

# システム開発を成功に導く 契約・プロセス・プロジェクト管理の実践

# 本日のセミナー概要

## 対象者

- ・システム開発に詳しくない営業・企画・バックオフィス向け
- ・営業、プロダクトオーナー、業務企画、バックオフィス担当
- ・システム開発の外注・内製の判断や折衝が必要なマネージャー

## タイムスケジュール

- ・オープニング（5分）：ありがちな課題
- ・第1部：トラブルを未然に防ぐ「契約」とは（30分）
- ・第2部：「開発プロセス」の理解（15分）
- ・第3部：プロジェクトを完遂する「プロジェクト管理」（30分）
- ・まとめ・質疑応答（10分）

**最初に：なぜシステム開発はトラブルが多いのか？**

## よくある失敗パターン①

「言った・言わない」問題

こんな経験、ありませんか？

「その機能、打ち合わせで話しましたよね？」

「いえ、聞いていません。それは契約外です」

根本原因：

- ドキュメントでの証跡と承認プロセスが機能していない
- 口頭での合意に頼りすぎている

## よくある失敗パターン②

### 仕様変更の想定外コスト

「仕様変更は軽微なので無償対応してください」

「いえ、これは大規模改修です。追加で〇〇万円かかります」

根本原因： 変更の影響が可視化できていない

## よくある失敗パターン③

プライマリ特有の「板挟み」

上（発注元）から：

- 「まだ決まりません」

下（開発会社）から：

- 「仕様が決まらないと動けない」
- 「それは契約外です」

結果：スケジュール遅延、コスト増加

# なぜこれらの問題が起きるのか？

## 根本原因

### 1. 契約書にやるべきことや範囲が記述されていない

- 成果物の定義が曖昧
- 「何をどこまでやるか」のスコープが不明確
- トラブル時の責任範囲、費用負担が未記載

### 2. プロセスに「バッファ」と「検証ポイント」がない

- 開発納期 = 本番稼働日（テスト期間ゼロ）
- 中間レビューなしで、終盤まで動くものが見えない

### 3. プロジェクト管理の「仕組み」がない

- 変更管理、進捗の可視化が属人的

# 本日のゴール

「契約」と「開発のプロセス」、「プロジェクト管理」の仕組みを理解して、トラブルを未然に防ぐ

## 1. 契約=エラー処理の実装

- ・「もし〇〇が起きたら？」という例外処理を契約条項に組み込む方法

## 2. 開発プロセス=手戻りを防ぐ検証ポイント設計

- ・ダブル検収、バッファ確保、中間レビューの組み込み方

## 3. プロジェクト管理=QCDのトレードオフを可視化する仕組み

- ・変更管理、進捗管理、リスク管理の実践手法

# 第1部

## トラブルを未然に防ぐ「契約」とは

# 契約締結までの流れ

【発注者（イベント主催者等）】 【受託者（開発会社）】  
【プライマリ（自社）】

ステップ1：NDA（秘密保持契約）

情報交換の前に機密保持を約束



ステップ2：MOU/LOI（基本合意）

大枠の合意、概算予算・スケジュール



ステップ3：RFI → RFP

ベンダー情報収集 → 提案依頼



ステップ4：見積り・提案

# NDA・MOU/LOIの役割

## NDA（秘密保持契約）

- 目的：情報交換の前に締結し、要件・仕様・見積り情報などの漏洩を防ぐ
- タイミング：初回打ち合わせ前、遅くとも詳細情報を共有する前

## MOU/LOI（基本合意・意向表明）

- 目的：正式契約前に「どこまで合意しているか」を可視化する
- 内容：
  - 役割分担の大枠
  - 概算予算・スケジュール上の前提
  - プロジェクトの方向性
- 特徴：まだ拘束力は弱いため、変更が多い案件に向いている

# RFI・RFPの使い分け

## RFI (Request for Information : 情報提供依頼)

- 目的：ベンダーの技術力や提供可能なサービスを把握する情報収集段階
- 実施内容：
  - ベンダーの実績・技術スタッフ・体制の確認
  - 概算費用やスケジュールの目安を把握
  - 複数ベンダーの比較検討材料を集める

## RFP (Request for Proposal : 提案依頼)

- 目的：仕様が固まっていない段階で、方向性をすり合わせる
- プライマリの役割：この段階で「翻訳作業」が始まる
  - 発注元の要望をシステム要件に変換
  - 開発会社が見積もれる形に整理

# 契約形態の基礎

## 基本の2つの契約形態

### 請負契約（成果物完成型）

- 特徴：「完成責任」を重視。成果物の完成がゴール
- 責任：納期までに仕様通りの成果物を完成させる義務
- 報酬：成果物の完成・検収をもって支払い
- 向いている場面：要件が固まっている、予算・納期を明確にしたい

