があります。

では、見積もりを可能にするために何らかのユースケースの実現が必要なのでしょうか。おそらくユースケースから直接見積もりを出すことへの期待があまりにも高く、ファンクション・ポイントとユースケース・ポイントの概念の間の例が誤っています。ファンクション・ポイント・カウントの計算にはいずれにしてもシステムのモデルが必要です。ユースケースの記述からファンクション・ポイントを導出するには、ユースケースの表現レベルが統一されている必要があります。その実現が明らかになり始めるときに、ファンクション・ポイント・カウントを信頼できるようになるでしょう。Fetcke97 はユースケースからファンクション・ポイントまでのマッピングを説明しますが、マッピングを有効にするには、ユースケースのレベルが適切でなければなりません。他の方法では、例えば、PRICE オブジェクト・ポイントなどのクラス/オブジェクト・ベースのメトリックスをソースとして使用します (Minkiewicz96)。

その他の研究

ユースケースを記述し、形式化するためにはかなりの量の作業があります。 これについて、Hurlbut97 は優れた調査を行っています。ユースケースから見積もりメトリックスを引き出す際はそれほど作業が発生しません。Graham95 および Graham98 には、ユースケースに関するかなり厳しい批判が含まれており (ただし、私はなぜ彼がそのように考えたのか、なぜユースケースがかけ離れたものであるかを完全に理解していません)、長さや複雑さがまちまちになるというユースケースの問題を克服する方法として、「タスク・スクリプト」という考えを提案しています。Graham 氏の「アトミック・タスク・スクリプト」は、「タスク・ポイント」目トリックの集合の基礎です。アトミック・タスク・スクリプトの問題は非常に低いレベルであることです。そして、アトミック・タスク・スクリプトは理想的には単一の文であるべきで、ただ 1 つのドメイン用語でそれ以上分割不可能であるということです。「ルート・タスク」は 1 つ以上のアトミック・タスク・スクリプトを含み、各ルート・タスクは「計画を開始するクラスの中で、ただ 1 つのシステム操作に」対応します (Graham98)。ルート・タスクは低レベルのユースケースと非常によく似ているように思えますし、アトミック・タスク・スクリプトはそうしたユースケース内のステップに似ています。しかし、レベルの問題は残ります。

その他の研究が Karner 氏 (Karner93)、Major 氏 (Major98)、Armour 氏 と Catherwood 氏 (Armour96)、Thomson 氏 (Thomson94) によって行われました。Karner 氏の論文はユースケース・ポイントの計算方法を指摘しますが、やはり、ユースケースがクラスにより実現可能な方法で表現されること (つまり、サブシステムよりも詳細なレベル) を仮定しています。

では、見積もりのためにユースケースを使用することは避け、代わりに発生する分析と設計の実現に頼るべきなのでしょうか。ここで問題になるのは、見積もり作成が遅れ、この技術を選択したプロジェクト管理者を満足さ

せないということです。 初期の見積もりが要求されれば、他の方法を使用せざるを得なくなります。プロジェクト 管理者にしてみれば、見積もりが遅れ、未計画のままプロジェクトを進行するよりも、計画目的のための初期見積もりを取得し、反復ごとに洗練できれば、それに越したことはありません。

このホワイト・ペーパーでは、任意レベルのユースケースを使用して作業量見積もりを作成するフレームワークを説明します。この考えを提示するために、経験に基づいて、関連のある規模を持つ、単純な標準的な構造を説明します。このホワイト・ペーパーの中には、対象領域における研究事例とデータの不足から、止むを得ず大胆かつ率直に推測している部分が多くあります。定式化においては「相互接続されたシステムのシステム」の考え方を採用しました。

次に、少し話が逸れますが、こうした考えに至った背景について紹介します。

機能分割を回避するべきか

機能分割という考えはソフトウェア開発に携わる人の多くが忌み嫌うもののようです。また、私個人が、徹底的な機能分割を経験しているため (非常に大規模なデータ・フロー図で 3,000 の基本変換を行い、レベルは 5 または 6 個あり、基盤レベルを除きアーキテクチャーはまったく考慮されていませんでした)、あまり楽観的には考えられませんでした。しかし、この事例の場合、機能分割だけではなく、仕様の長さが 1 頁未満でなければならない機能上の基本レベルに達するまで、プロセスを記述しないと言う考えも問題でした。

結果は非常に理解しにくいものです。 これらの基本変換から高レベルで求められる適切な動作がどのように現れるかを認識することは困難です。また、パフォーマンスやその他の品質要求を満たす機能構造が物理構造とどのように対応するのかも明らかではありません。したがって、問題を解決できるようになる (基本レベル) まで分割を続けるのですが、互いに連携する基本要素が高レベルの目標を実際に満たすことは不明か、または証明できないという矛盾した状況となりました。この方法の中で機能外要求を考慮する手段はありません。基盤 (通信、オペレーティング・システム、など) だけでなく、アーキテクチャー全体が分割と共に進化し、それぞれが互いに影響を与えなければならなかったのです。

Bauhaus の「形は機能に従う」という考えはどうでしょうか。彼らの設計に対する機能主義的な手法には多くのメリットがありましたが、どこでも平らな屋根を使用してしまうなどのデメリットもありました。屋根の機能だけを考え、設計を軽視して居住者の保護の面を優先させると、少なくとも一定の領域で不満足な結果となります。そうした屋根に防水処理を施すのは困難です。また、雪も多量に積もるでしょう。

これらの問題は解決することができますが、**別の設計を選んでいたらもっと出費は抑えられたでしょう。**ありふれた言い方かもしれませんが、形は機能上、機能外を問わず、すべての要求に従わなければなりません。また、機能外要求は審美眼を含む場合があります。よくあるアーキテクトの問題は、機能外要求が十分に記述されておらず、「こうあるべきだ」というアーキテクトの経験に多く依存することです。したがって、機能分割が単にアーキテクチャーを動かし (分割がいくつかのレベルまで進み、機能の基本要素が「モジュール」と 1 対 1 で対応する)、インターフェースを定義するのは不適切です。

こうした考察の結果、**アーキテクチャーの研究を行う前に**、ユースケースを一定の標準化されたレベル (クラスのコラボレーションにより実現可能) まで分割するのは意味がないことであると確信しました。一定の規模のシステムでは確かに分割は発生します (「参考資料」Jacobson97 を参照) が、分割の基準とエンジニアリング・プロセスは重要です。 場当たり的な機能分割では不十分です。

システムの考慮事項

システム・エンジニアは機能分析、分割、割り当て (設計の統合時) を行いますが、アーキテクチャーの原動力は機能だけではありません。 専門エンジニアのチームは代替設計の評価に貢献します。「IEEE Std 1220, the Standard for Application and Management of the Systems Engineering Process」は、セクション「6.3, Functional Analysis」のサブセクション「6.3.1 Functional Decomposition」で機能分割の使用を、セクション「「6.5 Synthesis」でシステム製品ソリューションをそれぞれ説明しています。特に興味深いのはサブセクション「6.5.1 Group and Allocate Functions」と「6.5.2 Physical Solution Alternatives」です。セクション 6.3.1 で、

