

# アジャイル概要

2020 年 5 月 18 日～5 月 22 日

(研修テキスト)



Web Applications  
Windows Store Apps Using C#

2020 年 5 月 8 日



Windows Azure Developer  
Windows Phone Developer  
Windows® Developer 4  
Enterprise Application Developer 3.5

ニュートラル株式会社  
林 広宣

<http://www.neut.co.jp/>  
[hayashi.hironori@neut.co.jp](mailto:hayashi.hironori@neut.co.jp)

複製を禁ず

(このページが、表紙の裏になるように印刷をお願いします。)

ご意見・ご質問は下記まで

ニュートラル株式会社 林広宣

## 目 次

アジャイル概要 .....	1
目 次 .....	3
第1章. アジャイル概要 .....	5
1.1. アジャイル概要 .....	6
1.2. アジャイル成功のために必要な外的要因 .....	7
1.3. アジャイル成功のために必要な内的要因 .....	8
1.4. 必要なテクニカルスキル .....	11
第2章. スクラム概要 .....	21
2.1. スクラムとは .....	22
2.2. スクラムの特色 .....	23
2.3. スクラムチーム .....	25
第3章. スプリント（イテレーション） .....	29
3.1. スプリント（イテレーション）とは .....	30
3.2. プロジェクト計画（リリース計画） .....	32
3.3. リリース計画ミーティング（プロジェクト計画ミーティング） .....	36
3.4. スプリント計画ミーティング（スプリント・プランニング） .....	37
3.5. デイリー・スクラム（デイリー・スタンドアップ） .....	40
3.6. 開発作業 .....	42
3.7. スプリント・レビュー .....	48
3.8. レトロスペクト（スプリント・レトロスペクティブ） .....	50
3.9. スプリント0（ゼロ）とリリース・スプリント .....	52
第4章. ツールと概念 .....	53
4.1. プロダクト・バックログ関連 .....	54
4.2. スプリント・バックログ関連 .....	58
4.3. 問題点リストとリスク一覧 .....	62
4.4. スプリントの評価とPDCAサイクル .....	66
4.5. 振り返り用ツール（KPT） .....	70

(空白ページ)

## 第1章. アジャイル概要

## 1.1. アジャイル概要

### 1.1.1.説明

1990年代に、スクラム (Scrum)、Extreme Programming (XP)、Adaptive などの反復型の手法が広がりつつあった。

2001年、ユタ州のスノーバード (Snowbird) で、それらの手法を実践する 17 名が集い「アジャイルソフトウェア開発宣言 (Manifesto for Agile Software Development または Agile Manifesto)」が発表された。これがアジャイルという言葉の起源である。

アジャイルなどの反復手法は、どの手法も非常に似ているが、スクラムではプロジェクトの進め方(フレームワーク)に重点がおかれ、XPでPは技術的なノウハウに重点が置かれている。

### 1.1.2.アジャイルの成功の条件と失敗の要因

#### 1. 外的要因

- ① 中間層が厚い組織のアジャイルに対する理解

#### 2. 内的要因

- ① 心理的安全性の確保
- ② アジャイルフレームワークの実践
- ③ 十分なテクニカルスキル

外的要因は我々 (チームメンバー) としてはどうしようもできない部分があるが、内的要因はチームメンバーの取り組み方や話し合いなどの努力によって改善できる部分である。

## 1.2. アジャイル成功のために必要な外的要因

### 1.2.1.中間層が厚い組織のアジャイルに対する理解

中間層が厚い組織は、業務命令と管理が主体の組織運営や事前の計画を重視する。一方、アジャイルは、フラットな組織で、常に計画を改善していく考え方をする。その他、アジャイルと従来の組織運営は、多くの点で考え方が異なる。

- ・プロジェクトの予算の考え方
- ・個々人の評価（査定）の方法
- ・仕事の進め方

ロバート・マーチンの **Clean Agile** (2019) によると、中間層が厚い組織は、アジャイルに移行しようと努力しているにもかかわらず、うまくいっていない場合がほとんどであると述べている。

また、中間層が厚い組織でも、アジャイルに成功している場合があり、それは（１）社長や事業部長直轄のチームを作る場合か、（２）アジャイルだということは内緒にして実践している場合ぐらいに限られると述べている。

また、組織を少しずつアジャイルに移行する方法もうまくいかない場合が多く、最初からアジャイルを行う新規のチームを編成するほうがうまくいくと述べている。

従って、アジャイルがうまくいくかどうかは、組織の中間層のアジャイルに対する理解の度合いにかかっている。

## 1.3. アジャイル成功のために必要な内的要因

### 1.3.1.心理的安全性の確保

心理的安全性 (Psychological Safety) とは、「チーム内で安心して仕事ができる状態」を指す。

2014 年の Google の大規模な組織内調査 Project Aristotle で有名になった言葉。

「生産性の高いチームに共通の特色は何か」を特定することを目的に行われた調査。

全世界の Google の何万人という社員を対象に行われた。

この調査は 2000 年代の半ばから始められ、最初の調査では、チームの生産性とチームの仕事ぶりの間に明確な相関関係は認められなかった。

つまり、正反対の雰囲気、正反対の仕事ぶりのチームがあったとして、それぞれどちらも、生産性の高いチームもあれば低いチームもあった。

そこで、心理学者なども加えより広い観点から 2 回目の調査を行ったところ、ようやく相関関係が明らかになった。

それによると、「生産性が高く、品質の高い仕事ができているチームの特色は何か」の一番の要因は「心理的安全性」であると結論づけられた。

「どんな発言をしても否定されない」「発言が批判されない」「このチームで仕事をするのが楽しい」「相互に信頼し合っている」というような状態に置かれているチームが、実は、最も効率的に仕事ができることがわかった。

一方、「計画がしっかりしている」「個人の役割が与えられている」「目標が明確である」という要因も重要ではあるが、心理的安全性にはおよばないことがわかった。

言い換えると、このような要因が高くて、心理的安全性が低いチームは生産性が低いことがわかった。

また、心理的安全性が高いチームは、「静かなチームか、賑やかなチームか」「リーダーシップが高いチームか低いチームか」という要因とは無関係に、効率の良い仕事ができていることがわかった。

<https://rework.withgoogle.com/jp/guides/understanding-team-effectiveness/steps/identify-dynamics-of-effective-teams/>

アジャイルではチームを「自己組織化 (Self-organization)」することを重視し、個人の責任にせずチームの責任で仕事を進めていくという考え方をするが、その前提となるのがこの心理的安全性である。



### 1.3.2.アジャイルフレームワークの実践

心理的安全性はどんな種類の組織にも必要であるが、プロジェクト（目的と期限が限られた業務）を完遂させるためには、さらに必要な要素がある。それがフレームワークである。

アジャイル（特にスクラム）は、下記の原則を実践できるフレームワークを備えている。広義では、下記の原則や実践のことをアジャイルと呼ぶ場合がある。

- ① よくない兆候を早期に発見し、直ちに軌道修正する。
- ② 自己組織化されたチームで作業を行う。
- ③ 個人に責任を割り当てない。（チーム全員で責任を持つ）
- ④ 時間を区切って反復する。
- ⑤ 価値に基づいて優先順位をつける。

上記の原則は、1980年代後半からの、特にシステム開発を行うプロジェクトにおいて、うまくいった事例（ベストプラクティス）の「やり方」をまとめたものである。

「心理的安全性」が人間の心の内部の要因に注目したものだとなれば、上記の原則は「プロジェクトに取り組む思想」や「具体的な行動規範」に注目したものである。

従って、上記の原則が守られていないと、そもそもアジャイルを実践していると言えない。

また、上記のそれぞれの原則を実践するために、具体的にどのような行動をしたらよいかが決められている。例えば、

1. よくない兆候を早期に発見し、直ちに軌道修正する。  
デイリースタンドアップや、イテレーションごとのレトロスペクティブ（振り返り）というイベント。  
KPT、YWT、Fun-Done-Learn、LAMDAなどのツールの活用。
2. 自己組織化されたチームで作業を行う。  
プロジェクトマネージャーとかチームリーダーという立場は存在しない。  
全員で計画を立て、全員が協力して遂行し、全員で振り返って、軌道修正していく。  
そのために、イテレーションやイテレーション内の「計画ミーティング」「レトロスペクティブ（振り返り）」というイベントがある。
3. 個人に責任を割り当てない。（チーム全員で責任を持つ）  
担当を決めず、全員がどの仕事もできるようにもっていく。  
ペアプログラミング、モブプログラミングなどの実践。
4. 時間を区切って反復する。  
イテレーション（スプリント）を、例えば1週間と決めたら、その期間を守って実践する。  
また、各イテレーション内の「計画ミーティング」「レトロスペクティブ（振り返り）」などのイベントも、決められた時間を超えて行わないようにする。
5. 価値に基づいて優先順位をつける。  
優先順位の高い作業が終わる前に、それより低い優先順位のものに着手しないようにする。  
機能やタスクをWBSに分解してそれぞれを個人に割り当て、個々人の中で優先順位を付けるという方法は採用しない。  
優先順位の高い順に1つずつ全員でとりくみ、それが終わったら次の優先順位に着手するという考え方をする。

### 1.3.3.十分なテクニカルスキル

組織の中間層からアジャイルが認知されている、心理的安全性が確保されている、アジャイルの実践もできている、という3つの要素がそろったら、外見上は順調にアジャイルを進めているようにみえるが、必ずしも、予定通りに品質の確保された製品が開発されるとは限らない。

チームメンバーに技術力が足りないと、良い製品を納期までに完成させられない。

この場合の「技術力」とは、「プログラミング言語を理解している」、「ネットワークやデータベースの知識があり、そのためのソースコードも記述できる」という程度の知識や技術力では足りない。

それらを満たした上で、さらに下記の知識や実践力が必要である。  
詳細は、次のページ以降で述べる。

- ① 継続的インテグレーションの環境を整えられる。
- ② 自動ユニットテストやテスト駆動開発ができる。
- ③ アプリケーションアーキテクチャが正しく設計できる（理解できる）。
- ④ ペアプログラミングやモブプログラミングを実践できる。
- ⑤ リファクタリングやデザインパターンなどの技術的知識を持っている。

プロジェクト開始前の時点では、全員が上記のスキルを持っているとは限らない。  
しかし、少なくとも一人は上記の技術のどれかを実践でき、チーム全体では上記のすべての技術を実践できる必要がある。

つまり、特定の個人に関していえば、上記の技術に関して、やり方を知らないものがあったてもよい。

そして、プロジェクトが進むにつれて、全員が上記のすべての技術の理解が深まり、実践できるようにもっていくことが重要である。

つまり、特定の個人に関しても、上記すべての技術を実践できるようにすべきである。

## 1.4. 必要なテクニカルスキル

### 1.4.1.継続的インテグレーションの環境を整えられる。

バージョン管理システムや、(ローカルではなく) サーバー上での自動ビルドや自動テスト、自動デプロイの整備ができること。

また、その環境に適した開発作業が行えること。

具体的な製品名としては、Sub Version、Git、TFS (Team Foundation Server)、Azure DevOps、Jenkinsなどが適切に構成・使用できるということである。

これができることのメリットは、次の通りとなる。

1. 開発初期の段階から、常に、顧客に引き渡せる状態の製品が準備できている。(もちろん開発初期ではほとんどの機能が未実装だが、いくつかの機能は 100%完成している)
2. 常にコンパイルエラーの無い状態を保つことができる。
3. 常に一定の品質を持った製品が完成している。(繰り返しになるが、機能は未実装のものがある)
4. 他の開発者のコードをマージした結果、不整合があれば直ちに知らせが届き、短時間で整合性の取れた状態に修正できる。

まとめると、コードのメンテナンスに関わる時間を削減でき、コーディングの時間を増やすことができることになる。

これは、結果的に品質の高いものを短時間で開発でき、いつでもリリースできるようになるということである。

### 1.4.2.自動ユニットテストやテスト駆動開発ができる。

#### 1. 自動テストのメリット

自動テストの作成は、開発初期の段階では進捗を遅らせる要因になる場合があるが、開発の中期以降では、自動テストを作成しない場合と比べ、何倍もの開発効率を達成させる原動力となる。

また、そもそもテストなので、開発のどの段階でも、一定の水準を満たした品質を保証できる。  
(自動テストを作成しないと、開発の後期になればなるほど、表面化しない品質の低下が増える)

自動ユニットテストを作成しても、手動のテストフェーズを省略できるわけではないが、手戻りが圧倒的に減る。

#### 2. テスト駆動開発の理解

慣れていない開発者が、テスト駆動開発 (TDD: Test Driven Development) を実践することは非常に困難であり、本当の意味でテスト駆動開発ができる人は、日本ではほんの一握りしかいないとも言われている。

従って、全てのメソッド (関数) をテスト駆動開発で作成する必要はない。

ただし、テスト駆動開発の理念や方法論は知っておくべきである。  
例えば、テスト駆動開発は設計手法であり、テスト手法ではない。  
また、テストを記述することで、バグが少なくメンテナンスしやすいコードとなる。

#### 3. 外部環境の接続を伴うテストは別途行う

外部環境の接続を伴う機能とは、データベース接続やネットワーク接続などのことである。フォームやウィンドウなどのユーザーインターフェースも外部との接続を伴う機能である。