### 準委任契約（業務遂行型）

- 特徴：「善管注意義務」を重視。業務遂行がゴール
- 責任：誠実に業務を遂行する義務（完成義務はない）
- 報酬：稼働時間・期間に応じて支払い

向いている場面：要件定義フェーズ、SES、保守・運用

# 請負契約の特徴

## メリット（発注者側）

-  予算・納期が明確
-  完成責任がある
-  リスクの転嫁（開発の進捗遅延や技術的トラブルはベンダー側の責任）

## デメリット（発注者側）

-  仕様変更に弱い（「ちょっとした変更」でも追加費用・納期延長）
-  見積もりリスクの上乗せ（ベンダーはリスクを見込んで高めの見積もり）
-  途中脱出が困難（違約金を払わないと解除できない）

# 準委任契約の特徴

## メリット（発注者側）

-  仕様変更への柔軟性
-  パートナーシップ型（ベンダーと協力して仕様を作り上げる）
-  契約解除が比較的容易

## デメリット（発注者側）

-  予算・納期が見えにくい
-  完成保証がない（「頑張りましたが完成しませんでした」でもOK）
-  発注者の管理負担が大きい（スコープ管理、進捗管理を発注者が主導）

# 善管注意義務とは

## 定義

「善良な管理者の注意をもって、委任事務を処理する義務」（民法第644条）

その道のプロとして、通常期待される注意を払って業務を行う義務

## 具体的な意味

- システム開発で言えば、「業界標準の技術力・知識を持つエンジニアとして、通常期待されるレベルの仕事をする」こと
- 完成義務はないが、「手抜き」や「明らかな技術的ミス」は契約違反になる
- 定期的な進捗報告・課題報告を怠ることも善管注意義務違反になる

## 請負契約と準委任契約の違い

項目	請負契約	準委任契約
責任の対象	成果物の完成	業務遂行プロセス
未完成の場合	契約不履行（責任あり）	善管注意義務を果たしていればOK
判断基準	「できたか、できなかつたか」	「適切に努力したか」

## 契約形態の比較表（発注者視点）

項目	請負契約	準委任契約
予算の明確性	○ 総額固定	△ 稼働時間次第で変動
納期の確定性	○ 納期コミットあり	△ 納期保証なし
成果物の保証	○ 完成義務あり	× 完成義務なし
仕様変更への柔軟性	× 変更ごとに追加費用	○ 柔軟に対応可能
品質責任	○ 検収基準を満たす責任	△ 善管注意義務のみ
契約解除	△ 違約金発生、困難	○ 比較的容易

# 準委任契約での「納期」の扱い

## 重要ポイント

準委任契約では、契約書に納期を記載しても、完成義務は発生しない

(民法第656条)

## 納期の法的性格

- ・ 準委任契約の納期は「目標」または「業務遂行期間」として扱われる
- ・ 納期に間に合わなくても、善管注意義務を果たしていれば契約違反ではない
- ・ 請負契約のような「納期までに完成させる義務」とは全く異なる

## プライマリが注意すべき最大のリスク

上流（発注元）：請負契約（完成義務あり、納期厳守）

下流（開発会社）：準委任契約（完成義務なし、納期は目標）

# イベント業務における3社間構造

## プライマリの立ち位置：サンドイッチ構造

[発注元（イベント主催者）]

↓ (請負契約)

[自社（プライマリ）] ← 板挟みの位置

↓ (発注)

[開発会社（再委託先）]

## 構造上のリスク

- 自社は発注元に対し「完成責任」を負っている
- しかし開発会社への発注ミスで足元をすくわれる
- 「下」の失敗を「上」にかぶる構造 = 自社がバッファとしてリスクを吸収

# 知的財産権の扱い①

## パターン1：知的財産権を発注者（自社）に完全移転

概要：

- 成果物の著作権・知的財産権を、検収完了と同時に発注者（自社）に完全移転する
- 最も一般的な形態で、プライマリとして上流への権利移転がしやすい

メリット（発注者側）：

- 成果物を自由に使用・改変・第三者への提供が可能
- 上流（イベント主催者など）への権利移転・利用許諾がスムーズ
- 将来的な機能追加や保守を別の会社に依頼できる

## 知的財産権の扱い②

### パターン2：知的財産権を開発会社に残す（利用許諾型）

概要：

- ・著作権は開発会社が保持し、発注者には「利用許諾」のみを与える
- ・特に大手SI会社では、この契約形態を要求する場合が多い

なぜ大手SI会社がこの形態を好むのか：

- ・開発したシステムのコア部分を他の案件でも再利用したい（資産化）
- ・自社の技術ノウハウを社外に流出させたくない

プライマリとしてのリスク：

- ・上流契約で「権利を主催者に移転する」と約束している場合、矛盾が生じる
- ・下流で利用許諾しか得られないと、上流への権利移転義務を果たせない

## 知的財産権の扱い③

パターン3：見積もり減額交渉の材料として使う

交渉戦略：

知的財産権を開発会社に残すことを認める代わりに、見積もりの減額交渉を行う

交渉の論理：

- 開発会社側：「知的財産権を自社に残せば、他案件での再利用が可能になり、開発コストを下げられる」
- 発注者側：「それなら、見積もりを〇〇%減額してください」

注意点：

- この交渉は、上流契約で権利移転が必須でない場合のみ有効
- イベント主催者が「権利を譲渡せよ」と要求している場合は使えない

# 再委託のチェックポイント

プライマリとして押さえるべきポイント

## 1. 上流契約と下流契約の連動を確認

上流契約（主催者 ← 自社）	下流契約（自社 → 開発会社）	リスク
権利を主催者に移転	権利を自社に移転	✓ OK
権利を主催者に移転	利用許諾のみ	✗ NG
利用許諾のみ	利用許諾のみ	✓ OK