分割はシステムが達成しなければならないものを明確に**理解する**ために実行され、**一般に 1 つのレベルの分割で十分である**と述べられています。

機能分割の目的はシステムを形成することではなく (これは統合が行います)、システムが行うことを理解し、伝達することです。 機能モデルはこれを行う有効な方法です。統合において、ソリューション構造にサブファンクションが割り当てられた後、他のすべての要求を考慮してソリューションが評価されます。この方法と複数レベルの機能分割との違いは、各レベルで要求される動作を記述し、実装するソリューションを見つけようとすることです。これらは、次のレベルの動作がさらなる洗練を要するかどうか、より低いレベルのコンポーネントに割り当てる必要があるかどうかを判断する前に行います。

以上の結論として、任意のレベルの動作を記述するために何百ものユースケースは必要ないといえます。システム、サブシステム、クラスなど、記述されたものの動作を十分にカバーする外部ユースケース (や関連シナリオ) の数はかなり少なくすることができます。ここで、外部ユースケースの意味を説明しなければなりません。例として、複数のサブシステムから構成されるシステムを考えます。各サブシステムは複数のクラスから構成されます。システムとそのアクターの動作を記述するユースケースを外部ユースケースといいます。サブシステムも独自のユースケースを持つ場合があります。これらのユースケースはシステムに対しては内部ですが、サブシステムに対しては外部です。非常に大きな (コードが 100 万行以上になる場合など) システムを構築するために最終的に使用されるユースケースは**外部と内部を合わせて**数百になることが考えられます。 なぜなら、そうしたサイズのシステムはシステムの中のシステムとして、または少なくともサブシステムのシステムとして構築されるからです。

構造とサイズに関する前提

ユースケースの数

Rational® では、一般にユースケースの数は小さくなければならない (10 ～ 50 など) と教えており、ユースケースの数が多い (100 を超える場合など) と、機能分割に陥って、ユースケースがアクターに価値を提供できなくなると述べてきました。それにもかかわらず、実際のプロジェクトでは多数のユースケースが存在することがあり、すべてが「悪い」というわけではありません。そうしたユースケースはさまざまなレベルをカバーします。例えば、Rational 内の電子メールで、著者は Ericsson の例を引用します。

> Ericsson は、新世代の電話交換機の大部分をモデリング中で、600 人年 (ピーク時の開発者は 300 ～ 400 人)、200 個のユースケースを見積もりました **(複数レベルのユースケースの使用については、「Systems of Interconnected Systems」を参照)** ―このカッコの部分は著者のコメントです。

600 人以上のシステム (どれだけの規模でしょうか。C++ コードで 1,500,000 行以上でしょうか) の場合、ユースケース分析はサブシステムの 1 つ上のレベルで止まったと思います (つまり、サブシステムを 7000 ～ 10000 コード行で定義する場合)、さもなければ、カウントはもっと高くなったかもしれません。

したがって、小数の外部ユースケースが適切であるという点には引き続き注意するつもりです。私が提案した構造と規模に一致させる場合、**10 個の外部ユースケース**とそれぞれに **30 個の関連シナリオ**[1]が動作を記述するために適切です[2]。実際の例でユースケースの数が 10 を超え、それらがそのレベルで真に外部である場合、記述されるシステムは対応する標準形よりも大規模です。これらの数の適切である根拠については後で述べます。

---

[1] UML1.3 で、シナリオは次のように説明されています。「**シナリオ**: 振る舞いを説明するアクションの特定の手順です。シナリオはユースケース・インスタンスの対話または実行を例示するために使用できます。ここでは、ユースケース・インスタンスの実行の例示という 2 番目の意味で使用されています。

[2] この数 (シナリオ内) はユースケースの複雑さを反映するためのものであることに注意してください。この数はすべてのユースケースについて、開発者が 30 個のシナリオを作成しなければならないことを提案しているわけではなく、ユースケースを経由するパスがより多数あるとしても、30 個のシナリオがユースケースの興味深い動作のほとんどを把握することを示します。

構造上の階層

提案される構造上の階層は以下のとおりです。

4 — システムのシステム

3 — システム

2 — サブシステム・グループ

1 — サブシステム

0 — クラス

クラスとサブシステムは UML で定義されます。より大きな集約は UML におけるサブシステム (サブシステムを含む) です。説明を容易にするためにそれらに別々の名前を付けました。集約サブシステム・グループは、2167 または 498 (サブシステムを CSC に、クラスを CSU にします) といった軍事標準の用語を知っている人にとっては CSCI に似たサイズです。思い出してみると、2167 日間にわたり、どの Ada 構造がどのレベルにマップされるべきか議論した末、騒ぎが収まると Ada パッケージはたいてい CSU にマップされていました。システムはこの階層に厳格に従わなければならないと提案しているわけではありません。レベル間の混在もあるでしょう。しかし、この階層により、ユースケース単位の作業におけるサイズの効果を説明することが可能になります。

各レベルのユースケースが存在します (おそらく個々のクラスには存在しません) が、詳細ではなく、そのレベルの各コンポーネント (つまり、サブシステム、サブシステム・グループ、など) に関するユースケースです[3]。私は、各レベルの各コンポーネントについて、10 個のユースケースがあるべきだと主張しています。ユースケースの記述が平均 10 頁とすると、仕様書は 100 頁になります (機能外要求について同程度かもっと少ない分量の書類が追加されます)。これは Stevens98 で好まれている数であり、Royce98 で提案されている数に近いものです。しかし、なぜ 10 個のユースケースなのでしょうか。この結論に至るまでにボトムアップ式に考察しました。 基本は、サブシステムあたりのクラス数、クラス・サイズ、操作サイズ、などの合理的な規模です。参照用に収集したこれらのデータを他の前提と共に次の表に示します。

操作サイズ	70 sloc
クラスあたりの操作数	12
サブシステムあたりのクラス数	8
サブシステム・グループあたりのサブシステム数	8
システムあたりのサブシステム・グループ数	8
システムのシステムあたりのシステム数	8
外部ユースケース数 (システム、サブシステム、などの単位)	10

[3] レビュー担当者の中には 4 つのレベルのユースケースの可能性について警告を発する人もいますが、それは一般に大規模になるシステムの中のシステムについてのみである点に注意してください。そうした場合、4 つのレベルでユースケースがあっても驚きません。特に、1 次請負業者 (システムのシステムについて)、2 次請負業者 (システムについて)、3 次請負業者 (サブシステムについて) が作業する場合などです。

ユースケースあたりのシナリオ数	30
ユースケース記述あたりの頁数[4]	10

経験的なデータはそれほど多くありません。テキスト全体で少しずつ紹介します。Lorentz94 と Henderson-Sellers96 はいくつかのデータを含み、私はオーストラリアでのプロジェクトからデータを得ています。ほとんどは 宇宙航空分野です。いずれの場合も、この段階では、フレームワークを適切な場所に位置付けることが重要です。

