性能要件を実現する性能品質確保戦略

The Strategy for Ensuring System Performance Which Satisfies the Performance Requirements

川口真一，下村剛士

要　約　システム性能は，多様なシステム構成要素に加えシステムの利用状況の影響も受ける．それ故，性能問題の原因も多様であり，その検出や是正は難しい．結果として，性能問題はシステム開発終盤になって顕在化する傾向が強い．

開発過程で性能問題を作り込んでしまうことと，その問題検出が遅いということの 2 点について要因分析を行い，対策を立案した．対策は開発工程全般にわたる四つのステップに整理し，これを性能品質確保戦略として定義した．

取引管理システム（後続稼働）プロジェクトで性能品質確保戦略を実践し，機能品質実現と同時に性能品質実現を達成することができた．性能品質を確保するためにはシステム開発全工程を通じ戦略的に性能を管理することが有効である．

Abstract　System usage conditions as well as a variety of system elements influence system performance. Thus the causes of performance problems are various, and it is difficult to find out those problems. As a result, many of the performance problems tend to become obvious in the end of the system development project.

We analyzed the reasons why the sources of performance problems are made through the system development and why it takes so much time to find out those problems, and worked out the answers for those reasons. We coordinated the answers in four steps which executed during the system development, and defined it as “The Strategy for Ensuring System Performance”.

On the project of JAPAN POST trade management system development, we practiced the strategy and got a result that we ensured system performance and functional quality at the same time. To ensure system performance, it is necessary to manage the system performance strategically through the system development.

1. はじめに

システムに対する品質要求のひとつに性能がある．しかし，性能への要求や検証基準について，同じ品質要求である機能品質では個々の機能ごとに様々なバリエーションを網羅的に定義するのに比べると，性能品質では対象や前提条件およびバリエーションなどの定義が粗い．また，機能品質が高まらないと性能検証の意味がないという理由から，性能検証は後回しとなりやすい．その結果，性能品質の評価基準が曖昧だったり，システム開発終盤になって性能問題が発生したりするケースが多い．

性能品質は，システム構成や H/W リソースの制約の下，アプリケーションプログラム（以降，AP）のアーキテクチャ，機能処理アルゴリズム，DB 構造やデータ量，業務処理パターンと多岐にわたる要素が絡み合った結果として実現される．そのため，性能品質を確保するに

は，APのこれらの要素に対する性能面での対応・対策がシステム開発工程全般にわたって必要である．

日本ユニシスには，大規模トランザクションシステム開発のテスト工程において性能品質を段階的に成長させる戦略（性能品質成長戦略*1）を実践した経験がある[1]．取引管理システム（後続稼働）*2プロジェクト（以降，本プロジェクト）では，システム開発の全工程にわたって性能品質確保を戦略的に進めるように性能品質成長戦略を昇華させて適用することで，良い結果を得た．本プロジェクトの事例を紹介しつつ，性能品質確保について論じる．2章で性能問題が発生する原因，3章で性能品質を確保する戦略，4章で本プロジェクトでの実践内容，5章でまとめを述べる．

2. 性能問題はなぜ発生するのか

本章では，どのようなAPが性能問題を引き起こすのか，また性能問題がなぜシステム開発プロジェクトの終盤や本番稼働後に発生するのかについて，過去のシステム開発事例を基に分析する．

