コンピュータリテラシ＃8 – ソフトウェア開発とテストケース

久野 靖 (電気通信大学)

2017.5.19

1　今回の目標

今回の目標は次の通りです。

- 高水準言語によるプログラムの記述について理解する — プログラムとはどのようなものか知っておくことはコンピュータの理解に有用です
- ソフトウェア開発とその難しさについて理解する — ソフトウェア開発は仕事としてのほかに研究活動として行うこともあるのでその難しさを知っておくことは有用です
- テストの考え方とテストケースについて理解する — ソフトウェアをその「仕様」に基づいて判断するという考え方は実際にプログラムを書く時に大変有用です。

2　プログラミングと手順

2.1　高水準言語と低水準言語 exam

以前の回で、CPU がメモリに格納された命令を順に取り出し実行して行く装置であることを説明し、「小さなコンピュータ」の命令を使ってプログラムを組み立ててみました。CPU の命令は機種ごとに違っているので、機械語やアセンブリ言語のプログラムは別機種でそのまま動かすことができません。このようなプログラミング言語のことを**低水準言語** (low level language) と呼びます。低水準言語のプログラムは、繁雑で書くのが大変という欠点も持ちます。

そこで今日では、特定 CPU の命令に依存せず、人間の考え方に近い記法でプログラムを書きます。これを**高水準言語** (high level language) と呼びます。皆様は C++、Java、JavaScript、Ruby などの名前に見覚えがあると思いますが、これらは高水準言語の名前です。

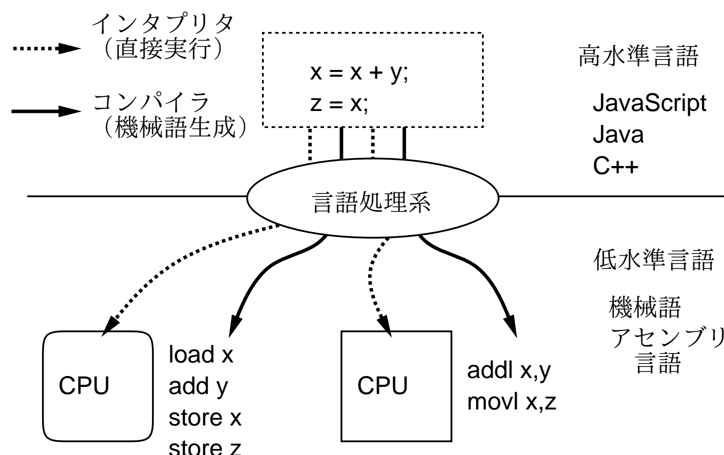

図 1: 高水準言語と低水準言語

高水準言語で記述したプログラムは、**言語処理系** (language processor) と呼ばれるソフトウェアによって実行可能になります (図 1)。言語処理系の種別として、高水準言語をアセンブリ言語や機械語に変換する**コンパイラ** (compiler)、変換する代わりに高水準言語に記述された動作を直接実行する**インタプリタ** (interpreter) があります。

2.2 JavaScript 言語 exam

以下では皆様に、**JavaScript** という言語でプログラムを作る経験をして頂きます。この言語は処理系が Web ブラウザに標準的に内蔵されており、Web システム開発で広く使われています。

今回はこの特徴を利用して、皆様に簡単に練習していただくため、ブラウザ内の入力欄に直接 JavaScript コードを打ち込めるページを作成しました (図 2)。使い方は簡単で、左側の欄に JavaScript コードを打ち込み、「Run」ボタンを押すと実行が始まり、出力などは右側の欄に表示されます。

