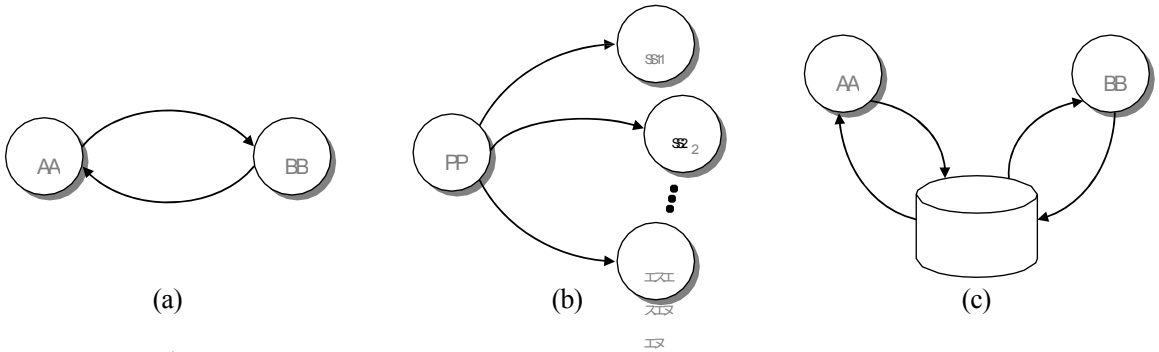


図2-37：オブジェクト通信パターンの例。(a) 1対1の直接メッセージ。(b) 1対多の非ターゲット・メッセージ。(c)共有データ要素経由。

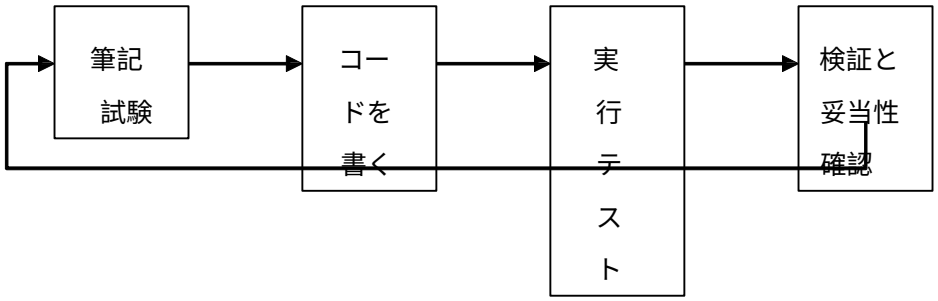


図2-38：テスト駆動実装

オブジェクトが排他的な役割を果たすことはほとんどなく、複数の役割がそれぞれのオブジェクトに程度の差こそあれ付与されているのが普通である。

オブジェクト・コミュニケーション・パターン

通信パターンとは、オブジェクトの集合に課される*メッセージ送信関係である。*どのような関係でもそうであるように、それは1対1であったり1対多であったりし、決定論的であったりランダムであったりする（セクション3.1.1）。これらのパターンのいくつかを図 2-37 に示す。

オブジェクト指向設計、特にデザインパターンについては、第5章でさらに詳しく説明する。

2.6.3ソフトウェア工学が難しい　理由 (3)

もう一つの重要な原因は、ソフトウェア設計の分析手法が不足していることである。ソフトウェア・エンジニアは最適な設計を目指しているが、最適なソフトウェア設計の定

量的基準はほとんど知られていない。最適性の基準は、主に判断と経験に基づいているように見える。

2.7テスト駆動実装

「コンピュータの良いところは、指示通りに動いてくれるところだ。悪いニュースは"あなたの言うとおりにする"-テッド・ネルソン

実装する機能を決めたら、テスト駆動実装は、テスト用のコードを書き、その機能を実装するコードを書き、テストを実行し、最後にテスト結果を検証して妥当性を確認する、という手順で行われます（図2-38）。そうでない場合は、コードをデバッグし、問題を特定して修正し、再度テストする必要があります。

2.7.1 ソフトウェア・テストの概要

"テストはバグの有無ではなく、存在を示す"-エドガー・W・ダイクストラ

テストとは、与えられた入力に対して正しい出力が得られるかどうかを確認するためにプログラムを実行することである。これは、最終成果物であるソフトウェアそのものをテストすることを意味し、ひいてはテスト活動がライフサイクルの後半まで先送りされることを意味する。経験上、ソフトウェアのライフサイクルの初期段階で発生したエラーは、最もコストがかかり、発見が最も困難である。より一般的な定義として、テストとは、UML ダイアグラムやコードなどのソフトウェア成果物の欠陥を発見するプロセスである。**フォールトとは**、「欠陥」や「バグ」とも呼ばれ、システムに*不具合を生じさせる*、*つまり*、望ましくない、あるいは有害でさえあるような振る舞いをする、システムの誤ったハードウェア要素やソフトウェア要素のことである。システムが*故障したのは*、内蔵された欠陥のせいである。

要求仕様、ドメインモデル、設計仕様など、どのようなソフトウェア成果物でもテストすることができる。テスト活動は、できるだけ早い時期に開始すべきである。このアプローチの極端な形が、エクストリーム・プログラミング（XP）のプラクティスの一つである*テスト駆動開発*（TDD）である。テストの形式と厳密さは、テストされる成果物の性質に合わせるべきである。デザインスケッチのテストは、ソフトウェアコードのテストとは異なるアプローチをとることになる。

テストは、さまざまな入力の組み合わせでプログラムを*プローブし*、欠陥を検出することで機能する。したがって、テストは欠陥の存在のみを示すものであり、欠陥の不存在を示すものではない。欠陥がないことを示すには、可能なすべての入力の組み合わせを徹底的に試す必要がある（あるいは、プログラムを通じて可能なすべての経路をたどる必要がある）。可能な組み合わせの数は、一般にソフトウェアのサイズとともに指数関数的に増加する。意図の悪いプログラマーが、個人的な利益や復讐のために、意図的に

悪意のある機能を導入しているかもしれない。したがって、想像しうるすべての入力シーケンスに対してプログラムが正しく動作することをテストすることは不可能である。ブルートフォース・アプローチによるテストに代わる方法は、*推論*（または定理証明）によってソフトウェアの正しさを証明することである。残念ながら、正しさの証明は一般に自動化できず、人間の労力を必要とする。さらに、この方法は、要求が形式的な（数学的な）言語で指定されているプロジェクトにのみ適用できる。このトピックについては、第3章で詳しく説明する。

テストの重要なトレードオフは、可能な限り多くのケースをテストしつつ、経済的コストを抑えることである。われわれの目標は、できるだけ安く、迅速に故障を発見することである。理想的には、各欠陥を発見するための "正しい "テストケースを1つ設計し、それを実行することである。しかし実際には、不具合を発見できない "不成功 "のテストケースを数多く実行しなければならない。コストを抑えるための戦略としては、(*i*) 設計／コードレビュー、推論、静的解析など、他の手法でテストを補完する、(*ii*) 自動化を利用してテストのカバレッジと頻度を増やす、(*iii*) ライフサイクルの早い段階で頻繁にテストを行う、などがある。テスト結果の自動チェックは