ローカルの自動テストでは、純粹にビジネスロジックをテストするだけにする。  
従って、データベース接続やネットワーク接続の部分は、モック (Mock) やフェイク (Fake) を作成し、それらを使ってテストすることになる。

モックやフェイクとは、物理的な外部環境とは接続せず「接続したふり」をするオブジェクトのことである。  
空の処理で実装したり、テストに必要な最低限の (オンメモリの) データを返すオブジェクトである。  
前者をモック、後者をフェイクと呼ぶことが多い。

物理データベースや実際のネットワーク接続でテストしてしまうと、ローカルの自動テストだけで数時間かかることもある。

もちろん、物理的な環境と接続するテストも必要であるが、それは、ローカルの自動テストとは別に作成し、夜間バッチやテスト専用 PC などで行う。  
一般的には、前述の継続的インテグレーションの環境で行われる。  
(なお、外部接続のテストが短時間で終われば、ローカルで行ってもよい)

#### 4. テストのカバレッジ

ローカルでの自動テスト（広義のユニットテスト）は、「狭義のユニットテスト」と「インテグレーションテスト」に分かれる。

狭義のユニットテストは個々のメソッドに対する自動テストであり、インテグレーションテストとは、ユースケースに対応するメソッドを自動テストするものである。  
ユースケースとは、ユーザーからみた機能や仕様と理解してもよい。

そして、現在では、インテグレーションテストを必ず実装し、そのために必要に応じて狭義のユニットテストを実装するというのが現実的であると言われている。

例えば、「図書の貸出システム」というアプリケーションがあったとして、考えられるユースケースは「貸出」と「返却」の2つである。

従って、「貸出」は **Lend** メソッドで「返却」は **Return** メソッドだとしたら、その2つのメソッドの自動テストがインテグレーションテストとなる。

システム全体としては数十個のメソッドが使用されていても、インテグレーションテストを行うのは、**Lend** と **Return** の2つだけでよい。

一方、正しく貸し出しできているかを確認するためには、**FindByBookNo** というメソッドを呼び出さなければテストコードを記述できないとする。

従って、**FindByBookNo** が正しく動作している保証がないと、**Lend** メソッドのテストも保証できないことになる。

この場合、**FindByBookNo** のテストコードを記述することが「狭義のユニットテスト」となる。

よって、「貸出」については、**Lend** メソッドと **FindByBookNo** メソッドのテストコードを記述すれば十分である、というのが最近のテスト手法の考え方である。

#### 5. ソースコード修正時

プロジェクトの中盤以降になるとメソッドを統合したり、分割したりといったソースコードの修正がリスクを伴う。

例えば、修正する前までは仕様を満たしていたのに修正後は一部の使用を満たさなくなったが、それを発見するのが遅れた。

その他に、バグを生み出したが、どこを直せばよいか発見するのに時間がかかった、などである。

このような場合でも、自動テストが記述されていると安心である。

当然、プロダクトコードの側のメソッド統合や分割を行うわけなので、テストコード側もある程度の修正が必要だが、テストケース（ユースケースといってもよい）自体に変更はないので、テストを実行して失敗が報告されたら、その間に修正したプロダクトコード側かテストコード側かのどちらかの修正が間違っていることになる。

つまり、安心してソースコードを修正でき、正しさの証明にもなる。

### 1.4.3.アプリケーションアーキテクチャーが正しく設計できる。

レイヤー型アーキテクチャーは従来の手続き型手法向けのアーキテクチャーであり、オブジェクト指向に適したアーキテクチャーではない。

オブジェクト指向に適したアーキテクチャーは、同心円型アーキテクチャーであると言われている。これは、「オニオンアーキテクチャー」「ヘキサゴナルアーキテクチャー」「クリーンアーキテクチャー」などと呼ばれている。

これらのアーキテクチャーで設計するためには、最低限「依存性の逆転 (DI: Dependency Inversion)」あるいは「制御の逆転 (IoC: Inversion of Control)」を理解している必要がある。

そして、それに関連して「依存性の注入 (これも DI と略す。ただし Dependency Injection の略)」が適切にできるアーキテクチャーにすることが重要である。

具体的な例をあげる。あるビジネスオブジェクト **A** があるとして、そのオブジェクトは最終的にデータベース処理を行う別のオブジェクト **B** を呼び出しているとする。

一方で、ローカルの自動テストでは、データベースをアクセスするのではなく、モック **C** を使ってテストを記述すべきである。

そういう場合、レイヤー型アーキテクチャーでは、**A** の中で **B** を呼び出している部分を、**C** の呼び出しに書き換えなければならない。

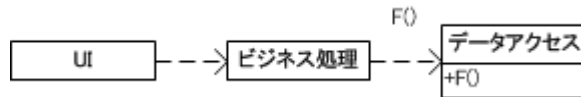
これは、ビジネスロジック内のコード **A** を修正することになってしまう。これは大変によくないコードである。

そこで、ユーザーインターフェース (本番) の側からは **B** のオブジェクトを指定して **A** を呼び出し、テストプログラムの側からは **C** のオブジェクトを指定して **A** を呼び出すようにすれば、**A** のコードを修正する必要はなくなる。

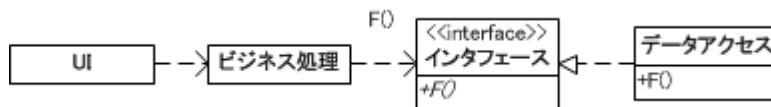
これを実現するためには、依存性の逆転を備えたアーキテクチャーにする必要がある。

## 1. 従来のレイヤー型アーキテクチャー

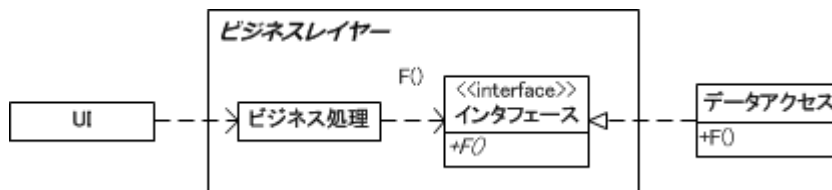
ビジネス処理がデータアクセス処理に依存している。  
 データアクセス部を入れ替えようと思ったら、ビジネス処理のコードを修正しなければならない。



## 2. 依存性を逆転させたアーキテクチャー

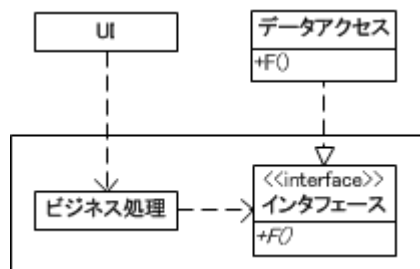


インタフェースはビジネス処理と同じレイヤーに属する。(下図)



## 3. 上図で、UI とデータアクセス部を同じ側にもって来た図

ビジネスレイヤーは、他のどのレイヤーにも依存しなくなった。



#### 1.4.4.ペアプログラミングやモブプログラミングを実践できる。

ペアプログラミングとは2人で1つのコードをプログラムすることで、モブプログラミングは3人以上で行うことである。

よって、ペアプログラミングはモブプログラミングの特殊な形態（人数が2人だけの形態）ということができる。

どちらも、1人がキーボードに向かってコードを記述する。その他の者はアドバイスを与えたり、逆にコードの書き方を勉強したりする。

キーボードに向かうものは10分から15分で、別の者に交代する。

そして、30分から1時間程度で、ペアプログラミングやモブプログラミングを終了する。

（もちろん、もっと短くても長くてもよい）

- ・ソースコードレビューをしたことになる。
- ・スキルの高い技術者から初級者へ向かってスキルの継承ができる。
- ・一人だけでは発見できないようなバグを発見できる。（品質の向上が期待できる）

プログラミングだけでなく、設計書の作成、テスト仕様書の作成、データベースの設計など、プログラミング以外の作業に適用することもできる。

統計では、エンジニアの作業時間の15%~30%程度をこのペアプログラミングやモブプログラミングに費やしているチームが、生産性の高いチームだということである。（「レガシーコードからの脱却」より）

また、開発作業をタスクに分解した結果、同一のソースコードを複数の開発者が編集しなければならないような場合にも、この方法が適している。

例えば、優先順位の最も高い作業が、あるフォームの `button1_Click()` というメソッドの処理の記述であるような場合で、かつその作業が完了しないことには、`button2_Click()` や `button3_Click()` の作業に取り掛かりにくいような場合、全員で `button1_Click()` の処理をモブプログラミングで作成する、というようなことがよく行われる。

また、WBSでタスクを分解した場合、最も優先順位の高いタスクから取り掛かるわけだが、そのタスクを複数人のタスクに分解できない場合にも、ペアプログラミングやモブプログラミングを行う。

（なお、アジャイルでは、通常WBS方式は採用せず、ユーザーストーリーとかプロダクトバックログというツールを使って作業を行う。）



この方法は、教育的な効果が絶大である。

開発初期の段階では、「自分はデータベースは得意だが、ネットワークは不得意」とか「テスト駆動開発を行ったことがない」というように、チームメンバーのスキルに偏りがある。

この偏りを少なくし、チーム全体のスキルアップを促すツールがこのペアプログラミングでありモブプログラミングである。

ペアプログラミングは、スキルの高いものとそうでない者のペアで行うと効果がある。  
また、常に異なった組み合わせで行うとよい。

ただし、スキルの高い者どうしで行うと、コーディングの細かな部分で意見が対立したりすることがある。またスキルの低い者どうしで行うと、コードの品質も低くスキルアップにもつながらないことがある。そういうようなペアの組み合わせをする必要はない。(してはいけないというわけではなく、杓子定規にペアプロを適用する必要はない)

#### 1.4.5.リファクタリングやデザインパターンなどの技術的知識を持っている。

一言でいうと「あなたはあとから修正しやすいコードを書いているか」ということになる。

そのためには SOLID の原則を踏まえてコードが書かれている必要がある。

SOLID の原則とは次のようなものである。

##### 1. SRP (Single Responsibility Principle 単一責任の原則)

「1つのモジュールはただ1つの理由だけで変更されるべき」という原則。

なお、SRP の誤解されている説明として「1つのクラスに複数の責務を負わせないこと」というものがあるが、これは SRP とは異なる。

この誤解された内容の実践が悪いわけではないが（むしろ良いことであるが）、SRP の説明ではない。

##### 2. OCP (Open-Closed Principle オープン・クローズの原則 または 開放/閉鎖の原則)

「機能追加に対しては開いて（オープンして）いなければならない、機能変更に対しては閉じて（クローズして）いなければならない」という原則。

これも「オープンしたオブジェクトを使い終わったらクローズしなければならない」と誤解される場合があるが、それとは異なる。

この誤解も実践内容としては正しいが、OCP のことではない。

##### 3. LSP (Liskov Substitution Principle リスコフの置換原則)

「ある操作 F がオブジェクト T に対して成り立つとき、T の派生クラス S（サブクラス）に対しても成り立つべき」という原則。

これは「ポリモーフィズムが成り立つようにサブクラスを設計すべきである」と解釈することができる。

##### 4. ISP (Interface Segregation Principle インタフェース分離の原則)

「あるオブジェクトの操作すべてをユーザーに提供するのではなく、そのユーザーに必要な操作だけを提供すべき」という原則。

これを実現するためには、クラスではなくインタフェースを提供することになるため「インタフェース分離の原則」とも呼ばれる。

##### 5. DIP (Dependency Inversion Principle 依存性逆転の原則)

「抽象は詳細に依存してはならない」という原則。

言い換えると「抽象が詳細に依存しているようなコードは、（依存関係を逆転させて）詳細が抽象に依存するように書き換えるべき」となる。

これは、オブジェクト指向に基づいたアプリケーション設計にとって非常に重要な原則である。

**SOLID** の原則を場面や目的に応じて具体的なソースコードのパターンとして示したものがデザインパターンである。

前節で述べた、**Dependency Injection** もデザインパターンの一つである。

カオス化したコードを、どのようにしたらすっきりしたコード、つまり修正しやすいコードにできるかをイメージするためには、デザインパターンの知識が必須である。

そして、そのカオス化したコードを、目標とするコードに改善していく作業がファクタリングである

(空白ページ)

## 第2章. スクラム概要

## 2.1. スクラムとは

### 2.1.1.概要

スクラム (Scrum) とはアジャイル手法の1つである。

このプリントではスクラムの概要を下記の順に説明する。

1. 特色と留意事項
2. 組織体制
3. 進め方の手順とイベント
4. ツールや概念

### 2.1.2.スクラムの歴史

1986年にハーバード・ビジネス・レビューに掲載された、野中郁次郎氏と竹内弘高氏の共同執筆による論文『The New New Product Development Game』で、自由度の高い日本発の開発手法をラグビーのスクラムに喩えて「Scrum (スクラム)」として紹介した。

この論文に着想を得て、Jeff Sutherland (ジェフ・サザランド) と Ken Schwaber (ケン・シュウェイバー) が、1990年の半ばまでに、ソフトウェア開発手法としてのスクラムを構築した。

スクラムの方法論は、まず理論(規範)があってそれに基づいて開発を進めていくという手法ではなく、産業界でのさまざまなベストプラクティスを観察し、「うまく行く方法」を体系化したものである。