## 2. 再委託の承諾

- ・自社がさらに下請けを使う場合、主催者の許可が必要か確認
- ・無断再委託で契約違反、信頼失墜のリスク

## 3. SLA（サービスレベル）のバック・トゥ・バック契約

# SOW (Statement of Work) とは

## 定義

\*\*SOW (Statement of Work: 作業範囲記述書) \*\*は、契約書本体とは別に、プロジェクトの具体的な作業内容を詳細に定義する文書

## 契約書とSOWの役割分担

項目	契約書	SOW
内容	法的な権利義務関係	具体的な作業内容・成果物
記載例	契約金額、支払条件、損害賠償	作業スコープ、成果物の詳細、マイルストーン
性質	包括的・抽象的	具体的・詳細
変更頻度	低い（基本的に変更しない）	高い（フェーズごとに更新）

# SOWの重要性

## 理由1：「契約外」トラブルの防止

よくある失敗例：

- 開発会社：「その機能はSOWに含まれていないので、追加費用が必要です」
- 発注側：「いや、当然含まれていると思っていた」

対策：

- SOWに作業範囲を具体的に列挙する（箇条書きで明記）
- 「含まれるもの」だけでなく、「含まれないもの（Out of Scope）」も明記する
- 例：「データ移行作業は本SOWの範囲外とする」

# SOWに含めるべき項目

## 必須項目チェックリスト

1. プロジェクト概要 - プロジェクト名、目的、背景
2. 作業範囲 (スコープ)
  - 実施する作業 : 要件定義、設計、開発、テスト、リリース支援など
  - 実施しない作業 (Out of Scope) : 運用保守、データ移行、既存システム改修など
3. 成果物 (Deliverables) - 各成果物の名称、内容、形式、納品期日
4. マイルストーン・スケジュール - 主要な節目と各マイルストーンの期日
5. 前提条件・制約事項 - 発注側の提供物、第三者の協力、技術的的前提
6. 役割分担 (RACI) - 誰が何を担当するか

# 準委任契約でのSOWの重要性

なぜ準委任契約でSOWが特に重要なのか

理由1：完成義務がないからこそ「境界」が必要

準委任契約には完成義務がないため、「どこまでやるか」の境界が曖昧になりがち

SOWで明記すべき内容：

- 実施する作業： 要件ヒアリング（〇〇回）、要件定義書作成、レビュー対応（〇〇回まで）
- 実施しない作業（Out of Scope）： 基本設計、詳細設計、プロトタイプ開発
- 成果物の範囲： 要件定義書（〇〇ページ程度、Word形式）、議事録

## 理由2：稼働ベースの報酬だからこそ、稼働内容の定義が必要

項目	明記すべき内容
稼働体制	シニアエンジニア1名（単価〇〇万円/月）、ジュニアエンジニア2名（単価〇〇万円/月）
稼働期間	2024年4月1日～2024年6月30日（3ヶ月間）
想定稼働時間	1ヶ月あたり160時間（週40時間×4週）を想定
上限設定	総稼働時間480時間を上限とし、超過時は事前承認を要する
報告義務	毎週金曜日に作業実績報告書を提出

# 多段階契約（フェーズ契約）

## 一括請負の危険性

問題点：

- 仕様未定での総額固定はギャンブル
  - 開発会社はリスクを見込んで高めの見積もりを出す
  - 発注側は「ぼったくり？」と感じる
- 途中脱出ができない
  - 「この会社、ダメだ」と気づいても契約上逃げられない

## 解決策：多段階契約（フェーズ分割）

### Phase 1：要件定義・設計（準委任契約）

- 目的：パートナーとして仕様を固める
- 成果物：要件定義書、基本設計書、画面エック、詳細見積もり（Phase 2用）

# Phase 1とPhase 2のメリット

## メリット

### 1. 見積もり精度の向上

- 仕様が固まってから金額を確定できる

### 2. Exit Strategy (脱出装置)

- Phase 1で「合わない」と判断したら、Phase 2は別の会社に発注できる

### 3. リスクの分散

- 大金を一度に投じるリスクを避けられる

## デメリット

### 1. 契約手続きの手間

- 2回契約を結ぶ必要がある

# 契約書は「エラー処理」の実装

プログラムコードとの類似

エンジニア的思考の応用：

プログラミングで、入力値が正しい場合しか動かないコードは書かない

```
try {  
    // 正常系の処理  
} catch (例外) {  
    // エラー時の処理  
}
```