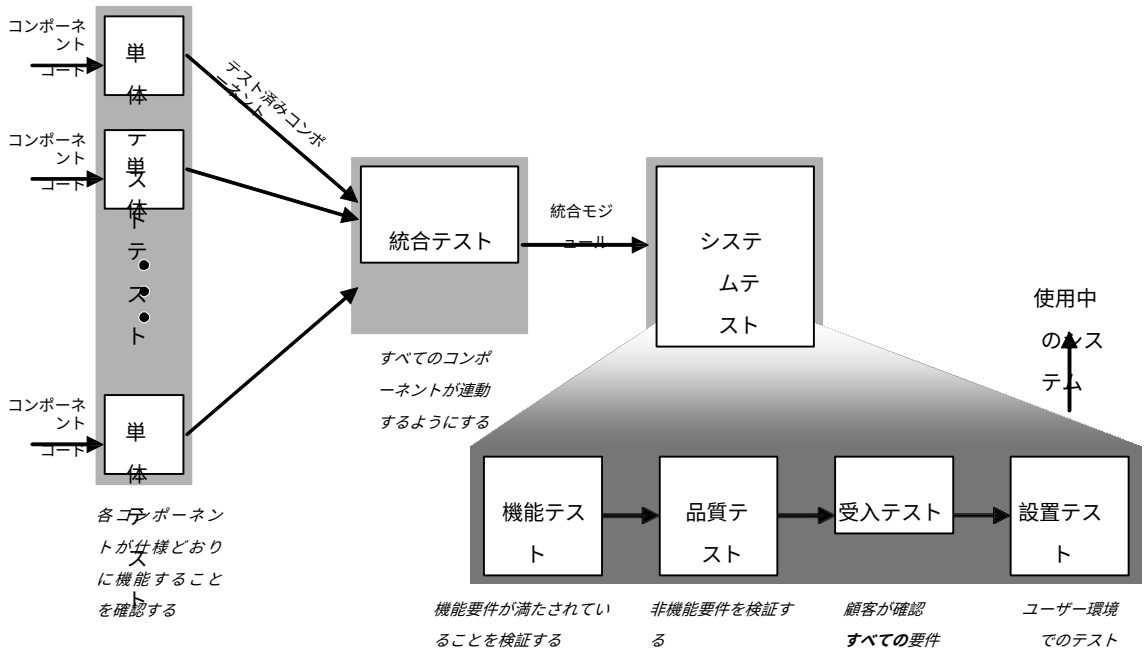


図2-39：ソフトウェアテストの論理構成

コストを低く抑えることが望ましいが、必ずしも実現可能とは限らない。例えば、グラフィカル・ユーザー・インターフェースの表示内容をチェックするにはどうすればよいだろうか。

テストは通常、分析・設計フェーズで設計されたシステムの階層構造（2.3節「ソフトウェアアーキテ クチャ」）によって導かれます（図2-39）。**ユニットテストと呼ばれる、**個々のコンポーネントのテストから始めることもあります。これらのコンポーネントは、段階的にシステムに統合されます。システムコンポーネントの構成をテストすることは、**統合テストと呼ばれる。システムテストは、**システム全体が機能要件および非機能要件に適合していることを保証する。顧客は、システム全体の**受入テストを実施する。**(受入テストとその例については、2.2節と2.4節の要求工学のところで説明する)。いつものように、論理的な構成は、図 2-39 に示すようにテストステップを時間的に順序付けることを意味しない。むしろ、開発ライフサイクルは漸進的かつ反復的に進化し、テストにおいても同様のサイクルが発生する。

単体テストは、オブジェクトの設計モデルと、それに対応する実装との間の差異を発見する。個々のコンポーネントに焦点を当てることには、いくつかの利点がある。ひとつは、分割統治（divide-and-conquer）アプローチに共通する利点で、問題の複雑さを軽減し、システムの小さな部分を個別に扱うことができる。第二に、単体テストでは、関与

するコンポーネントが少ないため、不具合の発見と修正が容易になる。最後に、ユニットテストは分業をサポートするので、複数のチームメンバーが異なるコンポーネントを並行してテストすることができる。ユニットテストの実際的な問題点については、セクション2.7.3で述べる。

リグレッション・テスト（回帰テスト）は、システムに変更が加えられた後に、既存の機能における新たなエラー、つまり「リグレッション」を明らかにしようとするものである。発見された不具合ごとに新しいテストが追加され、コードを変更するたびにテストが実行されます。なぜなら、すべての回帰テストは、あるバージョンのコードで不具合を発見した*後に*追加されるからです。なぜなら、すべての回帰テストは、あるバージョンのコードで不具合を発見した後に追加されるからです。なぜなら、リグレッションテストを追加することになった不具合は、すでに起こったことなので、再びエラーを起こすのは簡単かもしれません。

テストアプローチのもう一つの有用な区別は、テストケースの設計にどのような文書や成果物を使うかである。**ブラックボックス**テストとは、実行中のプログラムをさまざまな入力でプローブして分析することである。これは、実装を見ずに、仕様書からテストデータのみを選択することを含む。このテスト手法は、受入テストなどで顧客によって一般的に使用される。**ホワイトボックステストは、**システムアーキテクチャの知識、使用されているアルゴリズム、あるいはプログラムコードなど、実装に関する知識を持ってテストデータを選択します。このテストアプローチでは、コードが仕様のすべての部分を実装していることを前提としますが、バグ（プログラミングエラー）がある可能性もあります。コードが仕様の一部を省略している場合、そのコードから導かれるホワイトボックステストケースは、仕様の不完全なカバレッジを持つことになります。ホワイトボックステストは、実装の特定の詳細に依存すべきではない。

2.7.2　テスト・カバレッジとコード・カバレッジ

徹底的なテストは現実的に達成できないことが多いため、重要な問題は、いつ十分なテストを行ったかを知ることである。**テスト・カバレッジは、**ソフトウェア・プログラムの仕様やコードがテストによってどの程度実行されたかを測定する。このセクションでは、プログラムの *ソース*コードがどの程度テストされたかを測定する、**コード・カバレッジという**狭い概念に関心を持つ。コード・カバレッジには、等価テスト、境界テスト、制御フロー・テスト、ステート・ベース・テストなど、多くの基準がある。

テスト入力を選択するために、適切な入力値であると「感じる」ものを*恣意的に選択してもよい。*より良い方法は、乱数発生器を用いて *ランダムに入力を*選択することである。さらにもう一つの選択肢は、大きな入力空間をいくつかの代表値に分割して、*系統的に入力を*選択することである。恣意的な選択は通常最悪の結果をもたらす。ランダムな選択は多くのシナリオでうまく機能する。

同等性テスト

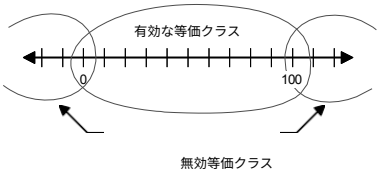
*等価性テストは*ブラックボックステスト手法の一つであり、すべての入力可能空間を等価グループに分割し、各グループでプログラムが「同じ動作をする」ようにする。目的は、各同値グループから代表的な入力値を選択することにより、テストケースの総数を減らすことである。システムは同値群からのすべての入力に対して同じような振る舞い

をするという前提があるため、各群の単一要素だけでテストすれば十分である。等価性テストには2つのステップがある：(*i*) 入力パラメーターの値を同値群に分割する。(*ii*) テスト入力値の選択。

このアプローチの問題点は、正しさを証明するのと同じくらい、入力の等価クラスを見つけるのが難しいことである。そのため、ヒューリスティック（一般的に有用だが、正しさを保証するものではない経験則）を使ってテストケースのセットを選択する。私たちは基本的に、経験とドメイン知識に基づいて推測し、選択されたテストケースの少なくとも1つが、真の（未知の）等価クラスのそれぞれに属することを望みます。

入力パラメータの値を同値クラスに分割することは、以下のヒューリスティックスに従って実行することができる：

- 値の範囲で指定された入力パラメータについて、値空間を1つの有効な等価クラスと2つの無効な等価クラスに分割する。例えば、許容される入力値が0から100までの整数、有効な同値クラス



は0から100までの整数を含み、一方の無効な等価クラスはすべての負の整数を含み、もう一方の無効な等価クラスは100より大きい整数をすべて含む。

- 1つの値で指定された入力パラメータについて、値空間を1つの有効な等価クラスと2つの無効な等価クラスに分割する。例えば、許容される値が実数の数1.4142の場合、有効な同値クラスは単一の要素{1.4142}を含み、一方の無効な同値クラスは1.4142より小さいすべての実数を含み、もう一方の無効な同値クラスは1.4142より大きいすべての実数を含む。
- 値の集合で指定された入力パラメータについて、値空間を1つの有効な等価クラスと1つの無効な等価クラスに分割する。例えば、許容される値が任意の要素の集合{1,2,4,8,16}のうち、有効な同値クラスは要素{1,2,4,8,16}を含み、無効な同値クラスはそれ以外のすべての要素を含む。
- ブール値として指定された入力パラメータについて、値空間を1つの有効な等価クラスと1つの無効な等価クラス（1つは`TRUE`、もう1つは`FALSE`）に分割する。

入力パラメータに対して定義される等価クラスは、以下の基準を満たさなければならない：

1. *カバレッジ*：すべての入力値が等価クラスに属する。
2. *不連続性*：入力値が複数の等価クラスに属することはない。
3. *表現*：表現：ある演算が等価クラスの1つの要素を入力パラメータとして呼び出され、特定の結果を返す場合、そのクラスの他の要素が入力として使われても同じ結果を返さなければならない。

ある演算が複数の入力パラメーターを持つ場合、入力パラメーターの組み合わせ（デカルト積またはクロス積と呼ばれる、セクション3.2.1参照）に対して新しい等価クラスを定義しなければならない。