つまり、実証的 (Empirical 経験的) な方法論である。

そのため、Jeff Sutherland と Ken Schwaber は、スクラムを「方法論」とは呼ばず「継続的な改善を可能にするフレームワーク」と呼んでいる。

現在では、ソフトウェア開発にとどまらず、ハードウェア、自動運転車、学校、政府、マーケティング、組織運営など多方面でスクラムが利用されている。

## 2.2. スクラムの特色

### 2.2.1.説明

スクラムが他の方法論と違う点、つまり、スクラムならではの特色は下記の通りである。

1. よくない兆候を早期に発見し、直ちに軌道修正する。  
プロジェクト終盤になって、問題点が顕在化することを防ぐ。
2. 自己組織化されたチームで作業を行う。  
リーダーの指示によって行動するのではなく、チームメンバーが自律的に行動する。
3. 個人に責任を割り当てない。(チーム全員で責任を持つ)  
個人の原因による遅延を防ぎ、メンバー全体のスキルアップという効果が期待できる。
4. 時間を区切って反復する。  
区切りごとに PDCA サイクルを回せるので、作業効率・やり方などをどんどん改善できる。
5. 価値に基づいて優先順位をつける。  
例として、「9割完成しているシステム」を考えてみる。  
「すべての機能が90%完成しているが、100%完成している機能は1つもない」ではなく、  
「10%の機能は未着手だが、優先順位の高い残り9割は100%完成している」という状況が実現できる。  
前者はユーザーにとってまったく使えないものしか提供できないが、後者は、ほぼ実用的なものが提供できる。

### 2.2.2.作業中に常にチェックする事柄（スクラム実施時の留意点）

#### 1. チームがスクラムらしく機能しているか

「一人に責任や作業が集中していないか」「同じ人が同じ分野の作業ばかり行っていないか」  
「時間を延長していないか」「よくない兆候を放置していないか」  
「改善点としてあげられた作業がきちんと実行されているか」などを常にチェックしながら作業を行う。

#### 2. 価値に基づいた優先順位となっているか

「優先順位の高い機能を未完成のまま放置していないか」  
「優先順位を常に見直しているか」など

#### 3. 製品やシステム（機能）の品質が保たれているか

「完了の定義」がきちんとされているか。  
「完了の定義」に漏れはないか。

#### 4. （仕様などの）変更に対して適切に対処しているか

顧客から（仕様や機能の）変更の要請があったときに、あるいは逆に、チーム内で変更の要請が発生したとき、適切に判断されているか。（勝手に拒否したり、受け入れたりしていないか）

#### 5. どんなリスクがあり、どんな対処方法があるか

リスク（将来の不安材料。インフルエンザなどの健康状態、台風や鉄道機関の乱れなどの自然や社会の現象、スキル不足やツールのパフォーマンスなどの技術的な問題など）対処の優先順位がつけられているか。  
リスクに対する予防方法・対処方法が提示されているか。

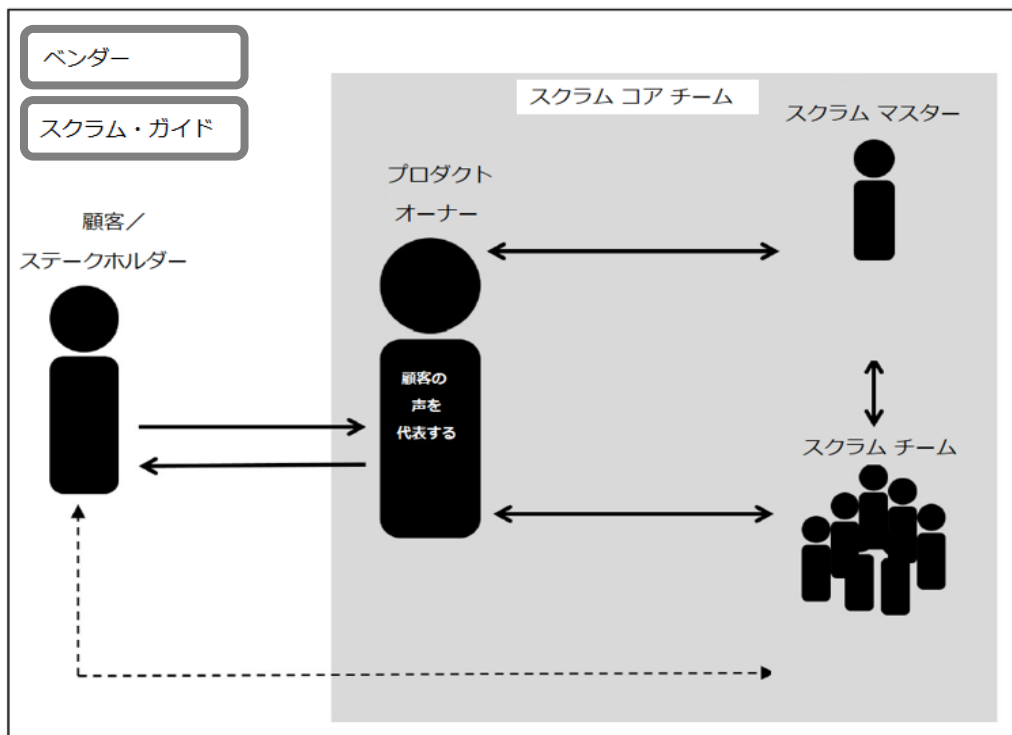


## 2.3. スクラムチーム

### 2.3.1.スクラム・コア・チーム

1 チーム 5 名～10 名程度で構成する。  
大きなプロジェクトで、10 名以上のメンバーが必要な場合は、複数チームに分割する。

1. プロダクト・オーナー（各チーム 1 名）
2. スクラム・マスター（各チーム 1 名）
3. 開発チーム（各チーム 3 名～ 8 名）



### 2.3.2.プロダクト・オーナー

顧客やエンドユーザーの視点で発言する人。

プログラムやシステム設計について詳しい知識は必要ない。(もちろん、あればなお良い)

その代わり、これから作成しようとしているシステムや製品（総称してプロダクトと呼ぶ）の仕様や完成時のイメージについては、チームの誰よりも深く理解している必要がある。

よってプロジェクトの計画、進行状況の確認、仕様変更の可否などの役割も担う。

リスク管理も行う。

日常的に開発チームと共に作業する必要はないが、いくつかのイベント（スプリント計画ミーティングとスプリント・レビュー）には必ず参加する。

日常的には、開発チームから製品に関する質問を受けた場合、適切に回答する責務がある。

従来の PM つまりプロジェクトマネージャーがスクラムに参加できるとしたら、この立場（プロダクトオーナー）が最もふさわしい。

開発チームとの兼任やスクラム・マスターとの兼任は、基本的には NG である。

演習では、計画時とレビュー時にプロダクト・オーナーが必要なので、この立場を設定する。

#### プロダクト・オーナーの役割

- ① ユーザー側の視点に立つ
- ② 完成の可否を決める
- ③ リスク管理、仕様変更の管理などを行う

### 2.3.3.スクラム・マスター

スクラムの手法で開発を進められるように、メンバーに助言したり、イベント（ミーティングなど）を招集したりする立場。

よく「サーバント・リーダー **Servant Leader**」と例えられる。  
サーバント・リーダーとは、上から指示するのではなく、下から支えるリーダーのこと。

スクラムについて知っているのはスクラム・マスターだけで、その他のメンバーはスクラムについて全く知らないというような環境で開発を進めなければならない場合もある。

そのような場合でも、適切にスクラムの方法で進めていけるようにファシリテートしていくことが主な使命となる。

現在発生している問題点はチーム全体で解決していくが、スクラム・マスターはそのサポートを行う。

プロジェクトの進行状況や難易度を正しく把握する必要があるため、プログラミング言語や設計手法などに関する知識が必要となる。

前章で述べた「必要なテクニカルスキル」を網羅していることが望ましい。

開発チームとの兼任を行うことも可能だが、あまり推奨されない。

演習では、講師（メイン講師およびサブ講師）がこの役を担当する。

#### スクラム・マスターの役割

- ① サーバント・リーダーとしてチームを支える
- ② ファシリテーターとしてスクラムを円滑に進める
- ③ 問題解決に取り組む

### 2.3.4.開発チーム（スクラムチーム）

プロダクト・オーナーとスクラム・マスターを除くチームメンバー全員。

開発チームの立場は平等である。（誰が上、誰が下というような組織上のヒエラルキーはない）  
（演習では、円滑な進行のため「演習のためのリーダー」を設けることはある）

もちろん、メンバーによって技術的なレベルや経験年数などの差が出てくるのは当然なので、お互いに補いながら開発を進めていくことになる。

例えば、1つの作業を行うのに、経験者と未経験者でペアを組んで、二人で作業を行うとよい。  
これは、一見、作業効率が落ちるように思われるが（実際その時点では落ちる）、時間がたつにつれて、逆に効率が上がることが経験的に知られている。  
未経験者のスキルがあがり、一人でも作業ができるようになるからである。  
理想的なケースでは、「全員」が「すべての分野」で「ほぼ同じレベル」に達することもある。

ネットワークが得意だがデータベースは未経験のAさんと、データベースは得意だがネットワークは未経験のBさんがチームにいたとして、スクラム開発の中盤以降は、二人ともどちらの分野の作業も問題なくできるようになる、というのが理想的である。実際そうなる場合が多い。

### 2.3.5.従来のチームとの比較

従来のプロジェクトマネージャーが単独で担っていた役割は、リスク管理や進捗管理などをプロダクト・オーナーが行い、問題解決やチームのサポートをスクラム・マスターが行うというような、役割分担が行われている印象を持ったかもしれない。

実は、スクラムでは、チーム全員でプロジェクトマネジメントを行う。  
前述した「スクラムの特色」で挙げた5点は、チーム全員でマネジメントできるようにするために、過去のベストプラクティスから抽出されたものである。

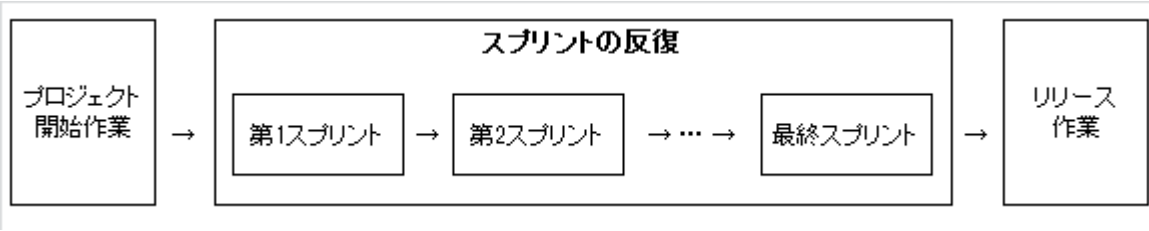
逆に言えば、技術的スキルが低く、スケジュール管理しかできないようなプロジェクトマネージャーは、スクラムでは不要といってよい。

### 第3章. スプリント（イテレーション）

3.1. スプリント（イテレーション）とは

3.1.1.スクラムの進め方

スクラムでは、1つのプロジェクトを、おおよそ次のような順序で進める。



「プロジェクト開始作業」は「リリース計画」と呼ばれることもあり、プロジェクト全体の計画を立てたり、納品までにいつ・何回リリースするのかの計画を立てたりする。

「リリース作業」には納品のための準備とプロジェクト全体の振り返り作業も含む。

このうちスプリントの反復が最も長く重要な期間となる。

各スプリントは、おおよそ「計画」→「実装」→「振り返り」の順序で行われる。

スプリント期間									
第1スプリント			第2スプリント			・・・	最終スプリント		
計画	設計・実装・ テスト等	振り返り	計画	設計・実装・ テスト等	振り返り		計画	設計・実装・ テスト等	振り返り
1週間～1か月			1週間～1か月				1週間～1か月		

スプリントの準備作業を「スプリント0（ゼロ）」などとして、通常のスプリントとは異なる期間、異なる内容で進める場合もある。

最後のスプリントも、同様に、納品直前のスプリントとなるため、「リリース・スプリント」として通常のスプリントとは異なる期間、異なる内容で進める場合がある。

リリース・スプリントでは、通常の開発は行わず、納品に必要な作業（マニュアル作成、運用環境の構築、最終テストの実施など）のみを行う。

### 3.1.2. スプリント

1 スプリントを1 週間(5 日間)だと仮定すると、1 回のスプリントの内容は、おおよそ下記のようになる。(h は時間、m は分を表す)

1 回のスプリント(第 N スプリント) - 1 週間(5 日間)				
1 日目	2 日目	3 日目	4 日目	5 日目
スプリント 計画 ミーティング (2h)	デイリー スクラム(10m)	デイリー スクラム(10m)	デイリー スクラム(10m)	デイリー スクラム(10m)
デイリースクラム(10m)	開発作業	開発作業	開発作業	開発作業
開発作業				スプリント レビュー (1h)
				レトロスペクト ミーティング (1h)