2.1 性能問題につながるAPの特徴分析

性能問題が発生したAPを分析すると，図1のとおり共通する特性がある．

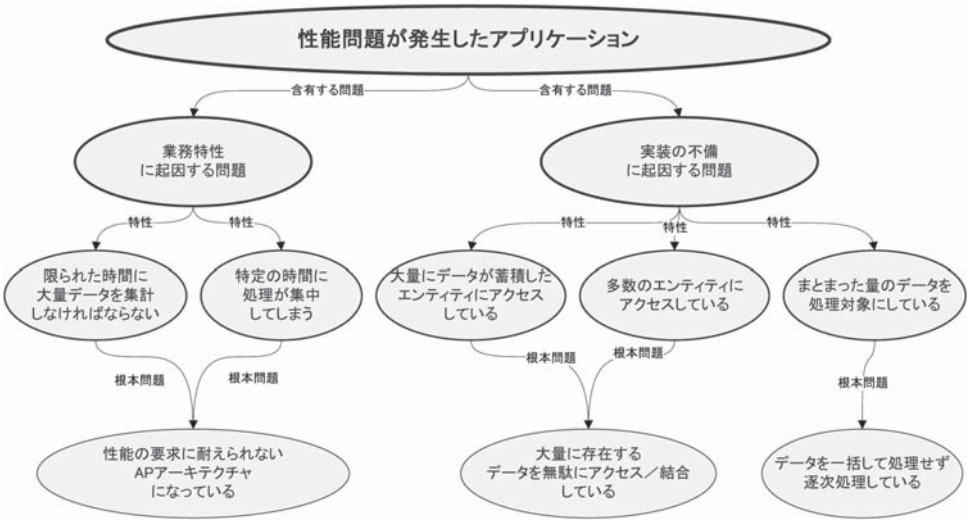


図1 性能問題につながるAPの特徴と根本問題

APの特性に対して，根本問題につながるようなアーキテクチャや設計および実装が存在すると性能問題が表面化する．

2.2 性能問題がプロジェクト終盤や本番稼働後に発生する要因分析

性能問題がプロジェクト終盤や本番稼働後に発生する場合，必ず以下の二つの基本要因が存在する．

・APの開発過程で性能問題を作り込んでしまう
・テスト過程で性能問題の検知が遅れてしまう

本節では，この二つの基本要因についてより深く要因分析する．

2.2.1　AP の開発過程で性能問題を作り込んでしまう要因の分析

AP の開発過程で性能問題を作り込んでしまう要因を分析すると，図 2 のようになる．

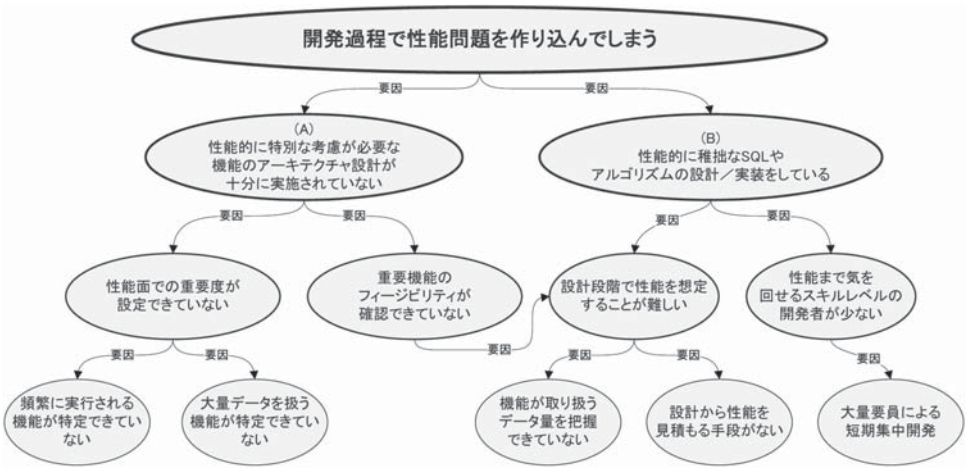


図 2　AP の開発過程で性能問題を作り込んでしまう要因の分析ツリー

AP の機能の中には，特定の時間に集中してアクセスされたり，大量のデータを限られた時間で集計するようなものがある．これらの機能を，他の機能と同様に扱って開発すると，アクセスの集中や大量データ処理に耐えられない AP となる可能性が高い．一方，アクセスの集中や大量データを取り扱わない機能は，本来は性能に関して特別な考慮をして設計/実装する必要がない．しかし，実装者のスキルレベルが低いと非効率な設計/実装をして性能問題を作り込むケースがある．特に大規模開発や短期開発の場合，大量の開発者を動員するためスキルのばらつきが発生しやすい．

また，ある機能で取り扱うデータ量を正確に把握できない，設計書の静的情報から処理時間を判断できない，等の理由により，開発者は設計時点で性能品質を満たしているか否か判断できない．その結果，非効率な設計/実装に気づかず性能問題を作り込んでしまう．

2.2.2　テスト過程で性能問題の検知が遅れてしまう要因の分析

テスト過程で性能問題の検知が遅れてしまう要因を分析すると，図 3 のようになる．

性能品質の基準が曖昧だと，テスト実施中に応答時間や処理時間が遅い機能があったとしても問題視されず，対応の機会を逸する場合がある．

性能テストは機能的に動作するものが前提となるので，機能テストと性能テストを並列で実施することは難しい．また，開発期間が短いと直列で実施することも難しい．そのため，性能テストを実施するのに十分な期間が確保できなかったり，性能テストが次工程へ先送りされたりすることで，問題検知の機会も遅れてしまう．

単体テスト工程や結合テスト工程では実際に AP を実行する．しかし，一般的にテスト環境は本番環境に比べて性能的に劣る環境である．テストデータの量やバリエーションも本番データと比べてかなり少ない．また，呼び出す API（Application Programming Interface）がス

タブ[*3]であるなど，実行するAPも全てが本番稼働用のモジュールではない．そのため，当該工程の実測値では性能品質を正しく評価できず問題を検知できないケースがある．

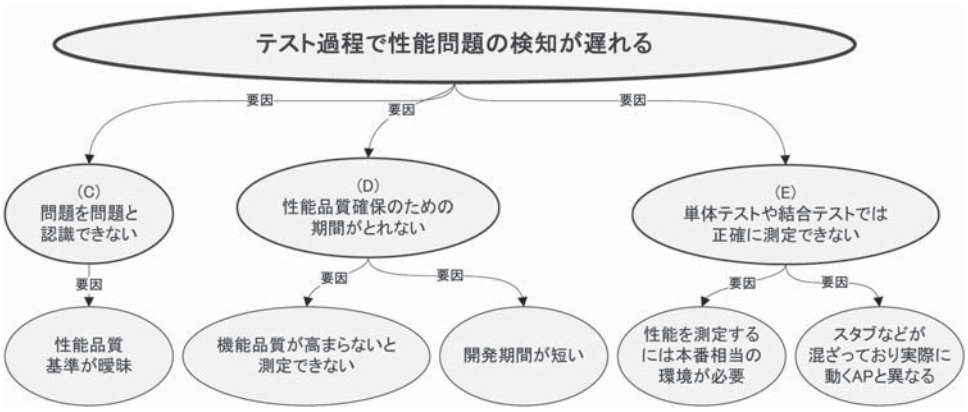


図3　テスト過程で性能問題の検知が遅れる要因の分析ツリー

2.3　性能品質確保のための重要成功要因

性能品質を確保するためには，根本問題となる要素をできるだけ作り込まない，作り込んだとしてもできるだけ早期に発見できる戦略が必要になる．戦略を立案する上での重要成功要因と対応する問題要因を図4に示す．