契約書も同じ。「無事に納品されました」というハッピーエンドしか書いていない契約書は、バグだらけのプログラムと同じ

PMの最大の防御

事前に「最悪の事態」を想定し、その時の挙動を契約条項にしておく

## 契約書に実装すべき「例外処理」

Catch 1：受入テストでNGが出たら？

契約条項例：

- ・「検収時に不具合が発見された場合、受託者は〇〇日以内に無償で修正する」
- ・「〇回以上の修正が必要な場合、発注者は契約を解除できる」

Catch 2：主催者の都合でイベントが中止になつたら？

契約条項例：

- ・「発注者都合で契約解除する場合、既に発生した費用+〇〇%を支払う」
- ・「不可抗力（災害等）の場合、双方協議の上、費用を按分する」

Catch 3：仕様変更が止まらなかつたら？

契約条項例：

# 第1部のまとめ

## 契約で押さえるべきポイント

-  契約は「信頼関係」だけでは守れない。明文化が必須
-  一括請負はリスク。フェーズ分割で「見極め」と「脱出装置」を確保
-  契約書は「エラー処理」。トラブル時の責任・費用・手順を事前定義
-  プライマリは上下の契約を連動させ、リスクを一方的に負わない設計を
-  SOWで作業範囲を明確化し、「言った・言わない」を防ぐ
-  準委任契約では納期に完成義務がない。上流請負・下流準委任は最大のリスク

## 第2部

### 「開発プロセス」の理解

# 開発プロセスの全体像

## 典型的な開発の流れ

要件定義 → 設計 → 実装 → テスト → リリース → 保守・運用

### 各フェーズの期間目安：

- 要件定義：全体の20-30%
- 設計：全体の20-30%
- 実装：全体の30-40%
- テスト：全体の20-30%
- リリース：1-2週間

重要： これらは重なり合うこともあり、厳密に区切られるわけではない

# 各フェーズの概要

フェーズ	目的	主な成果物	発注側の役割
要件定義	「何を作るか」を明確にする	要件定義書、機能一覧、画面遷移図	最も重要。業務要件を正確に伝える
設計	「どう作るか」を決める	基本設計書、詳細設計書、DB設計書	画面モックや設計書のレビュー・承認
実装	実際にコードを書く	ソースコード、単体テスト結果	定期的な進捗確認、デモの確認
テスト	品質を担保する	テスト仕様書、テスト結果報告書	受入テストの実施、不具合の優先順位判断
リリース	本番環境へ移行する	リリース手順書、運用マニュアル	リリース判断、ユーザー周知

# ウォーターフォール開発

## 特徴

- 各工程を順番に完了させる
- 前工程の完了が次工程の開始条件
- 計画重視、ドキュメント重視

## メリット

- スケジュール・予算が見えやすい
- 大規模プロジェクトの管理がしやすい
- 契約（請負）と相性が良い

## デメリット

- 途中で動くものが見えない（ブラックボックス化）

# アジャイル開発

## 特徴

- 短期間（1-4週間）のサイクルで開発とリリースを繰り返す
- 動くものを早期に確認しながら進める
- 変化への対応を重視

## メリット

- 早期に動くものが見られる（安心感）
- 仕様変更に柔軟に対応できる
- フィードバックを反映しやすい

## デメリット

- 最終的な総額・納期が見えにくい

# どちらを選ぶべきか

## 判断基準

項目	ウォーターフォール	アジャイル
要件の確定度	確定している	不確定、変わりやすい
予算の制約	固定したい	柔軟に調整可能
納期の制約	厳格（イベント開催日など）	柔軟
発注側の関与	要件定義と受入テストのみ	継続的に必要
プロジェクト規模	大規模	小～中規模