研修では、仮想的な1 日を設定し、2 日間から4 日間を1 スプリントとして実施する。  
仮想的な1 日は60 分程度にする。

計画や振り返りには、やや多めの時間を配分する。

仮想的な1 日を60 分とし、1 スプリント3 日間とした場合の時間配分例は下記の通り。

研修時のスプリント例 - 60分×3日		
1 日目	2 日目	3 日目
スプリント 計画 ミーティング (30分)	デイリー スクラム(3分)	デイリー スクラム(3分)
デイリースクラム(3分)	開発作業 (60分)	開発作業 (30分)
開発作業 (30分)		スプリント レビュー (15分)
		レトロスペクト ミーティング (15分)

## 3.2. プロジェクト計画（リリース計画）

### 3.2.1.説明

反復型開発（スプリント）に入る前に、プロジェクト全体の計画を行う。

### 3.2.2.内容

SBOKによるとスプリントに入る前のフェーズ（Initiation）には、次のページに掲げる6つの工程が説明されている。

このうち第2工程まででプロダクト・オーナーとスクラム・マスターが決定し、第3工程でスクラム・チームが編成される。

従って、スクラム・コア・チームとしての作業は第4工程から始まることになる。

注意点としては、「要件定義書の作成」とか「基本設計」というような、ウォーターフォールのいわゆる「上流工程」で登場する作業は一切存在しないということである。

プロジェクト・ビジョンとエピックという言葉が、唯一、要件定義に類似する概念であるが、要件を詳細・厳密に定義するわけではない。プロジェクト・ビジョンは「おおまかなビジネスの要件をまとめたもの」であり、エピックは要件定義書の要件一覧、機能一覧、ユースケース一覧のようなものである。

では、「要件定義書の作成」とか「基本設計」などが必要な場合、スクラムではいつ行うのか。それは、（1）この時点で既に完了している、（2）このフェーズで行う、（3）次のスプリント期間に行う、という3つの選択肢がある。

ただし、（2）の方法を採用してしまうと、その作業だけで数週間から数か月かかってしまう場合がある。それではスクラムを採用する意味が薄れるので、（3）の方法を採用したほうが良い。

しかし、（3）の方法を採用する場合でも、「おおよそどんなものを作成するのか」がわかっていないとスプリントに進むことができないので、このフェーズで「プロジェクト・ビジョン」や「エピック」を作成しておくのである。

プロジェクト・ビジョンは「どんなものを作るか」「どんな解決策を提供するか」という観点よりも「現状どのような問題点があるか」ということを重点に置いて作成するとよいとされている。

一方、エピックは、高いレベルで（詳細ではない、という意味）記述されたユーザー・ストーリーや、作成物の説明や要件項目が広範囲わたって定義されたものである。

このエピックを詳細化してプロダクト・バックログ（Product Backlog Item, PBI）を作成することになるので、エピックとはプロダクト・バックログの概要版と理解してもよい。



### 3.2.3. プロジェクトの開始フェーズにおける主な作業

下記は、プロジェクト開始時に実施するとよいとされている作業である。  
SBOK には、さらに詳細な実施項目が例示されている。(リスクの定義、品質の定義など)

1. プロジェクト・ビジョンの作成
  - ① プロジェクト・ビジョンの作成  
ビジネス要件をまとめる。
  - ② プロダクト・オーナーの選定
2. スクラム・マスターとステークホルダーの特定
  - ① スクラム・マスターの選定
  - ② ステークホルダーの特定  
スクラム・チームだけでは解決できない問題が発生したときに、プロダクト・オーナーが誰に質問すればよいかを特定する。
3. スクラム・チームの編成
  - ① スクラム・チームの選定
4. エピックの作成
  - ① エピック (Epic) の作成  
おおまかな機能一覧を記述する。
  - ② ペルソナ (Persona 人物モデル) の作成
5. プロダクト・バックログの作成
  - ① 優先付けされたプロダクト・バックログの作成  
エピックを詳細化しプロダクト・バックログの第1版を作成する。
  - ② 完了の定義 (Done Criteria) の作成  
プロダクト・バックログのすべての項目について、完了の定義を行う。  
品質を厳密に定義したい場合は、別途、承認基準 (Acceptance Criteria) も作成する。  
(これはスクラム・ガイドに記述してもよい)  
演習では「完了の定義」ではなく「デモ手順」程度の記述でよい。
6. リリース計画の作成
  - ① リリース計画のスケジューリング  
通常は、各スプリント終了時に、その時点で完成した部分までがリリース可能だが、それとは別に、大きな単位で顧客にリリースする計画を立てる。(アルファ版、ベータ版、最終版などの納期)
  - ② スプリントの長さを決定する。(3 日、1 週間、2 週間、1 か月など)

上記の 4. と 5. にある Epic の作成やプロダクトバックログの作成や 6. のリリース計画の作成は、次節で述べる「リリース計画ミーティング」でもう少し詳しく述べる。

### 3.2.4.留意事項

#### 1. スプリント・ゼロ（イテレーション・ゼロ）

スクラム・コア・チームが編成されたあとの作業（前ページの「4. エピックの作成」以降の作業）は、このプロジェクト計画作業の中で実施するのではなく、通常のスプリントとして実施してもよい。

この場合、最初のスプリントだけは、特別な作業となるので、通常のスプリントと区別して、スプリント・ゼロと呼ぶ。

小規模プロジェクトでは、「プロジェクト計画」自体をスプリント・ゼロで行ってもよい。

スプリント・ゼロについては、後述する。

#### 2. エピックやプロダクト・バックログの作成原則

エピックは機能概要で、プロダクト・バックログは機能詳細のような対応関係となるので、1つのエピックは複数のプロダクト・バックログ・アイテムに分割されることになる。

どちらにも「実現したいこと（要求仕様）」を記述し、「実現方法（設計）」は記述しない。

「実現方法（設計）」は、各スプリントの中で「スプリント・バックログ」として記述する。

#### 3. エピックやプロダクト・バックログに記述する内容

エピックやプロダクト・バックログには、アプリケーションの機能を記述するだけでは足りない。

顧客から、納品物として「要件定義書」「基本設計書」「外部設計書」「内部設計書」「詳細設計書」「テスト仕様書」「ユーザーマニュアル」などの提出が求められている場合は、それらの文書の作成も、プロダクト・バックログ（ユーザー・ストーリー）に明記すること。

#### 4. エピックとプロダクト・バックログの規模感

エピックは、各スプリントの期間やチームメンバーの数にかかわらず、理解しやすい粒度で機能やストーリーを記述する。

無駄に細かく分割する必要はない。

一方、プロダクト・バックログ・アイテムは、1回のスプリントで最低1つは完了できる程度の小ささに分割しなければならない。

できれば1回のスプリントで複数のバックログ・アイテムが完了できるような規模が望ましい。

理由は、スプリント・レビュー時に完了したバックログ・アイテムだけをレビューするからである。その回のスプリント終了時に、レビューするものが1つもないという状況は避けなければならない。

## 5. 要件が定まらないとき

スプリントが始まるまでに、要件が完全に定義されるのは望ましいことであるが、実際には、要件の一部が決まっていなかったり曖昧だったりすることがある。

スクラムなどのアジャイルの手法では、このような場合、要件が厳密に定義されるのを待ってからプロジェクトを始めるのではなく、先にプロジェクトを開始してから、スプリント期間中に要件を固めていくという考え方をする。

そのために、スクラムでは、頻繁に要件をチェックするイベント(スプリント計画ミーティングなど)やツール(プロダクト・バックログ)が用意されている(後述)。

### 3.3. リリース計画ミーティング（プロジェクト計画ミーティング）

#### 3.3.1.説明

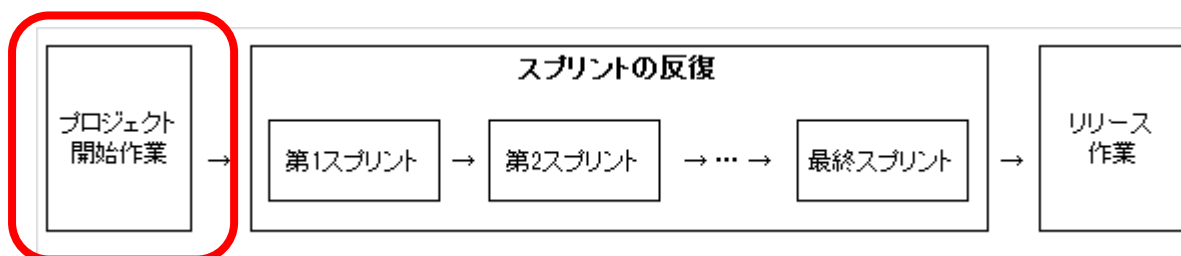
スクラムチームが編成された後に、プロジェクトの理解やエピックの作成などを行い、可能な限りプロダクトバックログを定義する。

このやり方については、次節のスプリント計画ミーティングを参照のこと。

また、リリースを何回に分けて行うか、リリース時期はいつにするか、それぞれのリリースでどこまで作成するかなども話し合う。

前出のプロジェクト計画の節には3番目の項目にチームの編成とあるので、4番目以降の作業をこのリリース計画ミーティングで行うことになる。

期間はプロジェクトの規模に応じて決める。(数時間の場合もあれば、1週間以上の場合もある)



なお、小規模プロジェクトでは、おおげさにリリース計画ミーティングを開くのではなく、最初のスプリントで行う場合もある。(次節の「スプリント・ゼロ」参照)

### 3.4. スプリント計画ミーティング（スプリント・プランニング）

### 3.4.1.説明

これから始まるスプリント（今回のスプリント）の計画を作るミーティング

スプリント全体の 5%程度の時間を割いて行う。(1 回のスプリントの期間が 1 か月の場合は 8 時間、1 週間の場合は 2 時間等。演習では 10 分～15 分程度とする)

1回のスプリント(第 N スプリント) - 1週間(5日間)				
1日目	2日目	3日目	4日目	5日目
スプリント 計画 ミーティング (2h)	デイリー スクラム (5m)	デイリー スクラム (5m)	デイリー スクラム (5m)	デイリー スクラム (5m)
デイリースクラム(5m)	開発作業	開発作業	開発作業	開発作業
開発作業				スプリント レビュー (1h)
				レトロスペクト ミーティング (1h)

### 3.4.2.内容

スプリント全体の5%程度の時間を割いて行う。(1回のスプリントの期間が1か月の場合は8時間、1週間の場合は2時間等。演習では1部と2部を合わせて30分程度とする)

一部と二部に分けて行うことが多い。

#### 1. 一部（重要）

1部では、プロダクト・オーナーと開発チームで、今回のスプリントで実現することを決める。

- ① 実現することは「プロダクト・バックログ」(後述)に記述する。
- ② 各バックログアイテム (PBI: Product Backlog Item) の優先順位を決める。
- ③ 各 PBI の「完了の定義」を行う。(演習では「デモ手順」の定義でもよい)
- ④ 各 PBI の規模を「ポイント」という単位で見積る。(相対見積)

今回のスプリント内で実現できなかった PBI は、次回のスプリントに回す。  
ただ、時間が余った場合のことも考慮して、少し多めに予定を立てるとよい。  
演習では、④の工数見積りは、フィボナッチ数列を使った見積りを行う。

#### 2. 二部（あまり時間をかけ過ぎない）

2部では、開発チームが中心となって実際の開発作業を洗い出し、より現実的で具体的な計画づくりを行う。(「どのように実現するか」を話し合う)

- ① 各 PBI を実現するために必要な作業を洗い出す。  
「ツールのインストール」「専門家へのインタビュー」「不明点の調査」など、PBI を表面的に読んだだけではわからない作業もしっかり洗い出すことが必要。
- ② それらはスプリント・バックログ (SBL: Sprint BackLog) に記述する。  
つまり1つの PBI (機能) に複数の SBL (作業) が対応付けられることになる。
- ③ 各 SBL は「時間」で見積る。(絶対見積)  
時間で見積った結果、今回のスプリント期間に収まる PBI を、最終的に今回のスプリントで実現する項目とする。(PBI の優先順位は変更しない)

この計画ミーティング時には想定しなかった作業が、スプリント期間中に増えてくる場合があるが、それらは随時、SBL に追加する。

### 注意事項

プロダクトバックログ (PBI) とスプリントバックログ (SBL) は名前が似ているので注意。

簡単にいうと、プロダクトバックログは顧客側から見た機能(ストーリー)の集まりであるのに対し、スプリントバックログはチームが行う作業(タスク)一覧である。

プロダクトバックログは顧客に提示することがあるが、スプリントバックログはチームメンバーが理解できればよく、後日破棄してもよい。

### 3.4.3.1 部と2部の対比

1部と2部で行う内容をまとめると、下記のようなになる。

なお、必ずしも1部と2部に分けて実施しなければならないわけではない。  
(最近では、分けずに実施するチームも増えているようである)

	スプリント計画ミーティング	
	1部	2部
テーマ	何(どれ)を実現するか。 ・優先順位は？ ・受け入れ条件(完了条件)は？	どのように実現するか。 ・誰がどれを担当するか ・個々の作業時間は？
進行役	プロダクト・オーナー	開発チーム
作成・更新する バックログ	プロダクト・バックログ ・優先順位づけ ・相対見積(ポイント) ・完了の定義	スプリント・バックログ ・担当者 ・絶対見積(時間)
バックログ変更の タイミング	プロダクト・バックログは、スプリント 期間中は、なるべく変更しない。	スプリント・バックログの内容は 開発中に随時変更してよい。
その他に作成・更新 するドキュメント	・問題点リスト ・リスク一覧 ・変更の要請一覧	・問題点リスト ・リスク一覧 ・バーンダウンチャート