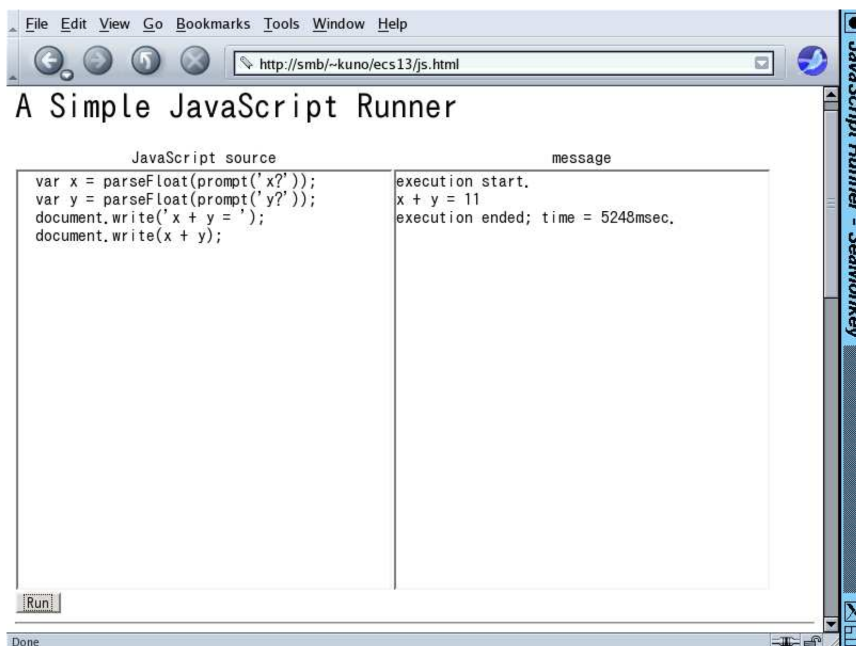

図 2: JavaScript 実行用のページ

ではさっそく、2 つの数の和を計算して表示する例題を見て頂きましょう。

```
var x = parseFloat(prompt('x?'));
var y = parseFloat(prompt('y?'));
document.write('x + y = ');
document.write(x + y);
```

ここで「`var x = △△`」というのは、x という名前の変数 (値を入れておく場所) を用意し、そこに△△の部分で計算した値を入れる、という意味です。「`=`」は等しいという意味ではなく「値を代入する」ことを表します (機械語の store 命令)。これ以外は全て「関数名 (···)」という形で、いずれも、JavaScript 環境に予め用意された機能です。今回使うものの一覧を表 1 に示します。

実際に実行すると、`prompt(...);` を実行したところでメッセージを表示したダイアログボックスが現れ、そこに文字列を打ち込んで OK を押すと先に進む、というふうに動いていきます。`prompt(...)` で打ち込んだ文字列は `parseFloat(...)` によって数値に変換され、変数 x と y に入ります。次に、「`x + y =`」というメッセージを表示し、それに続いて x+y を計算した結果を表示します。このように、足し算などが普通の計算の式らしく書けるのが、高水準言語のよい所です。

表 1: JavaScript のライブラリ関数一覧

書き方	説明
prompt(s)	s を表示して文字列を入力してもらいそれを返す
parseFloat(s)	文字列 s を数値に変換して返す
document.write(s)	文字列 s を出力する
document.writeln(s)	文字列 s を改行つきで出力する

2.3 JavaScript による枝分かれの記述 exam

次に、これも以前やった例のやき直してすが、「2 つの数のうち大きい方を表示する」例題を示します。以前は「条件によって枝分かれ」するのに条件分岐命令を使いましたが、高水準言語では「枝分かれ全体を表す構文」(if 文) を使います。具体的に見てみましょう。

```
var x = parseFloat(prompt('x?'));
var y = parseFloat(prompt('y?'));
var max;
if(x > y) {
  max = x;
} else {
  max = y;
}
document.write('largest = ');
document.write(max);
```

if 文ではかっこ内の条件 (ここでは `x > y`) の成否 (YES/NO) によって分岐が起こり、YES なら 1 番目、NO なら 2 番目の枝の中を実行します。それぞれの枝の中は字下げして書いてありますが、これは「プログラムを見やすくするための習慣」です (見にくいと間違えますから、必ず守りましょう)。

ところで、if 文で「NO の時にやること」が無い場合は、else 以降は書かなくて構いません。この形を使った、プログラムの別バージョンを示します (枝が短ければこのように 1 行に書いて構いません)。

```
var x = parseFloat(prompt('x?'));
var max = x;
var y = parseFloat(prompt('y?'));
if(y > max) { max = y; }
document.write('largest = ');
document.write(max);
```

上の 2 つのプログラムはやることは同じで、どちらも「正解」ですが、処理の進み方や見た目は違います。このように、ソフトウェアは動作や効果が同じであっても、さまざまな「正解」があり、さまざまな書き方が可能です。では、どの書き方がいいでしょう? それは、プログラムを書く人のセンスで、「この方が分かりやすい」「将来直すときに直しやすい」などの見通しに基づいて選びます。皆様なら、上の 2 つのどちらを選びますか?

