

# 第4章

## ソフトウェアの測定と見積もり

"測定したものは改善される"

-ドナルド・ラムズフェルド著『Known and Unknown: 回想録』

測定とは、物体の特性に数値や記号を割り当てるプロセスである。意味のある数値の割り当てを行うには、ルールや理論（あるいはモデル）に支配されなければならない。ソフトウェアには、測定可能な多くの特性がある。高さ、幅、重さ、化学組成など、物理的な物体の特性を測定することができる。高さ、幅、重さ、化学組成など、物理的な物体の特性を測定することができます。このような測定によって得られる数値は、それ自体ではほとんど価値がありません。例えば、物体を持ち上げるのに必要な重さを知ることができる。あるいは、物理的な寸法を知ることで、その物体が特定のスペースに収まるかどうかを判断することができる。同様に、ソフトウェアの測定は通常、目的を持って行われる。一般的な目的は、マネジメントの意思決定である。例えば、プロジェクトマネージャーは、開発コストやソフトウェア製品を開発し提供するのにかかる時間を見積もることができるようにしたい。物体の重さを知ることで、それを持ち上げるために必要なことを決定できるのと同様に、あるソフトウェアの特性を測定することで、必要な開発工数を見積もることができるようになることを期待している。

ソフトウェア測定の用途：

### 内容

#### 4.1 計測理論の基礎

- 4.1.1 測定理論
- 4.1.2
- 4.1.3

#### 4.2 何を測るのか？

- 4.2.1 サイクロマティック複素数
- 4.2.2 ユースケース・
- ポイント 4.2.3
- 4.2.4

#### 4.3 モジュールの凝集性の測定

- 4.3.1 内部凝集性または構文的凝集性
- 4.3.2 意味的結束
- 4.3.3
- 4.3.4
- 4.2.3

#### 4.4 心理的複雑性

- 4.4.1 アルゴリズム情報コンテンツ
- 4.4.2
- 4.4.3
- 4.4.4

#### 4.5

- 4.5.1
- 4.5.2
- 4.5.3
- 4.5.4

#### 4.6 まとめと文献的メモ 問題点

- コストと労力の見積もり（ライフサイクルの初期が望ましい）
- 設計と実施の質を向上させるためのフィードバック

明らかに、ソフトウェア製品が完成すれば、それを完成させるためにどれだけの労力がかかったかがわかる。ソフトウェアの他の特性から間接的に推測する必要はなく、投入された労力が直接わかる。しかし、それでは経営上の意思決定には遅すぎる。経営上の意思決定には、開発に着手する前に、あるいは少なくともプロセスの十分早い段階で、労力を知る（あるいは見積もる）ことが必要であり、そうすれば顧客と予算や納期について有意義な交渉ができる。



また、あるエンティティの特性を別のエンティティの特性から推論しようとしていることにも注意してください：測定されるエンティティの特性はソフトウェア（設計文書またはコード）であり、推定されるエンティティの特性は**開発プロセス**（人々の努力）です。通常、「見積もりモデル」は経験則に基づいている。つまり、過去のプロジェクトの観察に基づいて導き出される。過去のプロジェクトでは、ソフトウェアとプロセスの両方の特性が知られている。このことから、例えば機能的特徴の数と、例えば必要な開発工数との相関を計算しようとすることができる。ある範囲の値で相関が高ければ、機能的特徴の数は必要な開発工数の良い予測因子であると推測できます。しかし残念ながら、相関関係が因果関係とイコールでないことは分かっています。可能であれば、関係を立証するだけでなく、その理由も説明できるような**因果関係モデル**があった方がよいでしょう。

ソフトウェア・モジュールやシステムの特性として最も一般的に測定されるのは、そのサイズと複雑性に関連するものである。セクション2.5では、結合や凝集といったいくつかのソフトウェア特性について触れ、"良い設計"は"低い結合"と"高い凝集"によって特徴付けられると主張した。本章では、結合と凝集を測定し、ソフトウェア設計と実装の品質を定量化するための技法をいくつか紹介する。ユビキタスなサイズの尺度は、コード行数（LOC）である。複雑さはソフトウェア製品の重要な特性として容易に観察されるが、複雑さを測定できるように運用するのは難しい。

問題の最も表面的な細部間の最も単純な相関関係を超えた、理路整然とした、思慮深い

アプローチをとること。

ソフトウェアが複雑であればあるほど、開発や保守が難しくなることに同意するのは簡単だが、複雑さを測定できるように運用するのは難しい。読者はすでに、*計算複雑性*の尺度であるビッグO（またはビッグオー）、 $O(n)$ になじみがあるかもしれない。 $O(n)$ は、入力データのサイズがアルゴリズムの計算資源（通常は実行時間やメモリ）の使用量にどのような影響を与えるかという観点から、マシンの視点からソフトウェアの複雑性を測定するものである。しかし、ソフトウェア工学で必要とされる複雑さの測定は、人間の開発者の視点から複雑さを測定する必要がある。

## 4.1 計測理論の基礎

---

"正確に間違っているよりも、おおそ正しい方がよい"-ジョン・メイナード・ケインズ

ホーソン効果-特別視され、重要であると感じられるという心理的刺激によって生じる労働者の生産性の向上。ホーソン効果とは、環境条件の変化に対する行動やパフォーマンスの一時的な変化を表す。この変化は典型的には改善である。また、この定義を拡大し、新たな注目や注目の高まりをきっかけに、人々の行動やパフォーマンスが変化することを意味する人もいる。

自分が研究されていることを知っているために、個人の行動が変化する可能性があることは、イリノイ州シセロにあるウェスタン・エレクトリック社のホーソン工場での研究プロジェクト（1927-1932年）で実証されている。

生産工程をつぶさに観察することによって、その工程が最初に改善されること。この効果が最初に注目されたのは、ウェスタン・エレクトリック社のホーソン工場であった。生産量が増加したのは、工場の経営陣が導入した労働条件の実際の変化の結果ではなく、経営陣がそのような改善に関心を示したからである（関連：自己実現仮説）。



### 4.1.1 測定理論

測定理論は応用数学の一分野である。私たちが用いる具体的な理論は、*測定の表現論*と呼ばれる。これは、世界が実際に機能する方法についての直観を形式化したものである。

測定理論では、統計と確率を使って、データに含まれる可能性のある分散、範囲、誤差の種類を定量的に理解することができる。

## 測定スケール

測定理論では、名目、順序、間隔、比率、絶対の5種類の尺度がある。

**名目尺度では**、対象をさまざまなカテゴリーに分類することができる。例えば、天候を"晴れ"、"曇り"、"雨"、"雪"とする。カテゴリの2つの重要な要件は、共同網羅性と相互排他性である。相互排他的とは、測定された属性が1つのカテゴリに分類されることを意味します。共同排他的とは、すべてのカテゴリが一緒になって属性のすべての可能な値をカバーすべきであることを意味する。測定された属性が我々の関心よりも多くの力

カテゴリを持つ場合、カテゴリを共同排他的にするために "その他" のカテゴリを導入することができる。カテゴリが共同網羅的で相互に排他的であれば、統計分析の適用に必要な最小条件が得られる。たとえば、ソフトウェア製品の異なるカテゴリ間で、欠陥率、サイクルタイム、要件欠陥などのソフトウェア属性の値を比較したい場合があります。

**順序尺度とは、**対象を順番に比較できる測定操作を指す。序数尺度の例としては「Bad」、「Good」、「Excellent」、またはWeb上の製品やサービスに使われる「星」評価などである。順序尺度は、 $A > B$  が真なら  $B > A$  は偽という意味で非対称である。また、 $A > B$  かつ  $B > C$  であれば、 $A > C$  であるという他律性を持つ。順序尺度は、測定された特性の大きさによって対象を序列化するが、次のような情報は提供しない。

ス大学

被験者間の差の相対的な大きさについて。例えば、"優秀"は"優秀"よりも優れており、"優秀"は"不良"よりも優れていることだけはわかる。しかし、「優れている」-「良い」、「良い」-「悪い」のペア間の相対的な違いを比較することはできません。よく使われる順序尺度は、リカートの5点、7点、10点などの $n$ 点リカート尺度である。例えば、本や映画を評価するための5段階リッカート尺度では、次のような値を割り当てることができる：1="嫌い"、2="嫌い"、3="どちらでもない"、4="好き"、5="好き"。私たちは、 $5 > 4$ 、 $4 > 3$ 、 $5 > 2$ 、などということだけは知っているが、5が4よりどれだけ大きいかは言えないし、カテゴリー5と4の差が3と2の差に等しいとも言えない。このことは、足し算、引き算、掛け算、割り算といった算術演算が使えないことを意味する。それにもかかわらず、距離が等しいという仮定がなされ、平均評価が報告されることが多い（例えば、Amazon.comの商品評価では、星3.7といった小数値が使われている）。

**インターバル・スケール**は、測定点間の正確な差を示す。インターバル・スケールには、共通の基準として合意でき、再現性のある、明確に定義された固定測定単位が必要です。良い例は、伝統的な温度スケール（摂氏または華氏スケール）です。どちらのスケールでもゼロ点は定義されているが、それは恣意的なものであり、何の意味も持たない。したがって、フロリダの平均気温 $80^{\circ}\text{F}$ とアラスカの平均気温 $20^{\circ}\text{F}$ の差は $60^{\circ}\text{F}$ であると言うことはできるが、 $80^{\circ}\text{F}$ が $20^{\circ}\text{F}$ の4倍暑いとは言わない。加算と減算の算術演算は、区間スケールのデータにも適用できる。

**比率尺度**とは、絶対的または非恣意的なゼロ点を置くことができる区間尺度のことである。絶対ゼロまたは真のゼロは、ゼロ点が測定される特性の不在を表すことを意味する（例えば、お金がない、行動がない、正しいものがない）。例としては、質量、温度（ケルビン）、長さ、時間間隔などがある。比率尺度は測定の最高レベルであり、割り算や掛け算を含め、すべての算術演算を適用できる。

区間尺度と比率尺度の場合、測定値は整数データと非整数データの両方で表すことができる。整数データは通常、頻度カウントで与えられる（例えば、テスト段階で遭遇する可能性のある欠陥の数）。

**絶対尺度**は、ある特性を測定する方法が1つしかない場合に用いられる。特定の物質の物理的特性とは無関係である。実際には、絶対尺度の値は（常にではないにせよ）通常数えることによって得られる。例えば、部屋の中の椅子など、実体を数えることである。