### 3.5. デイリー・スクラム（デイリー・スタンドアップ）

### 3.5.1.概要

毎日 5 分～15 分程度行うミーティング。(演習では、2～3 分)

立ち上がって行くと効果的で、かつ、短時間で終了するため、このように呼ばれる。(演習では、着席したままでもよい)

最低限次の3つの内容について話し合う。

1. 前回のデイリー・スタンドアップからやってきたこと（昨日やったこと）
2. 次回のデイリー・スタンドアップまでにやること（今日やること）
3. 問題点

1回のスプリント(第 N スプリント) - 1週間(5日間)				
1日目	2日目	3日目	4日目	5日目
スプリント計画 ミーティング (2h)	デイリースクラム(10m)	デイリースクラム(10m)	デイリースクラム(10m)	デイリースクラム(10m)
開発作業	開発作業	開発作業	開発作業	開発作業
				スプリントレビュー (1h)
				レトロスペクト ミーティング (1h)



### 3.5.2. 説明

日本の「朝会」のようなもの。  
必ずしも午前中に行う必要はないが、毎日同じ時間に行うことが推奨されている。

このイベントの目的は一言でいうと問題点の早期発見である。

上記の1.と2.で「順調に進んでいるか」「遅れ気味であるか」というような進捗状況をおおざっぱに把握できる。  
また、3.では、進捗以外の問題点を早期に発見できる。

問題点はチーム全体で解決していくことが前提だが、開発チームは現場の作業で忙殺されている場合が多いので、スクラム・マスターとプロダクト・オーナーが連携を取りながら、問題解決の方途をさぐる。

時間内に収まらない場合は、いったん終了し、必要なメンバーだけで別途ミーティングを行う。

スプリント・バーンダウンチャートを作成している場合は、デイリー・スクラムが始まる前に、その更新も行っておく。

#### デイリースクラムのポイント

- ① 短時間で実施する
- ② 問題点を顕在化させる

### 3.6. 開発作業

#### 3.6.1.説明

スプリント・バックログに定義されている項目を、プロダクト・バックログの優先順位に基づいて実施する。

具体的には、開発環境の準備、顧客等との打合せ、調査作業、ユーザーインターフェースの作成、プログラミング、テスト、マニュアル作成など、一般的な開発作業を行う。

要件定義や基本設計など、いわゆる上流工程と呼ばれている作業も、プロジェクトの作業項目に入っている場合は、この期間に行う。

スプリント・バックログに「担当者」という項目を設けることが多いが、基本的には、スプリント・バックログの項目はだれが作業してもよいので、スプリント期間中に担当者を入れ替えてよい。

むしろ、チームの誰もがどの作業も行えるように進めていくのがよいやり方である。

また、ソースコードや設計書などの作業対象物は、だれか一人の担当者のものではなく、チーム全員のものである。

つまり、1つのソースコード（あるいはドキュメント等）を、ある日はAさんがアップデートして、別の日はBさんがアップデートするというやり方が、何の支障もなくできるように進めていくべきである。

「あのソースコード（ドキュメント）は私の担当ではなくで、Bさんの担当なので、私はわかりません。Bさんに聞いてください」ということは、基本的には、あってはならない。  
(もちろん、チーム内のスキルの差はプロジェクトの規模は考慮する)

1回のスプリント(第Nスプリント) - 1週間(5日間)				
1日目	2日目	3日目	4日目	5日目
スプリント計画 ミーティング (2h)	デイリースクラム(5m)	デイリースクラム(5m)	デイリースクラム(5m)	デイリースクラム(5m)
デイリースクラム(5m)	開発作業			開発作業
開発作業				スプリントレビュー (1h)
				レトロスペクト ミーティング (1h)

### 3.6.2.留意事項

前ページで列挙した作業はスプリント・バックログに記入されている必要がある。  
記入忘れがあったら、直ちにスプリント・バックログに記入し、スクラム・チーム内で情報共有する。

スプリント・バックログは、開発期間中に、スクラム・チームによって変更してよい。  
しかし、プロダクト・バックログを変更してはいけない。

プロダクト・バックログの優先順位や項目自体を変更するのは、スプリント終了時あるいは次回のスプリント計画時とする。スプリント期間中に変更すると、混乱するからである。

スプリント・バックログは開発期間中に変更可能と述べたが、当然、スプリント・バックログの内容に変化があると、そのスプリント期間中に予定されていたタスクの完了に影響を与える。

最悪の場合、完了できないタスクができてしまうかもしれない。

スクラムの手法ではこのような場合を想定して、スプリント・レビューやレトロスペクト時に、問題点の洗い出しや振り返りを行い、次回のスプリントで改善していくようになっている。

このように、短期間で問題点を解決できる仕組みになっていることが、スクラムをはじめとするアジャイル開発の強みである。

### 3.6.3.開発中に実施することが必須あるいは推奨される作業

スクラムは、短い期間（スプリント）で問題発見と解決、品質の保証を行うフレームワークである。また、担当者制ではなく、チーム全員がまんべなくすべての作業を行えるようにする方法論である。

以上の点を実現するための仕組みが、開発作業期間中にも必要となる。

そのために、たとえば、次のようなプラクティスを実践することが推奨されている。

#### 1. ユニットテスト（自動テスト）

関数（メソッド）ごとのユニットテストを作成し、チームが作成したロジックが正しいことをいつでも示せるようにしておく。

これがないと、スプリント・レビュー時に動作が正しいという根拠が直ちに示せない。

逆に、仕様と異なるという指摘を受けた場合でも、ユニットテストがあると、間違っている場所がすぐにわかる。

#### 2. リファクタリング

ソースコードをチームで共有しやすくするためには、ソースコードが常に理解しやすい状態に保たれていなければならない。（誰か一人にしか理解できないコードではいけない）

そのためには、ソースコードのリファクタリングは必須である。

#### 3. ペア・プログラミング（ペア作業）

チーム全員がまんべなくすべての作業を行えるようにするためには、チーム内の初心者が短期間でスキルアップする仕組みが必要である。

そのためには、熟練者と初心者がペアで作業するのが最も効率の良い方法である。

プログラミングだけでなく、データベース設計やネットワーク設定もペアで行うとよい。

#### 4. アプリケーション・アーキテクチャーの理解

プログラミング言語が理解できている、データベースやネットワークのプログラムが記述できる、画面設計ができるという個々のスキルだけでは、プロジェクトの中盤以降、プログラムが肥大化したときに、ソースコードを見渡しにくくなってしまう。

チームメンバーの少なくとも一人は、プロジェクトの開始時から、きちんとしたアーキテクチャーに基づいてアプリケーションを作成していくスキルを持っていることが求められる。

アーキテクチャーは、プロジェクト進行につれて他のメンバーにも共有され、改善され、当初のものから変貌を遂げてよい。

#### 5. 継続的インテグレーションの実践

バージョン管理システムや、(ローカルではなく)サーバー上での自動ビルドや自動テスト、自動デプロイの整備ができるようになると、生産性が飛躍的に向上する。

すべて自動化するのは困難だが、最低限、継続的インテグレーションに関する理論やツールの知識を持っていることが望ましい。

具体的な製品名としては、Sub Version、Git、TFS (Team Foundation Server)、Azure DevOps、Jenkinsなどが適切に構成・使用できるということである。

#### 3.6.4.作業を円滑に行うためによく使われるツール

チームでスムーズに作業を行うためには、プラクティスだけでなくツールも重要である。

ソースコードの管理やタスクの管理を手作業で行うと混乱し、作業漏れや重複がおきてしまう。

そこで、次のようなツールを使うとよい。

##### 1. バージョン管理システム (Git、SVN、TFS)

ギット (Git)、サブバージョン (SVN) やチーム・ファウンデーション・サービス (TFS) など。

##### 2. タスク管理ツール

スプリント・バックログの進捗具合を管理するのによく使われる。

Redmine、Trello、Jira などが有名である。

##### 3. コミュニケーションツール

遠隔地で作業を行う場合、ビジネスチャットツールが有効な場合がある。

Slack、Teams、ビジネス LINE など。

### 3.6.5.開発以外に必要な作業

開発作業そのものではないが、開発期間中に必要となる作業がある。

下記の項目は、開発期間中に随時更新し、レトロスペクティブ・ミーティングでまとめを行う。

#### 1. スクラム・ガイド (SGB, Scrum Guidance Body) の更新

SGB は、品質、社内・社外の規約、法令、セキュリティ、その他（組織上の制約など）を定義した文書や部署。

開発中に得られた知見や、問題解決方法など、今後のプロジェクトにも伝えておきたいような事柄を追加する。

#### 2. 妨害リスト（問題点リスト Impediments Log）に対する対処と、リストの更新

#### 3. リスク一覧

#### 4. 変更の要請

#### 5. バーンダウンチャートの更新

#### 6. スプリント・バックログ

計画ミーティング時に漏れていた作業項目を追加する。

プロダクト・バックログの優先順位は変更しない。

プロダクト・バックログの優先順位は、（レトロスペクティブ・ミーティングの結果にもとづいて）スプリント計画ミーティングで行う。

## 3.7. スプリント・レビュー

### 3.7.1.概要

各スプリントの最後のほうに実施するイベントで、製品を評価する。

スプリント全体の 2%~3%程度の時間を割いて行う。(1 回のスプリントの期間が 1 か月の場合は 4 時間、1 週間の場合は 1 時間等。演習では 15 分程度とする)

開発チームがプロダクト・オーナーに対し、今回のスプリントで完成した部分をプレゼンテーションする。場合によっては、顧客やエンドユーザーも参加する。

1回のスプリント(第 N スプリント) - 1週間(5日間)				
1 日目	2 日目	3 日目	4 日目	5 日目
スプリント 計画 ミーティング (2h)	デイリー スクラム (5m)	デイリー スクラム (5m)	デイリー スクラム (5m)	デイリー スクラム (5m)
デイリースクラム(5m)	開発作業	開発作業	開発作業	開発作業
開発作業				スプリント レビュー (1h)
				レトロスペクト ミーティング (1h)

### 3.7.2.説明

開発途中のものや完成しなかった部分は取り上げない。

スライドの上映などで代替してはいけない。あくまで出来上がった製品を評価する。

プロダクト・オーナーは、プロダクト・バックログ・アイテム (PBI) の「完了の定義」あるいは「デモ手順」にもとづいて、完成部分が仕様に合致しているかどうかレビューする。

仕様を満たしていないことがわかったら、次回のスプリント以降で開発を継続する。

もし、この時点で「完了の定義」自体に誤りや漏れがあった場合は、「完了の定義」を再定義し、次回のスプリントで完成をめざす。

完成が確認されたら、プロダクト・バックログに完了の目印をつける。(演習では、目印に加えて、かかった時間も分単位で記入する)

演習では、ここでベロシティ (チームの開発スピードのこと。後述) の計算も行う。



### 3.7.3. その他に実施する事柄の例

製品の評価の他に、下記のような議論も行う。

- ・品質に関する議論
- ・リスクや不確定事項の特定に関する議論
- ・ベロシティやバーンダウンチャートに基づく進捗状況（あるいはリリース時期）の議論

### 3.7.4. 留意点

スプリント・レビューは略式ミーティングであり、公式なミーティングではない。  
従って、公式な納品（リリース）作業ではない。

言い換えると、公式なリリースは、プロジェクト開始時のプロジェクト計画ミーティングなどで作成される「リリース計画」に基づいて行う。

たとえば、1回のスプリントを1週間とすると、スプリント・レビューも1週間ごとに行われるが、公式リリースは「1か月ごと」とか「アルファ版は〇月〇日、ベータ版は〇月〇日、最終リリースは〇月〇日」というような決め方をする。

しかしながら、スプリント・レビューでは、「完了の定義」あるいは「承認基準」を満たすことを厳密にチェックし、仮に公式リリースしたとしても問題ない品質を確保しなければならない。

#### スプリント・レビューのポイント

- ① PBI ごとに完成か未完成かを判定する
- ② プロジェクト全体の進捗が見える化する

## 3.8. レトロスペクト (スプリント・レトロスペクティブ)

### 3.8.1.概要

各スプリントの最後のほうに実施するイベントで、通常はスプリント・レビューよりも後に実施する。

レトロスペクト、レトロスペクト・スプリント、スプリント・レトロスペクティブ、レトロスペクティブ・ミーティングなどの呼び方がある。

日本の「反省会」や「振り返り」のようなミーティング。  
製品の評価を行うのではなく、「やり方」を評価するミーティングとなる。  
「うまくいったこと」と「うまくいかなかったこと」を話し合う。

スプリント全体の 2%~3%程度の時間を割いて行う。(1 回のスプリントの期間が 1 か月の場合は 4 時間、1 週間の場合は 1 時間等。演習では 5 分から 10 分とする)

スクラム・チームが実施する。

このミーティングは、形式的な内容にせず、ポジティブで生産的になるようにする。  
例えば、ティーブレイク (コーヒーブレイク) のような形式で行ってもよい。

具体的には、「作業は順調だったか」「開発ツールは適切だったか(別のツールに変えたほうがよいか)」「チーム間のコミュニケーションや連携はうまくいったか」「その他に (製品ではなく) プロジェクトに関する事柄なら何でも」などを話し合う。