## 実務的な落とし穴

- 「アジャイルなら柔軟」と聞いて安易に選ぶと、予算超過・終わらないプロジェクトになる

# チェックポイント①：中間レビューの設定

## 問題

- ・ ウォーターフォールでは「テスト工程まで動くものが見えない」
- ・ 「期待と違った」と気づくのが遅すぎる

## 対策

設計レビュー：画面モック、DB設計を必ずレビューする

開発途中のデモ：月1回程度、動くものを見せてもらう

契約条項に明記：「月次で進捗報告とデモを実施する」

## 重要

中間レビューを契約書やSOWに明記することで、開発会社に「見せる義務」を課す

## チェックポイント②：テスト期間の確保

### 問題

- ・「開発完了 = 本番リリース日」になっている
- ・テストで不具合が見つかっても修正時間がない

### 対策：ダブル検収の設計

開発会社の検収： システムテスト完了

発注側の検収： 受入テスト完了

バッファ期間： 本番リリース日の2-4週間前を納期に設定

不具合修正期間： テスト後の修正期間を明示的に確保

### 契約条項例

- ・開発完了期日：2025年3月15日

## チェックポイント③：受入基準の明確化

### 問題

- ・「検収」の基準が曖昧
- ・「これで完成ですか？」「まだ直すべきでは？」で揉める

### 対策：受入基準（Acceptance Criteria）を事前定義

機能要件：「〇〇ができること」を列挙

性能要件：「同時100ユーザーで5秒以内にレスポンス」

品質基準：「致命的バグ0件、重大バグ3件以内」

### 具体例：チケット販売システムの受入基準

- ・ ✓ 一般ユーザーがチケットを購入できること
- ・ ✓ 決済が正常に完了すること

# 技術動向①：クラウド・サーバーレス

なぜ技術動向を知る必要があるのか

発注側の課題：

- 開発会社から「最新技術で開発します」と提案される
- 「それって本当に必要?」「コストに見合う?」と判断できない

本セクションの目的：

- 「提案の妥当性を判断できる」レベルの知識を持つ
- コスト削減・リスク低減につながる選択肢を知る

サーバーレスとは

- サーバーを意識せず、プログラム（関数）だけを配置する
- アクセスがあった時だけ実行され、その分だけ課金

# サーバーレスのメリットと注意点

## メリット

- 初期コストの削減（大規模なハードウェア投資が不要）
- 柔軟性（イベント時だけサーバーを増強、終了後は縮小）
- 使わない時間帯はコストゼロ

## 注意点

### 1. ランニングコストの見積もりが難しい

- 「使った分だけ」なので、月々の費用が読めない
- アクセスが急増すると、予想外のコストが発生
- 対策： 契約書に「月額上限の設定」を明記

### 2. コールドスタート問題

## 技術動向②：ローコード/ノーコード開発

### 定義

ローコード/ノーコード： プログラミングをほとんど書かずに、GUI操作でシステムを構築する手法

### 主要ツール：

- **Salesforce**：顧客管理（CRM）
- **kintone**：業務アプリ作成
- **PowerApps (Microsoft)**：社内業務アプリ

### メリット（発注側）

- 開発期間の短縮（従来3ヶ月 → 1ヶ月で完成）
- コストの削減（フルスクラッチの50-70%程度）
- 自動化の可能性（データマigrationsも簡単化可能）

## 技術動向③：SaaS（作らずに借りる）

SaaS (Software as a Service) とは

定義：インターネット経由で利用するソフトウェアサービス

例：

- Slack / Teams : チャットツール
- Salesforce : 顧客管理
- Stripe / PayPal : 決済サービス

「作る」 vs 「借りる」の判断

項目	作る（フルスクラッチ）	借りる（SaaS）
初期コスト	高（500万円～）	低（数万円～）
ランニングコスト	低（保守費用のみ）	高（月額課金）

# 技術提案の評価チェックリスト

開発会社から技術提案を受けたら、以下を確認

## 1. 「最新技術」の必然性

- [ ] なぜその技術が必要なのか、ビジネス上のメリットは？
- [ ] 枯れた技術（実績のある技術）で代替できないか？
- [ ] 「最新技術」を使うことで、リスクは増えないか？

## 2. コストの妥当性

- [ ] 初期コストだけでなく、ランニングコストも試算されているか？
- [ ] SaaS・ローコードで代替した場合のコスト比較はしたか？
- [ ] 5年間のTCO（Total Cost of Ownership：総保有コスト）は？

## 3. ベンダーロックインのリスク

## 第2部のまとめ

### 開発プロセスで押さえるべきポイント

- 開発プロセスは「要件定義」が最重要。ここでの曖昧さが後の火種になる
- ウォーターフォール vs アジャイルは、要件の確定度と予算制約で判断
- 中間レビュー・テスト期間・受入基準を契約時に明記することで手戻りを防ぐ
- 「動くものを早めに見る」仕組みがトラブル回避の鍵
- 技術動向を知ることで、開発会社の提案を適切に評価できる
- クラウド・サーバーレス・ローコード・SaaSは、コスト削減・期間短縮の選択肢
- 「最新技術」に飛びつかず、費用対効果・リスクを冷静に評価

## 第3部

プロジェクトを完遂する「プロジェクト管理」

# プロジェクト管理とは何か

## 定義

プロジェクト管理（Project Management）とは、限られたリソース（予算・時間・人員）で、目標を達成するための仕組み

システム開発では、「契約で決めたこと」を「決められた期間・予算」で実現することがゴール

## プライマリの立場での課題

上（発注元）からの要求：

- 「予定通りに完成せよ」
- 「予算オーバーは許さない」
- 「品質も妥協するな」

# プロジェクト管理の5つの柱

## 1. QCD管理 (Quality / Cost / Delivery)

- ・品質・コスト・納期のバランスを取る
- ・トレードオフを可視化し、優先順位を判断する

## 2. スコープ管理 (Scope Management)

- ・「何をやるか・やらないか」を明確にする
- ・スコープクリープ（限界ない機能追加）を防ぐ

## 3. 進捗管理 (Progress Management)

- ・計画と実績のギャップを可視化する
- ・遅延の兆候を早期発見する

## 4. 変更管理 (Change Management)

# PMBOKとは

**PMBOK = Project Management Body of Knowledge**

発行元： PMI (Project Management Institute : プロジェクトマネジメント協会)