例えば、キー・チェッカーの操作 `checkKey(k : Key) : boolean` をテストすることを考えてみよう。図2-35に示すように、Keyクラスは3つの文字列属性を持つ: `code`、`timestamp`、`doorLocation`。リスト2-4で実装されている`checkKey()`オペレーションはタイムスタンプを使わないので、その値は関係ありません。ただし、`checkKey()` の出力が `timestamp` の値に依存しないことをテストする必要があります。他の 2 つの属性、`code` および `doorLocation` は、それぞれ一連の値で指定される。システムが、{196, 198, 200, 202, 204, 206, 208, 210} という部屋番号のマンションに設置されているとする。属性 `doorLocation` は、関連するアパート番号の値を取るとします。一方、入

居者は4桁のアクセスコードを{9415, 7717, 8290, ..., 4592}と選択しているかもしれません。コード値 "9415 "とdoorLocation値 "198 "はそれぞれ個別に有効であるが、198号室のテナントのコード値は "7717 "であるため、これらの組み合わせは無効である。

したがって、コード値と `doorLocation` 値のクロス積を作成し、この値空間を有効な等価クラスと無効な等価クラスに分割する必要があります。有効な等価クラスから選択されたテスト入力値のペアに対して、`checkKey()` 演算は論理値 `TRUE` を返す必要があります。逆に、無効な等価クラスからのテスト入力値のペアに対しては、`FALSE` を返す必要があります。

テストカバレッジを確保する際には、現在のスナップショットだけでなく、過去のスナップショットも考慮する必要があります。たとえば、キー・チェッカーの `checkKey()` という操作をテストするとき

あるアパートの元入居者の以前有効だった鍵は、過去には有効な等価クラスに属していたが、無効な 等価クラスに属していた。特に統合テスト（セクション 2.7.4）では、対応するテストケースを含める必要がある。

バウンダリー・テスト

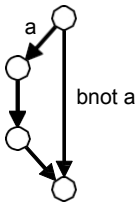
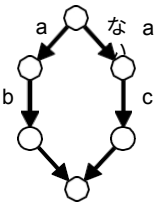
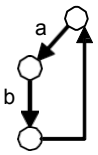
境界テストは、入力パラメーターの境界値に注目する同値テストの特殊なケースである。入力ドメインを同値クラスに分割した後、クラスの「内側」だけでなく、その境界の入力値も使ってプログラムをテストする。境界テストでは、同値クラスから任意の要素を選択するのではなく、同値クラスの「エッジ」、つまりゼロ、最小/最大値、空集合、空文字列、ヌルなどの「外れ値」から要素を選択する。この種のテストの背後にある前提は、開発者は等価クラスの境界で特殊なケースを見落とすことが多いということです。

例えば、ある入力パラメータがaから*b までの*値の範囲で指定されている場合、テストケースはaと*b*の値だけでなく、aと*b*のすぐ上とすぐ下の値で設計する必要があります。

コントロール・フロー・テスト

ステートメント・カバレッジは、プログラム内のすべての基本ステートメントが、テスト・セット内のテスト・ケースによって少なくとも一度は実行されるようなテスト・セットを選択する。

エッジカバレッジは、制御フローのすべてのエッジ（分岐）が、あるテストケースによって少なくとも一度は通過するようなテストセットを選択する。ステートメントがグラフのエッジとなり、エッジで結ばれたノードがステートメントへの出入りを表すように、プログラムの*制御グラフを*構成する。一連のエッジ（分岐なし）は1つのエッジに折りたたむ。

a;	a; b;	*if* a *then* b;	*if* a *then* b *else* c;	*while* a *do* b;
				

条件カバレッジ（*述語カバレッジとも呼ばれる）は*、あるテストケースにおいて、すべ

ての条件（ブーリアンステートメント）が少なくとも一度は TRUE と FALSE の結果をとるようなテストセットを選択する。

パス・カバレッジは、正しさを検証するために少なくとも一度はトラバース（通過）しなければならない、プログラム中の明確なパスの数を決定する。この方法では、ループの繰り返しや再帰呼び出しは考慮されない。サイクロマティック複雑度メトリック（セクション4.2.2）は、独立したパスの数を決定する簡単な方法を提供する。

ステート・ベース・テスト

状態ベースのテストは、ソフトウェア・ユニットが取り得る抽象的な状態のセットを定義し、実際の状態と期待される状態を比較することによって、ユニットの動作をテストする。このアプローチは、オブジェクト指向システムで普及している。オブジェクトの*状態は*、オブジェクトの属性の値に対する制約として定義される。メソッドはオブジェクトの振る舞いを計算する際に属性を使用するため、振る舞いはオブジェクトの状態に依存する。

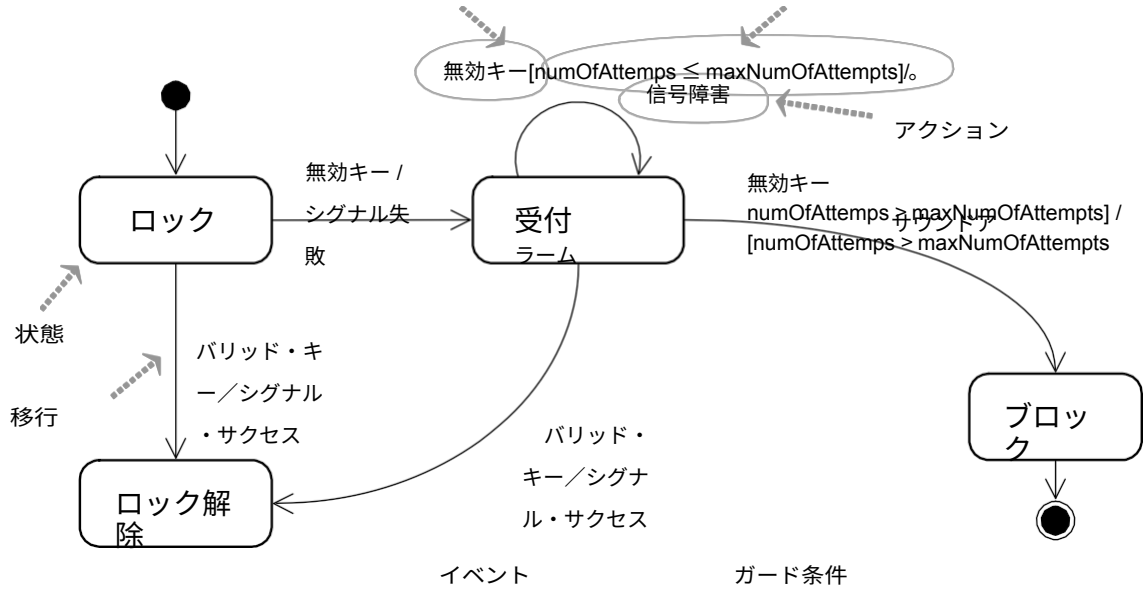


図 2-40：図 2-35「Controller」クラスの UML 状態図。UML 状態図の記法については 3.2 節で説明します。

ステート・ベース・テストを使用する最初のステップは、テストされるユニットの状態図を導出することである。まず、状態を定義することから始めます。次に、状態間で起こり得る遷移を定義し、ある状態から別の状態への遷移のトリガーとなるものを決 定します。ソフトウェアクラスの場合、状態遷移は通常、メソッドが呼び出されたときに発生します。次に、個々の状態についてテスト値を選択します。

第2のステップは、ユニットを初期化してテストを実行することである。セクション2.7.3で説明するように、テスト・ドライバはユニットのメソッドを呼び出してユニットを実行します。ドライバがユニットの実行を終了すると、エラーがまだ発生していないと仮定して、テストはユニットの実際の状 態と期待される状態を比較します。ユニットが期待される状態に到達した場合、その状態に到達した方法に関係なく、そのユニットは正 しい と みなされる。

安全な自宅へのアクセス事例 (図 2-35 のクラス図) の Controller クラスをテストするとします。状態図の導出プロセスと UML 状態図の記法については、第 3 章で説明します。コントローラの重要な責務は、無効なキーのために失敗した試みを追跡することで、辞書 攻撃を防ぐことである。通常、ドアは施錠されていると仮定する(表2-1のREQ1で必須)。利用者は有効な鍵を提供することで、ドアのロックを解除する。ユーザが無効な

キーを提示した場合、コントローラは最大 maxNumOfAttempts の失敗を許容し、それ以降はブロックしてアラームを鳴らす。したがって、状態図の以下の要素を特定する（図 2-40）：