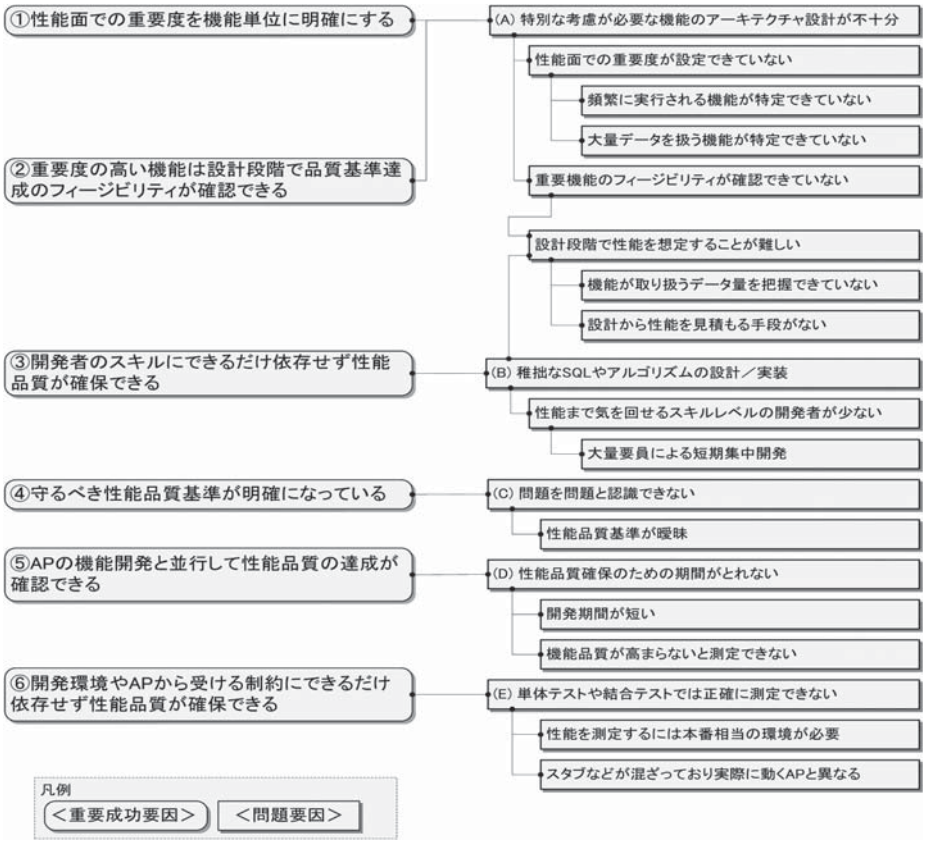


図4　性能品質確保のための重要成功要因と対応する問題要因

3. 性能品質確保戦略

本プロジェクトでは，前章で記述した重要成功要因を踏まえた性能品質確保戦略を立案・推進した．それにより性能問題を早期に解消し，予定期間内で性能品質を確保することができた．本章では，本プロジェクトで採用した戦略にもとづき，その改善点も加えて今後のプロジェクトが取るべき性能品質確保戦略（以降，本戦略）を以下の四つの STEP に整理した．

【STEP1】ゴール定義と重要機能の抽出
【STEP2】重要機能の性能品質作り込み
【STEP3】性能問題の早期検出と是正
【STEP4】性能テスト戦略

本戦略を構成する上記 4 STEP は，図 5 に示すプロセスで推進する．各 STEP の目的やポイントは本章の各節で解説する．

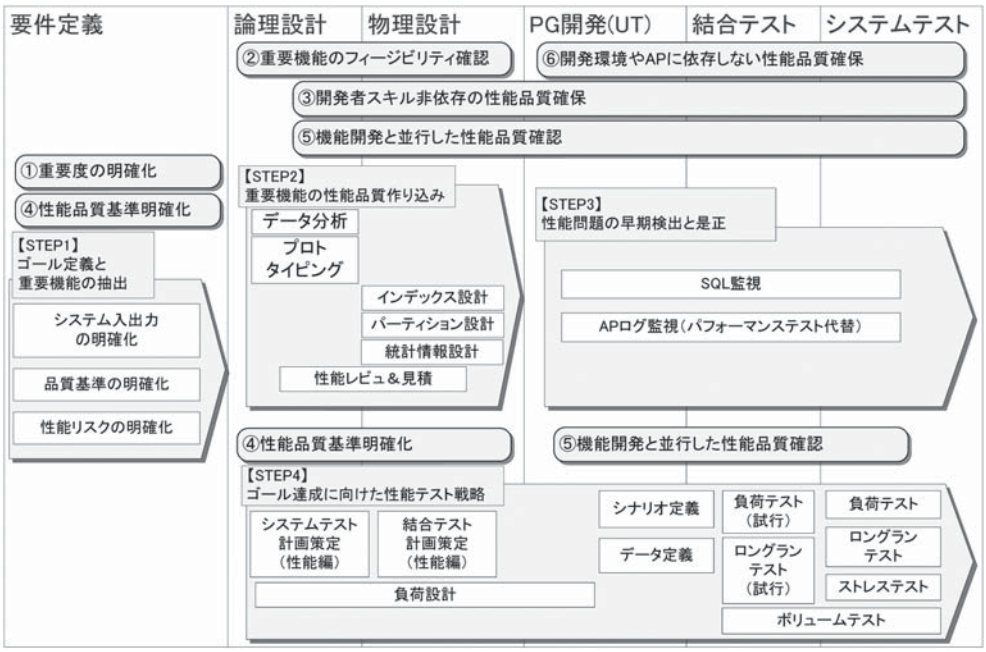


図 5　性能品質確保戦略のプロセス概要フロー

3.1　ゴール定義と重要機能の抽出

【STEP1】では，以下の重要成功要因を踏まえた施策を実行する．

①重要度の明確化
④性能品質基準明確化

3.1.1　システム入出力の明確化

達成すべきゴールはシステムの入出力経路ごとに定義する．入出力経路としては，例えば次のような経路が考えられる．

・画面の入出力（レスポンスタイム）
・他システム連携からの入力（オンライン連携，バッチ連携）
・他システム連携への出力（オンライン連携，バッチ連携）

入出力経路が定義できたら，それぞれの経路に対してどれだけのトランザクションが発生するかを分析，定義する．

3.1.2　品質基準の明確化

ゴールを明確にするということは，性能品質が基準に達したことを【STEP4】のテストにて測定できるようにするということである．たとえば，「画面の応答時間は3秒以内」という要求仕様だけではどのような条件で測定して3秒を達成すれば良いかわからない．以下の点を明らかにすることで評価基準を明確にすることができる．