。

## 基本的な対策

比率 比率 率

シックスシグマ

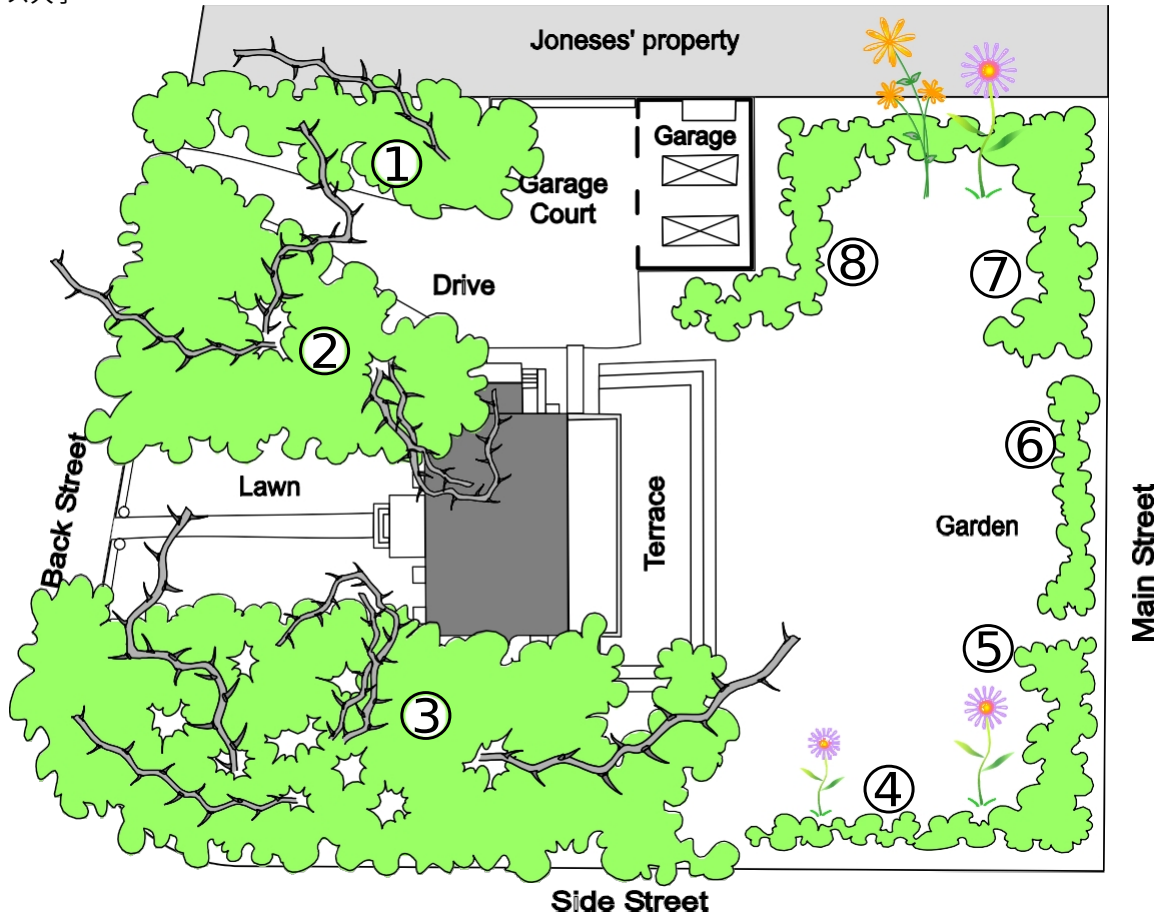


図4-1: 主観的なサイズ測定の問題点 (図1-10との比較)。悲観主義者が見たヘッジの左側、楽観主義者が見たヘッジの右側。

## 4.2 何を測るのか?

ソフトウェアの成果物（設計文書やソースコード）が与えられた場合、一般的に次のような測定が可能です。

1. 問題や解決策に関するあらゆる表現や記述の属性。表現の2つの主なカテゴリーは、構造と振る舞いである。
2. 開発プロセスや方法論の属性。測定された側面

:

- 量
- 複雑さ

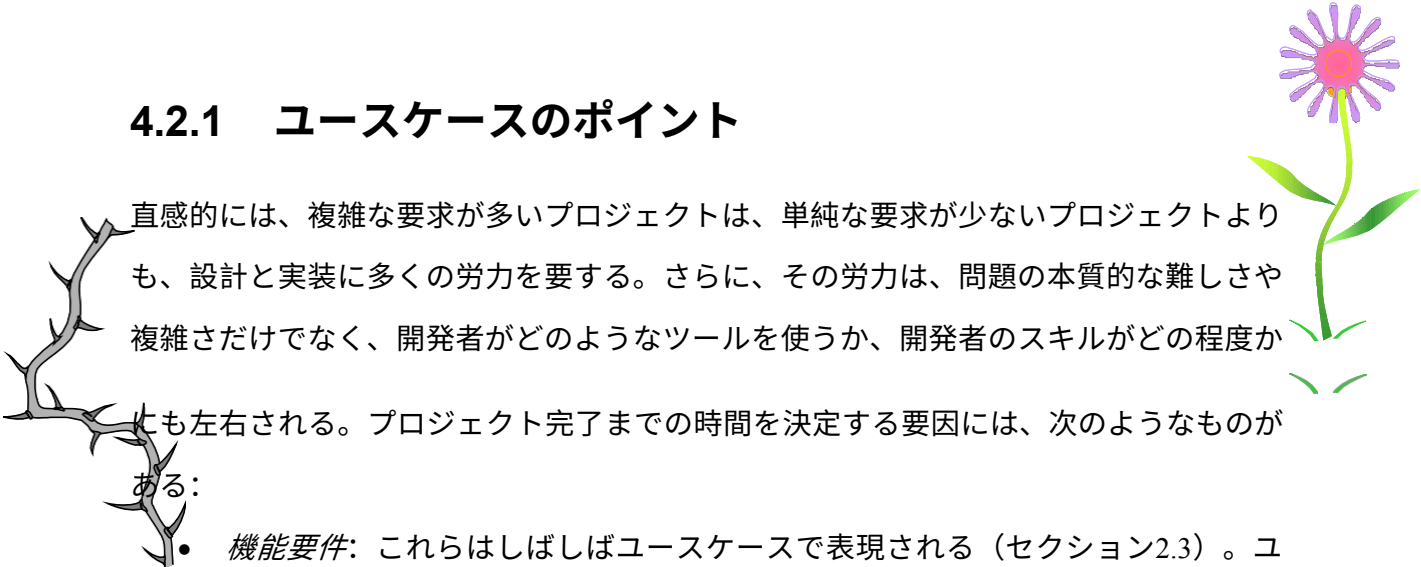
ソフトウェア計測の目的がコストと工数の見積もりであれば、ソフトウェアライフサイクルの早い段階で計測したい。通常、予算配分は調達プロセスの初期段階で設定され、契約価格の決定はこの予算制約と

ス大学

サプライヤーの入札回答その結果、計画されたシステムの機能分解はハイレベルである必要があるが、暗黙の要件や隠れた複雑性を可能な限り多く、可能な限り早期に洗い出すのに十分な詳細さでなければならない。理想的な世界では、これはユースケースの完全かつ詳細な分解となるが、見積もりプロセスでは、厳しい時間枠の中で見積もりを作成する必要があるため、現実的ではない。

図4-1

### 4.2.1 ユースケースのポイント



直感的には、複雑な要求が多いプロジェクトは、単純な要求が少ないプロジェクトよりも、設計と実装に多くの労力を要する。さらに、その労力は、問題の本質的な難しさや複雑さだけでなく、開発者がどのようなツールを使うか、開発者のスキルがどの程度かにも左右される。プロジェクト完了までの時間を決定する要因には、次のようなものがある：

- **機能要件:** これらはしばしばユースケースで表現される（セクション2.3）。ユースケースの複雑さは、アクターの数と複雑さ、および各ユースケースを実行するためのステップ（トランザクション）の数に依存します。
- **非機能要件:** セキュリティ、ユーザビリティ、パフォーマンスなど、FURPS+（セクション2.2.1参照）として知られるシステムの非機能特性を記述する。これらは、"技術的複雑さ要因"とも呼ばれる。
- **環境要因:** 開発チームの経験や知識、開発に使用するツールの洗練度など、さまざまな要因。

プロジェクトのライフサイクルの早い段階で上記の要素を考慮し、実際の完成時間の20%以内といった妥当な精度の見積もりを出す見積もり方法は、プロジェクトのスケジューリング、コスト、資源配分に非常に役立つだろう。

ユースケースはシステム設計の初期段階または想定段階で開発されるため、ソフトウェアライフサイクルの早い段階でプロジェクトのスコープを理解する機会が得られます。ユースケースポイント（UCP）法は、ユースケースに基づいてソフトウェアプロジェク

トに必要な工数を見積もる能力を提供する。UCPメソッドは、ユースケースのアクター、シナリオ、非機能要件、および環境要因を分析し、式に抽象化します。UCP法を適用する前に、詳細なユースケースの記述（セクション2.3.3）を導き出す必要があります。UCP法は、大雑把なユースケースには適用できません。セクション2.3.1で説明したように、この非常に早い段階でプロジェクトの工数見積もりにユーザーストーリーポイント（セクション2.2.3で説明）を適用することができます。

UCPの計算式は3つの変数で構成される：

1. *未調整ユースケースポイント (UUCP)*：機能要件の複雑さを測定する。
2. *技術的複雑性係数 (TCF)*：非機能要求の複雑性を測定する。
3. 開発チームの経験と開発環境を評価する *環境複雑性因子 (ECF)*。

表4-1: アクターの分類と関連する重み。

俳優タイプ	アクター・タイプの認識方法の説明	重量
シンプル	アクターは、定義されたアプリケーション・プログラミング・インターフェース(API)を通して私たちのシステムと相互作用する別のシステムです。	1
平均	アクターとは、テキストベースのユーザーインターフェースを通じて対話する人、またはネットワーク通信プロトコルなどのプロトコルを通じて対話する別のシステムのことであり。	2
コンプレックス	アクターとは、グラフィカル・ユーザー・インターフェースを介して対話する人のことであり。	3