このミーティングは、以降のスプリントをより円滑、効率的に進めていくための土台となる。

また、PDCA サイクルをうまく回すために重要なミーティングである。

1回のスプリント(第 N スプリント) - 1週間(5日間)				
1日目	2日目	3日目	4日目	5日目
スプリント 計画 ミーティング (2h)	デイリー スクラム (5m)	デイリー スクラム (5m)	デイリー スクラム (5m)	デイリー スクラム (5m)
デイリースクラム(5m)	開発作業	開発作業	開発作業	開発作業
開発作業				スプリント レビュー (1h)
				レトロスペクト ミーティング (1h)

### 3.8.2.話し合う内容の例

1. 前回のレトロスペクトで合意された事項の進行状況の確認
2. 問題点（妨害）リストに関する議論
3. 今回のスプリントでうまく行ったこととうまく行かなかったことと改善点。  
KPT（Keep-Problem-Try, ケプト）を議論する。
  - ① KEEP: 引き続き維持していくこと（うまくいったこと）、
  - ② PROBLEM: 止めたほうが良いこと（問題点やボトルネック）
  - ③ TRY: 問題解決のために新しく始めたほうがよいこと（改善点）
4. スプリント・レビューで提示されたり指摘された内容のうち、やり方に関する議論

### 3.8.3.守るべきこと

1. 前回のレトロスペクト時に作成した問題点リストや KPT の実施状況の確認
2. 今回議論された解決案や Try のうち、遂行可能な改善点の合意
3. 改善点や Try の実行完了日の決定

### 3.8.4.アウトプット

1. 問題点（妨害）リストの更新
2. レトロスペクトの議事録（必要ならば）  
KPT などでも代用してもよい。
3. 成功や失敗や教訓の共有  
Scrum Guidance Body を更新して、他のチームと共有する。

#### レトロスペクトのポイント

- ① PDCA サイクルのための重要なイベント
- ② 次回のスプリントをよりよいものにするために実施する

## 3.9. スプリント0（ゼロ）とリリース・スプリント

### 3.9.1.説明

最初のスプリントと最後のスプリントは、通常のスプリントとは期間と作業内容を分けて実施する場合がある。

特に、最初のスプリントでは、プロジェクトの全体像の理解、ツールのインストールなど、以降のスプリントのための「準備」だけを行うと決めてしまう場合がある。

さらに、それだけでなく、「機能はゼロだが、エンド・ツー・エンドの動作だけは確認できるアプリケーションを作成する」ところまでを、最初のスプリントで作成する場合がある。（注1）

前者の場合は通常のスプリントより短い期間になる傾向があり、後者の場合は通常のスプリントよりも長い期間になる傾向がある。

このように、最初のスプリントを特別構成にして行う場合、特に「スプリント・ゼロ」と呼ぶ。

同様に、最後のスプリントは、顧客に納品するための作業を行わなければならないので、開発やテストなどの実装作業は一切行わず、リリースの準備だけを行う場合がある。

このような場合、最終スプリントを、特に「リリース・スプリント」と呼んで、期間や作業内容を通常のスプリントと異なる構成にする。

もちろん、スプリント・ゼロやリリース・スプリントを特別に設けずに、通常のスプリントに組み込んで実施してもよい。

規模が小さい場合は、リリース計画（プロジェクト計画）をスプリント・ゼロに含めることもできる。

#### 注1

「エンド・ツー・エンドの動作を実装するダミー機能のアプリケーション」を作成することは、XP やスクラムなどのアジャイル開発では、非常に重視されている。

（XP では、「イテレーション・ゼロ」と呼んでいる）

「エンド・ツー・エンド」とは、「端から端まで」というような意味である。

「エンド・ツー・エンドの動作」とは、例えば、「画面とキーボードで入力したデータが、ネットワークを通じて、データベースに正しく保存される」というように、フロントエンド（画面・キーボード）からバックエンド（データベースサーバー）までが、きちんとつながって動作することを指す。

ただし、本格的な機能を実装する必要は全くなく、単に、「エンド・ツー・エンド」の処理が実行されることが確認できるだけの処理でよい。つまり、処理はダミーでよい。

画面機能とデータベース機能を別々のチームが開発し、開発の最終段階で結合テストを行ってみたら、ネットワーク性能により、要求仕様を満たさなかったというようなことが起こることがある。

そこで、開発の初期の段階で、全ての末端をつなぐようなダミーのアプリケーションを作成して、リスクを早期に発見することがよいことだとされている。

## 第4章. ツールと概念

## 4.1. プロダクト・バックログ関連

### 4.1.1. プロダクト・バックログ

実現したいことがすべて書かれているドキュメント。  
「要件定義書」や「基本設計書」の「機能一覧」に相当するといえる。

バックログとは「未処理分」「未着手分」というような意味。つまり「これから行う事柄」である。

どのように実現するかや詳細な実現項目は、プロダクト・バックログではなくスプリント・バックログに記述する。

決められたフォーマットはないが、少なくとも「ユーザー・ストーリー」「完了の定義」「見積ポイント」というような項目が必要だと考えられている。

ユーザー・ストーリーは、機能名だけでなく、その機能の実際の操作を文章で表したものである。

スプリントに入る前のプロジェクト開始時に、おおまかなユーザー・ストーリーを作成するが、これはプロダクト・バックログよりも粗い一覧なので、「エピック (epic)」と呼ぶ。

### 4.1.2. プロダクト・バックログの粒度

1 スプリントの期間で、最低1つのユーザー・ストーリーの機能を完成させられる粒度でプロダクト・バックログ・アイテムの規模を決める。

実際には、チームメンバーのモチベーションを維持するために、もっと多い数が完了できるようにするとよい。

スプリント・レビューでは、完了の条件を満たしているものだけをレビューするが、そのスプリントで完成したバックログ・アイテムが一つもないということは、スプリント・レビューする意味がなくなってしまう。

これは、スクラムのフレームワークが破綻していることを示す。

以上のような理由で、プロダクト・バックログの工数を見積ったとき、規模が大きすぎると感じたら、躊躇することなく、バックログ・アイテムの分割を行うとよい。

## 4.1.3.プロダクト・バックログの例

単に機能を列挙するだけでなく、その機能の実際の操作を文章（ストーリー）として表現したほうがよいとされている。

下図では「デモ手順」を「完了の定義」としている。

No.	ストーリー	デモ手順	工数見積 (ポイント)	実工数 (分)	備考
1	今まで担当者ごとにまちまちだったので、社内で使用する見積書のフォーマットを統一したい。	ボタンをクリックしたら、日付・顧客名・商品明細等を入力するフォームが開く。入力後「発行」ボタンをクリックすると見積書が発行される。	8		
2	見積書の検索を担当者別と顧客別と日付順のものと3種類作成したい。	見積書一覧はまず日付順で表示される。その画面の「担当者別」リストボックスでその担当者が作成したものが日付順で表示される。「顧客別」リストボックスでその顧客のものが日付順で表示される。	5		
3	今まで請求書は紙の見積書を見ながら作成していた。そこで見積書にもとづいて請求書を自動発行したい。	見積書画面の「請求書発行」ボタンをクリックすると見積書の内容がコピーされた請求書の編集画面が開く。	5		

短い表現（見出し項目）があったほうが良いと思ったら、下記のように「機能」のような項目を設けてもよい。

No.	機能 (要求仕様)	ストーリー	デモ手順	工数見積 (ポイント)	実工数 (時間)	備考
1	見積書フォーマットの統一	今まで担当者ごとにまちまちだったので、社内で使用する見積書のフォーマットを統一したい。	ボタンをクリックしたら、日付・顧客名・商品明細等を入力するフォームが開く。入力後「発行」ボタンをクリックすると見積書が発行される。	8		
2	見積書の検索機能	見積書の検索を担当者別と顧客別と日付順のものと3種類作成したい。	見積書一覧はまず日付順で表示される。その画面の「担当者別」リストボックスでその担当者が作成したものが日付順で表示される。「顧客別」リストボックスでその顧客のものが日付順で表示される。	5		
3	見積書に基づく請求書の自動生成機能	今まで請求書は紙の見積書を見ながら作成していた。そこで見積書にもとづいて請求書を自動発行したい。	見積書画面の「請求書発行」ボタンをクリックすると見積書の内容がコピーされた請求書の編集画面が開く。	5		

注意する点として、「見積」は時間ではなく（プランニングポーカーなどによる）ポイントを記述し、実際にかかった「工数」は時間で記述すると、細かなベロシティを計算する時に役立つ。

#### 4.1.4. プランニングポーカー（工数見積りの方法）

##### 1. プロダクト・バックログの見積は全員で行う

各ストーリー（つまりプロダクト・バックログの各項目）の工数は、開発チーム全員で見積もる。

この作業は開発チームごとに行うので、相対的な見積となる。

この見積値を1つのチーム内で比較することには意味があるが、複数のチーム間で比較するのはあまり意味がない。（チームごとに基準となる値が異なるため）

スクラムでの見積（時間）は、厳格である必要はなく、「だいたいこれぐらいだろう」という値でよい。

各スプリントの初めに毎回この作業を行うので、回数を重ねるに連れて自然と厳密な値に収束していく。

（もちろん、プロジェクト開始時に、全体の作業時間や金額を見積めることは重要だが、各スプリントで、そのスプリントの見積を行うことにより、見積の精度が高くなり、PDCA サイクルを回しやすくなる）

##### 2. 見積方法の一つとしてのフィボナッチ数列

この見積には、フィボナッチ数列（1, 2, 3, 5, 8, 13...）がよく使用される。

「1 は超簡単、2 は簡単、3 はふつう、5 はやや時間がかかりそう、8 はかなり時間がかかりそう」というような基準で使用する。

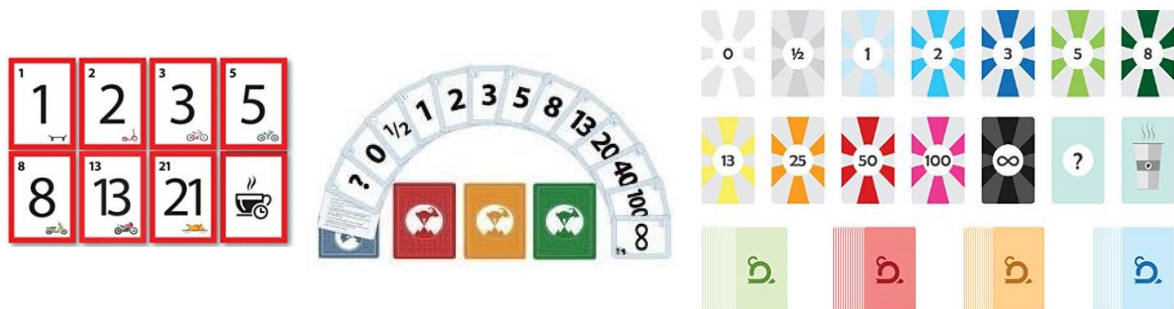
1 から始めなくても、途中の数字（例えば5 から始めてもよい）

フィボナッチ数列は、その値と次の値の比が、約 1.62 倍（黄金比）となるので、どの数字で始めてもよい。

簡単な作業は細かく見積りやすいが、難解な作業は細かな数字では見積りにくい。

フィボナッチ数列は、そのような数値を算出したいときに便利である。

下記は、プランニングポーカーの例：



基本的には、フィボナッチ数列だが、20 以上はきりのいい数字にしたり、0 や 1/2 (0.5) という小さな値を採用しているものもある。

コーヒーのカードは「そろそろ休憩したい」、? マークのカードは「よくわからない」、∞ マークのカードは「規模が大きすぎる（分割したい）」というような用途で使う。



### 3. 見積方法 ([https://tracpath.com/works/development/agile\\_planning\\_poker\\_and\\_userstory/](https://tracpath.com/works/development/agile_planning_poker_and_userstory/))

#### ① 基準となるストーリーを決める

参加者全員が理解できる小さめのストーリー（プロダクト・バックログ・アイテム）を選び、それを「2」あるいは「3」ポイントなどとする。このストーリーを基準に相対的に見積もりを行う。

#### ② カードを出す

全員が、カードを1セットずつ持ち、ストーリーごとに、上記①の基準と比べてどのぐらいの難易度であるかをカードで表明する。（トランプのゲームのポーカーのように）

#### ③ 見積を確認する

一番大きな番号と一番小さな番号を出した人は、その番号にした理由を述べる。

多くのメンバーは2や3を出しているのに、一人だけ8のようなかけ離れたカードを出している場合は、作業内容を誤解している可能性が高いので、誤解を解くためにも有効な方法である。

#### ④ 全員がほぼ同じ番号になるまで②と③を繰り返す

同じ番号で一致するのが理想だが、1つだけ違う（たとえば、「3」と「5」に分かれてしまう）場合は大きな番号をそのストーリーの見積ポイントとするというようなルールを設けてもよい。