位置づけ： プロジェクト管理の国際標準・ベストプラクティス集

適用範囲： システム開発だけでなく、建設、製造、イベント、研究開発など、あらゆる業種のプロジェクトに適用可能

**なぜPMBOKを知るべきか？**

- 世界中で使われている共通言語
- 開発会社やコンサルが「PMBOK的に...」と言ったとき、何を指しているか理解できる
- プロジェクト管理の「型」を知ることで、トラブルを未然に防げる

# PMBOK第7版の特徴

## 第6版までとの大きな違い

項目	第6版まで	第7版 (2021年)
構成	10の知識エリア	8つのパフォーマンスドメイン（より柔軟で統合的）
アプローチ	ウォーターフォール中心	アジャイル・ウォーターフォール両対応
思想	プロセス重視	原則ベース・成果重視

## 第7版のメリット

- より実践的で柔軟
- 「何をすべきか」より「何を達成すべきか」に焦点

# PMBOK第7版の12の原則

## プロジェクトマネジメントの基本的な考え方

1. スチュワードシップ（責任ある管理者としての行動）
2. チーム（協働と相互尊重）
3. ステークホルダー（利害関係者への積極的な関与）
4. 價値（ビジネス価値の創出に焦点）
5. システム思考（全体最適の視点）
6. リーダーシップ（状況に応じたリーダーシップ）
7. テーラリング（プロジェクトに応じたカスタマイズ）
8. 品質（品質への継続的な注力）
9. 複雑さ（複雑性への適切な対応）
10. リスク（リスクへの積極的な対処）

# 原則3：ステークホルダー

## プライマリの構造

[発注元（イベント主催者）]



[自社（プライマリ）] ← 板挟みの位置



[開発会社（再委託先）]

## 実践のポイント

- 上（発注元）と下（開発会社）の期待値を常に管理
- 「言った・言わない」を防ぐため、全ステークホルダーとの合意を文書化
- 定期的なコミュニケーション（週次定例など）

# 原則5：システム思考

## システム思考とは

- ・部分最適ではなく、全体最適を考える
- ・一つの変更が他にどう影響するかを考える

## 具体例：仕様変更の影響

仕様変更の依頼



開発コスト増 (+100万円)



納期への影響 (+2週間)



他の機能への影響 (リソース取られる)

## 原則7：テーラリング

### テーラリングとは

PMBOKの「型」を盲目的に適用するのではなく、プロジェクトの特性に合わせてカスタマイズする

### 判断基準

- プロジェクトの規模（小規模 vs 大規模）
- 要件の確定度（固定 vs 流動的）
- チームの経験値
- ビジネス上の制約（納期厳守 vs 予算固定）

### 実践例

ケース	テーラリングの例
-----	----------

# PMBOK第7版の8つのパフォーマンスドメイン

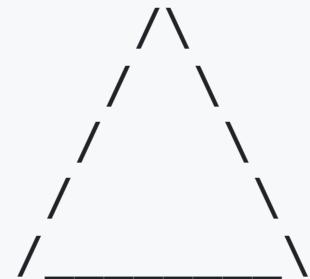
プロジェクトを成功させるために取り組むべき8つの重要領域

1. ステークホルダー - すべての人・組織との関係構築
2. チーム - チーム編成、協働、モチベーション管理
3. 開発アプローチとライフサイクル - ウォーターフォール、アジャイル、ハイブリッドの選択
4. 計画 - スコープ、スケジュール、予算、品質、リスクなどの計画策定
5. プロジェクト作業 - 作業を実際に遂行し、リソースを配分
6. デリバリー - 成果物を提供し、期待される価値を実現
7. 測定 - パフォーマンスを測定・評価し、意思決定に活用
8. 不確かさ - リスク（マイナス）と機会（プラス）を管理

# QCDのトレードオフ

## プロジェクトの鉄の三角形

Quality (品質)



Cost Delivery

(コスト) (納期)

### 基本原則

- 3つ全てを同時に最大化することはできない
- 何かを優先すれば、他が犠牲になる

# トレードオフのパターン

## パターン1：納期優先（Delivery First）

シナリオ：イベント開催日が確定している

トレードオフ：

- **Quality ↓**：一部機能を簡易実装、バグ修正を後回し
- **Cost ↑**：人員追加、残業増でコスト増

プライマリの判断：「最小限の機能（MVP）でリリース。残りは運用開始後にリリース2.0で対応」

## パターン2：予算優先（Cost First）

シナリオ：予算が固定されている

トレードオフ：

- Delivery ↓：納期を延ばして、少人数で開発
- Quality ↓：一部機能を削る、手動運用で代替

プライマリの判断：「Phase 1は必須機能のみ。Phase 2は予算が取れたら実施」

## パターン3：品質優先 (Quality First)

シナリオ： 大規模イベント、障害が許されない

トレードオフ：

- Cost ↑ : テスト期間延長、冗長化設計でコスト増
- Delivery ↓ : 十分なテスト期間を確保し、納期後ろ倒し

プライマリの判断： 「リリース日を1ヶ月延期。負荷テスト・障害訓練を徹底」

# トレードオフの可視化

## 問題

「納期も守れ、予算も守れ、品質も妥協するな」という無理な要求

対策：トレードオフを数値化して見る

選択肢	納期	予算	品質（主要機能）
A案：最速リリース	4/15	1,200万円	80%（一部後回し）
B案：予算優先	5/31	1,000万円	90%（時間かけて実装）
C案：品質最優先	6/30	1,500万円	100%（全機能完璧）