各変数は、重み付けされた値、主観的な値、制約定数を使って個別に定義され、計算される。主観的な値は、開発チームがプロジェクトの技術的な複雑さとチームの効率に対する認識に基づいて決定する。以下はその式である：

$$ucp = uucp \times tcf \times ecf \quad (4.1)$$

未調整ユースケース・ポイント (UUCP) は、これら2つの要素の合計として計算される：

1. 未調整アクターウェイト (UAW) は、すべてのユースケースにおけるすべてのアクターの複合性に基づきます。
2. すべてのユースケースシナリオに含まれるアクティビティ（またはステップ）の総数に基づく、未調整ユースケースウェイト (UUCW) 。

これらの成分の計算については、次に説明する。

## 未調整アクターウェイト (UAW)

ユースケースのアクターは、人、別のプログラム、ハードウェアの一部などです。各アクターの重みは、アクターとシステム間のインターフェースがどの程度洗練されているかによって決まります。テキストベースのコマンドライン・インターフェースで作業するユーザーのようなアクターには、非常に単純なニーズがあり、ユースケースの複雑さをわずかに増加させるだけです。高度に対話的なグラフィカル・ユーザー・インターフェースで作業するユーザーのような他のアクターは、ユースケースを開発する労力

にはるかに大きな影響を与えます。これらの違いを把握するために、システム内の各アクターは単純、平均、複雑の3つに分類され、表4-1に示すような重み付けがなされる。アクターの複雑さを評価するこの尺度は、専門家である開発者が経験に基づいて考案したものです。これは順序尺度であることに注意してください（セクション 4.1.1）。これは、本（Amazon.com）、映画（IMDb.com）、レストラン（yelp.com）の「星評価」に似た、「星評価」のための尺度と考えることができます。あなたの仕事は、このスケールを使って、システム内のすべての俳優に「星の評価」を割り当てることです。私たちの場合、俳優に1つ、2つ、3つの「星」を割り当てることができ、それぞれ「単純」、「平均」、「複雑」に対応します。表 4-2 は、2.3.1 節でアクタを説明した、家庭のアクセス制御のケーススタディにおけるアクタに対する私の評価です。UAWは、各カテゴリのアクターの数合計し、それぞれの合計に指定の重み付け係数を掛け、得られた製品を加算することで計算されます：

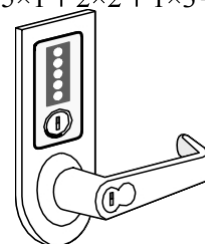
表4-2: 家庭の入退室管理のケーススタディにおけるアクターの分類（セクション2.3参照）。

俳優名	関連する特徴の説明	複雑さ	重量
家主	家主はグラフィカル・ユーザー・インターフェースを介してシステムとやりとりする（中央コンピューターでユーザーを管理する場合）。	コンプレックス	3
テナント	テナントは、テキストベースのユーザーインターフェース（キーパッドによる本人確認を想定。複雑な俳優）。	平均	2
ロックデバイス	LockDeviceは、定義されたAPIを通じて我々のシステムと相互作用する別のシステムである。	シンプル	1
ライトスイッチ	LockDeviceと同じ。	シンプル	1
アラームベル	LockDeviceと同じ。	シンプル	1
データベース	データベースは、プロトコルを介して相互作用する別のシステムである。	平均	2
タイマー	LockDeviceと同じ。	シンプル	1
警察	私たちのシステムは、警察にテキスト通知を送るだけです。	シンプル	1

$$UAW（ホームアクセス）= 5 \times \text{シンプル} + 2 \times \text{アベレージ} + 1 \times \text{コンプレックス} = 5 \times 1 + 2 \times 2 + 1 \times 3 = 12$$

## 未調整ユースケース・ウェイト（UUCW）

UUCWは、単純、平均、複雑の3つのカテゴリーに分類されたユースケースの数から導き出される（表4-3参照）。各ユースケースは、主な成功シナリオと代替シナリオ（拡張）の両方を含むイベントフロー内のステップ（または、トランザクション）の数に基づいて分類される。



シナリオのステップ数は見積もりに影響する。ユースケースシナリオのステップ数が多いと、UUCWが複雑な方に偏り、UCPが増加する。ステップ数が少ないと、UUCWは単純な方に偏り、UCPは減少します。ビジネスプロセスに影響を与えることなく、多くのステップを削減できる場合もあります。

UUCWは、各カテゴリーのユースケースの数を集計し、それぞれの合計に指定された重み付け係数を乗じて、製品を加算することによって算出される。例えば、表4-4はサンプル事例のUUCWを計算したものである。

代替シナリオ（エクステンション）のカウント方法については議論がある。当初は、主要な成功シナリオを除くすべてのシナリオを無視することが提案されていた。現在の見



解は、拡張シナリオはかなりの作業量に相当し、工数見積もりに含める必要があるというものである。しかし、どのように含めるかは合意されていない。問題は、単純に延長シナリオの行数を数え、それをメインの成功シナリオの行数に加えることはできないということである。

表4-3: トランザクション数に基づくユースケースの重み。

ユースケースカテゴリー	ユースケースカテゴリーの認識方法の説明	重量
シンプル	シンプルなユーザーインターフェース。最大1人の参加アクター（プラス開始アクター）。成功シナリオのステップ数: $\leq 3$ 。現在利用可能な場合、そのドメインモデルには3つ以下の概念が含まれます。	5
平均	適度なインターフェースデザイン。参加アクターは2人以上。成功シナリオのステップ数: 4から7。現在利用可能な場合、そのドメインモデルには5から10のコンセプトが含まれる。	10
コンプレックス	複雑なユーザーインターフェースや処理。参加アクターが3人以上成功シナリオのステップ数: $\geq 7$ 。もしあれば、そのドメインモデルには10以上の概念が含まれる。	15

UC-7:AuthenticateUser(セクション2.3)で見たように、各拡張は新しいトランザクションそのものではなく、トランザクションの結果から始まる。例えば、拡張2a("Tenant/Landlord enters an invalid identification key")は、メインサクセスシナリオのステップ2("Tenant/Landlord supplies an identification key")で述べられているトランザクションの結果である。したがって、UC-7: AuthenticateUser の extensions セクションの項目 2a はカウントされない。もちろん、2b、2c、3aについても同様である。UC-7:AuthenticateUserのユースケースのトランザクション数は10である。2b1と2b2を一度だけカウントしたいかもしれないが、それは割に合わない労力である。

ユースケースの複雑さを測定するもう1つのメカニズムは、ドメインモデリングによって得られる概念を数えることである（セクション2.4）。もちろん、これは見積もりを行う時点でドメインモデルがすでに導出されていることを前提とする。どの概念が特定のユースケースをモデル化するかが決まれば、概念をトランザクションの代わりに使用することができる。表4-3に示すように、単純なユースケースは5以下の概念、平均的なユースケースは5～10の概念、複雑なユースケースは10以上の概念によって実装される。重みは前述と同じである。次に、ユースケースの各タイプに重み付け係数を乗じ、そ

の積を合計してUUCWを求める。

UUCWは、各カテゴリーのユースケースを集計し、それぞれのカウントに指定の重み付け係数（表4-3）を乗じ、製品を加算することで算出される：

$$UUCW \text{ (ホームアクセス)} = 1 \times \text{単純} + 5 \times \text{平均} + 2 \times \text{複雑} = 1 \times 5 + 5 \times 10 + 2 \times 15 = 85$$

UUCPは、UAWとUUCWを足して計算される。表4-2と表4-4のスコアに基づき、我々のケーススタディプロジェクトのUUCPは、 $UUCP = UAW + UUCW = 12 + 85 = 97$ となる。

UUCPは、非機能要求（TCF）と環境要因（ECF）を考慮していないため、調整されていないシステム全体のサイズを示す。

**表4-4: 家庭の入退室管理に関する事例研究のユースケース分類（セクション2.3参照）。**

使用例	説明	カテゴリー	重量
アンロック (UC-1)	シンプルなユーザーインターフェース。主な成功のための5つのステップ シナリオ。3つの参加アクタ（LockDevice、LightSwitch、Timer）。	平均	10
ロック (UC-2)	シンプルなユーザーインターフェース。すべてのシナリオで2+3=5ステップ。3つの参加アクター（LockDevice、LightSwitch、Timer）。	平均	10
ManageUsers (UC-3)	複雑なユーザーインターフェース。主なサクセスシナリオのための7以上のステップ（UC-6またはUC-7）。参加アクター2名（テナント、データベース）。	コンプレックス	15
ViewAccessHistory (UC-4)	複雑なユーザーインターフェース。主な成功シナリオのための8つのステップ。参加アクター2名（データベース、家主）。	コンプレックス	15
AuthenticateUser (UC-5)	シンプルなユーザーインターフェース。すべてのシナリオで3+1=4ステップ。参加アクター2名（アラームベル、警察）。	平均	10
AddUser (UC-6)	複雑なユーザーインターフェース。主な成功のための6つのステップ シナリオ（UC-3を除く）。参加アクターは2名（テナント、データベース）。	平均	10
RemoveUser (UC-7)	複雑なユーザーインターフェース。主な成功のための4つのステップ シナリオ（UC-3を除く）。参加アクターは1名（データベース）。	平均	10
ログイン (UC-8)	シンプルなユーザーインターフェース。主な成功シナリオのための2つのステップ。参加アクターなし。	シンプル	5

## 技術的複雑度係数（TCF）-非機能要件

プロジェクトの非機能要件が生産性に与える影響を推定するために、（専門家である開発者が）13の標準的な技術的要因を特定した（表 4-5 参照）。各要因は、相対的な影響度に応じて重み付けされている。

開発チームは、表4-5の各技術的要因の複雑さの認知度を、自分たちのプロジェクトの文脈で評価する必要がある。その評価に基づいて、別の「星の評価」、つまり複雑さの知覚値を0から5の間で割り当てる。複雑さの認識値は、ある非機能要件を満たすためにどれだけの労力が必要かというチームの主観的な認識を反映したものである。例えば、

分散システム（表4-5の要因T1）を開発する場合、1台のコンピュータで動作するシステムを開発する場合よりも、より多くのスキルと時間が必要になる。複雑さの認識値が0であれば、その技術的要因はこのプロジェクトには無関係であり、3であれば平均的な労力、5であれば大きな労力が必要であることを意味する。疑問がある場合は、3を使用する。