JavaScript には繰り返しを記述する構文などもありますが、本科目はプログラミング入門ではないので今回は枝分かれの説明までにします。なお、「式」についても説明していませんが、「+」「- 」以外に掛け算「*」、割り算「/」、剰余「%」などの演算子を書くこと、1 行に書くために適宜「(…)」で囲むことなどが必要です。

演習 1 JavaScript の例題を動かしてみなさい。動いたら、次の小問から 1 つ以上やってみなさい。

表 2: JavaScript の主要な構文

書き方	説明
`式;`	単に式を計算する (関数呼び出しなど)
`変数 = 式;`	式の値を計算して変数に入れる
`var 変数 = 式;`	変数を定義し、初期値を入れる

`if(条件) {`
`  文…`
`}`
	条件が成り立った時だけ「文…」を実行する

`if(条件) {`
`  文 1…`
`} else {`
`  文 2…`
`}`
	条件が成り立った時「文 1…」、そうでない時「文 2…」を実行する

`if(条件 1) {`
`  文 1…`
`} else if(条件 2) {`
`  文 2…`
`} else {`
`  文 N…`
`}`
	条件 1 が成り立った時「文 1…」、そうでなく条件 2 が成り立った時「文 2…」、どれも成り立たなかった時「文 N…」を実行する (条件と文はいくつでも追加できる)

a. 数値を 1 つ読み込み、それが正/負/零のとき「plus」「minus」「zero」と表示するプログラムを作る。2 つの違う書き方のバージョンを作成できるとなおよい。

b. 3 つの値を読み込み、その最大を表示するプログラムを作る。2 つの違う書き方のものを作成できるとなおよい。

c. 3 つの値を読み込み、その最大を表示するが、ただし 2 つ以上の値が等しい場合は代わりに「error」と表示するプログラム。2 つの違う書き方のものを作成できるとなおよい。

3 ソフトウェア開発

3.1 ソフトウェア開発とは

ここまでは「プログラム」を書いてきましたが、我々が普段耳にする用語はどちらかと言えば「ソフトウェア」です。**ソフトウェア** (software) とは、コンピュータ上で何らかの仕事をこなすために必要とされるプログラム、データ、手引き書など、ハードウェア以外のすべての部分を指します。そしてハードまで含めた動くもの全体が**システム** (system) です。

たとえば Word などを考えてみても、動作するプログラムに加えて、そのプログラムの画面に表示されるさまざまなアイコン (絵)、多様な文書のテンプレート等も必要なわけです。

ソフトウェアを作る際は、やみくもにプログラムを書くという方法ではうまく行きません。ここまでの体験でも分かるように、プログラムは非常に緻密であり、どこか少しでも間違っていたら、それだけで全体として動作しないこともたびたびです。数行のプログラムでも正しく作るのは大変なのに、何万行ものプログラムを作って動かすのが簡単なわけはありません。

これには、多くの理由があります。まず、製品となるソフトウェアを作る時には、進め方を文書化し、設計も文書化し、コードも文書化し、テスト計画を立ててテストし、等の付帯作業が多数必要です。そのため、単にプログラムを書くだけなら 1 日に数十行・数百行が書ける人でも、ソフトウェア開発プロジェクトとして書くと「開発者 1 日あたり数行」になることも多いのです。次にそうなると、数万行のプログラムを 1 年程度で完成させるには、開発者が 100 人以上必要になります。人が多くな

ると、それらの人の間で意思疎通する手間 (ミーティング・会議・連絡の手間) が大きくなり、その分だけコードを書く時間が削られます。これも「開発者 1 日あたり数行」の原因の 1 つです。