階層内のコンポーネントのサイズ

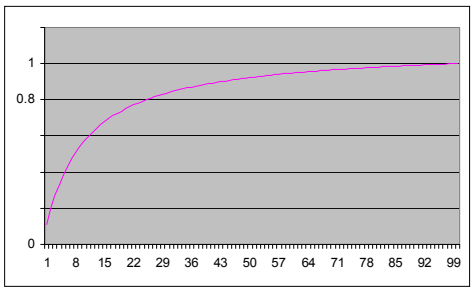
ここで、測定を嫌う人がいると知りながら、コード行を使用したことを告げなければなりません。これらは C++ (または同等レベルの言語) のコード行なので、ファンクション・ポイントを見つけるのは十分容易です。

コンテナー内のクラスの数と表現可能な動作の詳細さには何らかの関係があるでしょう。私はサブシステムあたり 8 個のクラス[5]、サブシステム・グループあたり 8 個のサブシステム、システムあたり 8 個のサブシステム・グループなどとしました。なぜ 8 個なのでしょうか。

- 7 ± 2 の範囲内。
- クラスあたり 850 sloc の C++ (12 個の操作があり、それぞれ 70 sloc) で、サブシステムのサイズは 7000 sloc 以下になります。小規模チーム (3 ～ 7 人程度のスタッフ) が 4 ～ 9 カ月で納品可能な機能とコードの量です。これは 300,000 ～ 1,000,000 sloc (RUP99) のシステムの反復期間と調和をとる必要があります。[6]

では、8 個のクラスの動作を表現 (外部的に) するユースケースはいくつで、サブシステムに密接に関連するものはどれでしょうか。詳細を定めるのは単にユースケースの数だけではなく、各ユースケースのシナリオの数でもあります。ユースケース当たりのシナリオの拡張に関するガイドラインは現在あまり存在しません。Grady Booch 氏は Booch98 の中でこう述べています。「ユースケースからシナリオへの拡張要因が存在する。やや複雑なシステムには、その動作を把握する数十個のユースケースが存在する場合があり、各ユースケースは数十個のシナリオを持つ場合がある」。また、Bruce Powel Douglass 氏は Douglass99 の中で、「ユースケースを精巧にするには多数のシナリオが必要である。通常、10 個あたりから数十個になる」と述べています。私はユースケースあたり 30 個のシナリオを選択しました。これは「数十個」の低い方ですが、Rechtin 氏は (Rechtin91 の中で)、エンジニアは 5 ～ 10 個の相互作用変数 (議論の目的のために 1 つのコラボレーション中 5 ～ 10 個のクラスと解釈します) と 10 ～ 50 個の対話 (シナリオとして解釈します) を処理できます。このように解釈すると、複数のユースケースはこの変数空間の複数のインスタンスです。

したがって、それぞれが 30 個のシナリオを持つ 10 個のユースケースは、合計で 300 個のシナリオを持ち (後に最大 300 個のテスト・ケースになります)、8 個のクラスの興味深い動作をカバーするのに十分です。これが合理的な数であることを示す根拠がほかにあるでしょうか。Pareto 氏の 80-20 ルールが適用される場合、クラスの 20% が機能の 80% を提供し、同様に、機能の 80% は各クラス内の操作の 20% により提供されます。控えめに考えて、機能の 75% を達成するためにクラスの 20% が必要であるとし、この点を通る Pareto 分布を作成します (図 1)。



[4] このホワイト・ペーパーの後の方で、シス

[5] 私はこの種のカウントが分析の典型であ があり、クラスの数は 3 倍以上増えますが、それに応じて操作サイズやクラス・サイズ

[6] より小規模なシステム (短い反復期間) (ます。また、各反復について部分的な納品を計画することは常に可能です。たた を納品しなければならないことがあります。

図 1:Pareto の分布

動作全体の 80% をカバーし、Pareto ルールがクラス、操作、シナリオの数に適用される場合、93% ($0.93^3 = 0.8$) の動作を対象にする必要があります。それぞれ 50% が必要です。つまり、4 個のクラスと 5 個の操作 (= (12 - 2 コンストラクター/デストラクター) /2)。それぞれが 5 個の操作を持つ 4 個のクラスの実行パターンを表現するために作成された複数ノードのツリーを巡回する回数は何千にもなる可能性があります。私は階層を仮定し、各ノードから最大 3 つのリンクを作成しました。私は階層を仮定し、各ノードから最大 3 つのリンクを作成しました。最上位に 10 個の操作 (インターフェース操作) があり、3 つのレベルのツリーを形成します。これにより約 1000 個のパスまたはシナリオができます。したがって、500 個のシナリオで 93% がカバーされるでしょう。300 個のシナリオならば、(同じ前提で) およそ 73% をカバーするでしょう。冗長な動作の仕様を除去するため、ツリーの刈り込み方法を検討すれば、選択するアルゴリズムによってはもっと少ない数でも十分かもしれません。

これに関するもう 1 つの方法は、7000 sloc の C++ コードについて、いくつのテスト・ケース (シナリオから導出) を期待するかを問うことです。これらのテストは単体テストのレベルを超えるもので、Jones91 と Boeing 777 プロジェクト (Pehrson96) から、この数が少なくとも実践を表すという点で安全だという証拠があります。これらのソースから 250 から 280[7] がほぼ適切だと考えられます。完全に異なるレベルで、Canadian Automated Air Traffic System (CAATS) プロジェクトは 200 個のシステム・テスト (プライベート通信) を使用します。

ユースケースのサイズ

ユースケースはどれくらいの「大きさ」を持つべきでしょうか。期待した動作を実現するのに十分な詳細さを持っていなければなりません。 これは内部と外部の複雑さに依存し、システムのタイプに関連します。ここで、システムの内部アクションについてどれくらい記述すべきかという問題に行き当たります。外部的な動作の記述からシステムを構築することは、明らかに出力が入力に関連することを要求します。例えば、動作が履歴の影響を受けやすく、複雑な場合、システムとそのアクションの内部的な概念モデルなしで動作を記述するのは非常に困難です。ただし、必ずしもシステムの内部的な作成方法を記述するわけではないことに注意してください。機能外要求を満たす任意の設計やモデルの動作に一致するものであればよいのです。

UML1.3 で提供される定義は、「**ユースケース [クラス]:** システムのアクターと対話して、システム (またはその他のエンティティー) が実行可能な一連のアクション (バリアントを含みます) の仕様」です。複雑な動作の場合、この定義を合理的に活用して内部アクションを含めることができます。ただし、実現まで延期される場合は除きます。また、この処理はエンド・ユーザーとは離れたステップです。ビジネス・ルールもユースケースに取り込み、アクターの動作を制限しなければなりません。例えば、ATM システムにおいて、残高にかかわらず、1 回のトランザクションで引き出すのは 500 ドル以内というルールが銀行にある場合があります。

この種の解釈を考慮すると、イベント記述のユースケース・フローは 2 ～ 20[8] ページになる可能性があります。単純な動作を持つアルゴリズム的に単純なシステムは明らかに長々とした記述が必要ありません。おそらく単純なビジネス・システムは 2 ～ 10 ページで特徴付けることが可能であり、平均は 5 ページ程度、ビジネスや科学分野のより複雑なシステムでは 6 ～ 15 ページで平均が 9 ページ、さらに複雑なコマンドや制御システムの場合は 8 ～ 20 ページで平均が 12 ページといえるでしょう (これらの比率は、同サイズのシステムのシステム・タイプに対する作業量の非線形関係を反映します)。ただし、根拠となるデータは持ち合わせていません。より表現力のある記述形式、状態マシン、またはアクティビティー図などを使用すればスペースが少なくて済むでしょう。テキストを重視する傾向がまだあるので他の形態はとりあえず無視します。 どちらしてもデータがほとんど、またはまったくありません。