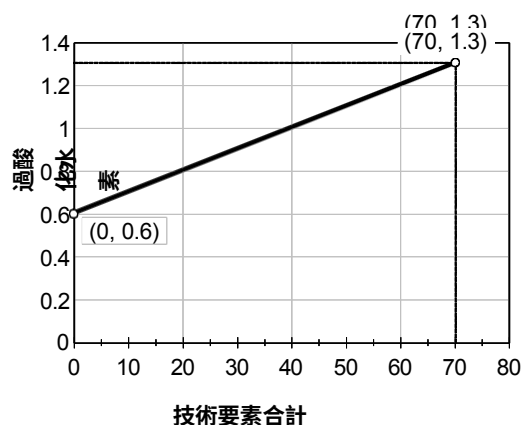
各要因の重み（表4-5）は、その知覚された複雑さの要因を掛け合わされ、計算された要因を生成する。算出されたファクターは合計され、技術的総合ファクターとなる。表4-6は、ケーススタディの技術的複雑さを計算したものである。

TCFを算出するために、2つの定数がTechnical Total Factorとともに使用される。この定数は、TCFがUCP方程式(4.1)に与える影響を、0.6（知覚される複雑さがすべて0の場合）から最大1.3（知覚される複雑さがすべて5の場合）の範囲に制限するものである（図4-2(a)を参照）。

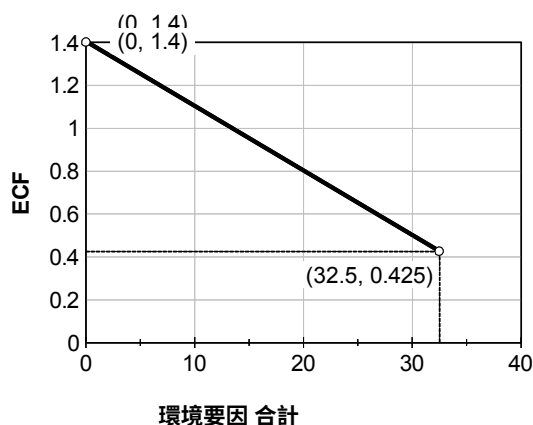
表4-5: 技術的複雑さの要因とその重み。

技術的要因	説明	重量
T1	分散システム（複数のマシン上で動作）	2
T2	パフォーマンス目標（レスポンスタイムとスループットが重要か？）	1 <sup>(*)</sup>
T3	エンドユーザーの効率	1
T4	複雑な内部処理	1
T5	再利用可能なデザインまたはコード	1
T6	インストールが簡単（自動変換とインストールがシステムに含まれているか？）	0.5
T7	簡単な操作（バックアップ、スタートアップ、リカバリなどの操作を含む）	0.5
T8	ポータブル	2
T9	変更が容易（新機能の追加や既存機能の変更）	1
T10	同時使用（複数のユーザー）	1
T11	特別なセキュリティ機能	1
T12	第三者への直接アクセスの提供（システムは異なる組織の複数のサイトから使用される）	1
T13	特別なユーザートレーニング施設が必要	1

(\*)いくつかの情報源では、パフォーマンス目標ファクター（T2）の重みとして2を割り当てている。



(a)



(b)

図4-2: 技術要因と環境要因のスケーリング定数。

TCFの値が1より小さいと、正の値に正の分数を乗じた値の大きさが小さくなるため、UCPは減少する： $100 \times 0.6 = 60$ （40%の減少）。TCFの値が1より大きいと、UCPは増加する。なぜなら、正の値に正の混合数を掛けると、 $100 \times 1.3 = 130$ （30%増加）となるからである。定数は、経験豊富な開発者へのインタビューによって、彼らの主観的な推定に基づいて決定された。

定数はTCFを0.6から1.3の範囲に制限しているため、TCFはUCP方程式に40%(0.6)から最大+30%(1.3)の影響を与える可能性がある。TCFの計算式は以下の通りである：

$$TCF = \text{定数-1} + \text{定数-2} \times \text{技術要素合計} = C_1 + c_2 \cdot \sum_{i=1} W_i \cdot F_i \quad (4.2)$$

どこだ？

表4-6: 家庭内アクセスのケーススタディ（2.3節参照）の技術的複雑さの要因。

技術的要因	説明	重量	知覚される複雑さ	計算係数 (重量×知覚される複雑さ)
T1	ViewAccessHistory（UC-4）のため、分散されたWebベースのシステム。	2	3	$2 \times 3 = 6$
T2	ユーザーは良いパフォーマンスを期待しているが 格別なものはない	1	3	$1 \times 3 = 3$
T3	エンドユーザーは効率を期待しているが、特別な要求はない	1	3	$1 \times 3 = 3$
T4	内部処理は比較的簡単	1	1	$1 \times 1 = 1$
T5	再利用性は要求されない	1	0	$1 \times 0 = 0$
T6	設置のしやすさは中程度に重要（おそらく技術者が設置するだろう）	0.5	3	$0.5 \times 3 = 1.5$
T7	使いやすさは非常に重要	0.5	5	$0.5 \times 5 = 2.5$
T8	データベース・ベンダーの選択肢を広げておきたいという願望以外に、移植性に関する懸念はない	2	2	$2 \times 2 = 4$
T9	必要最小限の交換が容易	1	1	$1 \times 1 = 1$
T10	同時使用が必要（5.3項）	1	4	$1 \times 4 = 4$
T11	セキュリティは重要な懸念事項である	1	5	$1 \times 5 = 5$
T12	第三者からの直接アクセス不可	1	0	$1 \times 0 = 0$
T13	独自のトレーニングは必要ない	1	0	$1 \times 0 = 0$
テクニカル・ファクター合計:				31

定数-1 ( $C_1$ ) = 0.6 定

数-2 ( $C_2$ ) = 0.01

$W_i = i$  の重量<sup>th</sup> テクニカル・ファクター（表4-5）

$F_i = i$  の知覚された複雑さ<sup>th</sup> 技術的要因（表4-6）

式(4.2)を図4-2(a)に示す。表4-6のデータから、 $TCF=0.6+(0.01 \times 31)=0.91$ 。式(4.1)によれば、UCPは9%減少する。

## 環境複雑性因子（ECF）

環境要因（表4-7）は、プロジェクトに参加する人々の経験レベルとプロジェクトの安定性を測るものである。経験が多ければ多いほどUCPの数は減り、経験が少なければ少ないほどUCPの数は増えます。利用可能な予算、会社の市場での地位、経済状況など、その他の外部要因も考慮するとよいでしょう。

開発チームは、各要因がプロジェクトの成功に与える影響を、自分たちの認識に基づい



て判断する。値 1 は、その要因がプロジェクトに強いマイナスの影響を与えることを意味し、3 は平均的、5 は強いプラスの影響を与えることを意味する。0 の場合は、プロジェクトの成功にまったく影響しません。E1～E4については、0はそのテーマに関する経験なし、3は平均、5はエキスパートを意味する。E5については、0はプロジェクトに対する意欲がないこと、3は平均的であること、5は意欲が高いことを意味する。E6では、0は変化しない要件、3は平均的な変化量、5は極めて不安定な要件を意味する。E7において、0はパートタイムの技術スタッフがいらないこと、3はチームの平均的な半数がパートタイムであること、5はチームの全員がパートタイムであることを意味する。

表4-7: 環境複雑性要因とその重み。

環境要因	説明	重量
E1	開発プロセスに精通している (UMLベースなど)	1.5
E2	アプリケーション問題の経験	0.5
E3	パラダイムの経験 (オブジェクト指向アプローチなど)	1
E4	リードアナリスト能力	0.5
E5	モチベーション	1
E6	安定した要件	2
E7	パートタイムスタッフ	-1
E8	難しいプログラミング言語	-1

のパートタイマーである。E8について、0は使いやすいプログラミング言語、3は平均的な難易度の言語、5は非常に難しい言語がプロジェクトで使用される予定であることを意味する。

各要因の加重にその影響度を掛け合わせ、算出された要因を算出する。算出されたファクターは合計され、環境ファクター合計が算出されます。環境要因合計の値が大きいほど、UCP方程式への影響は大きくなります。表4-8は、ケーススタディプロジェクト（家庭へのアクセス制御）の環境要因を計算したものである。

最終的なECFを作成するために、環境因子合計で2つの定数が計算される。上記のTCF定数と同様に、これらの定数も専門家へのインタビューに基づいて決定された。この定数は、ECFがUCP方程式に与える影響を以下のように制限する。

0.425（パートタイム労働者と難しいプログラミング言語=0、その他のすべての値=5）から1.4（知覚される影響はすべて0）。したがって、ECFはUCPを57.5%減少させ、UCPを40%増加させることができる（図4-2(b)参照）。ECFは、TCFよりもUCP数に大きな潜在的影響を与えます。計算式は

$$ECF = \text{定数-1} + \text{定数-2} \times \text{環境因子 合計} = C_1 + C_2 \cdot \sum_{i=1}^8 W_i \cdot F_i \quad (4.3)$$

どこだ?

定数-1 ( $C_1$ ) = 1.4 定数

-2 ( $C_2$ ) = 0.03

$W_i = i^{\text{th}}$  環境要因の重み (表4-7)

$F_i = i^{\text{th}}$  環境要因の知覚された影響 (表 4-8)

式(4.3)を図4-2(b)に示す。表4-8のデータを考慮すると、 $ECF = 1.4 + (0.03 \times 11) = 1.07$ とな

る。サンプル事例では、チームのソフトウェア開発経験が控えめであったため、平均的な EFT となった。4つの要因E1～E4はすべて比較的低いスコアであった。式(4.1)によると、この結果、UCPは7%増加した。

表4-8: 家庭内アクセスのケーススタディ（2.3節）の環境複雑性要因。

環境要因	説明	重量	認知されたインパクト	計算係数 (重量×インパクト)
E1	UMLベースの開発に慣れている。	1.5	1	$1.5 \times 1 = 1.5$
E2	アプリケーションに精通している問題	0.5	2	$0.5 \times 2 = 1$
E3	オブジェクト指向の知識	1	2	$1 \times 2 = 2$
E4	初級リードアナリスト	0.5	1	$0.5 \times 1 = 0.5$
E5	モチベーションは高いが、たまにサボるメンバーもいる	1	4	$1 \times 4 = 4$
E6	安定した要件が期待される	2	5	$2 \times 5 = 5$
E7	パートタイムのスタッフは関与しない	-1	0	$-1 \times 0 = 0$
E8	平均的な難易度のプログラミング言語を使用	-1	3	$-1 \times 3 = -3$
環境要因合計:				11

