ドメイン駆動設計によるシステム開発
生産ラインのイノベーション

清田康介

システム構築にかかるコスト・期間は20年単位で倍々に増加している。これは「2025年の崖」で示されているレガシーシステムの問題も大きいが、ウォーターフォールモデルで専門家による分業制をとっているシステム開発生産ラインのありようも看過できない。一方、アジャイル開発によるコスト削減や開発期間短縮の効果について、大規模な金融系システムでの実例はまだ少ない。さらに、昨今のマイクロサービスを実現するための設計手法も確立できてはいないと考える。

今回、筆者らチームは「ドメイン駆動設計」を活用し、システム構築コスト・期間を大幅に削減し、かつマイクロサービスに適合するシステム開発の可能性についてのPoCを実施した。

IT構築コスト・期間の増加

「システム構築はどうしてこれほどお金と時間がかかるのだろう」システム開発にかかわる人は誰しも思っていることである。実際、大手損害保険会社が第２次総合オンラインといわれている1980年代に基幹システムを構築した費用はおよそ300億円程度であったのに対し、20年後の2000年頃にはおよそ600～800億円、現在では1200億円は優にかかってしまう。

こうなってしまった理由は、システム構築における専門家間の役割分担、つまりシステム開発生産ライン（システムの作り方）の問題が大きいと考える。現在のシステム開発生産ラインを簡単に説明すると、次のようなステップとなる。

①要求定義や画面・帳票のUI、ビジネスエンジンなど外部設計を担当する業務アプリケーションの専門家、基盤・ネットワークや運用の方式設計するエンジニアなどがウォーターフォールモデルの上流工程を担当する。

②次に、データベース構成・項目やトランザクションなどを具体的に設計するアプリケーションエンジニア、基盤・ネットワーク・運用の諸条件を設計するエンジニアなどが、それぞれの中流工程を担う。

③プログラム仕様書からコーディング、単体テストの工程、いわゆる下流工程は、費用を抑えるためにオフショアなどに開発を委託する。

アプリケーション・基盤・ネットワーク・運用などの専門領域をつなぐため、そして、上・中・下流の開発工程をつなぐため、大量の設計書が必要となる。当然、設計書は作成する工数（コスト）だけではなく、読んで理解する工数（コスト）も必要となる。さらに、本番稼働後に障害を発生させないように品質を積み上げるため、領域の専門家間、開発工程間のそれぞれにおいて、多くの時間をかけてレビューを実施する。このような専門家間・工程ごとの分業体制が、大量の設計書作成と伝言ゲームを支えるための多くのレビューを必要とし、そのことがシステム構築費用や開発期間の大幅な増加につながっている。

一方、DXやマイクロサービスを実現させるためには、アジャイル開発による要求事項への柔軟な対応が必要といわれている。また、

アジャイル開発は、コスト低減や短期間開発の可能性があることからもその期待は大きい。しかし、アジャイル開発はどのように設計するべきかの方法論よりも、チーム組成の考え方や反復型開発の運営方法を中心に説明しているような印象を筆者は受けてしまう。実際、2001年に登場して現在に至るまで、SAFeやNexusFrameworkなどの開発フレームワークはあるものの、国内の金融系大規模システムでの開発実績は決して多くはない。つまり、アジャイル開発だけでは前述のような効果を得ることが難しいと考える。

筆者は、コスト削減や開発期間の短縮を実現するためには、開発プロセスだけではなくソフトウエアの設計手法も同時に検討が必要と考えている。今回、筆者らチームは、開発コスト・期間の削減に効果的なソフトウエアの設計手法として、「ドメイン駆動設計」に注目した。そして、ドメイン駆動設計をベースにした開発手法が金融系大規模システムに適用できるかどうか、またその効果についてのPoC（実証実験）を行った。

ドメイン駆動設計

ドメイン駆動設計（以下、DDD）とは、2003年にEric Evans氏の著書の中で提唱されたソフトウエアの設計手法のことである。ソフトウエアによって解決したい問題領域をドメインと呼び、ドメインの知識をソフトウエア構造に反映させることを目的としている。