- 4つの状態｛ロック、アンロック、アクセプト、ブロック｝。
- つのイベント{有効キー、無効キー}。
- 有効な5つの遷移{ロック→アンロック、ロック→アクセプタンス、アクセプタンス→アクセプタンス、アクセプタンス→アンロック、アクセプタンス→ブロック}。

テストセットは、状態図を通る所定のパスに沿ってオブジェクトを実行するシナリオで構成される。一般に、状態図要素の数は

全イベント、全状態≦全遷移≦全パス

状態図において可能なパスの数は一般に無限であるため、可能なパスそれぞれをテストするの は 現実的ではない。その代わりに、以下のカバレッジ条件を確保する：

- 特定されたすべての状態を少なくとも1回はカバーする（各状態は少なくとも1つのテストケースに含まれる）
- すべての有効なトランジションを少なくとも一度はカバーする
- すべての無効な遷移を少なくとも1回トリガーする。

すべての有効な遷移をテストすることは、全イベントカバレッジ、全状態カバレッジ、全アクションカバレッジ を意味する（包含する）。これは、状態図を責任を持ってテストするための、最低限受け入れ可能な戦略と考えられる。全遷移のテストは網羅的 *ではないことに注意してください*。なぜなら、網羅的なテストでは、ステートマシン上のすべてのパスを少なくとも1回はテストする必要があり、通常は不可能か、少なくとも現実的ではないからです。

2.7.3 ユニットテストの実践的側面

単一のコンポーネント（または「ユニット」）、あるいはコンポーネントの組み合わせに対してテストを実行するには、テスト対象がシステムの他の部分から隔離されている必要がある。そうでなければ、テストによって発見された問題を特定することができません。しかし、システムの部品は通常相互に関連しており、互いに無関係では動作しない。システムの欠落部分を補うために、私たちはテストドライバーとテストスタブを使います。**テストドライバーは、**テスト対象のコンポーネントの操作を呼び出すシステムの一部をシミュレートします。**テストスタブは、テスト対象のコンポーネントから**呼び出されるコンポーネントをシミュレートする最小限の実装です。テストされるものは**フィクスチャとも**呼ばれます。

スタブとは、単体テストを行うために存在する、インターフェースの些細な実装のことである。例えば、スタブは計算をせずに固定値を返すようにハードコードされているかもしれない。スタブを使うことで、*実際の*コードを書かずにインターフェイスをテストすることができる。インターフェイスが正しく動作しているかどうかを確認するために、実装は実際には必要ありません（クライアントの視点から見ると、インターフェイスはクライアントオブジェクトのためのものであることを思い出してください、セクション1.4）。ドライバとスタブは**モックオブジェクトとしても**知られています。

各試験方法はこのサイクルに従う：

1. テストするもの（フィクスチャ）、テストドライバ、テストスタブを作成する。
2. テスト・ドライバにフィクスチャの操作を呼び出させる
3. 結果が期待通りであることを評価する

より具体的には、**ユニットテストケースは、**テストドライバーによって実行される3つのステップから構成される：

1. オブジェクトの*セットアップ*：テストするオブジェクトとそれに依存するオブジェクトを作成し、セットアップする。
2. テストされたオブジェクトに*作用する*
3. 結果が期待通りであることを*確認する*

2.6節で設計した安全なホームアクセス事例のKey Checkerクラスをテストしたいとします。図 2-41(a) は、図 2-33 から抜粋した関連する抜粋シーケンス図です。`Checker` クラスはテストされるコンポーネントであり、`Controller` の代用となるテストドライバと、 `KeyStorage` クラスと `Key` クラスの代用となる*テストスタブを*実装する必要があります。

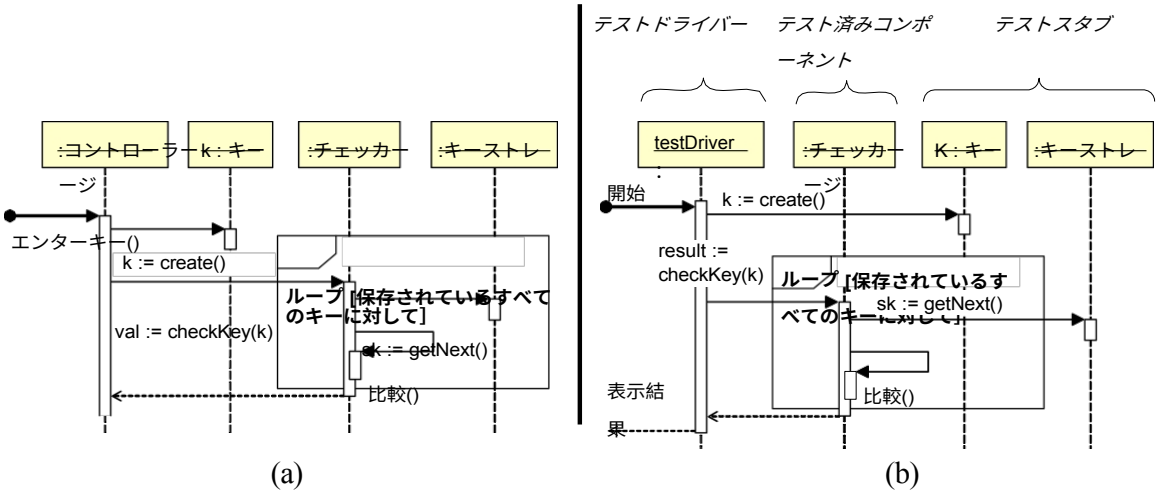


図 2-41：キー・チェッカーの操作 `checkKey()` のテスト（ユースケース Unlock）。(a）図 2-33 から抜粋したシーケンス図の関連部分。(b) キーチェッカーをテストするためのテストスタブとドライバー。

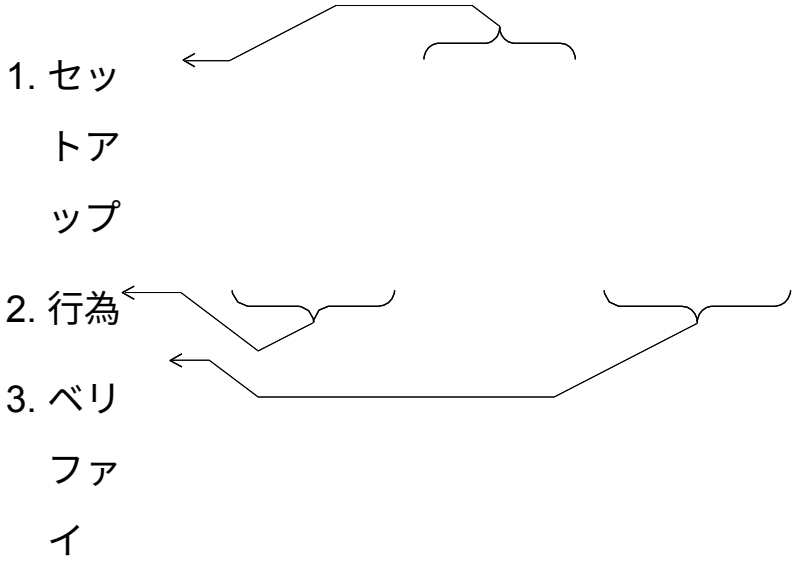
図 2-41(b)に示すように、テストドライバーはテストされたコンポーネントにテスト入力を渡し、 結果を表示します。Java のテストフレームワーク JUnit では、結果の検証は `assert*()` メソッドで行います。テストドライバはどんなオブジェクトでもかまいませんが、 `Controller` クラスのインスタンスである必要はありません。これとは異なり、テストスタブはシミュレートするコンポーネントと同じクラスでなければなりません。同じ操作 API を同じ返り値で提供しなければなりません。テストスタブの実装は自明な作業ではないので、 正確なテストスタブを実装することと実際のコンポーネントを使うことは トレードオフの関係にあります。つまり、KeyStorageと`Key`クラスの実装が利用可能であれば、Key Checkerクラスをテストする際にそれらを使用することができます。

リスト2-1：キーチェッカー・クラスのテストケース例。

```
public class CheckerTest {
  // 無効なキーが拒否されることをチェックするテストケース @Test
  public void
    checkKey_anyState_invalidKeyRejected(){」。

    // 1. セットアップ
    Checker checker = new Checker( /* コンストラクタのパラメータ */ );
    // 2.
    Key invalidTestKey = new Key( /* 無効なコードでセットアップ */ );
    boolean result = checker.checkKey(invalidTestKey);
    // 3. assertEqual(result,
    false)を検証する;
  }
}
```