## PMの役割

上記3案を提示し、発注元に選ばせる。丸投げで『全部やれ』は受けない

# 進捗管理の基本

## 目的

- ・「計画通りに進んでいるか」を可視化する
- ・遅延の兆候を早期発見し、対策を打つ

## 失敗パターン

- ・「順調です」という報告だけで、実態が見えない
- ・終盤になって「実は大幅遅延」が発覚

## 進捗の「見える化」

1. マイルストーン管理：プロジェクトの重要な節目を設定
2. WBS (Work Breakdown Structure)：大きな作業を小さなタスクに分解
3. バーンダウンチャート：残作業量を可視化（アジャイル開発でよく使われる）

# マイルストーン・WBS

## マイルストーン管理

設定例：

- M1：要件定義完了 (2025/2/28)
- M2：基本設計完了 (2025/3/31)
- M3：実装完了 (2025/5/15)
- M4：システムテスト完了 (2025/6/15)
- M5：受入テスト完了 (2025/6/30)
- M6：本番リリース (2025/7/15)

チェック方法：

各マイルストーンで「完了の定義」を明確にする

# 遅延の早期発見

## 危険信号 (Red Flag)

### 1. 「順調です」が続く

- 具体的な%や成果物がない報告は要注意
- 対策：「何が完了したか、成果物を見て」

### 2. 同じタスクがずっと「90%完了」

- 「あと少し」が永遠に終わらない
- 対策：「残り作業を具体的にリストアップして」

### 3. マイルストーンが遅延

- M1が1週間遅れると、最終納期は1ヶ月遅れる（経験則）
- 対策：初期段階の遅延こそ厳しくチェック

# 変更管理 (CR) とは

## 定義

変更管理 (Change Management) とは、仕様変更を適切にコントロールし、プロジェクトを破綻させない仕組み

## 問題

- ・ 「ちょっとした変更」の積み重ねでスコープが肥大化
- ・ 「これくらい無償対応してよ」「いや、追加費用が必要」で揉める

## 結果

- ・ スコープクリープ (Scope Creep) : 際限なく機能が増える
- ・ 納期遅延、予算超過、品質低下

# CRプロセス

## 基本フロー

1. 変更要求の提出



2. 影響分析 (Impact Analysis)

- コスト影響 : +〇〇万円
- 納期影響 : +〇週間
- 品質影響 : リスク評価



3. 承認・却下の判断



4. 変更の実施



# CRの実例

## ケース：「ちょっとした」変更要求

要求内容：

「チケット購入後に、確認メールを送るようにしてほしい」

一見： 軽微な変更に見える

実際の影響分析：

- メール送信機能の実装、テンプレートのデザイン、リトライ処理、履歴保存、テスト
- 合計：12人日 = 約120万円
- 納期影響：+1週間

判断：

不認可の場合：又答追加 → ニコ → プロット（化機能を後回し）

# 変更管理表

全ての変更を記録する

CR番号	提出日	要求内容	影響(コスト)	影響(納期)	判断	承認者	備考
CR-001	2/15	確認メール送信	+120万円	+1週間	承認	○○部長	Phase 1に含める
CR-002	3/1	決済方法追加	+200万円	+2週間	却下	○○部長	Phase 2で検討
CR-003	3/10	UI色変更	+5万円	影響なし	承認	PM判断	軽微な変更

## ポイント

- 小さな変更も記録する（累積で大きな影響になる）

# MoSCoW分析

## 要件の優先順位付け

- Must (必須) : これがないとリリースできない
- Should (重要) : あるべきだが、なくても代替可能
- Could (あればよい) : あると嬉しいが、優先度低い
- Won't (今回はやらない) : 将来的には検討するが、Phase 1では対象外

## 実務での使い方

Must :

- チケット購入機能
- 決済機能（クレジットカード）

Should :