DDDには、次のような特徴がある。

(1) ドメインモデル

ビジネスロジック（業務で利用される判断条件や計算式）の集合体であるドメインは、複雑な構造をしていることが多い。そのような複雑なドメインをかみ砕いて抽象化し、理解しやすいように体系化したものを「ドメインモデル」と呼ぶ。DDDでは、複雑なドメインを一つの大きなドメインモデルで表現するのではなく、特定のまとまった業務単位でさらに分割して表現する。この分割された業務領域のことを「境界づけられたコンテキスト」と呼ぶ。「境界づけられたコンテキスト」が複数存在する場合、コンテキスト間の依存関係や全体的な整合性をとるために、「コンテキストマップ」と呼ばれるビジネスロジックをコンテキストごとにマッピングしたドキュメントを作成する。

また、実際に業務を行うユーザーや業務知識に秀でたシステムエンジニアなど、ドメインの知識に精通した実業務の有識者を「ドメインエキスパート」と呼び、システムの開発者はドメインエキスパートの話す言葉をベースにシステムを設計していく。

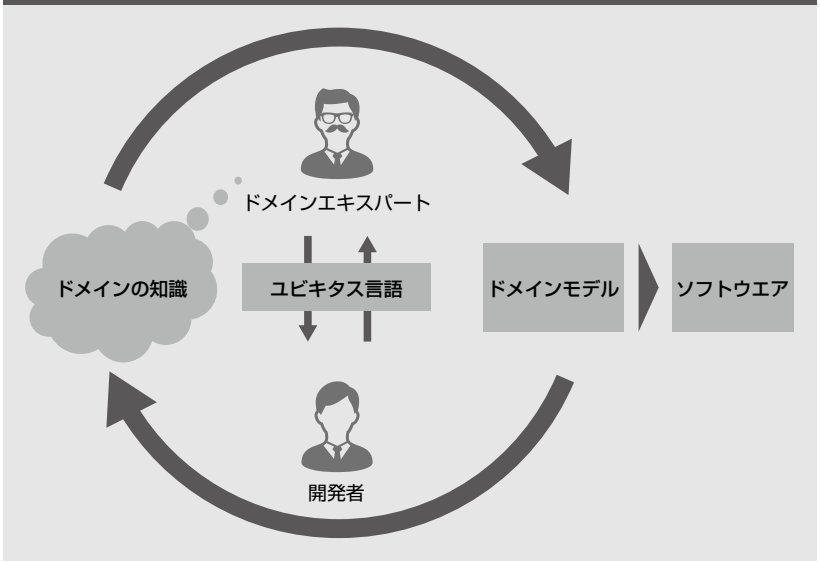
(2) ユビキタス言語

「境界づけられたコンテキスト」内では、業務を表現する用語を統一する必要があり、その用語を「ユビキタス言語」という。ユビキタス言語は関係者間のコミュニケーションに利用するだけではなく、開発者が作成するソースコードでも利用される必要がある。ユビキタス言語をソースコード上の変数名やメソッド名でも利用することで、業務知識をソースコードに反映することが可能となる。

(3) ドメインモデルのソフトウエア化

DDDでは、ドメインモデルをソフトウエアで具体的に表現する技術的な手法（エンティティ、値オブジェクト、集約など）についても定義されている。これらの手法は、ドメインモデルを実際に動くソフトウエアとして構成し、ソ

図1　ドメイン駆動設計（DDD）のイメージ



フトウエア内部のモジュール構造を疎結合化するための実践的なプラクティスとなっている。

(4) 継続的なモデルの改善

DDDでは、一度設計して終わりではなく、モデルがドメインをよりうまく記述できるように、ドメインを変化させつつ継続的にモデルを改善することが求められる（これを「DDDリファクタリング」と呼ぶ）。

このような継続的なモデルの改善活動を、ドメインエキスパートと開発者が対話をしながら行うことで、ドメインの知識を反映したより良いドメインモデルへと成長させることができる（図1）。

DDDを導入すると、一般的に次のようなメリットが得られる。

①ビジネスロジックの疎結合・高凝集化による効果

DDDを取り入れることで、一つのビジネスロジックを実際のビジネスに合わせて適切に整理・分割した最適な形で一箇所に集約する（単一定義する）ことができる。その結果、疎結合・高凝集なシステム構造を実現でき、ビジネス要求の変更時にシステムの修正箇所を極小化することが可能となる。結果、影響調査範囲やノンデグレードテストの範囲も削減できるため、システム開発のコストや期間が削減できる。

②共通認識できる設計書の削減

DDDでは、ユビキタス言語を用いて、業務知識やビジネスロジックの仕様をソースコードで表現し、ドメインエキスパートともソースコードをベースにして仕様の合意や調整を行う。これによって、従来必要であった多くの設計書を作成するコストを削減することが可能となる。