・入力経路：どの入力経路に適用されるのか？
・対象機能：どの機能に適用されるのか？
・データ量：処理対象データが最大量のときか，平均的なときか？
・対象時間：ピーク時間帯か，稼働時間を通してか？
・算出方法：応答時間の算出方法は？（平均値/パーセンタイル値/最大値など）
・測定範囲：システムが保証すべき範囲は？（レスポンスタイム/サーバ内滞留時間など）

3.1.3　性能リスクの明確化

ゴールとなる品質基準が明確になったら，その基準の達成に対してリスクがある機能を特定する．リスクの高低に応じて，その後の品質確保のアプローチに濃淡をつけることで，効果の高い施策を効率的に実施できる．

機能ごとに次のような観点でリスクを判定し，性能重要度を段階的に定義する（高い順にSABC）．

・現行システムなどで過去に発生した性能課題からの類推
・特定の時間帯に機能の実行が集中する度合い
・機能がアクセスするエンティティの特性（データ量とエンティティ数）

性能重要度は図6のような階層となるよう定義する．重要度の比率がその後に実施するプロセスの工数に直結するため，図6で示す割合を目安にする．

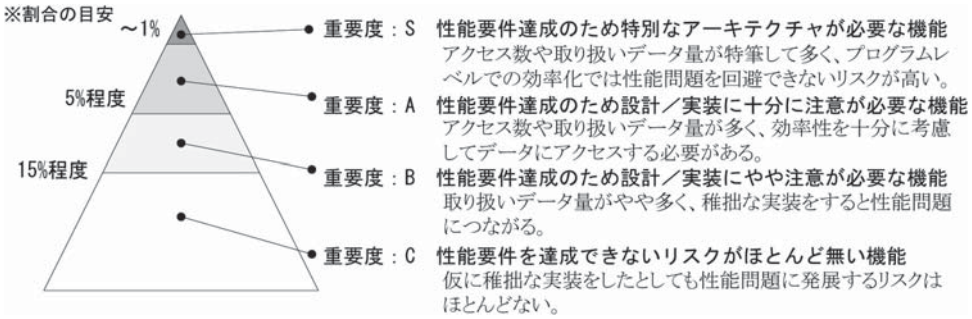


図6　性能重要度の階層イメージ

性能重要度の判断は，表1および表2のような性能重要度判断基準を使用するとよい．なお，過去に発生した性能課題からの類推などで，性能重要度が高いことが自明な機能は基準によらず個別に定義する．

表1　オンライン機能系のピーク負荷による性能重要度判断基準の例

性能重要度	アクセス数の割合
S	ピーク時アクセス数の上位30%に含まれる機能
A	ピーク時アクセス数の上位31%～70%に含まれる機能
B	ピーク時アクセス数の上位71%～90%に含まれる機能 アクセス数の割合が全体の1%以上の機能
C	その他の機能

表2　オンライン機能系のエンティティ特性による性能重要度判断基準の例

アクセステーブル数 / レコード件数/テーブル	0～2	3～4	5以上
1万件以下	C	C	B
1万件以上	C	B	B
100万件以上	B	B	A
1億件以上	B	A	S

性能重要度判断基準に採用する閾値は，求められる性能要件やハードウェア環境などにも依存する．そのため，必要に応じて類似システムの調査やSQL性能のベンチマークを採取し，基準値を決定する．

3.2　重要機能の性能品質作り込み

【STEP2】では，以下の重要成功要因を踏まえ，表3の施策を実行する．

②重要機能のフィージビリティ（実現可能性）確認

③開発者スキル非依存の性能品質確保

⑤機能開発と並行した性能品質確認

表3の施策は，AP開発者とは異なる性能担当者が実施する．対象機能は【STEP1】で定義した性能重要度にあわせて選択する．

表3　品質作り込み施策と性能重要度の関連

品質作り込み施策（◎：必須施策，○推奨施策）	性能重要度				備　　考
	S	A	B	C	
プロトタイピングに基づく設計	◎	—	—	—	特別なアーキテクチャを要するもののみ実施する
論理設計の性能見積/レビュ	◎	○	—	—	論理設計段階ではB以下に問題は混入しにくい
物理設計の性能見積/レビュ	◎	◎	○	—	C以下に問題は混入しにくい
インデックス設計	◎	◎	◎	—	C以下はインデックスの効果は低い
パーティション設計	◎				重要度によらずテーブルのデータ量で対象を決定
統計情報設計	◎				一定のルールに基づき全テーブルを対象とする

また，これらの施策の重要な入力情報となる「データ分析」を事前に実施する．本節ではこれらの施策について解説する．

3.2.1　データ分析

データ量見積では，エンティティ単位でのレコード数見積の実施にとどまることが多い．しかし，3.2.3項で述べる性能見積やレビュおよび3.2.4項のインデックス設計では，レコード

の総数だけでは入力情報として不十分である．データの項目値のバリエーションは必ずしも平均的に分布していないため，偏りを考慮して表4のサンプルに示すようにデータ分布情報を定義する必要がある．

表4　データ基礎件数一覧のサンプル

項番	算出対象	最小	平均	90パーセンタイル	最大	情報源
1	1販売あたりの販売明細	1	1	3	150	現行システムデータ
2	1販売所あたり契約している顧客数	7	25	120	8,200	マスタデータ
3	1顧客あたりの契約している部署数	1	3	15	60	マスタデータ
・ ・ ・	・ ・ ・	・ ・ ・	・ ・ ・	・ ・ ・	・ ・ ・	・ ・ ・

表4では，主要なエンティティ同士の関連（多重度）に着目し，平均的な件数だけでなく90パーセンタイルや最小/最大の件数も定義している．これらの値に基づいて，性能見積時の想定ループ処理回数やインデックスで絞り込んだ際の想定最大件数などが判断できる．