テストケースを表すメソッドには、以下の表記法を使います（リスト2-1参照）：



startingStateは、テストされるメソッドが呼び出される条件です。そして、expectResultは、指定された条件下でテストされるメソッドが生成することを期待する結果です。この例では、`Checker` のメソッド `checkKey()` をテストしています。`Checker` オブジェクトは属性を持たないので、常に初期状態にあります。期待される結果は、`checkKey()` が 無効な キーを拒否することです。 したがって は テスト ケース メソッド という名前のテスト・ケース・メソッドを呼び出します。

異なる状態を持つオブジェクトをテストするのは、もう少し複雑です。なぜなら、 オブジェクトをテストされる状態に持っていき、最終的に、オブジェクトが期待される状 態のままであることを検証しなければならないからです。図 2-40 に示す `Controller` オブジェクトとその状態図を考えてみましょう。あるテストケースでは、`Controller` が maxNumOfAttempts の無効なキーを受け取ると、正しく Blocked 状態に遷移することを検証する必要があります。

リスト2-2: Controller クラスのテストケースの例。

```
public class ControllerTest {
  // ブロックされた状態になったことを確認するテストケース @Test
  public void
    enterKey_accepting_toBlocked(){」。

    // 1. セットアップ: オブジェクトを開始状態にする コントローラ cntrl =
    new Controller( /* コンストラクタのパラメータ */ );
    // コントローラがブロックされる直前に、コントローラをAccepting状態にする
    Key invalidTestKey = new Key( /* 無効なコードでセットアップ */ ); for
    (i=0; i < cntrl.getMaxNumOfAttempts(); i++) {.
        cntrl.enterKey(invalidTestKey);
    }
    assertEqual(    // 開始状態が設定されていることをチェック
        cntrl.getNumOfAttempts(), cntrl.getMaxNumOfAttempts() - 1
    );

    // 2. act
    cntrl.enterKey(invalidTestKey)
    ;

    // 3. 検証
    assertEqual(    // 結果の状態は "Blocked" でなければならない
        cntrl.getNumOfAttempts(),
        cntrl.getMaxNumOfAttempts())
    );
    assertEqual(cntrl.isBlocked(), true);
  }
}
```

残りのテストケースを設計し、カバレッジ条件を確保することは読者に任されている（セクション2.7.2）。

ユニットテストの重要な課題は、各ユニットを個別にテストできるように、ユニットを十分に分離することである。そうしないと、統合テストに近い「ユニット」テストになってしまいます。この分離を達成するための最も重要なテクニックは、具体的なクラスではなくインターフェイスに対してプログラミングを行うことです。

2.7.4 統合テストとセキュリティテスト

伝統的な手法では、テストは開発ライフサイクルの比較的後期に行われ、図 2-39 の論理的な順序に従う。単体テストの後に統合テストが続き、その後にシステムテストが続く。統合テストは、個々のコンポーネント（「ユニット」）を連結し、結合されたコンポーネントの正しさをテストすることで、ステップバイステップで動作する。コンポーネントは*水平方向に*組み合わされ、水平方向の統合テスト戦略によって、異なる方向に統合プロセスが行われる。

アジャイル手法では、テストは開発サイクル全体に組み込まれる。エンドツーエンドの機能を実装するために、コンポーネントは*縦割りで*組み合わされる。各縦割りスライスはユーザーストーリー（セクション2.2.3）に対応し、ユーザーストーリーは並行して実装され、テストされる。

水平統合テスト戦略

テスト済みのユニットを組み合わせることから始めるには、さまざまな方法がある。最も単純なのは、**"ビッグバン "統合**アプローチと呼ばれるもので、すべてのコンポーネントを一度にリンクし、その組み合わせをテストする。

ボトムアップ統合は、最下層の階層にあるユニットを組み合わせることから始まる。階層」は、他のユニットに依存しないユニットから開始することで形成される。たとえば、図 2-35 のクラス図では、`PhotoSObsrv`、`Logger`、および `DeviceCtrl` の各クラスは、他のクラスを指し示すナビゲーシ ョン矢印を持たないため、これら 3 つのクラスがシステム階層の最下層を形成します（図 2-42(a)）。ボトムアップ統合テストでは、一番下のユニット（「リーフユニット」）がユニットテストによって最初にテストされま

す（図 2-42(b)）。次に、最下位ユニットへのナビゲーショ ンを持つユニットをリーフユニットと組み合わせてテストする。統合は、最上位レベルがテストされるまで、階層を上がっていきます。テストスタブを開発する必要はない：他のすべてのユニットについては、現在テストされているユニットが依存するユニットがすでにテストされている。ボトムアップ・テストのためのテスト・ドライバを開発する必要はあるが、それは比較的単純なものでよい。実世界のシステムでは、ユニット階層は必ずしも「ツリー」構造を形成しているとは限らず、むしろ、ユニットの正確なレベルを決定することを困難にするサイクルを含んでいるかもしれないことに注意されたい。

トップダウン統合は、他のユニットが依存しない最上位階層のユニットをテストすることから始まります（図2-42(c)）。この手法では、テストドライバーを開発する必要はありませんが、テストスタブは必要です。