## ユースケース・ポイント（UCP）の計算

念のため、UCPの式（4.1）をここにコピーしておく：

$$ucp = uucp \times tcf \times ecf$$

上記の計算から、UCP変数は以下の値を持つ：UUCP = 97

TCF = 0.91

ECF = 1.07

サンプル・ケーススタディの場合、最終的なUCPは以下のようになる：

$UCP = 97 \times 0.91 \times 1.07 = 94.45$ 、つまり94のユースケース・ポイント。

サンプルのケーススタディでは、TCFとECFの複合効果により、UUCPは約3%上昇した（ $94/97 \times 100 \div 100 = +3\%$ ）。これは微々たる調整であり、計算への他の多くの入力の主観的な見積もりであることを考えれば、無視できる。

## UCP指標の議論

UCPの式(4.1)は、計数が比率尺度であり、調整因子の得点が順序尺度であるため（セクション4.1.1参照）、測定理論に合致していないことに注意してください。しかし、この

ような式は実際にはよく使われます。

UUCW (Unadjusted Use Case Weight: 未調整ユースケースウエイト) は、個々のユースケースの認知されたウエイトを単純に合計することで算出されることに注目すべきです (表4-3)。これは、すべてのユースケースが完全に独立していると仮定していますが、通常はそうではありません。サイズ尺度の線形加算の利点については、1.2.5節と2.2.3節ですでに述べた。

ス大学

UCPは、主観的で一見恣意的なパラメータ、特に重み付け係数に基づくように見える。UCPが広く採用されるようになったのは、その不完全さにもかかわらず、多くの重要な意思決定が必要とされるプロジェクトの初期段階において、貴重な見積もりを提供してくれるからである。UCPに基づく推定の精度に関する実証的証拠に関する文献については、参考文献（セクション4.7）を参照のこと。

UCPは、ソフトウェア・システムが機能的にどの程度の「大きさ」になるかを測るものである。ソフトウェアの規模は、誰がシステムを構築するのか、あるいはシステムが構築される条件にかかわらず同じである。例えば、UCPが100のプロジェクトは、UCPが90のプロジェクトよりも時間がかかるかもしれませんが、どの程度かはわかりません。セクション1.2.5の考察から、式(1.2)を使ってプロジェクトを完了するまでの時間を計算するには、チームの速度を知る必要があることがわかります。チームのベロシティ（生産性）を考慮し、推定工数を計算する方法については、セクション4.6で後述します。

## 4.2.2 サイクロマティック複素数

プログラムの複雑さの最も一般的な領域の1つは、複雑な条件論理（または制御フロー）にある。Thomas McCabe [1974]は、プログラムの条件論理の複雑さを把握するために、サイクロマティック複雑度という尺度を考案した。分岐のないプログラムは最も複雑度が低く、ループのあるプログラムはより複雑で、2つのループが交差するプログラムはさらに複雑である。サイクロマティック複雑度は、プログラムを通るさまざまなパスの数という直感的な考え方にほぼ対応しており、プログラムを通るさまざまなパスの数が多ければ多いほど、複雑度は高くなる。

McCabeのメトリックはグラフ理論に基づいており、プログラム内の線形独立パスの数を数えることによって、 $V(G)$ で示されるグラフ $G$ のサイクロマティック数を計算する。サイクロマティック複雑度は

$$V(G) = e - n + 2 \quad (4.4)$$

ここで、 $e$ はエッジの数、 $n$ はノードの数である。

プログラムをグラフに変換したものが図4-3である。サイクロマティック複雑度は、プログラム中の2進数決定の数に1を足した数に等しい。すべての決定が2進数でない場合、3進数の決定は2進数の決定としてカウントされ、 $n$ 進数のcase（selectまたはswitch）文

は $n \times 1$ 進数の決定としてカウントされる。ループ文の反復テストは、1つのバイナリ判定としてカウントされる。

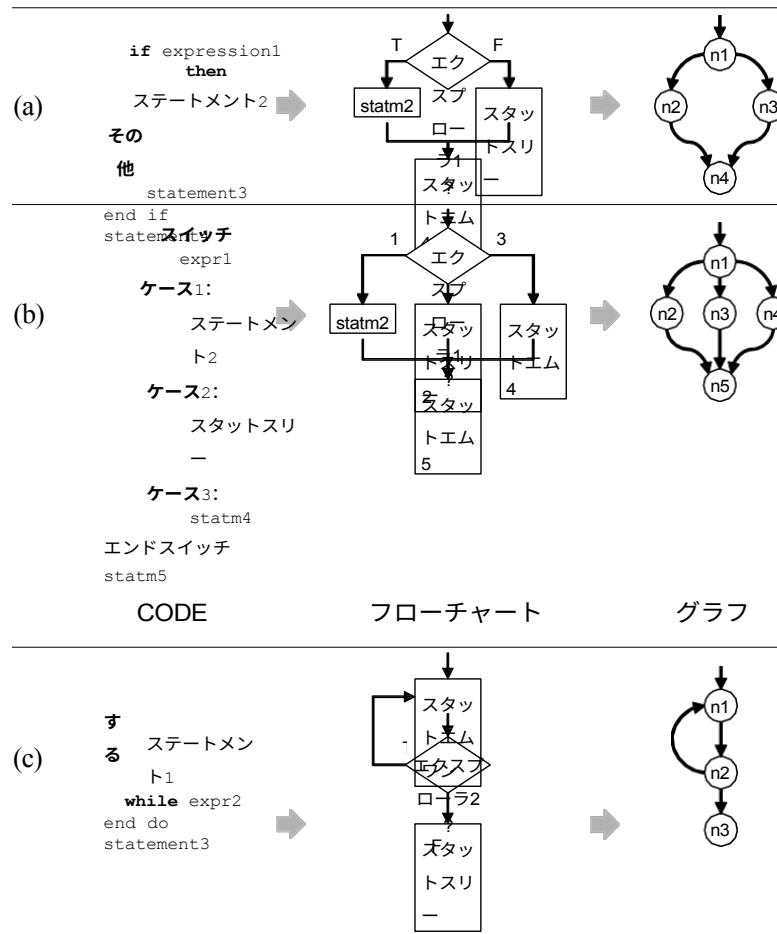


図4-3: ソフトウェア・コードを抽象グラフに変換する。

サイクロマティック複雑度は加法的である。複数のグラフを1つのグループとして考えたときの複雑さは、個々のグラフの複雑さの和に等しい。

グラフのサイクロマティック複雑度  $V(G)$  の計算には、2つの微妙に異なる公式がある。

G. McCabe [1974]によるオリジナルの公式は次の通りである。

$$V(G) = e \oslash n + 2p \quad (4.5)$$

ここで、 $e$ は辺の数、 $n$ はノードの数、 $p$ はグラフの連結成分の数である。あるいは、[Henderson-Sellers & Tegarden, 1994]は、線形に依存しないサイクロマティック複雑度を次のように提案している。

$$V_{LI}(G) = e \oslash n + p + 1 \quad (4.6)$$

サイクロマティック複雑さの指標は決定と分岐に基づいており、設計とプログラミングの論理パターンに合致しているため、ソフトウェアの専門家には魅力的である。しかし、欠点がないわけではない。サイクロマティック複雑度は逐次文の複雑さを無視してい



る。つまり、条件分岐のないプログラムは、サイクロマティック複雑度がゼロである！  
また、ループとIF-THEN-ELSE文、選択文とネストされたIF-THEN-ELSE文など、異なる種類の制御フローの複雑さを区別していない。

サイクロマティック複雑度メトリクスはもともと、プログラムのテスト可能性と理解可能性を示すために設計された。また、プログラム中のすべての実行可能なステートメントを実行するために実行しなければならない、ユニークなテストの最小数を決定することもできます。サイクロマティック複雑度が高いプログラムほど、その複雑さゆえにテストや保守が難しくなることが予想されます。優れたテスト容易性と保守性を実現するために、McCabeはどのプログラム・モジュールも環状複雑度が10を超えないようにすることを推奨している。多くのソフトウェア

ス大学

リファクタリングは、プログラムの条件ロジックの複雑さを軽減することを目的としている [Fowler, 2000; Kerievsky, 2005]。

## 4.3 モジュールの凝集性の測定

---

凝集性は、ソフトウェアユニットの機能における関連性または一貫性の尺度として定義される。これは、そのユニット内のパーツがどの程度一緒に属しているか、または互いに関連しているかを識別する属性である。オブジェクト指向のパラダイムでは、クラスがユニット、データが属性、メソッドが部品となる。凝集性の高いモジュールは、通常、堅牢で信頼性が高く、再利用可能で理解しやすい。一方、凝集性の低いモジュールは、理解、テスト、保守、再利用が難しいなど、好ましくない特徴を持つ。凝集性は順序型の測定で、通常 "凝集性が高い" か "凝集性が低い" かで表される。

第2章では、モジュール・レベルであれクラス・レベルであれ、設計の各単位は単一の目的に焦点を絞るべきだと主張した。つまり、論理的に関連する責務はごくわずかであるべきなのです。モジュール内の機能的関連性」や「モジュールの強度」といった用語が、設計の凝集性の概念を扱うために使われてきた。

### 4.3.1 内部凝集性または構文的凝集性

内部的な結合は、個々のモジュールのコードを調べることによって評価される構文的な結合として理解するのが最も適切である。したがって、大規模なプログラムをモジュール化する方法と密接な関係がある。モジュール化は、さまざまな理由と方法で達成することができる。

非常に粗雑なモジュール化は、各モジュールが一定のサイズ、例えば50行のコードを超えないように要求することである。これはプログラムを任意に約50行ずつのブロックに量子化することになる。あるいは、設計の各単位が一定の規定サイズを持つことを要求することもできる。例えば、パッケージにはある数のクラスが必要であるとか、各クラスにはある数の属性と操作が必要であるといった具合です。設計の単位やコードが、無

関係なタスクを実行することになるかもしれません。ここでの結束は、偶発的な*結束*、あるいは*偶然の結束*であろう。

通常、初期設計では偶然の一致は起こりません。しかし、要求事項の変更やバグの修正などによって設計が何度も変更・修正され、スケジュールに追われるようになると、元の設計が偶然の一致に進化することがあります。元の設計にパッチを当てて新たな要件を満たすようにしたり、再出発の代わりに関連する設計を採用して修正したりすることもある。このような場合、1つの設計単位に無関係な要素が複数存在することになりやすい。