3.2.2　プロトタイピングに基づく設計

重要度Sに指定した機能は，重要成功要因である「②重要機能のフィージビリティ確認」を行う．表5に示す分散方式により時間リソースやマシンリソースを有効活用するアーキテクチャを検討する．性能要件達成のフィージビリティを確保するため，プロトタイプモデルで性能を検証し，フィージビリティを得たアーキテクチャにしたがって設計を進める．

表5　分散方式事例の一覧

分散方式		方式説明
時間 分散方式	事前集計	スケジュールバッチなどで特定の時間に処理するデータ量が多い場合，その時間帯で実施する処理をできるだけ減らすため，その処理の実施前に中間集計などを実施する方式．
リソース 分散方式	プロセス （スレッド） 分散	大量データを処理するバッチなどで，対象データを複数のプロセス（もしくはスレッド）で処理する方式． DBサーバなどのCPU/メモリリソースを考慮し分散数を決定する． なお，Oracle RACを採用している場合，Oracleデータベースノード間のブロック競合によるクラスタ待機が原因で，プロセス分散効果が期待通りに得られない場合がある．その際は次のパーティションプルーニングもあわせて検討する．
	パーティション プルーニング	データベースの物理パーティションのデータ範囲にあわせて，プロセスの処理対象データを決める方式． これにより，プロセス間で物理的に異なるデータ領域を処理対象とするため，I/O効率が高まりプロセス分散がより効果的になる．
	APパーティショニング	Oracle RACを採用している場合に有効な方式で，データベースノードと処理対象データを固定化する方式． これにより，データベースノードをまたがって同一のデータを処理することがなくなり，キャッシュフュージョン抑制による処理効率改善およびバッファキャッシュの有効活用につながる． バッチ処理ではプロセス分散，パーティションプルーニング，APパーティショニングを組み合わせることで，プロセス分散の効果を最大化できる．

3. 2. 3　論理設計および物理設計の性能見積とレビュ

設計工程の性能に関する対策を，「べからず集」のチェックリストによるチェックで実施しているプロジェクトもあるだろう．しかし，やってはいけないことをやらなかったとして，対象機能の性能面でのフィージビリティが得られるわけではない．そのため，論理設計および物理設計の段階ではチェックリストに従ったレビュに加えて処理時間を机上で見積り，品質基準に耐えうる設計となっているかを評価する．

処理時間は，主に SQL の処理時間に着目し以下の二つを掛け合わせて見積る．

・処理構造や処理対象レコード数などから算出した SQL 発行回数

・アクセスレコード数に応じて設定した SQL 実行時間の基礎値

見積った処理時間から，性能要件が達成できないことが懸念される機能を洗い出し，処理構造やデータアクセス方法の改善を開発者に提言する．

3. 2. 4　インデックス設計

AP の性能を大きく左右するのがデータベースのインデックスである．そのため，インデックス設計は性能品質を確保する上で非常に重要である．インデックスの設計は多くの場合，AP 開発者が必要に応じて定義し，DBA（Database Administrator）がそれを承認するというプロセスであろう．しかし，DBA が AP の全てのアクセスを把握するのは難しいため，インデックス要否は厳密には判断できない．そのため，性能品質確保のプロセスとしては信頼性が低いものとなる．

インデックスを設計するためには，次の 3 点を明確にすることが重要である．

1）性能重要度が高い（B 以上の）機能

2）データ量が多い，もしくはアクセス頻度が高いテーブル

3）1)の機能が 2)のテーブルのどの項目を条件にしてアクセスしているか

これらの情報は，性能重要度の定義結果と AP の設計書から，性能担当者でも取得できる情報である．この情報を収集し，データベース化（リポジトリ化）することでテーブルに対するアクセスパターンを可視化できる．テーブルとアクセスパターンが可視化できれば，効果の高いインデックスを機械的に判断できるため，効率よくインデックスが定義できる．

3. 2. 5　パーティション設計

レコード件数が多いテーブルは，AP からのアクセス効率を高めるため適切にパーティション分割する．その際にどのキー項目で分割するかも重要である．これらの設計を AP 開発者の感覚によらず，以下の選定基準に基づいて性能担当者が対象テーブルやキー項目，分割キー範囲を設計できるようにする．

・パーティションを分割すべきレコード件数の閾値

・アクセス時の主要キー項目が同じトランザクションテーブルのグループ

・分割キー項目のデータ分布

3. 2. 6　統計情報設計

Oracle データベースは，オプティマイザ[*4] がコストベース[*5] で SQL をオプティマイズし，最適と考える SQL の実行計画を作成する．オプティマイズ時には，主に「インデックス定義」

と「パーティション定義」に加えて「オプティマイザ統計情報[*6]」を参照する．この統計情報の運用方式として，主に以下の二つがある．

方式①　実際に格納されているデータを動的に分析・更新する方式

方式②　あらかじめ定義した情報で固定化する方式

方式①は，Oracle データベースが常に最適と考える SQL の実行計画を作成する利点があるが，その実行計画がまれに性能劣化につながってしまうケースがある．

データ量の変化に応じて実行計画を変化させたくない場合は，方式②の運用が望ましい．方式②を採用することで，3.3.1 項で述べる SQL 実行計画監視にて問題の検出ができる利点もある．統計情報は，正確に定義しようとすると旧システムの本番データなどを詳細に分析してカーディナリティ[*7]を決定する必要がある．しかし，実行計画を固定化する目的に限るならば，統計情報の設計は正確性にとらわれず定義方法をデータ型や項目長などに基づいてルール化できる．全工程を通じて，ルールに基づいて画一的に定義できるようにすることで，開発の早い段階から実行計画を固定することが重要である．