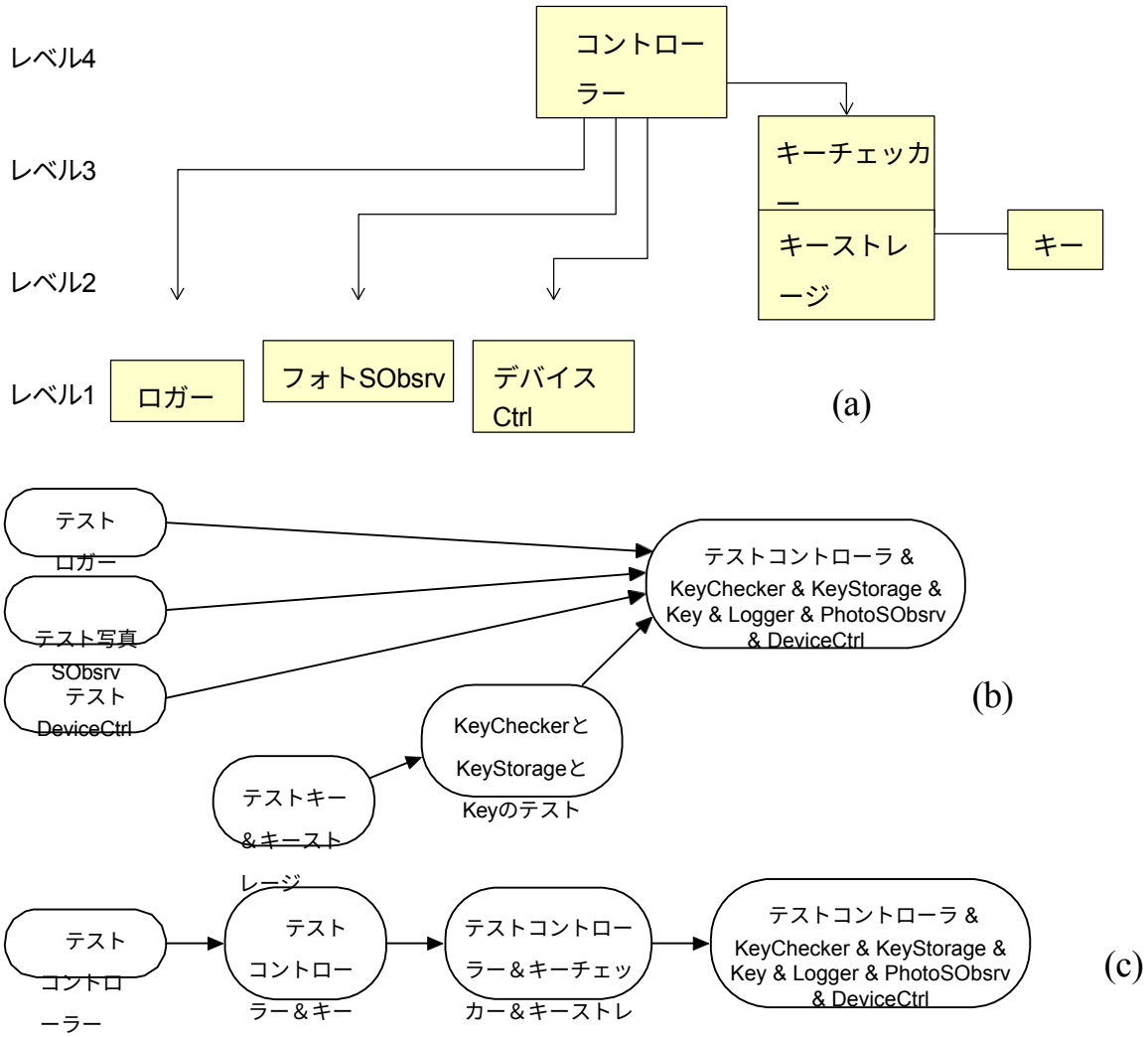


図 2-42：図 2-35 のシステムの統合テスト戦略。(a）ユニット階層、（b）ボトムアップ統合テスト、（c）トップダウン統合テスト。

サンドウィッチ統合アプローチは、トップダウンとボトムアップを組み合わせたもので、両端から出発し、中間レベルの構成要素を双方向に段階的に使用していく。中間レベルは*ターゲットレベル*と呼ばれる。サンドイッチテストでは、通常、トップレベルのコンポーネントをテストするためのスタブを記述する必要はない。同様に、実際のターゲットレベルのコンポーネントは、低レベルコンポーネントのボトムアップテストのドライバとして使用されます。図2-42(a)のシステム階層の例では、ターゲット層には1つのコンポーネントしか含まれていません：キーチェッカーである。まず、チェッカーを使用したコントローラのトップダウンテストから始めます。並行して、鍵ストレージのボトムアップテストをチェッカーを使って再度行う。最後に、すべてのコンポーネントをまとめてテストする。

それぞれの統合戦略には利点と欠点がある。ボトムアップ統合は、ユーティリティ・ライブラリのような低レベルのコンポーネントが多いシステムに適している。上位のコンポーネントが下位のコンポーネントの仮定に違反した場合、どこに問題があるかを見つけやすくなります。欠点は、最上位のコンポーネント（ユーザー・インターフェースのような、通常最も重要なコンポーネント）が最後にテストされることである。

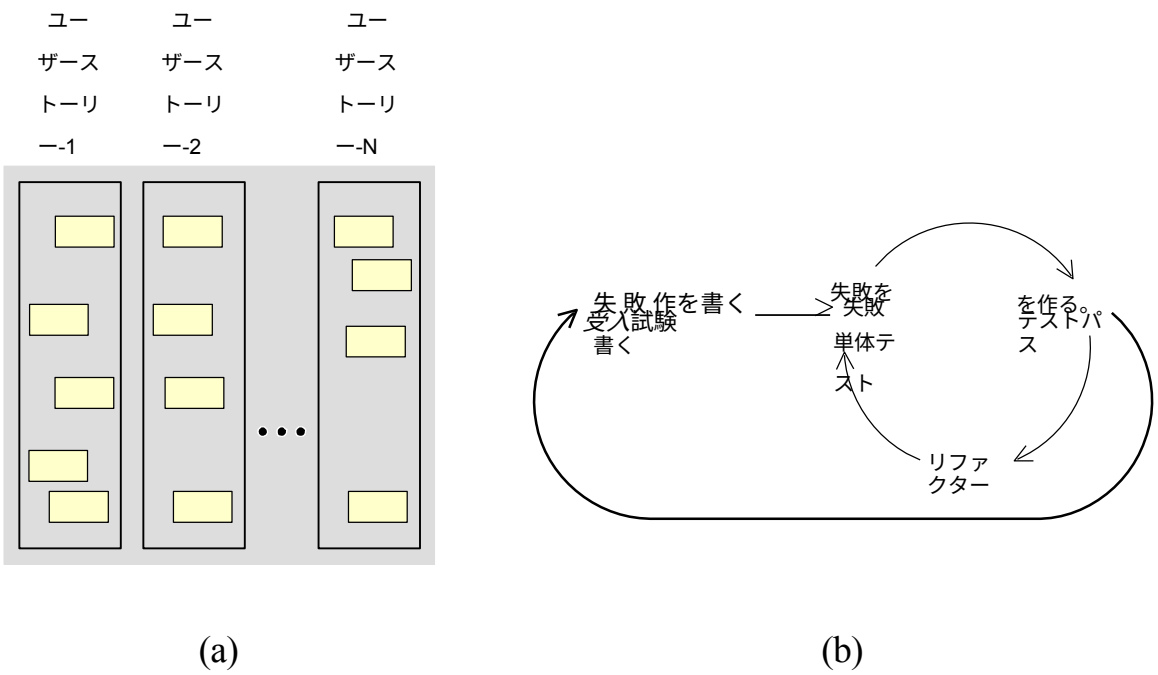


図2-43：アジャイル手法における垂直統合は、機能的な垂直スライス（ユーザーストーリー）を並行して開発する（a）。それぞれのストーリーは、内側のフィードバックループのユニットテストと外側のフィードバックループの受け入れテストを統合するサイクルで開発される（b）。

トップダウン統合には、最上位のコンポーネント（通常はユーザーインターフェース、つまりエンドユーザーが早期に関与できる可能性がある）から始めるという利点がある。テストケースは要件から直接導き出すことができる。欠点は、テストスタブの開発に時間がかかり、エラーが発生しやすいことです。

サンドイッチ・テストの利点は、スタブやドライバを書く必要がないこと、ユーザー・インタフェースを早期にテストできること、したがってエンド・ユーザーが早期に参加できることである。欠点は、サンドイッチテストでは、統合前にターゲット（中間）レベルのユニットを徹底的にテストできないことである。この問題は、下位、中位、上位の各レベルを個別にテストしてから、それらを互いにインクリメンタル・テストで結合する、*修正サンドイッチ・テストによって*改善することができる。

垂直統合テスト戦略

アジャイル手法では、*垂直統合*アプローチを使ってユーザーストーリーを並行して開発する（図2-43(a)）。各ストーリーはフィードバックループ（図2-43(b)）で開発され、開

発者は内側のループでユニットテストを使い、顧客は外側のループで受け入れテストを実行します。それぞれのサイクルは、顧客/ユーザーが特定のユーザーストーリーをテストする受け入れテストを書くことから始まります。受け入れテストに基づいて、開発者は単体テストを書き、関連する、つまり単体テストをパスするために必要なコードだけを開発する。単体テストは、コードが書かれた直後に毎日実行され、単体テストに合格したコードだけがコードベースにコミットされる。受け入れテストは、各サイクル（数週間から数カ月単位）の終わりに実行される。

垂直統合の利点は、実用的な成果物が迅速に得られることである。潜在的な欠点は、各サブシステム（垂直スライス-ユーザーストーリー）が独立して開発されるため、システムの統一性と "グランドデザイン "に欠ける可能性があることである。そのため、開発サイクルの後半でシステムの大幅な再設計が必要になる可能性がある。

セキュリティ・テスト

機能テストは、必要な機能や特徴が正しく実装されているかどうかという「正」のテストです。しかし、セキュリティの欠陥や脆弱性の大部分は、暗号化や特権管理のようなセキュリティ機能とは直接関係ありません。その代わりに、セキュリティの問題には、攻撃者が発見した、予期しない、しかし、意図的なシステムの悪用が関係することがよくあります。したがって、攻撃されたときにシステムがどのように振る舞うかを判断するために、悪用ケースのような「ネガティブ」なテストも必要です。セキュリティテストは、多くの場合、既知の攻撃パターンによって進められます。

2.7.5 テスト駆動実装

「本物のプログラマーはコードをコメントしない。書くのが難しいなら、理解するのも難しいはずだ"
-不明

このセクションでは、設計されたシステムがどのように実装されるかを示す。(先に進む前に、付録AのJavaプログラミングの復習を復習しておくとよいだろう)。プログラマーが軽視しがちなことの1つは、コードがエレガントで読みやすくなければならないということです。これは、コードを実行するコンピュータのためではなく、元のコードを読み、保守し、改良する人間のためである。良いコメントを書くことは、少なくとも良いコードを書くことと同じくらい難しいと私は信じている。なぜなら、コメントは開発者の*意図を*記述するものであり、コードは開発者が行ったことだけを表現するものだからだ。美的感覚に欠け、コメントの書き方が悪いコードは、質の低いコードである可能性が高い。[11]コメントのほかに、JavaやC#などの言語では、クラスやメソッドのドキュメントを書くための特別な構文が用意されている。Javadocは、ソースコードのドキュメント・コメントからHTML形式のAPIドキュメントを生成するツールだ。SandcastleはC#用の同等のツールだ。