## 結束力測定のための順序尺度

より合理的な設計は、モジュールの内容が互いに何らかの関係を持つようにすることだ。各モジュールの内容に対して、異なる関係を作り出すことができる。異なるタイプのモジュールの凝集性を特定することで、凝集性測定のための名義尺度を作成することができる。より強力な尺度は順序尺度で、専門家に異なるタイプのモジュールの結束の質を主観的に評価してもらい、順位を作成することで作成できます。以下は、凝集性測定のための順序尺度の例です：

順位	凝集タイプ	品質
6	機能的結束	グッド ↓ 悪い
5	連続的な結束	
4	コミュニケーションの結束	
3	手続きの結束	
2	時間的結束	
1	論理的結束	
0	偶然の結束	

**機能的結合**は、設計ユニット（モジュール）が単一の明確に定義された機能を実行する、または単一の目標を達成するため、最も緊密な関係を提供すると判断されます。

設計ユニットが複数の機能を実行するが、これらの機能は仕様で規定された順序で発生する、つまり、強く関連しているため、**逐次的な結束**はやや弱いと判断される。

設計ユニットが複数の機能を実行するが、そのすべてが同じデータまたは同じデータセットを対象としている場合、**通信の凝集性**が存在する。しかし、データはオブジェクト指向の方法で単一の型や構造として編成されるわけではない。

**手続き的な結合**は、設計ユニットが手続き的に関連する複数の機能を実行するときに存在する。各モジュールのコードは、アクティビティの制御シーケンスを定義する1つの機能を表します。

**時間的凝集性**（*Temporal Cohesion*）とは、1つの設計ユニットが複数の機能を実行し、それらが同じ時間内に実行されなければならないという事実によってのみ関連している場合です。例えば、すべてのデータ初期化を1つのユニットにまとめ、他の設計ユニットで定義され利用される可能性があるにもかかわらず、すべての初期化を同時に実行する設計が挙げられます。

**論理的なまとまり**は、一連の類似した機能を果たすデザイン・ユニットの特徴である。

一見すると、論理的結束は要素が関連しているという点で理にかなっているように見える。しかし、その関係は実のところ非常に弱い。例えば、Javaの`java.lang.Math`クラスは、指数、対数、平方根、三角関数のような基本的な数値演算を行うメソッドを含んでいます。このクラスのすべてのメソッドは、数学的演算を実行するという点では論理的に関連していますが、互いにまったく独立しています。

理想的には、オブジェクト指向の設計単位（クラス）は、上位2つのタイプの結合性（機能的または逐次的）を示すべきです。

この結束力測定法の重大な問題は、高レベルの結束力を達成するためのモジュールの成功が、純粹に人間の評価に依存していることである。

ス大学

## 凝集力測定のための間隔尺度

私たちは主に、クラスなどのオブジェクト指向ソフトウェアの結束に興味があります。

**クラスの凝集性は**、クラスのさまざまなメンバ（属性と操作（またはメソッド））間の関連性をキャプチャします。クラスの凝集性のメトリクスは、大きく2つのグループに分類できます：

1. **インターフェイスベースのメトリクス**は、メソッド・シグネチャの情報からクラスの凝集性を計算する
2. **コード・ベースのメトリクス**は、メソッドによる属性アクセスの観点からクラスの凝集性を計算する。

さらに、コードベースの凝集性メトリクスを、凝集性の定量化方法に基づいて4つのサブタイプに分類することができる：

- 2.a) このダイアグラムでは、「Constraints（制約）」、「Constraints（制約）」、「Constraints（制約）」、「Constraints（制約）」の各要素について説明します。
- 2.b) ペアワイズ接続ベースのメトリクスは、接続された不連続なメソッドペアの数の関数として凝集性を計算します。
- 2.c) 接続の大きさベースのメトリクスは、属性ごとにアクセスメソッドをカウントし、（メソッド間の直接的な属性共有を計算する代わりに）カウントの観点から間接的に属性共有インデックスを見つける。
- 2.d) 分解に基づくメトリクスは、与えられたクラスの再帰的分解の観点から結合度を計算します。分解は、クラスの連結を維持する極めて重要な要素を取り除くことによって生成される。

これらのメトリクスは、クラス要素の操作を使用して、クラスの凝集性を計算します。クラス  $C$  の主要な要素は、その  $a$  属性  $A_1, \dots, A_a$ 、 $m$  メソッド  $M_1, \dots, M_m$ 、およびメソッド  $P_1, \dots, P_m$  の  $p$  パラメータ（または引数）型のリストです。以下のセクションでは、クラスの凝集性を計算するためのさまざまなアプローチについて説明します。

既存のメトリクスの多くは、クラスを「凝集性」または「非凝集性」のどちらかに分類し、凝集性の強さの違いを捕捉しません。しかし、このアプローチでは、2つの凝集性

のあるクラスとないクラスを比較したり、コードの修正によって凝集性の程度が高まったか低下したかを知ることは難しい。もし2つの異なるバージョンのソフトウェアの凝集性を比較したいのであれば、モジュールが凝集しているかしていないだけでなく、その凝集性の程度も計算できる指標を使う必要がある。私たちのソフトウェアの両方のバージョンが凝集性があると仮定すると、どちらのバージョンがよりよく設計され、より凝集性が高いかを判断することができる。

### 4.3.2 インターフェイス・ベースの結束指標

インターフェイス・ベースの凝集性メトリクスは、分析および設計フェーズの早い段階で、クラスのメソッド間の凝集性を評価するのに役立つ設計メトリックです。これらのメトリックは、メソッドのパラメータのリストを使用して、クラスのインターフェイス内のメソッドの一貫性を評価します。メソッド・プロトタイプ（メソッド・タイプとパラメーター・タイプ）のみを含み、クラス実装コードを必要としないクラス宣言に適用できます。そのようなメトリックのひとつが、CAMC（**Cohesion Among Methods of Classes**）です。CAMC メトリックは、メソッドのパラメータが、そのメソッドが実装する可能性のある相互作用のタイプを合理的に定義するという仮定に基づいています。

図4-4

		シリアルポ ーストリング パラメータタイプ	
		0	1
方法	デバイスCtrl	1	0
	アクティブ	0	1
	非アクティ	0	1
	ブ化	0	1
ゲットステ		0	1
ータス		(b)	

デバイスCtrl

# devStatuses\_ : ベクトル

+ activate(dev : String) : boolean

+ 無効化(dev :String) : boolean

+ getStatus(dev : String) : オブジェ  
クト

+ getStatus(dev : String) : オブジ  
エクト

(a)

図4-4: クラス (a) とそのパラメータ出現行列 (b) 。

CAMC メトリック値を計算するには、クラスのすべてのメソッドのすべてのパラメータの和を決定します。各メソッドのパラメータ・オブジェクト・タイプの集合  $M_i$  も決定されます。 $M_i$  と結合セット  $T$  の交点 (セット  $P_i$ ) が、クラスのすべてのメソッドについて計算されます。すべてのメソッドについて、交差集合 ( $P_i$ ) のサイズと結合集合 ( $T$ ) のサイズの比が計算されます。すべての交差集合  $P_i$  の合計が、メソッド数と結合集合  $T$  のサイズの積で割られ、CAMC メトリックの値が得られます。形式的には、このメトリックは

$$CAMC(C) = \frac{1}{kl} \sum_{i=1}^k \sum_{j=1}^l o_{ij} = \frac{\sigma}{kl} \quad (4.7)$$

### 4.3.3 要素の不連続集合を用いた凝集度測定法

このタイプの初期のメトリクスは、メソッドの凝集性の欠如 (LCOM1) です。このメトリクスは、クラス属性を共有していないメソッドのペアの数をカウントします。これは、各メソッドによって使用されるクラス属性のセットの交差によって形成される不連続なセットがいくつあるかを考慮します。LCOM1の下では、すべてのメソッドがすべての属性にアクセスするとき、完全な結合が達成されます。反対に、各メソッドが1つの属性にしかアクセスしない場合 ( $m = a$  と仮定)。この場合、我々は  $LCOM = 1$  を期待し、これは極端な凝集性の欠如を示す。



LCOM1の正式な定義を以下に示す。属性の集合  $\{A_j\}$  ( $j = 1, \dots, a_j$ ) にアクセスするメソッドの集合  $\{M_i\}$  ( $i = 1, \dots, m$ ) を考える。各メソッド  $M_i$  がアクセスする属性の数を  $\alpha(M_i)$ 、各属性にアクセスするメソッドの数を  $\mu(A_j)$  とする。すると、クラス  $C_i$  のメソッドの結合の欠如は、形式的に次のように与えられる。

$$LCOM1(C)_i = \frac{m \times \left( \frac{1}{a} \cdot \sum_{j=1}^a \mu(A_j) \right)}{m - 1} \quad (4.8)$$

LCOMのこのバージョンは、LCOM2、LCOM3、LCOM4という後続のバリエーションを可能にするため、LCOM1とラベル付けされている。クラスの凝集性（LCOM3）は、連結された

ス大学

コンポーネントをグラフに追加します。LCOM2 は、クラス属性を共有するメソッド・ペアと共有しないメソッド・ペアの数の差を計算します。LCOM2 は、Pairwise Connection-Based メトリクスに分類されます（セクション 4.3.4）。LCOM メトリクスの参考文献については、参考文献（セクション 4.7）を参照してください。

### 4.3.4 意味的結束

凝集性またはモジュールの「強さ」は、システムの抽象化レベルで見たモジュールレベルの「一体感」の概念を指す。したがって、ある意味ではシステム設計の概念とみなすこともできるが、より適切には、凝集性をモジュールの外部で評価されるモジュールの意味論的関心とみなすことができる。

意味的凝集性は、モジュール（オブジェクト指向のアプローチではクラス）によって表現される抽象化が、意味的に「全体」であると考えられるかどうかを評価する、外部から識別可能な概念である。意味的複雑度メトリクスは、個々のクラスが、完全であり、かつ首尾一貫しているという意味で、本当に抽象データ型であるかどうかを評価します。つまり、意味的に首尾一貫しているためには、クラスは、その責任を持つクラスが持っている期待されるものすべてを含んでいなければなりません、それ以上のものは含んでいません。