3.3　性能問題の早期検出と是正

【STEP3】では，以下の重要成功要因を踏まえた施策を実行する．

③開発者スキル非依存の性能品質確保

⑤機能開発と並行した性能品質確認

⑥開発環境や AP に依存しない性能品質確保

3.3.1　SQL 実行計画監視

AP の性能を決定づける重要な要素が SQL の性能である．効率のよい SQL を早い段階で実装できれば，性能問題のほとんどを未然に防ぐことができる．Oracle データベースでは，SQL を実行しなくても実行計画を分析することでアクセス効率の悪い SQL をある程度特定できる．

実行計画を分析するため，単体テストや結合テストなどで AP が実行した SQL を Oracle データベースから定期的に採取[*8]する．採取した SQL の実行計画を分析し，問題があれば SQL の是正やインデックスの見直しなどの対策をとる．

3.3.2　AP ログ監視

SQL 単体性能以外での AP の性能劣化要因が SQL の大量発行である．マスタデータなどをループ処理内で都度取得しているケースが考えられる．このようなループ処理内での無駄な SQL 発行を早い段階で検出することで，SQL 単体の性能では判別できない性能問題を未然に防ぐことができる．そのため，AP の機能テストで各機能が発行した SQL の回数に着目して，問題機能を検出する．機能テストにおいて，当該機能が最も多く SQL を発行すると想定される最大件数（ex：画面に最大行表示するなど）のテストケースを実施すれば，性能問題検知の感度を高めることができる．

単体テストや結合テストでは，スタブが組み込まれているなどの理由により，本番時とは SQL の発行回数が異なる場合がある．そのため処理時間の見積りでは，実際に SQL を発行した回数に，メソッドの実行回数から推測した SQL 発行回数を加える．

見積った処理時間が性能問題となりうる場合は，設計書やソースコードの内容を確認し，必要に応じてSQLの統合などの処理内容の改善を行う．

3.4　性能テスト戦略

【STEP4】では，以下の重要成功要因を踏まえた施策を実行する．

④性能品質基準明確化

⑤機能開発と並行した性能品質確認

3.4.1　性能テスト計画の立案

結合テスト工程，システムテスト工程でスムーズに性能テストに着手するには，テスト計画を早期に立案し，テスト実施までに必要な作業を洗い出し計画化する必要がある．そのため，Wモデル[*9]に従い論理設計工程でシステムテスト計画書，物理設計工程で結合テスト計画書を作成する．各計画書では，性能要件で定めた品質基準を明示するとともに，それらの達成を客観的に判断できる指標やその評価方法などを明確化する．

また，性能テストでテスト対象とする機能と処理量を明確にする．そのために必要なのが負荷モデルである．負荷モデルでは，性能重要度の高い機能に着目し，それらの機能について時間別の負荷量，全体に占める負荷の割合を定義する．

3.4.2　性能テストの準備

性能テストを実施するために，事前に準備が必要なものは以下のとおりである．

・負荷モデルを再現するためのツール（負荷掛けツール）
・負荷掛けツールに投入するテストシナリオ（スクリプト，投入データ）
・APが動作するためのテストデータ（マスタデータ，トランザクションデータ）
・テスト実行環境
・テスト実行結果の集計/レポートツール

画面から投入される負荷を再現するには市販の負荷掛けツール（オープンソースツールも含む）の使用が有効である．市販の負荷掛けツールの使用にあたっては，相応のスキルが必要になるため，事前の教育や試行を計画的に実施する．他システム連携など，独自のプロトコルのため市販の負荷掛けツールで負荷が再現できない場合は，独自の負荷掛けツールを作成する．

負荷掛けツールに投入するテストシナリオは，機能テストのシナリオから，主要なシナリオを対象として抽出する．性能テスト対象とした機能テストシナリオは，機能テストの実施優先度を高くし，機能テストの序盤で機能テストをパスさせることで，性能テストのテストスクリプトを早い段階で完成させるようにする．

APが動作するためのテストデータは，機能テストで使用しているデータを必要に応じて増幅して，テストシナリオが正常に稼働できるようにする．

テスト実行環境は，テスト対象外の機能が動くことで測定値にノイズが入らないように，機能テストとは完全に独立した環境で実施する必要がある．そのため，個別の環境を準備するか，一定期間環境を専有してブロックするように事前に計画する．また，テスト実行環境と本番環境に性能差がある場合，その性能差に着目して評価基準を調整する．例えば，DBサーバのCPUのコア数が本番環境の半分しかない場合，評価基準となる目標スループット性能の値を

性能要件の半分にするなどである．

テスト実行後は，性能品質基準の達成を客観的に示すため，すばやくレポートする必要がある．そのために実行結果のログから品質基準を客観的に評価できる形式で，レポートを自動生成できるようにツール化しておく．

3. 4. 3　性能テストの実行

結合テスト工程では，工程後半で性能テストが実施できるように，工程前半までに性能テストの準備を終える必要がある．結合テスト工程の性能テストは，システムテストでの性能テストの信頼性を向上させるため，以下を目標に実施する．

・性能テスト自体の品質の確保（シナリオやデータの妥当性，実施手順の確立）

・AP やミドルウェア設定の主要なボトルネックの改善

システムテスト工程では，システムに負荷が掛かることで初めて顕在化する性能問題のチューニングに注力する．それによってシステムテスト工程内での性能品質基準達成のフィージビリティが高まる．

3. 4. 4　性能テストの評価と改善

性能テスト実行結果は性能品質基準と比較し，達成していない項目があれば，その原因がテスト実施上の問題なのか，システム側の問題なのかを切り分ける．テストシナリオやデータおよびテスト環境などのテスト実施上の問題であれば，テスト準備に立ち返り不備を見直す．システム側の問題であれば，必要なチューニングを施す[*10]．性能テストの実施/評価/改善は目標を達成するまで繰り返すことを想定し，各工程内で複数回計画しておく．