このように、ソフトウェア開発で「人が多くなる」ことは多くのマイナスをもたらします。しかし、ソフトウェア開発に詳しくない人には、それが分かりません。たとえば、100 人でソフトの残りを仕上げるのに 2 か月掛かるとします。しかしそれでは期限に遅れるとなると、ではもう 100 人増員して 200 人にすれば 1 か月で仕上がるとか思うわけです。

しかしそれは勘違いで、100 人増員したら、その人たちがいかに優秀でも、(1) これまでのメンバーが新しい人に情報を伝えて仕事ができるようにする手間 (教える側・教わる側とも)、(2) 人数増加による意志疎通の手間の増大により、ソフトウェアの完成時期はかえって伸びます。この「遅れているプロジェクトに追加人員を投入すると遅れは一層ひどくなる」という知見は、フレデリック・P・ブルックスという人が最初に指摘したことから「ブルックスの法則」と呼ばれます。

3.2 要求仕様の問題

そもそも一番最初に、「どのようなプログラムを作る」ということを決めることがまず大変です。このことを決めたものを**要求仕様** (requirements specification) と呼びますが、世の中のソフトウェア開発プロジェクトの多くは要求仕様がきちんと決まっていなかったためにトラブルに陥っています。

たとえば、次の要求仕様を見てください。

> 2 つの数値を入力し、その大きい方を打ち出す。…(A)

これが先に動かしたプログラム (2 バージョンありましたね) の元となる要求仕様だったとします。プログラムの片方を再掲します (動作はどっちでも同じです)。

```
var x = parseFloat(prompt('x?'));
var y = parseFloat(prompt('y?'));
var max;
if(x > y) {
  max = x;
} else {
  max = y;
}
document.write('largest = ');
document.write(max);
```

このプログラムは完璧ですか? 要求仕様に合致していますか? 当然じゃないか、と思われるかも知れませんが、次のことを考えてみてください。

- 要求仕様 (A) は、2 つの数値が同じだったときの動作について規定していない。

ということは、どうすればいいのでしょうか。2 つの数値が同じだったらまずいので、同じ場合をチェックしてエラーメッセージを出しますか? 待ってください。勝手にそんなことを決めてはいけません。お客さんに確認したところ、要求仕様を次のように改訂したものとします。

> 2 つの数値を入力し、その大きい方を打ち出す。2 つの数値が同じである場合は、その数値を打ち出す。…(B)

これだったら、先のプログラムで完璧です。早まって直さなくてよかったですね。でも次のようになるかも知れません。

> 2 つの数値を入力し、その大きい方を打ち出す。2 つの数値が同じである場合は、何が出力されてもよいものとする。…(C)

このプログラムがもっと大きなシステムの一部であって、値が同じだったら別の場所で別の処理をするので、このプログラムの出力は気にしないという場合はこのようになります。これも先のプログラムで OK です。では次の場合はどうでしょうか。

> 2 つの数値を入力し、その大きい方を打ち出す。2 つの数値が同じであることは決してないはずである。…(D)

決してないのだから、考えないでよい? それは違います。決してないことが起きていると分かったら、それは「おかしいことが起きている」と教えてあげるのが筋です。もっとも、本当にそうした方がよいかどうかは再度お客さんの意向を確認した方が安全ですが。このように、ごく簡単なプログラムでも「要求仕様をきちんと決める」「決めた要求仕様に正しく合致するコードを作る」ことはとても大変なのです。

3.3 テストとテストケース exam

作成したコードには大抵間違いが含まれているので (人間は必ず間違いを犯します)、この間違いを防いだり発見して修正する作業が必要です。その 1 つの方法は、コードをよく見直すことです。当り前みたいですが、ソフトウェア開発プロセスの中に複数人で集まってコードを見直す作業が必ず必要です。これには**レビュー** (review)、**インスペクション** (inspection)、**ウォークスルー** (walkthrough) など複数の呼び方があります。