一方で、DDDを実際のエンタープライズ系、特に大規模な金融システムで導入した実例はあまり知られておらず、開発プロセス標準化の確立や、定量的な導入効果の検証がなされていない。

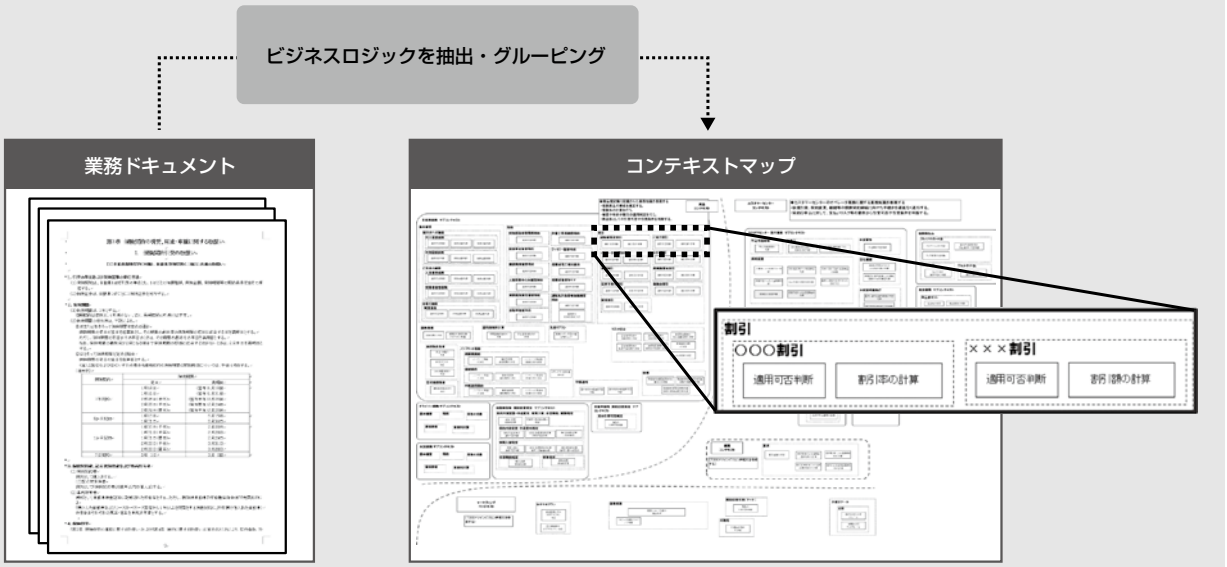
野村総合研究所（NRI）保険ソリューション事業本部では清田康介上級、江本龍二主任、春田晋作専門職を中心にDDDが大規模金融システムに適合できるのかについてのPoCを行った。

大規模システムへのドメイン駆動設計の適合性検証

(1) PoCの目的と方法

筆者らは、DDDを適用して構築したシステムの初期開発時の規模、およびビジネス変更要求時の修正規模によって、コスト削減や開発期間短縮の効果があるか否か

図2　コンテキストマップ



をPoCの目的とした。

PoCの方法は、現在本番で稼働する通販型損害保険の契約管理システムを対象とし（以下、本番システム）、その一部機能をDDDで試行的に構築（以下、試行システム）を行い、さらに実際に発生したビジネス変更要求を試行システムに適用した。そして、本番システムと試行システムで初期開発時の開発規模（ソースコード行数）や変更要求発生時のプログラム修正量、ドキュメント量などを比較した。