我々のシステムのハードウェア・アーキテクチャは、セクション[@@@]で説明されている（図2-7）。

以下のコードでは、プログラムの同時実行にスレッドを使用している。スレッドに詳しくない読者はセクション5.3を参照されたい。

メイン・クラスの主な目的は、外部情報（有効なキーのテーブルと、デバイスを制御す

る組み込みプロセッサへの接続）を取得することです。以下はメイン・システム・クラスの実装です。

リスト 2-3: 実装 Java コード の 実装Javaコード メインクラスのJavaコードです、 と呼ばれる HomeAccessControlSystem、ケーススタディー・ホームアクセス・システムの。

```
import java.util.TooManyListenersException;
import javax.comm.CommPortIdentifier;
import javax.comm.NoSuchPortException;
```

---

[11]関連して、ユーザーメッセージを書くことも同様に重要である。次の面白い話は、マイクロソフトの製品以外にも応用できる："*かつて、偉大な作家になり、何百万人もの人々が読んで感情的なレベルで反応し、泣いたり、痛みや怒りで吠えたりするようなものを書きたいと思った若者がいた。*""*それで今、彼はマイクロソフト社でエラーメッセージを書く仕事をしている*。"出典A Prairie Home Companion、2007年2月3日。オンライン: http://prairiehome.publicradio.org/programs/2007/02/03/scripts/showjokes.shtml].

```
import javax.comm.SerialPort;
import javax.comm.SerialPortEvent;
インポート javax.comm.SerialPortEventListener;
インポート javax.comm.UnsupportedCommOperationException;

public class HomeAccessControlSystem extends Thread
        implements SerialPortEventListener {.
    protected Controller ctrler_; // ドメインロジックへのエントリーポイント
    protected InputStream inputStream_; // シリアルポートから protected
    StringBuffer key_ = new StringBuffer(); // ユーザーのキーコード public
    static final long keyCodeLen_ = 4; // 4文字のキーコード

    public HomeAccessControlSystem(
        KeyStorage ks, SerialPort ctrlPort)
    ) {
        を試す。
            inputStream_ = ctrlPort.getInputStream();
        } catch (IOException e) { e.printStackTrace(); }.

        LockCtrl lkc = new LockCtrl(ctrlPort);
        LightCtrl lic = new LightCtrl(ctrlPort);
        PhotoObsrv sns = new PhotoObsrv(ctrlPort);
        AlarmCtrl ac = new AlarmCtrl(ctrlPort);

        ctrler_=
            new Controller(new KeyChecker(ks), lkc, lic, sns, ac);

        を試す。
            ctrlPort.addEventListener(this);
        } catch (TooManyListenersException e) { e.printStackTrace();
            // 1つのポートにつき1つのリスナーに制限されます。
        }
        start(); // スレッドを開始する
    }

    /** 最初の引数はハンドル（ファイル名、IPアドレス、...）です。
     * 有効なキーのデータベースの
     * 2番目の引数は省略可能で、存在する場合は次のようになります。
     * シリアルポート。*/
    public static void main(String[] args) {
        KeyStorage ks = new KeyStorage(args[1]);

        SerialPort ctrlPort;
        String portName = "COM1";
        if (args.length > 1) portName =args[1];
        try {       // 初期化
            CommPortIdentifier cpi =
                CommPortIdentifier.getPortIdentifier(portName);
            if (cpi.getPortType() == CommPortIdentifier.PORT_SERIAL) {
                ctrlPort = (SerialPort) cpi.open();
```

```
            // シリアルポートから読み込むスレッドを開始する new
            HomeAccessControlSystem(ks, ctrlPort);
        } catch (NoSuchPortException e) {
            System.err.println("使用法： .....ポート名");
        }
```

```
        を試す。
            ctrlPort.setSerialPortParams()。
                9600、SerialPort.DATABITS_8、SerialPort.STOPBITS_1、
                SerialPort.PARITY_NONE
            );
        } catch (UnsupportedCommOperationException e) {
            e.printStackTrace();
        }
    }

    /** 何もせず、ただ中断されるのを待つ。
     * シリアル・ポートからの入力による。*/
    public void run() {
        while (true) { // スリープとアウェイクを交互に繰り返す try {
            Thread.sleep(100); }.
            catch (InterruptedException e) { /* 何もしない */ }.
        }
    }

    /** シリアル・ポート・イベント・ハンドラ
     * 入力された文字が1つずつ送信されると仮定する。public void
    serialEvent(SerialPortEvent evt) { */ public void
    serialEvent(SerialPortEvent evt)
        if (evt.getEventType() == SerialPortEvent.DATA_AVAILABLE) {
            byte[] readBuffer = new byte[5]; // 念のため5文字。

            を試す。
                while (inputStream_.available() > 0) { {.
                    int numBytes = inputStream_.read(readBuffer);
                    // "numBytes" == 1かどうかをチェックできる ...
                }
            } catch (IOException e) { e.printStackTrace(); }.
            // 新しい文字列をユーザー・キーに追加
            key_.append(new String(readBuffer));

            if (key_.length() >= keyCodeLen_) { // キー全体を取得したか?
                // コントローラに渡す ctrler_.enterKey(key_.toString());
                // 新しいユーザーキー用の新しいバッファを取得 key_
                = new StringBuffer();
            }
        }
    }
}
```

HomeAccessControlSystemクラスは、永遠に実行され、シリアル・ポートからの入力を受け付けるスレッドである。メイン・スレッドはすべてをセットアップして終了する

が、新しいスレッドは生き続けるからである。スレッドについてはセクション5.3で説明する。

次に示すのは、図 2-33 で設計されたコア・システムの実装例である。システムのコーディングは相互作用図によって直接駆動されます。

リスト2-4：`Controller`、`KeyChecker`、および`KeyChecker`クラスの実装Javaコード。

Ctrlをロックする。

```
public class Controller { protected
    KeyChecker checker_; protected
    LockCtrl lockCtrl_; protected
    LightCtrl lightCtrl_; protected
    PhotoObsrv sensor_; protected

    AlarmCtrl alarmCtrl_;
    public static final long maxNumOfAttempts_ = 3;
    public static final long attemptPeriod_ = 600000; // msec [=10min]

    protected long numOfAttempts_ = 0;

    public コントローラ(
        KeyChecker kc, LockCtrl lkc, LightCtrl lic,
        PhotoObsrv sns, AlarmCtrl ac
    ) {
        checker_ = kc;
        lockCtrl_ = lkc; alarmCtrl_ = ac;
        lightCtrl_ = lic; sensor_ = sns;
    }

    public enterKey(String key_code) { Key
        user_key = new Key(key_code) if
        (checker_.checkKey(user_key)) { {
        { { { {
        (checker_.checkKey(user_key))
            lockCtrl_.setArmed(false);
            if (!sensor_.isDaylight()) { lightCtrl_.setLit(true; }
            numOfAttempts_ = 0;
        } else {
            // 試行期間もチェックする必要があるが、... if
            (++numOfAttempts_ >= maxNumOfAttempts_) { 。
                alarmCtrl_.soundAlarm();
                numOfAttempts_ = 0; // 次のユーザーのためにリセットする。
            }
        }
    }
}
```

インポート java.util.Iterator;

```
public class KeyChecker {.
    protected KeyStorage validKeys_;

    public KeyChecker(KeyStorage ks) { validKeys_ = ks; }
```

public boolean checkKey(Key user_key) { 以下のようになりま

す。

```
        for (イテレータ e = validKeys_.iterator(); e.hasNext(); ){
            if (compare((Key)e.next(), user_key) { return true; }.
        }
```

false を返す;

```
    }
```

```
    protected boolean compare(Key key1, Key key2) {.

    }
}
```

インポート javax.comm.SerialPort;

```
パブリッククラスLockCtrl {
    protected boolean armed_ = true;

    public LockCtrl(SerialPort ctrlPort) { {.
    }
}
```

リスト2-4では、KeyStorageが`java.util.ArrayList`のリストとして実装されていると仮定している。キーが単純なオブジェクト、例えば数字である場合は、`java.util.HashMap`のハッシュテーブルを使用する方法もあります。キーが与えられると、KeyStorageは有効なキーの値を返します。戻り値が`NULL`の場合、キーは無効です。キーは、リレーショナル・データベースやプレーン・ファイルなどの永続ストレージに格納し、リスト2-4には示されていないが、システム起動時にKeyStorageにロードする必要がある。