4.　本プロジェクトでの実践

本章では，本プロジェクトでの本戦略の実践内容と，得られた結果について述べる．

4. 1　実践内容

本戦略を実践するには，まず顧客と日本ユニシスが共に戦略を理解し，一体となって推進する必要がある．特に，性能要件を明確にする際には現行システム担当者や業務部門との連携を密にするよう，顧客側にも相応の負担をお願いすることになる．本プロジェクトでは，プロジェクト開始段階で顧客と日本ユニシス間で性能品質確保戦略について繰り返し議論し，性能リスクや品質基準について合意形成することができた．これが開発工程全般を通じて円滑に戦略を推進できた大きな要因である．

また，AP 開発チームにも本戦略を浸透させる必要がある．工程の節目や重要な施策の実施前などでは次のような説明会を繰り返し実施し，戦略の浸透を図った．

・性能品質確保戦略の説明

・システムの負荷モデルの周知

・性能レビュプロセスの説明

・SQL 監視および AP ログ監視プロセスの説明

・インデックス設計の考え方の説明

・性能テスト計画の周知

・性能テスト結果の共有

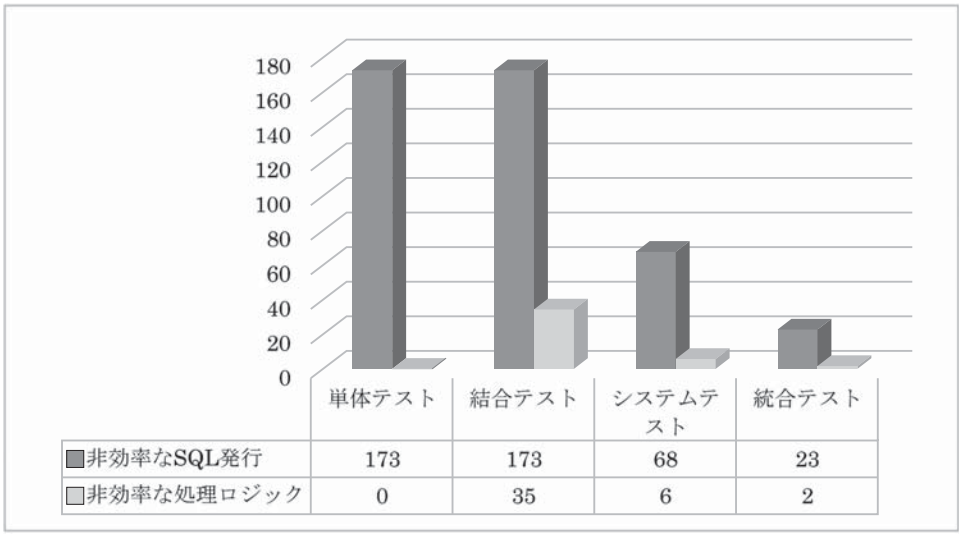
特に，SQL 監視や AP ログ監視は一般的な開発プロセスではなじみのない活動であるため，各担当者の作業との関係について開発プロセスフロー図などを定義して説明に努めた．

4. 2　実践結果

本節では，本戦略の実践結果として，成果分析と本番稼働後に明らかになった戦略の限界について述べる

4. 2. 1　成果

従来，性能問題の発覚はシステムテスト工程に偏りがちであったが，SQL 監視および AP ログ監視によって図 7 に示すとおり単体テスト，結合テスト工程でかなりの問題を検出している．本戦略によって性能問題の早期検出と是正が実現できていることがわかる．



	単体テスト	結合テスト	システムテスト	統合テスト
非効率なSQL発行	173	173	68	23
非効率な処理ロジック	0	35	6	2

図 7　SQL 監視・AP ログ監視による工程別性能問題検出数推移

次に工程別の性能問題の混入件数と戦略で実施した性能問題の検出・除去活動による性能問題除去率の関連を表 6 に示す．

性能問題混入数【A】は，各工程において作り込んでしまった性能問題の数を示す．

潜在性能問題数【B】は，前工程までに解決済みの不具合を除いた検知済の性能問題の数を示す．

性能問題除去数【C】は各性能問題検出・除去活動において，除去できた性能問題の数を示す．

レビュで検出できなかった性能問題の大半が SQL 監視及び AP ログ監視により発見できるため，SQL や AP ログの監視活動は机上でのレビュに比べ非常に効果の高い施策と判断できる．ただし，レビュは実装やテスト前に性能問題を検出できるメリットがあるため，それ自体の効果は十分にあると言える．また，システムテスト工程で実施した性能テストによって検出した性能問題は，性能問題の総件数のうち約 4.9%であった．性能テストを実施する前にできるだけ性能問題を解消するという，本戦略の狙いが実現できていることがわかる．

表 6　障害データ整理結果（不具合・障害マトリクス）

工程	性能問題混入数【A】	性能問題の検出・除去活動	潜在性能問題数【B】 算出式：（対象工程終了までの【A】）－（前工程までの【C】）	性能問題除去数【C】	性能問題除去率（対潜在問題） 算出式：【C】/【B】	性能問題除去率（対総計） 算出式：【C】/総計
詳細設計	294	設計レビュ	294	93	31.6%	11.5%
コーディング	456	コードレビュ	657	160	24.4%	19.7%
		SQL 監視/AP ログ監視		173	26.3%	21.3%
結合テスト	0	SQL 監視/AP ログ監視	324	208	64.2%	25.6%
		性能テスト		17	5.2%	2.1%
システムテスト	28	SQL 監視/AP ログ監視	144	74	51.4%	9.1%
		性能テスト		40	27.8%	4.9%
統合テスト	29	SQL 監視/AP ログ監視	42	25	59.5%	3.1%
		性能テスト		5	11.9%	0.6%
運用テスト	5	SQL 監視/AP ログ監視	22	17	77.3%	2.1%
総計	812			812		

4. 2. 2　限界

本戦略の各種施策は，本番稼働の負荷を想定した負荷モデルが前提となっている．すなわち，施策の効果は負荷モデルの精度に依存する．負荷モデルは，現行システムの繁忙日実績等を参考にすることで高い精度のモデルが策定できる．一方，現行システムの繁忙日実績の分析だけで想定した負荷モデルには，次の様な限界がある．