(2) PoCのステップ

PoCは、①システム化対象業務のドメインモデルの作成、②対象範囲の試作システムの構築、③ビジネス変更要求に対する修正の3つのステップで推進した。

①システム化対象業務のドメインモデルの作成

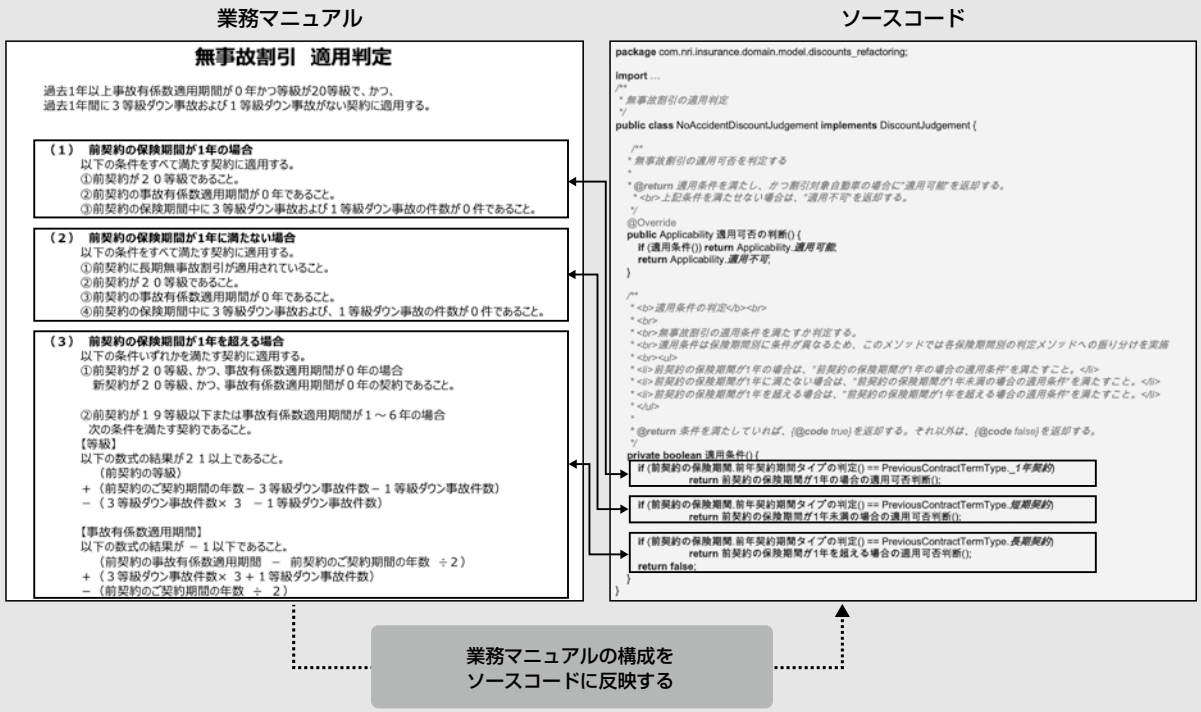
ドメインモデルの成果物の一つである「コンテキストマップ」を作成した（図2）。コンテキストマップを作成する上で2つのポイントがある。

1つ目のポイントは「業務的な目的・関心事の観点でビジネスロジックを分割・凝集できているかに留意してモデリングを行うこと」である。業務要件に対して自然な形でビジネスロジックを分割・凝集することで、よりビジネスの変更要求に対して柔軟なドメインモデルが実現できる。

2つ目のポイントは「ビジネスロジックの詳細仕様書として業務資料そのものを利用すること」である。それによって、要求定義書の多くは不要となる。

今回の試作システムにおいては、業務規定集や業務マニュアルなど、実際の業務で利用されているドキュメントからビジネスロジックを抽出した。また、ソースコードのクラス名、メソッド名、変数名についてもそれらの業務マニュアルに合わせることによって、

図3　業務マニュアルとソースコードの構成



ユーザーでも理解しやすいプログラムソースを追求した。

以上のことに留意しながら、自動車保険の契約管理システムにおける「コンテキストマップ」を作成した。

②対象範囲の試作システムの構築

整理したコンテキストマップをベースにしてパイロット開発を実施した。具体的にはまず、コンテキストマップで洗い出したビジネスロジックの入力・出力関係を紐づけることで、ビジネスロジック間の依存関係を整理した。その上で、業務マニュアルを基にプログラムを実装した。各ビジネスロジックを実装する際、「業務マニュアルの構造・用語とソースコードの構造・用語を対応させる」ことを意識した。ビジネスロジックを一通りコーディング作成したのち、APIやデータベースを用いたトランザクション処理を実装し、アプリケーションとして実行可能なプログラムを作成した。このアプリケーションは「境界づけられたコンテキスト」であるため、実行環境を分割すれば、マイクロサービス化が実現できることになる。実装例として図３を参照されたい。

③ビジネス変更要求に対する修正

本番システムと試作システムのビジネス変更要求に対する比較検証においては、自動車保険の保険料割引（無事故割引）の追加対応を対象テーマに選定した。自動車保険の商品改定は定期的に発生し、プログラムの修正やテストに掛かる開発期間・工数は決して少なくない。このテーマにおいて、