#### ⑤ 大きな過ぎる数値で合意したらストーリーを分割する

13や21のような高い数字で同意した場合は、作業量として多すぎると考えられるので、複数のストーリーに分解することも考慮に入れる。

「分割したい」という意思表示のために「∞」マークのカードを使ってもよい。

### 4. 留意点

重要な点は、時間で見積もらないこと。あくまで作業量（ポイント）で見積ることである。言い換えると、絶対量で見積るのではなく、相対量で見積るということである。

時間で見積ってしまうと、完了させようとするあまり残業しようとするような心理的要因となってしまう。

慣れないうちは、「カードの1は1時間程度、2は半日程度、3は1日程度、5は数日、8は1週間程度」というように、時間に置き換えた基準で判断することが効果的な場合もあるが、慣れてきたら純粋に相対量で見積りが行えるようにすべきである。

## 4.2. スプリント・バックログ関連

### 4.2.1.説明

プロダクト・バックログの項目を作成するために、その回のスプリントで、具体的にどのような手順でどのような作業を行うかを記述したもの。

プロダクト・バックログが要件定義の項目（機能一覧）だとすれば、スプリント・バックログは基本設計や詳細設計で「どうやって実現するか」を決めた手順だといえる。










別の言葉で表現すると、プロダクト・バックログは「ストーリーの一覧」、スプリント・バックログは「タスクの一覧」ということになる。

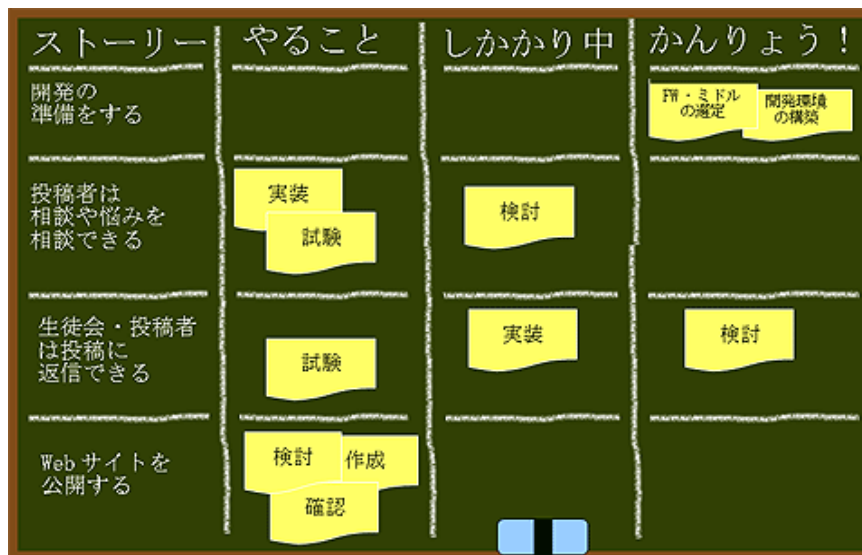
スプリント・バックログは単に一覧表にするだけでは有効活用できないため、次節で述べるタスクボード（スクラムボード）という形式で、随時更新できるようにしておく。

ホワイトボードに付箋を貼るような感じで管理されることも多い。

Excel で管理することもできるが、Trello のような使いやすツールもある。

### 4.2.2.タスクボード

	担当	TODO	DOING	DONE
PBI #1	猿田			
PBI #2	犬山			
PBI #3	雉尾			



### 4.2.3.Trello

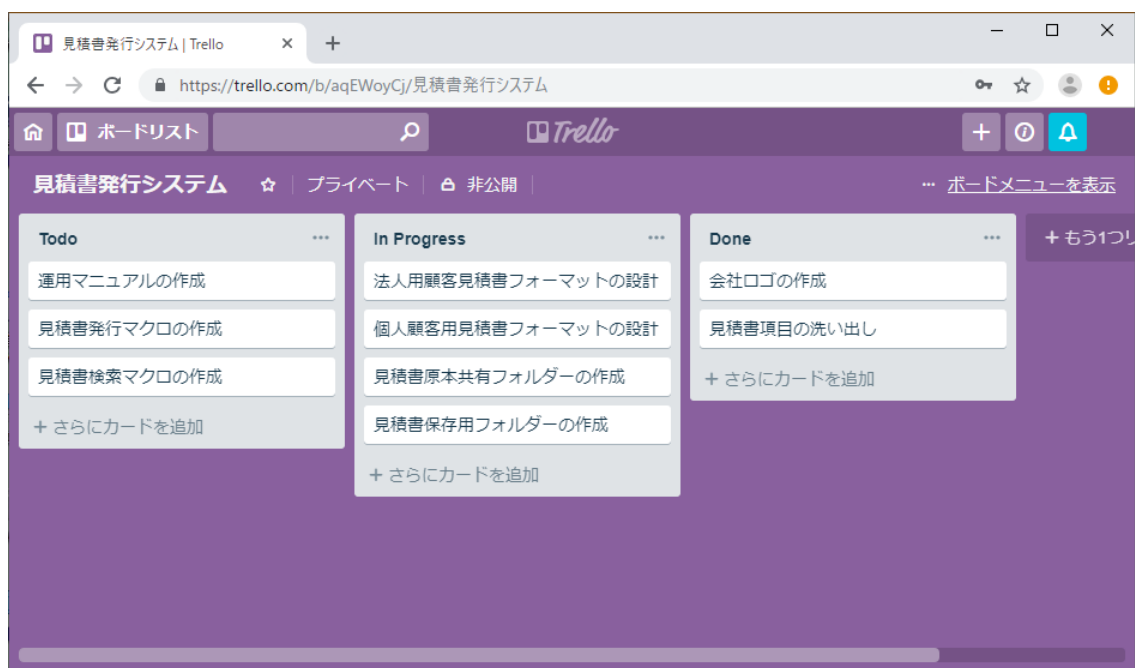
Trello (trello.com) は、アジャイルでチーム全体の進捗管理を行うときによく使われるツールである。

ボード、リスト、カードという概念があり、複数のカードで1つのリストが構成され、複数のリストで1つのボードが構成されるようになっている。

さらに一人で複数のボードを管理でき、複数のメンバーが1つのボードを共有できるようになっている。

カードは、ドラッグ・ドロップで、リスト間を簡単に移動できる。

Trello 自体は、どんな目的で使用してもいいが、アジャイル開発では、「やること (Todo)」「作業中 (In Progress)」「完了 (Done)」というようなリストを作成し、スプリント・バックログの管理を行う目的で使用されることが多い。



## 4.2.4. プロダクト・バックログとスプリント・バックログの比較

「第4章スプリント」の「スプリント計画ミーティング」の「1部と2部の対比」のセクションも参照。

	プロダクト・バックログ (PBI)	スプリント・バックログ (SBL)
目的	要求事項の見える化, 優先順位の見える化, 受入条件 (AC: Acceptance criteria) の見える化	実現手段の見える化, 担当の見える化, 工数の見える化
所有者	プロダクト・オーナー	開発チーム
見積もり手法	相対見積もり	絶対見積もり
見積もりの単位	ポイント	時間
作成のタイミング	最初のスプリント, PBL リファインメント	スプリント・プランニング, スプリント期間中

出典 : <https://www.cresco.co.jp/blog/entry/1395/>

## 4.3. 問題点リストとリスク一覧

### 4.3.1.説明

問題点 (impediment または issue) とは、今の時点ですでにプロジェクトに影響を与えているものである。

今の時点ではまだ影響が出ていないが、将来必ず影響が出るのが確実 (100%) なものも含める。

一方、リスクとは、将来起こるかもしれない不確実な事象のことである。

リスクは、それが現実となる可能性がなるべく低くなるように、かつ、現実になった場合でも影響が限定的になるようにする必要がある。つまり、マネジメントする必要がある。

問題点は、現時点での仕事のやり方に影響を与える。したがって、問題点リストは主にスクラム・マスターが所有 (管理) する。

(スクラム・マスターは、「やり方」をサポートする役割だから)

一方、リスクは、現時点での仕事のやり方には直接影響しない。

むしろ、プロダクトと同様にマネジメントすべきものである。したがって、リスク一覧はプロダクト・オーナーが所有 (管理) する。

しかし、問題点もリスクもチーム全体として取り組んでいく必要がある。

	問題点(Impediments)	リスク(Risk)
定義	現在プロジェクトに影響を与えている事象。 将来必ず影響が出るのが確実なもの	将来起こるかもしれない不確実な事象のうち、プロジェクトに影響を与えるもの
時	現在	未来
所有者	スクラム・マスター	プロダクト・オーナー
議論場所	レトロスペクトと計画ミーティング2部を中心に 開発期間全般	スプリント・レビュー時と計画ミーティング1部 を中心に開発期間全般
対処方法	スクラム・マスターを中心にメンバー全員で 解決に取り組む	メンバー全員で議論し、プロダクト・オーナー がマネジメントする
ツール等	KPT(Keep/Problem/Try)など	・可能性と影響度で評価 ・評価で優先順位づけし、対処方法を練る ・いくつかの方法論がある

## 4.3.2. リスク一覧

### 1. リスクとそのマネジメント方法

厳密には、リスクには2種類ある。

プロジェクトにプラスのインパクトを与えそうなリスクは「機会 (opportunity)」と呼ばれ、負の影響を与えるリスクは「脅威 (threat)」と呼ばれる。

リスク・マネージメントは、(a) リスクの特定、(b) リスクの事前評価、(c) 適切な対処方法というステップで行われる。

「リスクの事前評価」では、リスクの原因と影響の範囲を明らかにする。  
具体的には、リスクの(1) 現実となる可能性 (likelyhood, probability) と、(2) 影響度 (impact) という2つの観点から評価し、可能性が高く影響度も高いリスクから優先順位をつける。

「適切な対処方法」は、個々のリスクについて「可能性と影響度が低くなる方法」と「起こってしまったときの対処方法」を考察する。

### 2. リスクの例

不確実だが、もしそれが現実になった場合、プロジェクトに影響を与えるものなら何でもリスクと呼べるが、たとえば次のようなものがある。

- ① プロダクト・バックログの実行を妨げるような事象
- ② 人間関係
- ③ 知識やスキル不足
- ④ ツールの機能やパフォーマンス
- ⑤ 特定分野の難易度
- ⑥ 開発環境や職場の環境
- ⑦ 天候・気象
- ⑧ 交通機関
- ⑨ 病気
- ⑩ 家庭の事情
- ⑪ 文化や習慣の違い

## 3. リスクマネジメントの方法

リスク・マネジメントには多くの方法論があるが、前述の「可能性と影響度」を数値化して評価する Probability Impact Grid の方法を紹介する。(SBOK 参照)

可能性と影響度を 0 から 1 までの数値で表現すると決め、可能性と影響度を掛け算した値で評価する。(下表)

掛け算の結果が、0.05 以下のものを「低リスク」、0.20 以下のものを「中リスク」、それより大きなものを「高リスク」というような基準を決める。

		影響度		
		0.1 (低)	0.3 (中)	0.8 (高)
可能性	0.90	0.090	0.270	0.720
	0.75	0.075	0.225	0.600
	0.50	0.050	0.150	0.400
	0.30	0.030	0.090	0.240
	0.10	0.010	0.030	0.080
		<div>低リスク</div> <div>中リスク</div> <div>高リスク</div>		

そして、スプリント計画ミーティングなどで、リスクを特定し、可能性と影響度を数値化し、掛け算した値の順に優先順位をつけ、上記の基準で評価する。(下表)

リスク	可能性 (P)	影響度 (I)	評価 (P × I)	ランク
WPFの知識不足	0.5	0.9	0.45	高
インフルエンザ	0.1	0.8	0.08	中
客先担当者が不在がちである	0.5	0.1	0.05	中
台風	0.1	0.1	0.01	低
...	0.2	0.2	0.04	低

この表を見ると、WPF という技術の習得をもっとも優先して解決する必要があることがわかる。



## 4.3.3.問題点リスト（妨害リスト Impediments Log）

問題点（妨害点）とリスクの違いは、現在進行形で起こっているものなのか、それとも将来起こる可能性があるものなのかという点が一番大きな違いなので、問題点の項目はリスクとほぼ同じであると考えられる。

リスク一覧で「可能性と影響度の積が 1.0 のもの」ということもできる。

フォーマットは自由なので、Trello のようなツールを使って、スプリント・バックログと一緒に管理することもできる。

その他に、例えば次のようなフォーマットでもよい。

問題点・妨害項目	改善案	着手済	担当	完了	結果
ボタンAを押下したあとの処理が、基本設計書と直近の打ち合わせの議事録と異なる。	2月8日に担当者にTEL。	✓	林	✓	基本設計書が正しい。
森さんがインフルエンザで今週いっぱいお休み。	鈴木さんと青木さんが残業可。 残りは来週リスケジュールング。	✓	犬飼	✓	来週いっぱいに取り戻せる予定。
林のPCが故障した。	新規PCの購入。	✓	林		
画面Bに表示する折れ線グラフの仕様理解が困難。	顧客の設計担当者にヒアリングする。				
最近ミーティングの数が多く時間も長い。	ミーティング内容の見直し。				