---

[7] Rational 内のレビュー担当者によるフィードバックから、ほとんどの重要でないシステムについてこれは十分すぎると考えられます。これらのシステムの場合、ユースケースあたりのシナリオは 30 未満になります。この点とテスト・ケースの数と使用中に発見された欠陥の数との関係に関するデータを調べると興味深いでしょう。

[8] これは固定的な上限ではないことに注意してください。ユースケース記述の長さはある種の統計分布に従いますが、極端な数値はめったに発生しません。

これらのサイズと計画的に異なる開発はこうした発見的方法から派生するユースケースあたりの時間に乗数を適用しなければなりません (COCOMO 形式のコスト・ドライバーの追加を提案します。これはシステム分類について観察された平均サイズを提案された平均サイズで除算したものです。対象となるのは単純なビジネス、より複雑なビジネス、コマンドと制御、などです)。

ユースケース・サイズのもう 1 つの側面はシナリオ・カウントです。例えば、わずか 5 ページのユースケースが多くのパスの可能性がある複雑な構造を持つ場合があります。ここでもシナリオの数を見積もる必要があり、30 (ユースケースあたりのシナリオ数の初期予測) に対する比率をコスト・ドライバーとして使用します。

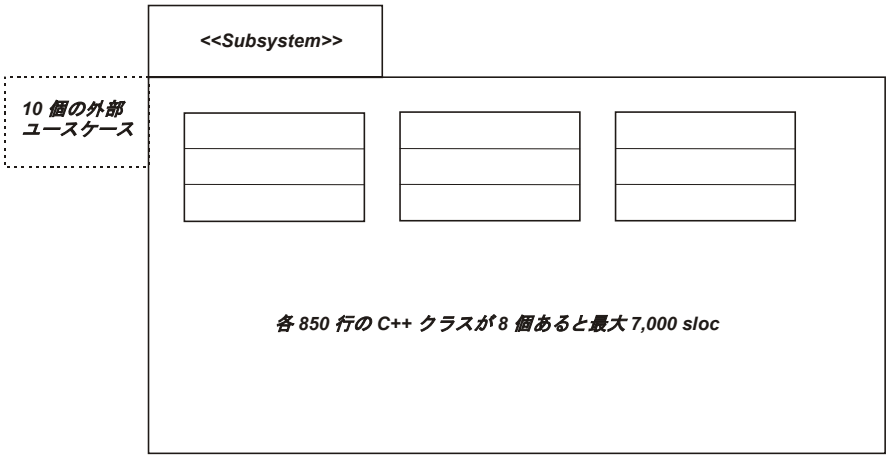
結論として、最大 100 ページのユースケース・ベースの仕様書は任意のレベルの外部仕様について十分で、これに補足仕様が追加されます。範囲は 20 ～ 200 ページ (この限度はあいまい) です。ただし、**最低レベルの** (サブシステム・グループの) システムに関する合計は 3 ～ 15 ページ/ksloc (単純なビジネス・システム) から 12 ～ 30 ページ/ksloc (複雑なコマンドと制御) となります。これによって、成果物のページ・カウントがかなり小さい Royce98 の表 14-9 と特に防衛などで大量の文書を生成した実際のプロジェクトとの矛盾が説明されるでしょう。このホワイト・ペーパーは書き留める必要のない仕様レベルから生まれたものです。Royce 氏の説は正しく、開発構想書のように重要なものは、表で示されている程度の分量であるべきです。大規模で複雑なシステムについては 200 頁ほどでしょう。

サブシステム階層

サブシステム階層とはどのようなものでしょうか。私は使用したことがある簡単な「標準形」を紹介します。これらはシステムを実現するために使用される概念形式であることに注意してください。実際のシステム境界はこれらの形式の集合の外側にあり、それぞれの外部ユースケースの合計はシステムに対する外部ユースケースの合計です。したがって、現実のシステムは 10 個を超える外部ユースケースを持つかもしれませんが、後で説明するように上限は定められません。すべての開発を 4 つのレベルのユースケースで記述しなければならないと提案しているわけではないことに注意してください。小規模のシステム (<50,000 sloc) は 1、2 個のユースケースで足りるかもしれません。

レベル 1

レベル 1 では、0 個以上のサブシステムにおけるクラスによりユースケースを実現します。



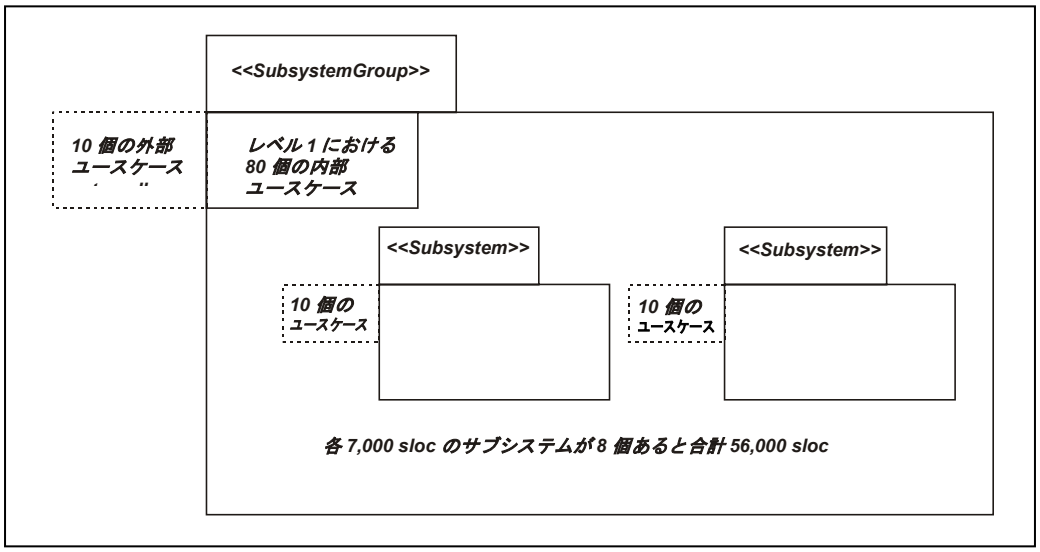
このレベルにおけるシステム・サイズの見積もり範囲 (7 ± 2 という概念を使用) は以下のとおりです。

- 2 ～ 9 クラス (サブシステムへの形成はなし) —1700 ～ 8000 sloc
- 1 サブシステム、5 クラス、合計 4000 sloc
- 9 サブシステム、7 クラス、合計 53,550 sloc

ユースケースはクラス・インスタンスにより実現されるものとして表現されます。ユースケースは 2 ～ 76 個の範囲になります。これらはあいまいな限度です。少なくとも上限はそうです。この方法 (このサイズ) でシステムを構築すると、高レベルの形式で期待した動作を表現できず、限度が 0 に減少する可能性があります。より大きなユースケース・カウントは異常を示すかもしれません。