要件からコードへの段階的な進行を注意深く追った読者は、プログラミング言語に関係なく、コードには多くの詳細が含まれており、通常、高レベルの設計の選択や抽象化を不明瞭にしていることに気づくかもしれない。細部まで正確に記述する必要があり、言語特有の特殊性も避けられないため、コードのみからソフトウェアの構造を理解し推論することは難しい。この時点で、読者が追跡可能な段階的進行と図式的表現の有用性を理解してくれることを願っている。


Coca Cola
Coca Cola
Coca Cola
Coca Cola

2.7.6 リファクタリング：既存のコードの設計を改善する

既存のコードの**リファクタリングとは、**動作を維持したまま設計を改善する変換のことである。リファクタリングは、ソフトウェアの内部構造を変更することで、理解しやすく、修正コストが安くなるようにするもので、観察可能な動作は変更しない。リファクタリングのプロセスには、重複を取り除き、複雑なロジックを単純化し、不明瞭なコードを明確にすることが含まれる。リファクタリングの例としては、変数名を変更するような小さな変更から、2つのクラス階層を統一するような大きな変更までがある。

リファクタリングは、コードに一連の低レベルの設計変換を適用する。各変換は、アイデアを統合し、冗長性を取り除き、あいまいさを明確にすることによって、シンプルな方法で、コードを少しずつ改善する。大きな改善は、段階を追って徐々に達成される。小さな改良に重点を置くのは、理解しやすく追跡しやすいからであり、それぞれの改良はコードに焦点を絞った変更をもたらすからである。影響を受けるのはコードの小さな局所的なブロックだけなので、改良によって欠陥が発生する可能性は低くなる。

アジャイル手法では、テスト駆動開発（TDD）と継続的なリファクタリングを推奨している。リファクタリング（コードの変更）には、ダメージがないことを確認するためのテストが必要だからだ。

条件論理の代わりにポリモーフィズムを使う

プログラミング言語の重要な機能に*条件文*がある。これは、特定の条件が真である場合にのみ、別のステートメントを実行させるステートメントである。簡単な "文 "を使って、コンピューターに、"この15個のことを次々に実行しなさい。もしそれまでに、まだこのようなことが達成されていなければ、ステップ5からやり直しなさい "と指示することができる。同様に、次のような複雑な条件付き命令も容易に記号化できる：「しかし、もし*そうなったら*、そのようなことをしなさい。もし何か他のことが起こったら、それが何であれ、そのようなことをしなさい。IF-THEN-ELSE、DO-WHILE、SWITCHといった言語構文を使うことで、動作の契機が正確に指定される。条件文の問題点は、コードをわかりにくくし、エラーを起こしやすくすることだ。

ポリモーフィズムを使えば、型によって振る舞いが異なるオブジェクトがあっても、明示的な条件分岐を避けることができる。その結果、オブジェクト指向プログラムでは、型コードを切り替える`switch`文や、型文字列を切り替えるif-hen-else文はあまり使われなくなる。ポリモーフィズムには多くの利点がある。最も大きな利点は、同じ条件セットがプログラムのあちこちに現れる場合だ。新しい型を追加したい場合、すべての条件式を見つけて更新しなければならない。しかし、サブクラスを使えば、新しいサブクラスを作り、適切なメソッドを提供するだけでいい。クラスのクライアントはサブクラスについて知る必要がないので、システム内の依存関係が減り、更新が簡単になります。

境界条件のチェックなど、いくつかの条件分岐は必要ですが、似たような変数を扱い続けながら、条件によって異なる処理を適用する場合、ポリモーフィズムの最適な場所となり、コードの複雑さを軽減することができます。さて、ポリモーフィズムで置き換えることのできない条件式が通常2種類あります。それは、比較級（>、<）（またはプリミティブの操作、通常）と境界ケース（時々）です。そして、この2つは言語

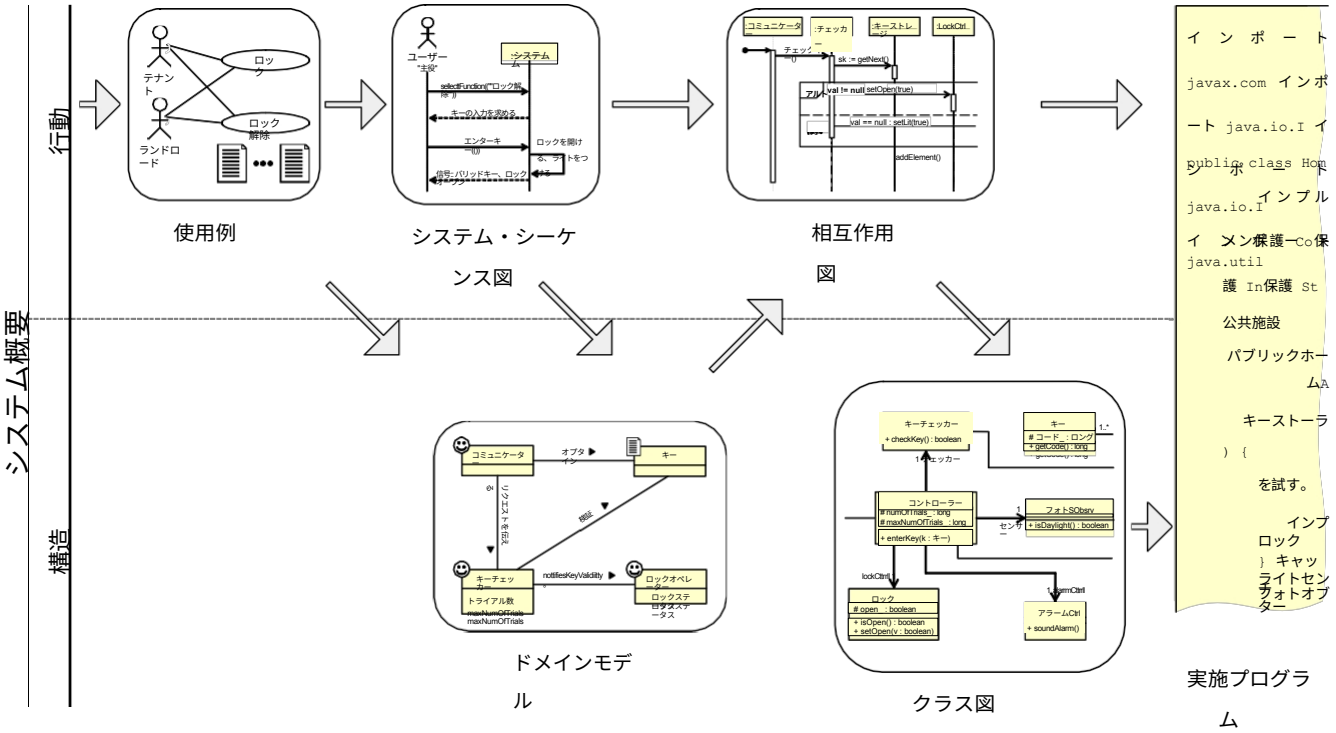


図2-44：ソフトウェア開発ライフサイクルの一反復の概要。このアクティビティは、システムの動作と構造を交互に詳しく説明する。選択したステップと成果物のみを示す。

Javaだけのように。他のいくつかの言語では、クロージャを渡すことができ、条件文の必要性を難しくしている。

2.8 要約と参考文献

「良い判断は経験から、経験は悪い判断から生まれる」。
-フレデリック・P・ブルックス

本章では、インクリメンタルかつ反復的なソフトウェア設計のアプローチを提示し、実行中のケーススタディーを用いてソフトウェアエンジニアリング技術を徐々に紹介する。プロセスの主要なフェーズを図2-44にまとめた。(なお、構造的な記述である*パッケージ図は*、スペース不足のため示していない)。連続するソフトウェア成果物間の意味のある対応を確実にするために、開発ライフサイクル全体にわたってトレーサビリティ・マトリクスを維持する。トレーサビリティ・マトリクスは、要求、設計仕様、ハザード、および検証をリンクする。これらの活動や文書間のトレーサビリティは不可欠である。

図2-44は、活動が行われる論理的な順序を示しているだけであり、ウォーターフォール法のようにソフトウェアライフサイクルが一方向に進行することを意味するものではない。実際には、各ステップの間には大きな絡み合いや後戻りがあり、図2-44はプロセスの*1つの反復を示し*ているに過ぎない。図2-44はプロセスの1つの反復に過ぎない。