## 4.4. スプリントの評価と PDCA サイクル

### 4.4.1.バーンダウンチャート

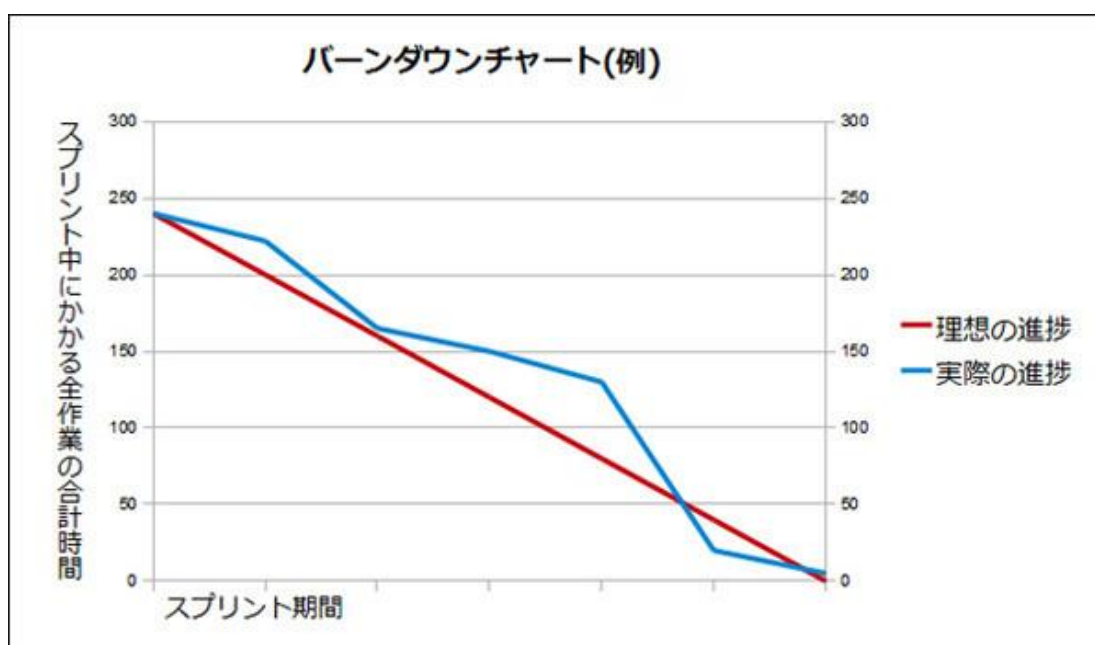
スプリント開始時に見積もった作業がどこまで進捗したかを図示したもの。

Y 軸は見積工数（ポイント）で、X 軸は1回のスプリント期間（つまり日付）である。

プロダクト・バックログのうち、その回のスプリントで行う項目の工数の合計を左端の Y 座標の値とし、右端（スプリント終了日）のゼロと直線で引く。  
これが理想の進捗を示す直線となる。

一方、日々の作業で完了したものを除いた工数を Y 座標に記録して線分を引く。  
これが実際の進捗を示す折れ線となる。

理想の直線と日々の折れ線を比較することによって、作業が順調であるかどうか判断できる。



## 4.4.2.バーンダウンチャートの書き方

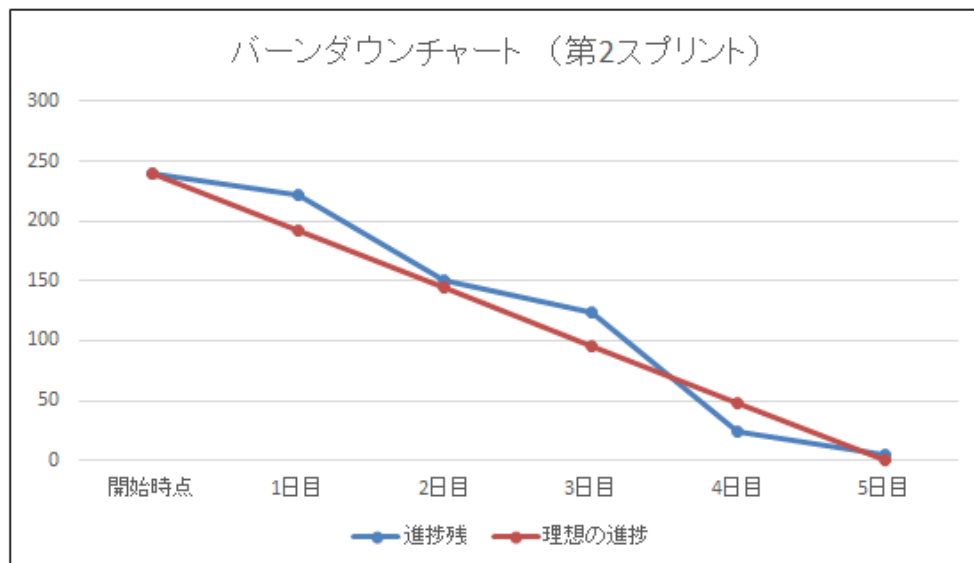
スプリント計画ミーティングで見積ったポイントの合計が 240 ポイントだとする。

5 日間のスプリントで、それぞれ 18 ポイント、72 ポイント、26 ポイント、100 ポイント、20 ポイントの作業を完了したとする。(下表の(1)の部分)

この場合のバーンダウンチャートは、下記のようになる。

- (1)は、各日の完了ポイント数。  
 (2)は、それを累積したもの。  
 (3)は、それを見積ポイント(240)から引き算したもの。つまり、残りポイント数。  
 (4)は、各日で平均的にポイントを消化していった場合の、残りポイント数。

	開始時点	1日目	2日目	3日目	4日目	5日目
(1) 作業数(ポイント)	0	18	72	26	100	20
(2) 累積作業完了数	0	18	90	116	216	236
(3) 進捗残	240	222	150	124	24	4
(4) 理想の進捗	240	192	144	96	48	0



### 4.4.3.ベロシティ

ベロシティ (velocity) は「速度」というような意味である。  
スクラムでは、「1回のスプリントで完了できる工数 (ポイント)」のことをいう。

ここでいう「工数」とは、スプリント計画ミーティングでたてた見積と同じ単位である。  
この値は、必ずしも、他のチームと同じ基準の数値を使うとは限らないので、他のチームの工数とは直接比較できない。

むしろ同一チームのパフォーマンス (作業効率) を「見える化」できるメリットがある。

スプリント終了時のレトロスペクトミーティングで、そのスプリントで完了した工数を記録しておく。  
最初のうちの慣れない時期はその工数 (完了工数) は少ないが、スプリントが進むにつれてその値が増えていく。そして、多くの場合は、一定の範囲内に収まっていく。  
一定の範囲の値の平均値がベロシティとなる。

(ここまで読んでわかったと思うが、実は、プロダクト・バックログに、各項目を完了させた時間 (分) を記入する必要はない。「1回のスプリントで何ポイント完了できたか」がわかればよい。しかし、そのチームの1ポイントあたりのおおよその所要時間を知る上では役立つ)

自分のチームのベロシティがわかれば、次回のスプリントでどのぐらいの作業が完了できるか予測できる。

また、システム全体の開発をどのぐらいの期間で完了できるかを予想することもできる。

下表の例では、このチームのベロシティは約 125 ポイントとみなすことができる。  
 $(117 + 123 + 131 + 127 + 128) \div 5 = 125.2$

	5月2週	5月3週	5月4週	6月1週	6月2週	6月3週	
スプリント	1	2	3	4	5	6	スプリント2-6 の平均
完了した工数 (ポイント)	98	117	123	131	127	128	<u>125.2</u>
↑ 1回目のスプリントは慣れない時期のため、平均の計算から外してもよい。							

仮に、システム完成までの残りの工数が 800 ポイントだと仮定すると、

$$800 \div 125.2 \approx 6.34$$

となり、今後、およそ7回のスプリントが必要であると見積もることができる。

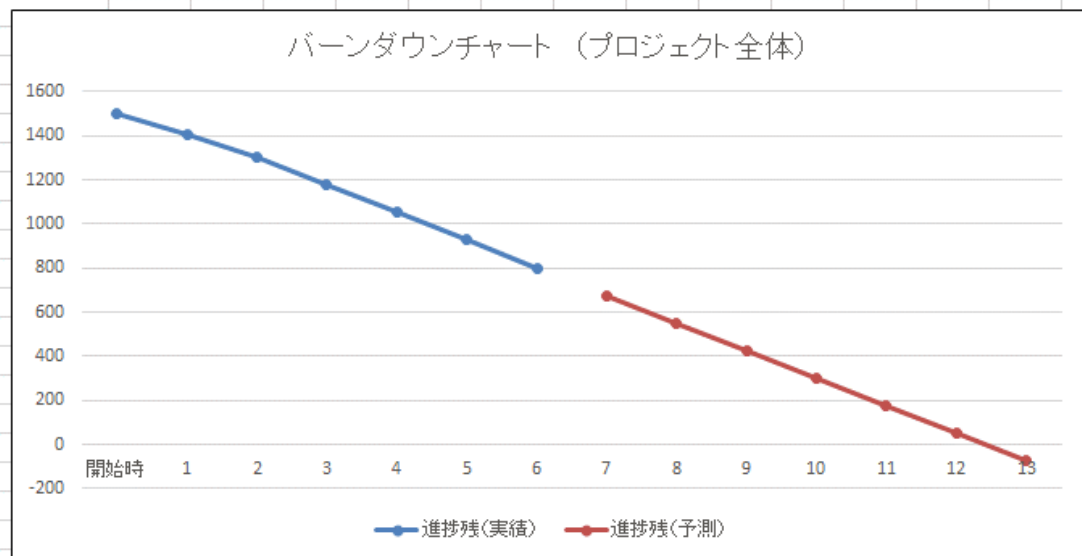
顧客の要求する納品期限がそれよりも前である場合は、メンバーを増員する、実現したい機能を減らしてもらうなどの措置が必要だが、いずれにしても、そういった措置の是非が早い段階でわかる。

前ページのベロシティをもとに、プロジェクト全体のバーンダウンチャートを作成してみると、下記のようなになる。

今後、ベロシティの値のとおりに進捗すると仮定すると（下図ではベロシティを整数の125とした）、第12スプリントで残ポイントが50、第13スプリントですべての作業が完了すると予測できる。

なお、下図の「見積総工数」がプロジェクト開始時には1500だったものが途中1520や1524という値に変化しているが、これはスプリント計画ミーティングでプロダクト・バックログの見直しを行なった実施した結果、見積ポイントが増えたということである。

	実績値							予測値						
スプリント	開始時	1	2	3	4	5	6	7	8	9	10	11	12	13
完了数	0	98	117	123	131	127	128	125	125	125	125	125	125	125
累計完了数	0	98	215	338	469	596	724	849	974	1099	1224	1349	1474	1599
見積総工数	1500	1500	1520	1520	1524	1524	1524	1524	1524	1524	1524	1524	1524	1524
進捗残(実績)	1500	1402	1305	1182	1055	928	800							
進捗残(予測)								675	550	425	300	175	50	-75

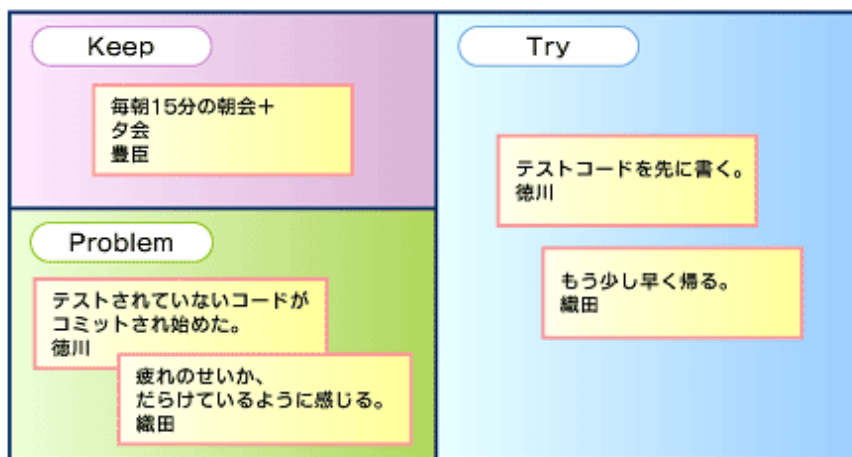


## 4.5. 振り返り用ツール（KPT）

### 4.5.1.KPT（ケイピーティー または ケプト）

Alistair Cockburn 氏が考案した The Keep/Try Reflection という手法では、「Keep these/Try these/Problems」という呼び方になっていたものを、平鍋健児氏が日本で紹介する時に「Keep/Try/Problem」という用語に代え、名前を KPT（ケプト）にしたもの。

1. Keep（うまくいったので今後も継続したいこと）
2. Problem（よくない兆候だと感じたり、うまく行かなかったり、やり方を変えたほうがいいと思ったこと）
3. Try（Keep と Problem を踏まえて、新しくチャレンジしてみたいと思ったこと）を、まとめる。



出典：<http://www.itmedia.co.jp/im/articles/0507/29/news110.html>

### 4.5.2.その他のツール

1. Fun-Done-Learn  
楽しかったこと、完了できたこと、学べたこと。  
最も初心者チーム向けの振り返り方法。
2. YWT  
よかったこと（Y）、わかったこと（W）、次にやること（T）  
その次に初心者向け。（次にやることが入っている）
3. KPT  
一般チーム向け。（問題点 Problem が入っている）
4. LAMDA  
Look（現場で見る）、Ask（見たものに対し問いかけて背景を理解する）、Model（理解したことをシンプルにモデル化する）、Discuss（議論して深める）、Act（決定したことを実行する）