レベル 2

次のレベルでは、8 個のサブシステムから構成されるサブシステム・グループがあります。これは、防衛用語のコンピューター・システム構成項目 (CSCI) と同等であると思います。このレベルで、ユースケースはサブシステムのコラボレーションにより実現されます。



このレベルにおけるシステム・サイズの見積もり範囲 (7 ± 2 という概念を使用) は以下のとおりです。

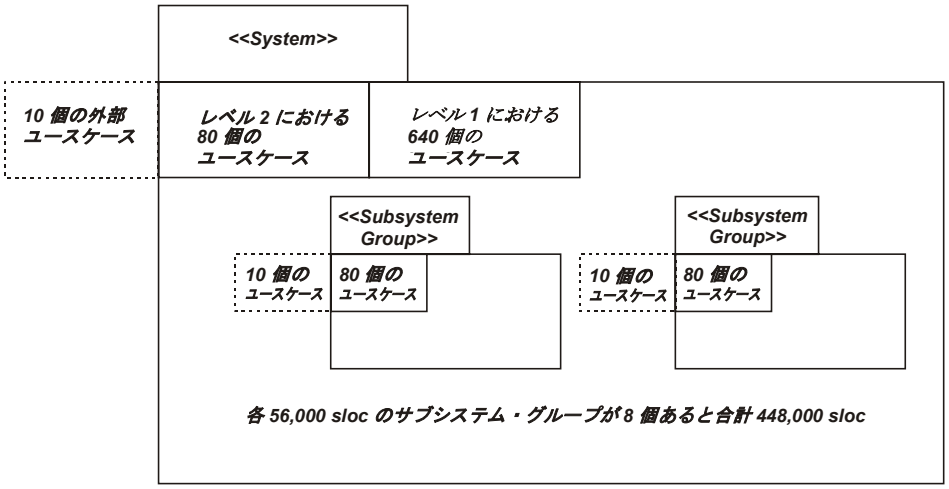
サブシステム・グループが 1 個、サブシステムが 5 個、クラスが 5 個で合計 22,000 sloc

サブシステム・グループが 9 個、各サブシステムが 7 個、クラスが 7 個で合計 370,000 sloc

外部ユースケースは 4 ～ 66 個の範囲になります。ここでも限度はあいまいです。

レベル 3

次のレベルでは、1 個のシステム (サブシステム・グループから構成) があります。レベル 3 で、ユースケースはサブシステム・グループのコラボレーションにより実現されます。



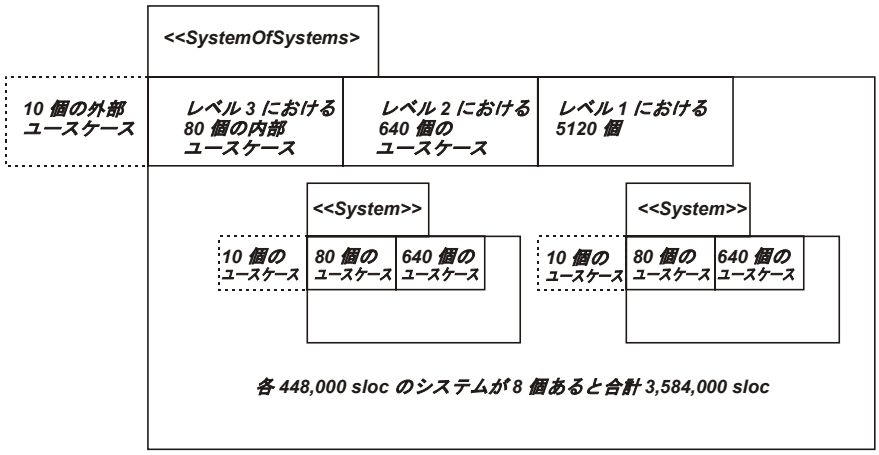
このレベルにおけるシステム・サイズの見積もり範囲 (7 ± 2 という概念を使用) は以下のとおりです。

- システムが 1 個、サブシステム・グループが 5 個、サブシステムが 5 個、クラスが 5 個で合計 110,000 sloc
- システムが 9 個、各サブシステム・グループが 7 個、各サブシステムが 7 個、クラスが 7 個で合計 2,600,000 sloc

外部ユースケースは 3 ～ 58 個の範囲になります。ここでも限度はあいまいです。

レベル 4

次のレベルでは、システムのシステムがあります。レベル 4 で、ユースケースはシステムのコラボレーションにより実現されます。



このレベルにおけるシステム・サイズの見積もり範囲 (7 ± 2 という概念を使用) は以下のとおりです。

- システムのシステムが 1 個、システムが 5 個、サブシステム・グループが 5 個、サブシステムが 5 個、クラスが 5 個で合計 540,000 sloc
- システムのシステムが 9 個、各システムが 7 個、各サブシステム・グループが 7 個、各サブシステムが 7 個、クラスが 7 個で合計 18,000,000 sloc

外部ユースケースは 2 ～ 51 個の範囲になります。ここでも限度はあいまいです。より大きな集約も可能ですが、考えたくありません。

ユースケースあたりの作業量

各レベルで以上のような名目上のサイズに対する作業量を見積もることにより、ユースケースあたりの作業量について理解を深めることができます。Estimate Professional™ ツール[9] (COCOMO 2[10] と Putnam 氏の SLIM[11] モデルに基づく) を使用し、言語を C++ に設定 (他のコスト・ドライバーを名目に設定) して、各名目サイズ・ポイントにおけるシステム・タイプ例のそれぞれに対する作業量 (10 個の外部ユースケースを仮定) を計算すると、表 1 の結果が得られます。

サイズ (sloc)	作業時間/ユースケース (単純なビジネス・システム)	作業時間/ユースケース (科学技術システム)	作業時間/ユースケース (複雑なコマンドおよび制御システム)
7000 (L1)	55 (範囲 40-75)	120 (範囲 90-160)	260 (範囲 190-350)
56000 (L2)	820 (範囲 710-950)	1700 (範囲 1500-2000)	3300 (範囲 2900-3900)
448000 (L3)	12000	21000	38000
3584000 (L4)	148000	252000	432000

表 1:さまざまなサンプル・タイプのユースケース当たりの作業量

[9] Software Productivity Center Inc http://www.spc.ca/ は Estimate Professional ツールを提供します。

[10] 参考資料 Boehm81 と http://sunset.usc.edu/COCOMOII/cocomo.html を参照してください。

[11] 参考資料 Putnam92 を参照してください。

表 1 のレベル 1 (L1) とレベル 2 (L2) で示される範囲は、個々のユースケースの複雑さを示します。この複雑さは COCOMO のコード複雑性マトリックスを使用した類推から見積もられます。L2 において、複雑さのばらつきは、複雑さの変動は、システム・タイプによってその特性に組み込まれ始めると考えられるため、例えば、より高いレベルの複雑なコマンドや制御システムのユースケースなどはより低いレベルの複雑さを混合して含みます。これらの対数を尺度としてプロットすると図 2 が得られます。