・現行システムと業務フローが異なる業務があり，負荷の掛かり方が異なる
・現行システムとシステムアーキテクチャが異なり，負荷の掛かり方が異なる（例：バッチ処理がオンライン処理に変更になっているなど）
・特定の日に特定の機能のみがよく使われるなど，繁忙日とは異なる負荷日がある

負荷の掛かり方が変わることが想定される機能は，現行システムの実績等を前提に負荷を机上算出する．特異な負荷日は，業務フローの点検や顧客の業務担当者からのヒアリングによってできるだけ洗い出し，負荷モデルに反映する．本番時の負荷を完全に想定することは難しいが，これらの対策により負荷モデルの精度をより高めることができる．また，負荷の想定値にリスク係数をかけることで，想定した負荷と実際の負荷の差を吸収できる余力を持たせることも検討する．

負荷モデルに対して，リクエストやレスポンスのデータサイズを掛け合わせることで，ネットワークの送受信データサイズの理論値を算出できる．ただし，以下の理由により，理論値をそのままネットワーク帯域の使用率としてネットワークリソースの充足度を判断してはならない．

・実際に使用するネットワーク帯域では，送受信データサイズは回線品質や通信プロトコル制御電文などに依存して理論値より増加する
・他システムが同一のネットワーク回線を共有している場合がある

ネットワークリソースの充足度を検証するためには，ネットワーク回線を共有する全システム稼働時の最大データサイズがネットワーク帯域に収まるかを評価する．最大データサイズは，各システムの時間帯別の送受信データサイズを集計し，それを時間帯別に合算した最大値とする．各システムの時間帯別最大データサイズを全システム分合算しても，全システム稼働時の最大データサイズにならない場合があることに注意が必要である．回線品質や通信プロトコル制御電文などによるデータ増分の考慮は，全システムの最大データサイズ理論値に十分なリスク係数をかけるか，実際の回線効率のデータなどを入手して使用帯域を想定する．結果として，ネットワーク帯域の使用率が高くなることが想定される場合，送受信データを圧縮するアーキテクチャの採用を検討する．

5. 評価

各工程での性能問題検出率から，性能要件実現のためにはシステム開発の全工程を通じて戦略的に性能を管理することが有効であることが証明できたと考える．

本プロジェクトでは，開発規模・期間の関係から性能専任のチームを構成し，戦略の立案・推進を行った．しかし，同様のチーム体制を中小規模の開発プロジェクトで構築するのは難しい．そのため，本プロジェクトでとった戦略を今後の開発プロジェクトでも実施するには次の様な対策を検討する必要がある．

・標準の開発プロセスに組み込む
・開発プロセスに組み込んだ内容を，SE の基本技術として教育する
・全社共通の性能品質評価部門を常設する

これらの対策により，プロジェクト内に専任チームを抱えなくても，必要なときに必要な施策を臨機応変に適用できるようになると想定している．

また，4.2.2 項「限界」に記述した点を踏まえ，負荷モデルの策定プロセスやネットワーク帯域使用量見積を改善することで，本戦略の精度をより高められると想定している．

6. お わ り に

本プロジェクトは，総工数 1 万人月に及ぶシステムの機能要件，性能要件をシステムテスト完了までの 18 ヶ月で達成するという挑戦であったが，性能品質確保戦略を実践することで，機能品質実現と同時に性能品質実現を達成することができた．

今後の開発プロジェクトでは，戦略をより洗練させることで性能品質のさらなる向上を図っていきたい．

＊1　単体テストからシステムテストにかけて，段階的に性能品質を向上させる戦略．テストレベルに応じた品質評価基準・対象・評価環境等を定義している．一方，性能要件や目標値の定め方，システム設計上の考慮等，システム開発上流工程で実施すべき事項は対象としていない．
＊2　日本郵便取引管理システムは，先行と後続の2段階に分けて開発を実施した．先行開発は2013年4月に，後続開発は2016年2月にそれぞれ本番稼働した．
＊3　呼び出し先APIの代用品．呼び出し先APIが開発中や環境依存処理の場合に本来の処理の代用目的で使用する．
＊4　SQLを最適に実行するための実行計画を立案するOracleデータベースの機能．
＊5　SQLの実行コストが最も低い実行計画を採用するオプティマイザのアプローチ方式．
＊6　表内の行数や列内の個別値数などを定義した情報．
＊7　表内の項目値数（値の種類数）のこと．
＊8　Oracleデータベースでは実行されたSQLをV$SQL表等に保持しているため，そこを検索することで実際に実行されたSQLを採取できる．
＊9　設計工程とテスト工程の対比を明確にしたVモデルに加え，テスト工程に対応した設計工程でテスト計画やテスト設計を事前に実施するという方法論．
＊10　参考文献[1]で述べている性能測定PDCAサイクルの概念に基づく．

参考文献　［1］　高井健志,「大規模トランザクション処理の性能対応」, ユニシス技報, 日本ユニシス, Vol.33 No.3 通巻118号，2013年12月，P114 ～ 119

執筆者紹介　川　口　真　一（Shinichi Kawaguchi）
1986年日本ユニシス(株)入社．電力会社の汎用機担当SE，オープン系システム開発の技術主管，大規模システム開発等に従事．2013年～2016年に本稿で紹介した日本郵便取引管理システム（後続稼働）プロジェクトに参画．

下　村　剛　士（Tsuyoshi Shimomura）
1994年日本ユニシス・ソフトウェア(株)入社．公共事業部門にて官公庁，エネルギー，航空系などの業種のシステム開発に従事．2007年日本ユニシス(株)へ転籍，アプリケーション基盤開発部門や品質保証部門を経て，2013年～2016年に本稿で紹介した日本郵便取引管理システム（後続稼働）プロジェクトに参画．