表1　本番システムとプロトタイプの開発規模

比較項目＼現・試システム		現行システム	試作システム
試作	開発規模	（試作と同範囲の規模）182.3kStep	97.0kStep
	設計書ボリューム	153本	1枚（規定集、マニュアルをインプット）
変更要求	新規・修正モジュール数	新規：3本、修正：65本	新規：1本、修正：2本
	修正ステップ	5.2kStep	0.68kStep
	設計書ボリューム	およそ26枚	0枚（変更された業務マニュアルをインプット）

プログラムの修正規模を抑え、テストを効率化できれば、DDDの有効性が判断できると考えたためである。

プログラム修正箇所の特定方法は、業務マニュアルの変更箇所からソースコードの影響箇所を判断した。業務マニュアルの構成と試作システムの構成が一致しているため、容易に修正を実施できた。

(3) 評価

この検証結果は表1の通りである。

本番システムと試行システムの初期開発規模を比較すると、ソースコードのStep数は本番システムの約53%へ減少できている。また、変更要求における修正ソースコード規模は約13%へ減少できた。さらに、DDDではソース自体が詳細仕様書となるため、試作システム開発時および変更要求時の設計書についても大幅に削減できている。

一方、試行錯誤しながらのPoCであったため、試作システム開発と変更要求の対応に実質2人でおよそ4カ月の期間を要した。紆余曲折がなくなるだけで、工数はさらに圧縮できることは確実であり、また、DDD設計の道筋を整理できたので、今後はDDDの基本的な訓練を行うだけでも開発期間・コストとも十分に短縮されていくものと考える。

ドメイン駆動設計の評価と今後の展望と課題

PoCの結果から、DDDはシステム初期開発規模の縮小に効果的であり、開発期間・コストの削減が期待できることが分かった。かつ、ビジネス変更要求時のシステム改修に対しても大きな効果があることが確認できた。DDDの導入における評価を整理し、今後の展望と課題について次のように考える。