# リスク管理の基本

## リスク (Risk) とは

「もし〇〇が起きたら、プロジェクトに悪影響が出る」という不確実性

## リスク管理の目的

- リスクを事前に洗い出す
- 発生確率と影響度を評価する
- 対策を準備し、被害を最小化する

## 典型的なリスク

### 技術リスク：

- 「新技術を使うが、社内に経験者がいない」
- 「外部APIの仕様が不明確」

# リスク評価マトリクス

発生確率 × 影響度でリスクを評価

影響度

↑

高 | [中] [高]

| リスクB リスクA

|

低 | [低] [中]

| リスクD リスクC

|

|

→ 発生確率

低 高

例

# リスク対応戦略

## 4つの対応戦略

### 1. 回避 (Avoid)

- リスクそのものを排除する
- 例：新技术を使わず、枯れた技術で実装

### 2. 軽減 (Mitigate)

- リスクの発生確率や影響を減らす
- 例：テスト期間を2倍確保する、予備メンバーを確保

### 3. 転嫁 (Transfer)

- リスクを第三者に移す
- 例：外部APIのSLAを契約条項に盛り込む、保険加入

# エスカレーション

エスカレーション (Escalation) とは

問題が発生したとき、上位の意思決定者に判断を仰ぐこと

いつエスカレーションすべきか

1. マイルストーンの遅延

- 「要件定義が1週間遅れそう」 → 即座にエスカレーション

2. 予算超過の見込み

- 「このままだと予算オーバー」 → 早期にエスカレーション

3. 重大な品質問題

- 「致命的なバグが見つかった」 → 即座にエスカレーション

4. 約外の要求

# エスカレーションの階層

## 3段階のエスカレーション

Level 3：経営層

↑ (重大な契約変更、プロジェクト中止判断)

|

Level 2：部長・事業部長

↑ (予算超過、納期大幅遅延)

|

Level 1：PM・リーダー

(日常的な課題解決)

## エスカレーション時のコミュニケーション

悪い例：「ヤバいです。どうしましょう？」

良い例：

# プロジェクト管理ツール

## コミュニケーションの仕組み化

失敗パターン：

- 「言った・言わない」問題
- メールが流れで情報が消える
- 誰が何を決めたか分からない

対策：

- 定例ミーティング：週次・隔週で進捗確認
- 議事録の徹底：全ての合意事項を記録
- 課題管理ツール：Jira、Redmine、Backlog等

## 第3部のまとめ

### プロジェクト管理で押さえるべきポイント

- PMBOKは国際標準のプロジェクト管理知識体系。第7版の8つのパフォーマンスドメインは、実践的なプロジェクト管理の指針
- QCDは全て最大化できない。トレードオフを可視化し、意思決定を促す
- 進捗管理は「見える化」が全て。マイルストーン・WBS・バーンダウンで早期発見
- 変更管理（CR）で「ちょっとした変更」を許さない。影響分析と承認プロセスを徹底
- リスク管理は「転ばぬ先の杖」。事前に想定し、対策を準備する
- エスカレーションは「丸投げ」ではなく「選択肢の提示」。事実とデータで交渉する

# 全体のまとめ

## まとめ①：契約

トラブルを未然に防ぐ「契約」のポイント

契約は「エラー処理」の実装

- ・ 「もし〇〇が起きたら？」という例外処理を契約条項に組み込む
- ・ 受入テストNG、イベント中止、仕様変更、納期遅延への対応を明文化

多段階契約（フェーズ分割）がベスト

- ・ Phase 1（準委任）で見極め、Phase 2（請負）で完成をコミット
- ・ Exit Strategy（脱出装置）を確保

上流契約と下流契約を連動させる

- ・ プライマリは上下の契約を連動させ、リスクを一方的に負わない設計を
- ・ Back-to-Back契約、知的財産権の連動、SLAの連動

## まとめ②：開発プロセス

手戻りを防ぐ「開発プロセス」のポイント

要件定義が最重要

- ここでの曖昧さが後の火種になる
- 発注側が業務要件を正確に伝える責任

中間レビュー・テスト期間・受入基準を契約時に明記

- 中間レビューで「動くものを早めに見る」仕組み
- ダブル検収の設計（開発会社の検収 → 発注側の検収）
- バッファ期間を確保（本番リリース日の2-4週間前を納期に設定）

技術提案を適切に評価

- クラウド・サーバーレス・ローコード・SaaSは、コスト削減・期間短縮の選択肢 82

## まとめ③：プロジェクト管理

プロジェクトを完遂する「プロジェクト管理」のポイント

QCDのトレードオフを可視化

- ・品質・コスト・納期は全て最大化できない
- ・複数の選択肢を提示し、発注元に判断させる

進捗管理は「見える化」が全て

- ・マイルストーン・WBS・バーンダウンで早期発見
- ・「順調です」報告を鵜呑みにせず、成果物で確認

変更管理（CR）で「ちょっとした変更」を許さない

- ・影響分析と承認プロセスを徹底
- ・変更管理表で全ての変更を記録

# 実践のために

## 明日からできること

### 1. 契約書・SOWをチェックする

- 「エラー処理」が実装されているか？
- 「Out of Scope」が明記されているか？
- 上流契約と下流契約が連動しているか？

### 2. プロジェクト計画に中間レビューを組み込む

- 月次で進捗報告とデモを実施する
- ダブル検収の設計
- バッファ期間を確保

### 3. 変更管理表を作成する

## 質疑応答

ご質問・ご相談をお受けします

ご清聴ありがとうございました