ですが、普通まずやるのは、実際にコードを走らせて正しく動作するか調べることですよね? これを**テスト** (test) と呼びます。そして、テストのためにプログラムに与える入力と、その入力に対応して出力されるであろう「正しい」結果を組にしたものを**テストケース** (test case) と言います。テストケースでは、プログラムを動かしてみる前に正解を用意しておくことがポイントです (間違った結果を出すプログラムでも、自分で書いたプログラムの結果を見たら、それで合っていると勘違いしがちなので)。

では、どのようにテストしますか? 思い付いた入力を与えて動かしてみて、結果が想定とあっているかをいく通りかやる? 最初のテストとしてはそれもいいですが、それだけでは全然不足です。実際に開発プロセスの中で行われるテストとしては、次のようなものがあります (これでも一部です)。

- **スモークテスト** (smoke test) — コードがとりあえず一通り実行されることを確かめる。上で「最初のテスト」と呼んだもの。[1]
- **機能テスト** (blackbox test) — コードの「仕様」に基づき、さまざまな可能性をひととおり網羅するようなテストケースを用意し、結果を確認するもの。
- **構造テスト** (whitebox test) — コードの「構造」に基づき、分岐などの境界の条件を調べるテストケースを作って確認するもの。この中に、「プログラム中のできるだけ多くの箇所を実行する」**被覆テスト** (coverage test) と呼ばれるものがある。[2]
- **ランダムテスト** (random test) — 乱数などで生成した値を与え、不具合が起きないか調べる。
- **回帰テスト** (regression test) — これまでに確認したテストケースをすべて保管しておき、プログラムを修正したあと再度すべて実行しなおして修正によって壊れた箇所がないか確認する。

近年では開発プロセスの一種として「テストファースト」つまり、コードを 1 行でも書くよりもまず先にテストを記述するという流儀も提案されており、それなりに使われています。

また、テストケースを予め格納しておき、自動的にテストを実行して結果を報告してくれるツール (自動テストツール) の使用も一般的です。ここでは簡単なテスト実行機能を提供してくれる Web ページを用意したので (図 3)、これを用いてテストを行ってみましょう。

[1]電気のスイッチを入れてみて、「煙」が出てこないかどうか確認するという意味でこう呼ばれています。

[2]実際には大きなコードになるとその全ての箇所を実行するというのは不可能です。その場合はカバー率を設定してそれを目標に被覆テストを行います。

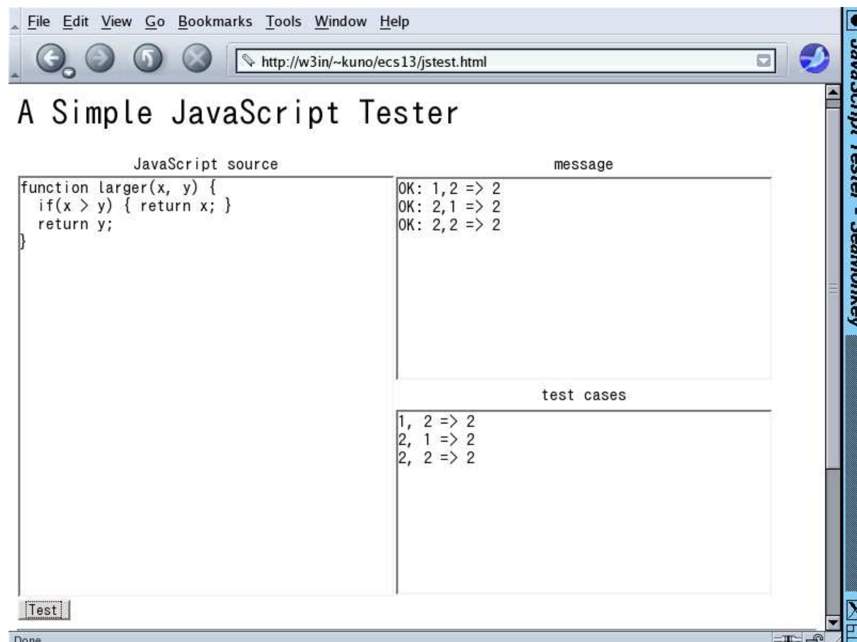

図 3: JavaScript のテスト機能つきページ

このページでは、テスト対象のコードを**関数** (function) という単位でパッケージして用意します。ここでいう関数とは数学の関数とは違い、単に「まとまったコードに名前をつけて呼び出せる」もので、次の形を取ります。

```
function 関数名(変数名, ...) {
  文…
}
```