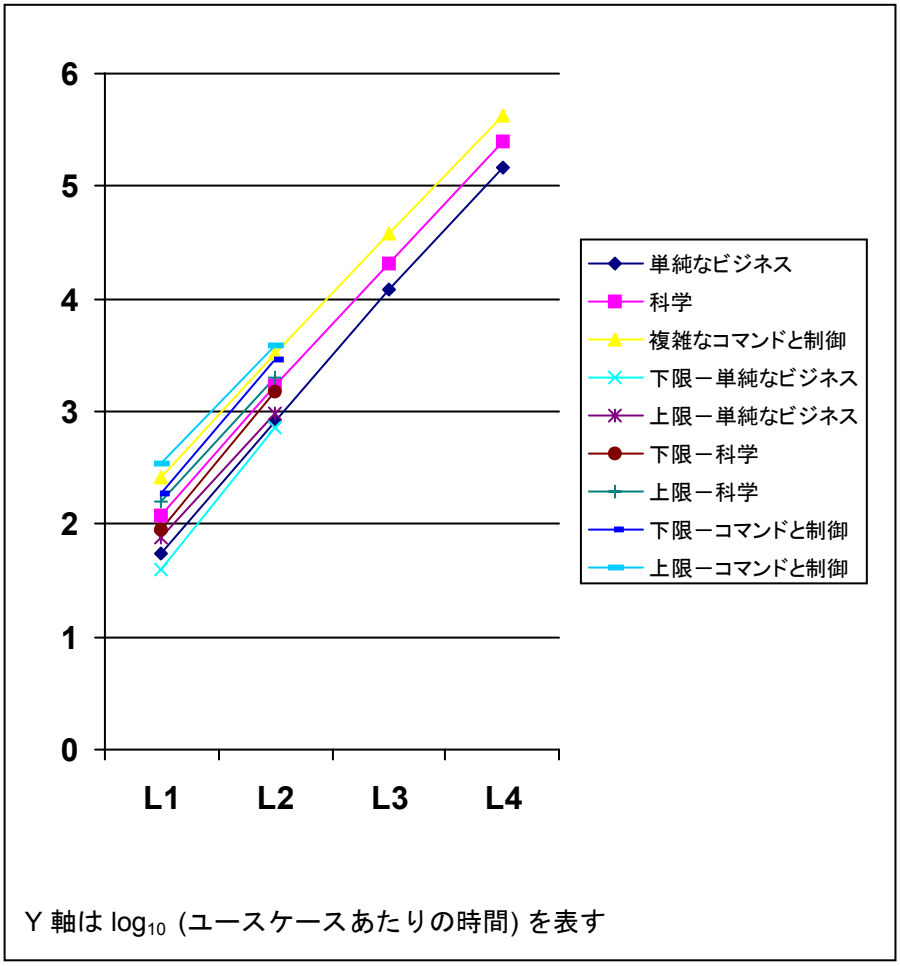


図 2:サイズ別のユースケース作業量

この図から、ユースケースあたり 150 ～ 350 時間 ($10^{2.17}$–$10^{2.54}$) という以前の Objectory 数が L1 に見事に適合することが読み取れます。つまり、これらはクラスのコラボレーションによって実現可能なユースケースです。したがって、最終的にこの数について何らかの正当性が存在します。しかし、分析中にすべてのプロジェクトを特徴付けることは適切ではありません。誰かが電子メールのやりとりで言っていたように、「あまりに『均一化』し過ぎます」。

作業量見積もり

現実のシステムは以上のように便利な枠に収まらないので、システムの特性を判断するために、これまでの考察から得られたあいまいな限度を使用することができます。 これらを図 3 にプロットして示します。

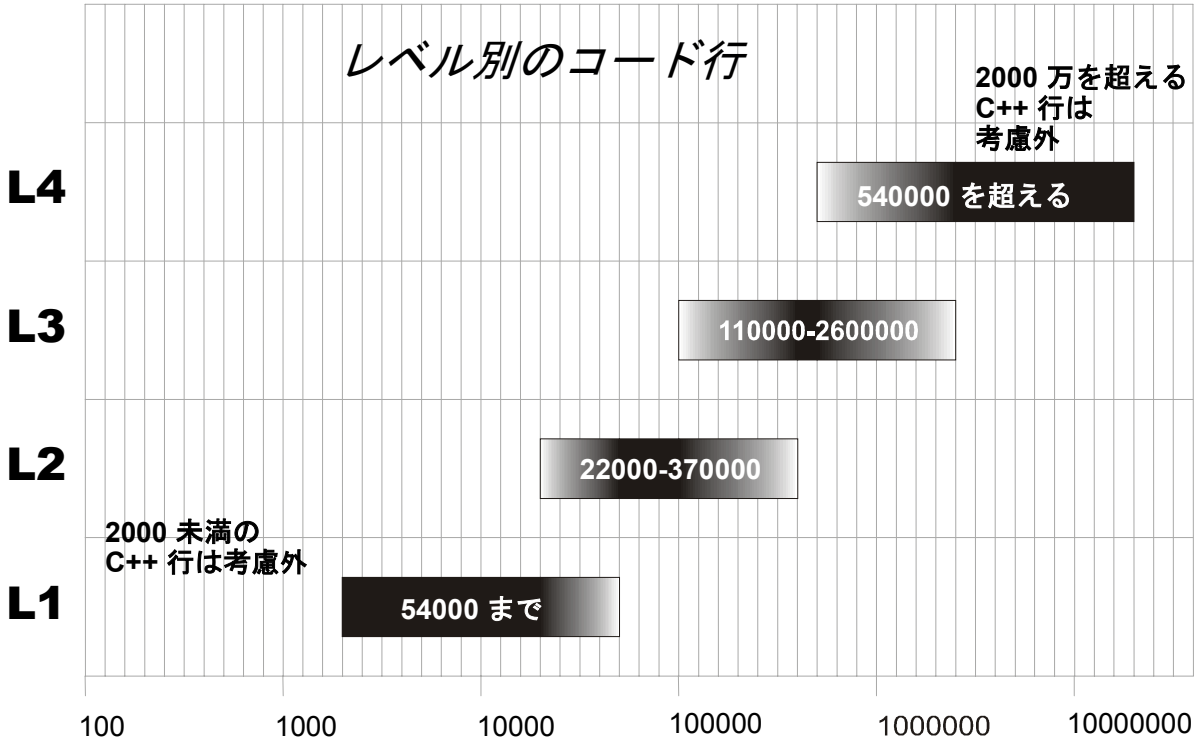


図 3:各レベルのサイズ帯

図 3 から、22000 sloc までのシステムはレベル 1 で記述され、ユースケース・カウントは 2 ～ 30 の可能性が最も高いことがわかります。このサイズでより高いユースケース・カウントは、ユースケースが細分化されすぎていることを意味するかもしれません。

22000 ～ 54000 sloc の場合は、レベル 1 とレベル 2 のユースケースの組み合わせになる可能性があり、ユースケース・カウントは 4 (すべてレベル 2) ～ 76 (すべてレベル 1) です。図からわかるように、こうした**極端な**値はめったに発生しません。

54000 ～ 110000 sloc の場合は、全体がレベル 2 で記述された構造がしっかりしたシステムの可能性があり、ユースケース・カウントは 10 ～ 20 となります。混在は L1/L2/L3 (1 ～ 160 個のユースケース。 ただし、極端な値はまれ) かもしれません。