内部的、構文的なまとまりは強いが、意味的なまとまりはほとんどないクラスを持つことは可能である。個別に意味的に凝集性の高いクラスが統合されると、内部的な構文的凝集性を保持したまま、外部的には意味的に無意味なクラスができることがあります。たとえば、人とその人が所有する車の両方の特徴を含むクラスがあるとします。各人は1台の車しか所有できず、各車は1人の人にしか所有できない（1対1の関連）とします。そうすると、 $person\_id \leftrightarrow car\_id$  となり、データの正規化と等価になる。しかし、クラスはデータだけでなく、様々なアクションを実行するためのオペレーションも持っている。これらは、(1)人の側面と、(2)提案するクラスの車の側面の行動パターンを提供する。PERSONとCARの間に交差する動作がないと仮定すると、CAR\_PERSONと名づけられたクラスはどのような意味を持つのだろうか？このようなクラスは、内部的には非常にまとまりのあるものである可能性がありますが、外部から見たクラス全体としての意味的には、表現された概念（ここでは人-車として知られているもの）は無意味です。

## 4.4 カップリング

---

カップリング・メトリクスは、異なるモジュールが互いにどの程度相互依存しているかを示す尺度です。高カップリングは、あるモジュールが他のモジュールの内部動作を変更したり依存したりする場合に発生します。このため、このようなモジュール間の相互作用が、あるモジュールとあるモジュールとの間で、どのように相互作用しているかが重要になる。結合は凝集と対比される。凝集性と結合性はともに序数的な尺度であり、"高い"か"低い"かで定義される。低結合と高結合を達成することが最も望ましい。

密結合と疎結合の比較

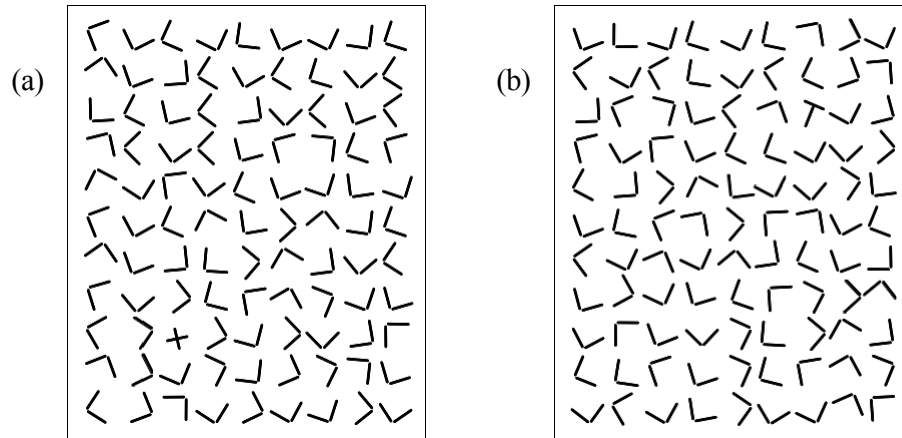


図4-5: ランダム配列による情報の複雑さと深さの比較。詳細は本文を参照。

## 4.5 心理的複雑性

「そして、何が観測できるかは理論によって決まると説明した。彼は言った。"何が観測できるかを知るには、まず理論を知らなければならない、あるいは理論を作らなければならない。"-アインシュタインと初めて会ったときのハイゼンベルクの回想。

「私は長い間、自分の結果を手にしてきた。  
-カール・フリードリヒ・ガウス

ソフトウェアの複雑さ測定におけるフラストレーションのひとつは、物理的な物体を秤に載せて重さを測るのとは異なり、ソフトウェアの物体を「複雑さ秤」に載せてその量を読み取ることができないことである。複雑さは、人の健康状態のように解釈される尺度のようで、"平均的なケース"として述べられなければならない。医師はあなたの血圧を正確に測ることができるが、特定の数値が必ずしも健康の良し悪しに対応するわけではない。医師は、あなたの健康状態について評価を下す前に、心拍数、体温、そしておそらく他のいくつかのパラメータも測定します。たとえそうであっても、その評価は最善の推測であり、単に生理学的測定値の平均的なこのような組み合わせが特定の健康状態に対応すると述べるに過ぎない。おそらく、ソフトウェア・オブジェクトの複雑さも同じように定義すべきなのだろう。直接測定可能な変数の集合に基づく統計的推論として。

### 4.5.1 アルゴリズム情報コンテンツ

私たちの抽象化はどうしても近似的にならざるを得ないことはすでに述べた。よく「粗視化」という言葉が使われるが、これは世界像の細部をぼかして、目の前の問題に関連すると思われる現象だけを抽出することを意味する。したがって、複雑さを定義する際には、システムを記述する詳細レベルを特定する必要がある。

プログラムやシステムの複雑さを定義する一つの方法は、その記述、つまり記述の長さである。前述したように、複雑さの尺度としてサイズの尺度を使うことの利点については、次のように述べた。

ス大学

を測定する必要がある。上記の問題点には、サイズの測り方が異なる可能性があること、プログラムコード（またはそれに関する他の正確な記述）が書かれた言語に依存すること、プログラムの記述が不必要に詰め込まれて複雑に見える可能性があること、などがある。解決策としては、言語の問題を無視し、記述の長さで複雑さを定義することである。

2人の人間が、離れた場所でシステムの説明を伝えたいとする。両者があらかじめ共有している（と知っている）言語、知識、理解を用いていると仮定する。システムの粗い複雑さは、一方の当事者が、ある粗い粒度で、遠方の当事者にシステムを説明するために必要な最短メッセージの長さとして定義することができる。

このような尺度としては、1960年代にアンドレイ・N・コルモゴロフ、グレゴリー・チャイティン、レイ・ソロモノフによって独自に導入されたアルゴリズム情報量と呼ばれるものがよく知られている。無限の記憶容量を持つ理想化された汎用コンピュータを仮定する。aaaaabbbbbbb "のような特定のメッセージ文字列を考える。我々が知りたいのは、その文字列をプリントアウトして計算を停止する、可能な限り最短のプログラムは何か、ということである。アルゴリズム情報量（AIC）は、与えられた文字列をプリントアウトする最短のプログラムの長さとして定義される。例の文字列の場合、プログラムは次のようになります：これは「'a'を5回、'b'を10回印刷する」という意味である。

## 情報理論

## 論理的深さと暗号性

「...長い手紙を書くことは、時間の損失を招くよりはましだと思う..."

-キケロ-キケロ "この手紙がとても長いことをお詫びします。短くする時間がなかった。-

マーク・トウェイン

「もっと時間があれば、もっと短い手紙を書いたのですが.....」。

-キケロ、パスカル、ヴォルテール、マーク・トウェイン、ジョージ・バーナード・ショー、T.S.エリオットなど、さまざまな人物によるとされている。

「信頼性の代償は、究極のシンプルさの追求である。それは、大金持ちには難しい代償である。

を支払う。"-C.A.R.ホア

アルゴリズムック・インフォメーション・コンテンツ（AIC）が、日常的な複雑さの概念と正確には一致しないことはすでに述べた。しかし、他にも考慮すべき点がある。次のような記述を考えてみよう："ランダムな文字Lの配列の中の文字X"。すると、"letter T in a random array of letters L"という記述もほぼ同じAICを持つはずである。図4-5は、私の大好きなベラ・ジュレシュ先生が開拓した方法で、両方の記述を描いたものである。両方の配列を見た場合、図4-5(a)ではXにすぐに気づくことができるが、図4-5(b)をスキャンしてTを検出するにはかなりの時間を費やすことになるに違いない！人間の視覚システムが、図4-5(a)のパターンを認識するために特別なメカニズムを開発したのに、図4-5(b)のパターンではそれができなかったと考える理由はない。より可能性が高いのは、どちらの場合も同じ一般的なパターン認識メカニズムが働いているが、図4-5(b)での成功率ははるかに低いということである。したがって、AICの概念には何かが欠けているようだ。

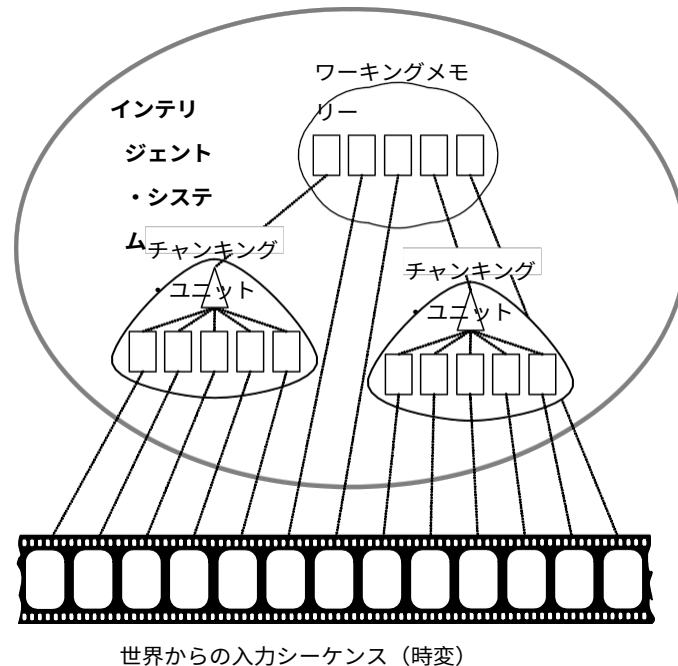


図4-6: 限られたワーキングメモリーのモデル。

一見複雑な記述が低いAICを持つ。解決策は、計算時間を含めることである。

チャールズ・ベネットは、記述の論理的な深さを定義し、その記述を印刷できる最短のプログラムから、システムの実際の記述に至る難易度を特徴付けた。文字列を出力する最短のプログラムだけでなく、同じ効果を持つ短いプログラムの集合を考える。これらのプログラムそれぞれについて、文字列を計算して印刷するのに必要な時間の長さ(またはステップ数)を求める。最後に、実行時間を平均して、より短いプログラムがより重視されるようにする。

## 4.6 努力の見積もり

「後発のソフトウェア・プロジェクトに人手を加えると、後発になる」。  
-フレデリック・P・ブルックスJr.