「関数名」はコードに対してつける任意の名前で、その機能を表す分かりやすい名前をつけます。変数名の並びは、この関数に対する「パラメタ」であり、ここに処理してもらうデータを渡します。そして、「文…」の部分はこれまで同様に処理を書きますが、最後に結果を「`return 式;`」という文によって返します。具体例ですが、先の「より大きい値」を関数にした「渡されたパラメタのうち大きい値を返す」関数 `larger()` は次のようになります (書き方は色々有り得ますので、あくまで一例です)。これをテスト対象とします。

```
function larger(x, y) {
  if(x > y) { return x; }
  return y;
}
```

次にテストケースですが、パラメタとして渡す 2 つの数の前者が大きい場合、後者が大きい場合、両方とも同じ場合の 3 つの場合をテストすれば十分ですね (理由を考えてみること)。そこで、テストケースとして次の 3 つを記述します (各ケースでは「`=>`」の左にパラメタ、右に想定出力を指定します— この書き方はこのツール用に決めたものです)。

```
1, 2 => 2
2, 1 => 2
2, 2 => 2
```

これらを入力した状態で Test ボタンを押すと、テストが実行され、3 つとものケースをパスする (実行結果とテストケースの想定結果が一致する) ことが確認されます。

演習 2 `larger()` のテストを自分でも実行してみなさい。「正しくない」テストケースを設定したらNG が出ることも確認しなさい。終わったら、次の 3 問から 1 つ以上やってみてください。必ず「なぜこのテストケースで十分と考えるか」を記述すること。関数のコードの正しいものを完成させせることは望ましいですが必須ではありません (テストケースを体験することが目的)。

a. 「3 つの数をパラメタとして受け取り、最も大きいものを返すが、ただし 3 つの数値がすべて同じなら-3、2 つだけが互いに同じなら-2 を返す」という関数を書くものとします。テストケースを書きなさい。その後で、関数の実装を書き、テストしなさい。自力で書くのがつらい人向けにバグあり版をつけておきます。

```
function largest(a, b, c) {
  if(a == b && b == c) { return -3; }
  var max = a;
  if(max < b) { max = b; }
  if(max < c) { max = c; }
  return max;
}
```

b. 「3 つの数をパラメタとして受け取り、小さい順 (等しいものがある場合も含めるので正確には大きくない順) に並べて返す」という関数を書くものとします (並びは「`[1, 2, 3]`」のようにかぎかっこで囲んだ数値のリストとして表現します)。テストケースを書きなさい。その後で、関数の実装を書き、テストしなさい。自力で書くのがつらい人向けにバグあり版をつけておきます。

```
function order3(a, b, c) {
  if(a > b) { var x = a; a = b; b = x; }
  if(b > c) { var x = b; b = c; c = x; }
  return [a, b, c];
}
```

c. 「h 時 m 分から、h_1 時間 m_1 分たったら、何時何分かを計算し、答えを (時と分の並びとして) 返す関数を書くものとします (24 時制とし、23:59 を過ぎたら翌日の 0 時になります)。」ただし、$0 \leq h, h_1 < 24$, $0 \leq m, m_1 < 60$ とします。テストケースを書きなさい。その後で、関数の実装を書き、テストしなさい。自力で書くのがつらい人向けにバグあり版をつけておきます。

```
function etime(h, m, h1, m1) {
  h = h + h1;
  if(h > 23) { h = h - 24; }
  m = m + m1;
  if(m > 59) { m = m - 60; }
  return [h, m];
}
```

3.4 ソフトウェア開発プロセス

要求仕様の獲得から始めて、最後にソフトウェアがテストを通過し納品されるまでの過程のことを**ソフトウェア開発プロセス** (software development process) と呼び、さまざまな流儀のやり方があります。一番古典的でしかし現在でも多く使われているのが、**ウォーターフォール** (waterfall) 型のプロセスです。これは、分析→設計→製造→テスト、というふうにステップが決まっており、各ステップとも一旦終わったら後戻りしない、ということが原則です (図 4)。それぞれ前の段階が終わってな