110000 ～ 370000 sloc の場合は、レベル 2 とレベル 3 の混在が考えられ、ユースケース・カウントは 3 (すべてレベル 3) ～ 66 (すべてレベル 2) です。

370000 ～ 540000 sloc の場合、全体がレベル 3 で記述される場合、ユースケース・カウントは 9 ～ 12 になるでしょう。混在は L2/L3/L4 (1 ～ 100 個のユースケース。ただし、**極端な**値はまれ) かもしれません。

540000 ～ 2600000 sloc の場合、レベル 3 とレベル 4 の混在が考えられ、ユースケース・カウントは 2 (すべてレベル 4) ～ 60 (すべてレベル 3) です。

2600000 sloc を超える場合、レベル 4 のユースケース・カウントは 8 以上になるでしょう。

十分な数量のユースケースとは

これまでの考察からいくつかの経験則を支持する興味深い見解が得られます。よくある質問は「いくつのユースケースだと多すぎるか」です。この質問はたいてい**要求把握時**に多すぎる数量を意味します。最大規模のシステムでも 70 を超える場合は設計前段階として、細分化されすぎているかもしれないというのが答えになりそうです。5 ～ 40 は適度ですが、レベルを考慮しない数そのものはサイズと作業量の見積もりに使用できません。これは特定のレベルにあてはまる**初期の**数です。大きなスーパーシステムが複数のシステムに、さらにサブシステムに、というように分割されるとユースケース・カウントが何百にもなります。クラス・レベルに達するまでユースケースが開発されると、最終的なカウントは何百または何千になる可能性があります (140 人年のプロジェクト、またはユースケースあたり 15 個のファンクション・ポイントがある場合は 最大 600)。しかし、これは、設計から独立して純粋なユースケース分割として発生するわけではありません。こうしたユースケースは Jacobson97 で記述されたプロセスから生まれます。 システム・レベルのユースケースはサブシステムに割り当てられた動作に分割されます。(他のサブシステムをアクターとして) それらについてさらに低レベルのユースケースを作成できます 。

作業量見積もり手順

では、どのように見積もりを進めたらよいのでしょうか。いくつか前提条件があります。ユースケースに基づく見積もりは問題領域の理解なしでは行えません。また、**提案されるシステム・サイズの概念や、アーキテクチャーの知識、見積もりを行うのに適した段階を理解する必要があります。**

見積もりの最初の概算は専門家の意見を聞くか、もう少し正式に Wideband Delphi テクニック (これは 1948 年に Rand 機関で考案されました。 詳細は「参考資料」Boehm81 を参照) を使用して実行できます。これにより、見積もり担当者はシステムを図 3 のサイズ帯のいずれかに配置できます。この配置はユースケース・カウントの範囲と表現のレベル (L1、L1/L2、など) を示します。次に、見積もり担当者は、現在のアーキテクチャーの知識や領域の語彙を元にして、ユースケースが 1 つのレベルに合致するか、分割するか、レベルを混在させるかを決めなければなりません (イベントのフローが表現される方法で)。

これらを考慮することで、データが異常である可能性についても明らかになるでしょう。例えば、Delphi の見積もりが 600,000 コード行 (または同等のファンクション・ポイント) で、アーキテクチャーに関する調査がほとんど行われていないため、まだシステム・アーキテクチャーがよくわからない場合、図 3 を参照すると、ユースケース・カウントは 2 (すべてレベル 4) ～ 14 (すべてレベル 3) が提案されます。実際のユースケース・カウントが 100 の場合、ユースケースの分割が時期尚早だったかもしれませんし、Delphi の見積もりが限度を越えているかもしれません。

この例の話を続けましょう。実際のユースケース・カウントが 20 であり、見積もり担当者がすべて L3 と定め、さらにユースケースの長さが平均 7 ページで、システムが複雑なビジネス・タイプの場合、ユースケースあたりの時間は (図 2 から) 20,000 です。これに 7/9 を掛けて、低レベルの複雑さを明らかにしなければなりません (ユースケースの長さに基づく)。したがって、この方法による合計作業量は 20*20000* (7/9) = 310,000 人時、または 2050 人月となります。Estimate Professional によると、複雑なビジネス・システムにおける 600,000 C++ コード行は 1928 人月を要します。したがって、この架空の例ではかなり一致しています。

実際のユースケース・カウントが 5 であり、見積もり担当者がこれらをレベル 4 に 1 個、レベル 3 に 4 個分割し、さらに、 L4 のユースケースが 12 ページ、L3 のユースケースが平均 10 ページだとすると、作業量は 1*250,000*12/9+4*21000* (10/9) = 最大 2800 人月となります。この結果から、Delphi の見積もりの見直しが必要のようです。システムの大半はまだ大まかにしか理解されていませんが、エラー範囲は大きくなっています。

元の Delphi の見積もりが 100,000 C++ コード行だった場合、図 3 から、ユースケースは L2 で 18 個と提案されます。最初の例のように実際は 20 個あった場合、実際のユースケース・レベルを考慮しないでこの方法を適用すると、Delphi の見積もりがかなり間違っているいう仮定で、ひどい欠陥を生む可能性があります。

したがって、見積もり担当者は、ユースケースが本当に提案される抽象レベル (L2) であること、サブシステムのコラボレーションにより実現可能であること、ユースケースが本当に L3 にないことをチェックしなければなりません。 もっとも、Wideband Delphi 手法はたいていそれほど悪くはありません (実際は 600,000 近くのものを 100,000 と予測)。しかし、要点は、ユースケース・レベルと連携するなんらかの概念もしくは概念的アーキテクチャーの構築なしに、信頼をもってこの見積もり手法を進めることができないということです。ある領域において熟達した見積もり担当者にとって、レベルの判断を可能にする精神的な何かなのかもしれません。 経験の十分でない見積もり担当者とチームにとっては、一定のアーキテクチャー・モデリングを行い、特定のレベルでユースケースがどのように実現可能かを確認することが賢明です。

混在表現ユースケース (つまり、レベル N とレベル N+1 の混合) のカウントは下限ユースケース・タイプの n=8 (2 つのレベル間の端数距離) としてカウントされなければなりません。したがって、50% の L1 と 50% の L2 で評価されるユースケースは $8^{0.5}$ = 3 個の L1 ユースケースとしてカウントし、全体のカウントを得なければなりません。L2 と L3 間の 30% の位置で評価されるユースケースは $8^{0.3}$ 個の L2 ユースケース = 2 個の L2 ユースケースとしてカウントしなければなりません。L2 と L3 間の 90% の位置で評価されるユースケースは $8^{0.9}$ = 7 個の L2 ユースケースとしてカウントしなければなりません。

テーブルのサイズ調整

全体サイズを明らかにするために、個々のユースケースあたりの時間数についてさらに調整する必要があります。各レベルの作業量数値は**そのサイズのシステム状況**で有効です。したがって、表 1 の L1 で、ユースケースあたり 55 時間という値は 7000 sloc のシステム構築時に適用されます。実際の数値は合計システム・サイズに依存するため、構築予定のシステムが例えば 40,000 sloc で、それを記述する 57 個のレベル 1 ユースケースが存在する場合、単純なビジネス・システム向けの作業量は 55*57 時間ではなく、 $(40/7)^{0.11}$ * 55 = 66 時間/ユースケースとなります。これは、サイズと作業量の COCOMO 2 関係に基づきます。COCOMO モデルによると、Effort = A * $(Size)^{1.11}$ となります。