「不注意に計画されたプロジェクトは、2倍の時間しかかからない」。  
-ゴルブによるコンピュータ社会の法則

「最初の90パーセントのタスクには10パーセントの時間がかかり、最後の10パーセントには残りの90パーセントの時間がか



かる。

パーセント"-プロジェクト・スケジュールの九割九分ルール

## 4.6.1 ユースケースポイントからプロジェクト期間を導き出す

ユースケースポイント (*UCP*) は、ソフトウェア規模の尺度である (セクション4.2.1)。セクション1.2.5で示した式(1.2)を使って、プロジェクト期間を導き出すことができます。このためには、チームの*ベロシティ*を知る必要があります。*ベロシティ*は、ユースケースの進捗率 (または、チームの生産性) を表します。以下に、式(1.2)と等価な式を示しますが、生産性係数(*PF*)を使用します:

$$\text{期間} = \text{UCP} \times \text{PF} \quad (4.9)$$

生産性係数 (Productivity Factor) とは、ユースケース1点あたりに必要な開発工数の割合である。過去のプロジェクトから収集した経験と統計データが、最初の*PF*を見積もるためのデータとなります。例えば、*UCP*が112の過去のプロジェクトで、完了までに2,550時間かかったとすると、2,550を112で割ると、ユースケース1ポイントあたりの*PF*は23人時間となります。

過去のデータが収集されていない場合、開発者はこれらのオプションのいずれかを検討することができる:

1. チームが過去に完了したプロジェクトの*UCP*を計算し、ベースラインを確立する (それが可能な場合)。
2. *PF*の値は、開発チームの全体的な経験や過去の実績 (通常、期限内に終了するか、予算内で終了するか、など) に応じて、15~30の間で設定してください。学部生のような初心者のチームの場合、最初のプロジェクトでは最も高い値 (すなわち30) を使用してください。

異なるアプローチは、Schneider and Winters [2001]によって提案された。環境要因 (表 4-7) は、プロジェクトに参加する人々の経験レベルとプロジェクトの安定性を測定するものであることを思い出してください。この領域でマイナスがあれば、人材育成や (要求事項の) 不安定性に起因する問題の解決に時間を費やす必要があることを意味する。マイナスが多ければ多いほど、問題解決や人材育成に多くの時間を費やすことになり、プロジェクトに割ける時間が少なくなる。

SchneiderとWintersは、E1からE6（表4-8）のうち、知覚される影響が3未満の環境要因の数と、E7とE8のうち影響が3以上の環境要因の数を数えることを提案した。合計が3または4の場合、ユースケース1点あたり28時間とする。合計が4より大きい場合は、プロジェクトに不利な環境要因が多すぎることを示す。いくつかの環境要因が改善されるまで、プロジェクトは保留されるべきである。

生産性係数を見積もるための最良の解決策は、過去のプロジェクトから組織独自の過去の平均値を算出することであろう。これが、将来のプロジェクトの工数見積りを改善するために、過去のデータを収集することが重要な理由です。プロジェクトが完了したら、プロジェクト完了までにかかった実際の時間数をUCPの数値で割ります。その結果が、今後のプロジェクトで使用できる新しいPFとなります。

期間を暦年で見積もる場合は、理想的な労働条件を想定しないことが重要である。見積もりには、電子メールへの対応や会議への出席など、会社のオーバーヘッドを考慮する必要があります。過去の経験から、ユースケースポイントあたりのPFは23人時で、現在のプロジェクトは94ユースケースポイントであるとします（セクション4.2.1で決定）。式(4.9)

ス大学

の場合、 $94 \times 23 = 2162$ 人時である。もちろん、これは $2162 / 24 \times 90$ 日でプロジェクトが完了することを意味するものではない！ 合理的な仮定は、各開発者が週に約30時間をプロジェクトのタスクに費やし、残りの時間は会社のオーバーヘッドに取られるということです。開発者4人のチームであれば、週に $4 \times 30 = 120$ 時間を費やすことになる。2162人の作業時間を週120時間で割ると、このプロジェクトを完了するのに約18週間かかることになる。

## 4.7 要約と参考文献

---

この章では、2種類のソフトウェア測定について説明した。1つは、プロジェクトの初期段階で利用可能な希少な成果物、例えば、計画されたシステムに対する顧客の要件定義書を対象とするものである。これらの測定には主観的な要素が大きく、主に推測と過去の類似プロジェクトの経験に基づいて行われる。この種の測定の目的は、プロジェクトの期間と作業コストを見積もることであり、プロジェクトのスポンサーである顧客との契約条件を交渉することである。

もう1つのソフトウェア測定は、UML設計やソースコードなど、実際のソフトウェア成果物を使用します。これは、ソフトウェアの本質的な特性を測定し、開発者の主観的な推測を避けることを目的としている。測定する成果物がすでに完成しているか、ほぼ完成した状態で存在する必要があるため、プロジェクトの初期段階では適用できない。この種の測定の目的は、製品の品質を評価することである。製品が配備可能かどうかのテストとして、あるいは、対処すべき潜在的な弱点について開発チームにフィードバックを提供することができる。

初期のプロジェクト工数の見積もりは、マネージャ、開発者、テスト担当者がプロジェクトに必要なリソースの計画を立てるのに役立ちます。ユースケースポイント（UCP）手法は、そのような手法の1つとして登場した。これは、未調整ユースケースポイント（UUCP）によって測定されるソフトウェア固有の特性と、開発者の主観的な見積もりに依存する技術的要因（TCF）と環境要因（ECF）を混合したものです。UCPメソッドは、これらの主観的な要因を方程式変数に定量化し、より正確な推定値を生成するため

に時間をかけて調整することができます。産業界のケーススタディによると、UCP法は、実際の労力の20%以内の早期見積もりが可能である。

## セクション4.2: 何を測定するか?

[Henderson-Sellers,1996]は、出版日までのソフトウェア・メトリクスの凝縮されたレビューを提供している。技術的な内容で、構造的な複雑さのメトリクスに焦点を当てています。

Horst Zuse, History of Software Measurement, Technische Universität Berlin, オンライン:  
<http://irb.cs.tu-berlin.de/~zuse/sme.html>

[Halstead, 1977]は、ソフトウェア科学をコンピュータ科学と区別している。ソフトウェア科学の前提は、あらゆるプログラミングタスクは、有限個のプログラム「トークン」を選択し、配置することで構成されるということである。トークンとは、コンパイラが区別できる基本的な構文単位であり、演算子とオペランドである。彼はこれらのトークンに基づいて、いくつかのソフトウェア・メトリクスを定義した。しかし、ソフトウェア科学

ス大学

は、その導入以来論争を巻き起こし、さまざまな方面から批判されてきた。Halsteadの研究は、主にソフトウェア計測において歴史的な重要性を持っている。なぜなら、コンピュータ科学者の間でメトリクス研究が問題になるきっかけとなったからである。

ユースケース・ポイント（UCP）は、Gustav Karner [1993]によって最初に記述されましたが、このテーマに関する彼の最初の仕事は、Rational Software, Inc.によって厳重に守られています。したがって、Karnerの仕事を説明する主なソースは、[Schneider & Winters, 2001]と[Ribu, 2001]です。UCPは、Allan Albrechtの "Function Point Analysis" [Albrecht, 1979]に触発されました。重み付けされた値と制約定数は、最初はAlbrechtに基づいていましたが、その後、オブジェクト指向アプリケーションを開発するためにIvar Jacobsonによって作成された方法論であるObjectoryの経験に基づいて、Objective Systems, LLCの人々によって修正されました。

ユースケース・ポイントの主な情報源は、[Schneider & Winters, 2001; Ribu, 2001; Cohn, 2005]です。[Kusumoto, et al., 2004]は、与えられたユースケースの合計UCPを自動的に計算するシステムのルールを記述しています。これらのルールは、初心者の人間がプロジェクトのUCPを計算するときに非常に役に立つと思います。

多くの産業事例研究が、UCP手法の見積もり精度を検証した。これらのケーススタディによると、UCP手法は、実際の工数の20%以内の早期見積もりを作成でき、多くの場合、専門家や他の見積もり手法よりも実際の工数に近いことがわかった。Mohagheghiら [2005]は、インクリメンタルな大規模開発プロジェクトのUCP見積もりについて、実際の工数の17%以内であったと述べている。Carroll[2005]は、5年間にわたる200以上のプロジェクトのケーススタディについて述べている。このプロセスを数百の大規模ソフトウェアプロジェクト（平均60人月）に適用した結果、調査したプロジェクトの95%で、実際コストと見積もりコストの乖離が9%未満という見積もり精度を達成した。より高い精度を達成するために、キャロルの見積もり方法には、UCP方程式にリスク係数が含まれている。

### セクション4.3: モジュールの凝集性の測定

凝集力測定のための7段階の凝集力を持つ順序尺度は、YurdonとConstantine[1979]によって提案された。

[Constantine *et al.*, 1974; Eder *et al.*, 1992; Allen & Khoshgoftaar, 1999; Henry & Gotterbarn, 1996; Mitchell & Power, 2005] 。

参照: <http://c2.com/cgi/wiki?CouplingAndCohesion>

B.Henderson-Sellers, L. L. Constantine, and I. M. Graham, "Coupling and cohesion: B. Henderson-Sellers, and I. M. Graham, "Coupling and cohesion: Towards a valid suite of object-oriented metrics," *Object-Oriented Systems*, vol. 3, no.3, 143-158, 1996.

[Joshi & Joshi, 2010; Al Dallal, 2011]は、オブジェクト指向のクラス結合メトリクスの識別力を調査した。

## セクション4.4: カップリング

## 第4.5節 心理的複雑性

[Bennett, 1986; 1987; 1990]は、物理システムの複雑さの定義について議論し、論理的な深さを定義している。

## セクション4.6: 努力の見積もり

---

## 問題点

### 問題 4.1

### 問題 4.2

### 問題4.3

(CYCLOMATIC/MCCABE COMPLEXITY) 以下のクイックソートのソートアルゴリズムを考える:

```
クイックソート( $A, p, r$ )
1   $p < r$ 
2  であれば,  $q \leftarrow \text{PARTITION}(A, p, r)$ 
3       $\text{QUICKSORT}(A, p, q - 1)$ 
4       $\text{QUICKSORT}(A, q + 1, r)$ 
```

ここで、PARTITIONの手順は以下の通りである:

```
パーティション( $A, p, r$ )
1   $x \leftarrow A[r]$ 
2   $i \leftarrow p - 1$ 
3  for  $j \leftarrow p$  to  $r - 1$ 
4      もし  $A[j] \leq x$  なら
5          なら,  $i \leftarrow i + 1$ 
6           $A[i] \leftrightarrow A[j]$ を交換する。
7   $A[i + 1] \leftrightarrow A[r]$ を交換する。
8   $i + 1$ を返す
```



- (a) 上記のアルゴリズムのフローチャートを描け。
- (b) 対応するグラフを描き、ノードを  $n_1, n_2, \dots$  とし、エッジを  $e_1, e_2, \dots$  とする。
- (c) 上記のアルゴリズムのサイクロマティック複雑度を計算せよ。

## 問題 4.4