かったら次の段階の作業はできない、というのは言われてみれば当然なので、このモデルは自然のように思えます。

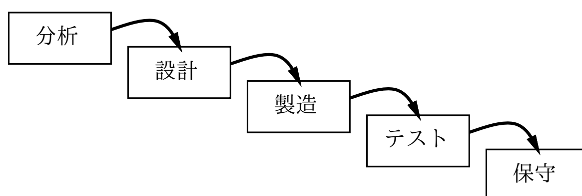

図 4: ウォーターフォール型プロセス

でも実際には、要求を確定してこれ以上変更しない、ということになっていたのに、後からお客さんの無理な要求が追加されたとか、途中までやってみたらできないことが明らかになったとかで、手戻りが発生しがちです。手戻りが発生すると、その分だけスケジュールが遅れてあとの方で大変になります。では一切変更を認めないのがいいのでしょうか? そうすると、最後に製品を納品した後で「肝心なところの機能が要求から落ちていて役に立たないと分かった」というふうな失敗に至ります。現実にはこの両方の失敗が組合わさって起きるというのが、ソフトウェア開発の実情です。

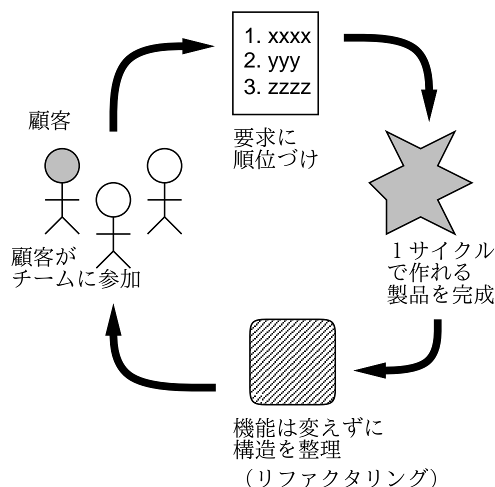

図 5: アジャイル型プロセス

そこで現在では、これとは別の**アジャイル型** (agile) のプロセスが普及してきました。アジャイル型では、要求仕様や設計に時間を掛ける代わりに、顧客にまず要求を聞き、一定期間 (数週間程度) で作れる箇所を選んでそこをまず開発・稼働し、コードをきれいに整理した後、また次にやることを決めて開発し、のように多数のサイクルを繰り返して開発を進めます (図 5)。これなら、後からの要求追加も容易ですし、動いているリリースを見れば顧客も何が足りないか確認できます。[3]

本日の課題 8A

本日の課題は「演習 1」「演習 2」に含まれる小問 (合計で 6 個) の中から 1 つ以上を選択し、結果をレポートとして報告して頂くことです。LMS の「レポート# 8」の入力欄に直接入力してください (別途作成したものをコピーペーストで貼っても構いません)。以下の内容がこの順に含まれるようにしてください。

[3]その一方で、じっくり設計して作るわけではないので、大きなものを作るときに設計が練れていなくて途中で行き詰まる危険があります。

- 冒頭に題名「コンピュータリテラシレポート＃8」、学籍番号、氏名、提出日付を書く。(グループでやったものはグループのメンバー全員の氏名も別途書く。)
- 課題の再掲を書く (どんな課題であるかをレポートを読む人が分かる程度に要約する)。
- レポート本体の内容 (やったこととその結果) を書く。必ず自分が書いたコードやテストケースを掲載すること。
- 考察 (課題をやった結果自分が新たに分かったことや考えたこと) を書く。
- 以下のアンケートに対する回答。

  Q1. JavaScript のような高水準言語と前にやった機械語 (アセンブリ言語) を比較してどのように感じますか。

  Q2. テストケースを書くなどのソフトウェア開発の考え方についてどのように思いましたか。

  Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

なお、課題はグループでやって構いません。その場合も、(メンバー氏名を明記した上で) レポートは必ず各自で執筆してください。レポート文面が同一 (コピー) と認められた場合は同一であると認めた全員について点数にペナルティを科すことがあります。