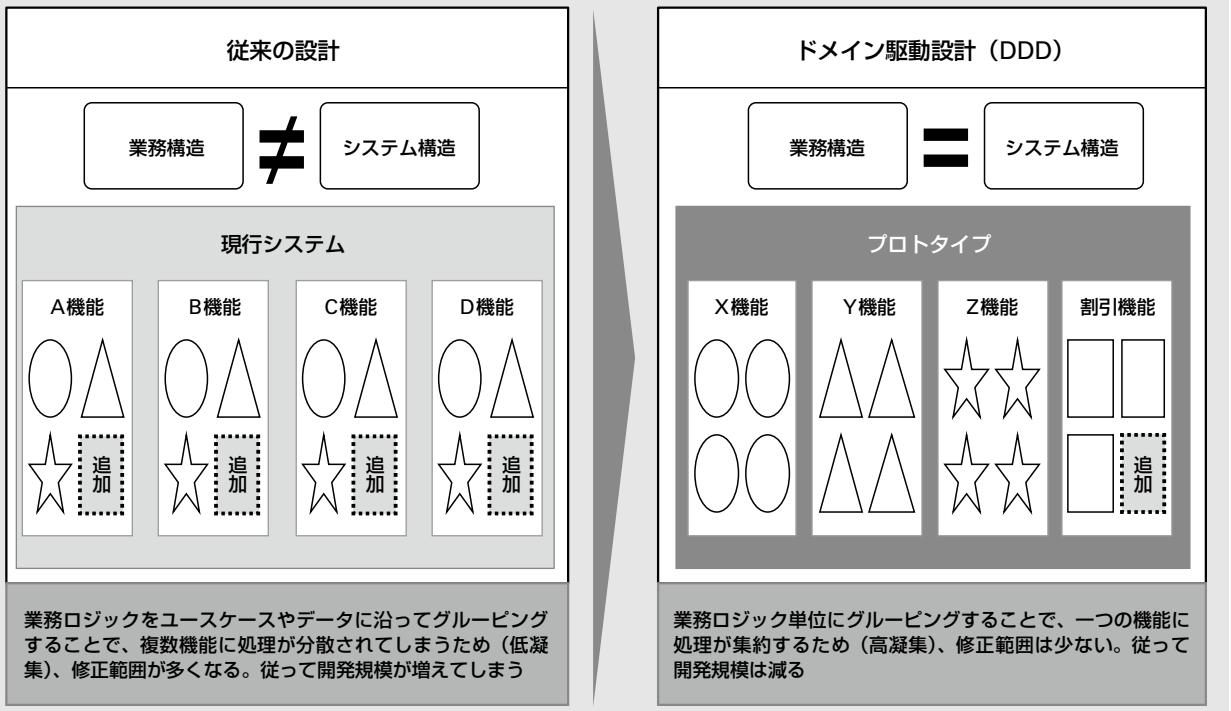
(1) ドメイン駆動設計の評価

①システム初期開発、ビジネス変更要求における効果

DDDを取り入れることで、システム初期開発規模の圧縮とビジネス変更要求時の修正箇所の極小化が期待できる。従来の設計では、業務ユースケースを起点として設計を行うため、ビジネスロジック（業務で利用される条件判断や計算式）がユースケース単位に分散する傾向にあった。

一方、DDDでは、ユースケースの設計の前にビジネスロジック

図4　ビジネスロジック凝縮化の比較



の集約設計を実施するため、ビジネスロジックの高凝集化が実現できる（図4）。従来分散していたビジネスロジックを単一的に集約した結果、システム規模が圧縮され、またビジネスロジック変更時の変更箇所の極小化が可能となった。また、DDDの導入により、仕様書作成のコストも軽減できることが分かった。前述したように、DDDでは「ユビキタス言語」という思想でソースコード自体でビジネスロジックを表現し、ドメインエキスパートと開発者はソースコードベースを基にシステム仕様を検討する。その結果、従来作成してきたような共通認識するための仕様書の作成コストは不要となる。

②金融系基幹システムへのドメイン駆動設計適用の可能性について

損害保険会社の契約管理システムをPoCの範囲とすることで、想定以上に金融系基幹システムに対してもDDDをスムーズに適用できることが確認できた。さらにいえるのは、金融系業務の場合、ビジネスロジックが業務規定書やマニュアルなどのドキュメントにしっかりと構造的に明文化されていることが多い。そのため、暗黙的なビジネスロジックが少なく、ほかの業種よりも金融系業務の方がDDDを適用しやすい領域かもしれないというのが筆者の受けた感想である。

③ドメイン駆動設計のマイクロサービスへの適合について

今回のPoCから、マイクロサー

ビスを実現するためにもDDDは非常に効果的であるということが判明した。マイクロサービスでは、個々のサービスが独立して稼働するようにサービス間を疎結合に設計する必要がある。そのような設計をするには、DDDの実践が欠かせない。また、既存システムをマイクロサービス化していく場合も、DDDの設計手法を適用すれば、既存システム内のモジュールを疎結合に整理した上でサービスに切り出すことができる。このことから、マイクロサービスとDDDの親和性は高いといえるだろう。

(2) 今後の展望と課題

DDDの導入は、金融系大規模システムにおいても効果的であることが現実的に見えてきた。一方で、ウォーターフォールモデルを前提とする大規模システムへDDDを適用するためには、解決すべき課題が2つある。

1つ目は、「継続的なモデルの改善」の適用である。前述のように、DDDは設計工程において「継続的なモデルの改善」が必要となる。これは工程ごとに完了判定を行いながら次工程に進むウォーターフォールモデルと相反する。工程の戻りを許容しない従来のウォーターフォールモデルではなく、ビジネスロジックの記述部分は設計と開発を繰り返して継続的に改善していくような従来のウォーターフォールモデルの再定義が必要だろう。

2つ目は、DDDの学習コストが大きいことである。大規模なシステムを構築するには多くの開発者が必要となる。当然、その開発担当者全員に対してDDDの思想を浸透させ、DDDの練度を高める育成活動をすることは現実的には難しい。そのため、少数のDDD有識者とその他大勢の有識者以外の開発者という体制で大規模開発を実践できるようなスキームや開発プロセスの検討が必要となる。

たとえば、少数の有識者だけでDDDを適用したモジュール分割設計を行い、開発者は分割済のモジュール単位で開発を行うなどの検討が必要となる。

筆者らチームは、2020年度以降も上記2つの課題を解消し、新たなシステム生産ラインを定義できるかの検証を継続していく予定である。

最後に、当PoCにご支援いただいている有限会社システム設計代表増田亨氏、株式会社エセントリック辻東正和氏および当レポートを校正いただいた保険PRM部児玉寛主席に感謝申し上げる。

清田康介（きよたこうすけ）
野村総合研究所（NRI）
保険システム事業三部損保システム開発第一グループ上級システムエンジニア