- Size は ksloc です
- A はコスト・ドライバー要因を持ちます
- プロジェクト・スケール・ファクターは名目上のものです (指数として 1.11 を与えます)

計算の負荷を取り除くため、これらの計算は Estimate Professional などのツールに組み込まれている場合があることに注意してください。ここでは完全性の目的で表示しています。

したがって、ksloc またはユニットあたりの作業量は A* $(Size)^{1.11}$/Size、つまり A* $(Size)^{0.11}$ となり、サイズ S1 における作業量/ユニットのサイズ S2 における作業量/ユニットの比率は $(S1/S2)^{0.11}$ となります。

Delphi の見積もりに加えて、システム・サイズはさまざまなレベルのユースケース・カウントから概算することができます。ユースケースの数がレベル 1 で N1、レベル 2 で N2、レベル 3 で N3、レベル 4 で N4 の場合、合計サイズは [(N1/10) *7 + (N2/10) *56 + (N3/10) *448 + (N4/10) *3584] ksloc です。このため、表 1 のユースケースあたりの各作業量数値について、作業量乗数を計算できます。 これは、合計サイズを表 1 の列 1 に示される各レベルのサイズ (単位：ksloc) で除算することにより行います。

- レベル 1　　$(0.1*N1 + 0.8*N2 + 6.4*N3 + 51.2*N4)^{0.11}$
- レベル 2　　$(0.0125*N1 + 0.1*N2 + 0.8*N3 + 6.4*N4)^{0.11}$
- レベル 3　　$(0.00156*N1 + 0.0125*N2 + 0.1*N3 + 0.8*N4)^{0.11}$
- レベル 4　　$(0.00002*N1 +0.00156*N2 +0.0125*N3 +0.1*N4)^{0.11}$

明らかに、例えばレベル 4 で、レベル 1 ユースケースの数は、レベル 3 またはレベル 4 の数にくらべて影響が非常に小さいことがわかります。

まとめ

ユースケースに基づく見積もりのフレームワークを述べてきました。説明に具体性を持たせるため、それほど誤りはないと考えるフレームワーク・パラメーターを紹介しました。例によって、こうした推測は現実の環境でテストされるべきで、データとして再見積もりされたパラメーターが収集されます。このフレームワークは、システムの異なるカテゴリーについて、ユースケースのレベル、サイズ、複雑さという概念を考慮しますが、細かい機能分割は行いません。計算の負荷を軽減するために、ユースケースに基づく入力サイズの代替手段を提供する Estimate Professional などのツールをフロントエンドとして構成することができます。

このホワイト・ペーパーに関するご意見やフィードバックは John Smith、jsmith@rational.com 宛てにお願いします。

参考資料

Armour96:"Experiences Measuring Object Oriented System Size with Use Cases", F. Armour, B. Catherwood, et al., Proc. "ESCOM", Wilmslow, UK, 1996

Boehm81:"Software Engineering Economics", Barry W. Boehm, Prentice-Hall, 1981

Booch98:"The Unified Modeling Language User Guide", Grady Booch, James Rumbaugh, Ivar Jacobson, Addison-Wesley, 1998 (邦訳:「UML ユーザーガイド」、オージス総研オブジェクト技術ソリューション事業部 訳、羽生田 栄一 監訳、ピアソン・エデュケーション)

Cockburn97:"Structuring Use Cases with Goals", Alistair Cockburn, Journal of Object-Oriented Programming, Sept-Oct 1997 and Nov-Dec 1997

Douglass99:"Doing Hard Time", Bruce Powel Douglass, Addison Wesley, 1999

Fetcke97:"Mapping the OO-Jacobson Approach into Function Point Analysis", T. Fetcke, A. Abran, et al., Proc. "TOOLS USA 97", Santa Barbara, California, 1997

Graham95:"Migrating to Object Technology", Ian Graham, Addison-Wesley, 1995

Graham98:"Requirements Engineering and Rapid Development", Ian Graham, Addison-Wesley, 1998

Henderson-Sellers96:"Object-Oriented Metrics", Brian Henderson-Sellers, Prentice Hall, 1996

Hurlbut97:"A Survey of Approaches For Describing and Formalizing Use Cases", Russell R. Hurlbut, Technical Report:XPT-TR-97-03, http://www.iit.edu/~rhurlbut/xpt-tr-97-03.pdf

Jacobson97:"Software Reuse – Architecture, Process and Organization for Business Success", Ivar Jacobson, Martin Griss, Patrik Jonsson, Addison-Wesley/ACM Press, 1997

Jones91:"Applied Software Measurement", Capers Jones, McGraw-Hill, 1991 (邦訳:「ソフトウェア開発の定量化手法」第 2 版 鶴保征城、富野壽監 訳、共立出版)

Karner93:"Use Case Points - Resource Estimation for Objectory Projects", Gustav Karner, Objective Systems SF AB (copyright owned by Rational Software) , 1993

Lorentz94:"Object-Oriented Software Metrics", Mark Lorentz, Jeff Kidd, Prentice Hall, 1994 (邦訳:「オブジェクト指向ソフトウェアメトリクス - 現実的な運用のためのガイド -」、オージス総研 訳、宇治邦明 監訳、株式会社プレンティスホール出版)

Major98:"A Qualitative Analysis of Two Requirements Capturing Techniques for Estimating the Size of Object-Oriented Software Projects", Melissa Major and John D. McGregor, Dept. of Computer Science Technical Report 98-002, Clemson University, 1998

Minkiewicz96:"Estimating Size for Object-Oriented Software", Arlene F. Minkiewicz, http://www.pricesystems.com/foresight/arlepops.htm, 1996

Pehrson96:"Software Development for the Boeing 777", Ron J. Pehrson, CrossTalk, January 1996

Putnam92:"Measures for Excellence", Lawrence H. Putnam, Ware Myers, Yourdon Press, 1992

Rechtin91:"Systems Architecting, Creating & Building Complex Systems", E. Rechtin, Prentice-Hall, 1991

Royce98:"Software Project Management", Walker Royce, Addison Wesley, 1998

RUP99:"Rational Unified Process", Rational Software, 1999

Stevens98:"Systems Engineering – Coping with Complexity", R. Stevens, P. Brook, et al., Prentice Hall, 1998

Thomson94:"Project Estimation Using an Adaptation of Function Points and Use Cases for OO Projects", N. Thomson, R. Johnson, et al., Proc. "Workshop on Pragmatic and Theoretical Directions in Object-Oriented Software Metrics", OOPSLA '94, 1994

Rational®
the software development company

Dual Headquarters:

Rational Software
18880 Homestead Road
Cupertino, CA 95014
Tel: (408) 863-9900

Rational Software
20 Maguire Road
Lexington, MA 02421
Tel: (781) 676-2400

Toll-free: (800) 728-1212
E-mail:info@rational.com
Web:www.rational.com
International Locations:www.rational.com/worldwide