ソフトウェア設計のコード化と数学的検証を実現する Goフレームワークの開発

― Design as Codeと新しい仕様駆動開発の実現 ―

1.　背景

日本のソフトウェア産業は大手SIerが受注し下請け企業が実装を担う「ITゼネコン」構造が広く根付いており、これが2つの構造的な問題を生んでいる。1つは人材育成の阻害である。設計と実装が分断されることで設計者は実装の現実を無視した設計を行い、実装者は設計を学ぶ機会を失う。もう1つは品質向上インセンティブの欠如である。SIerのビジネスモデルは「人月単価」に基づいており、開発や保守にかかる工数が増えるほど利益が増大するため、効率化や新技術の導入に消極的になりがちである。

こうした構造的な問題に加え、開発現場レベルでも論理設計は軽視されている。パブリッククラウドの普及により、現在の開発現場では分散システムが広く使われるようになった。こうした環境ではサービス間の相互作用の設計が、システム全体の整合性を保ち意図した挙動を保証する上で不可欠である。しかし実際の開発現場では十分に実施されていない。この原因は次の4点に整理できる。設計に用いられるツールやフォーマットが標準化されておらず知見が蓄積されないこと、設計が実装とは別の場所に管理されるため設計と実装の整合性を保つことが困難であること、設計の検証プロセスが存在しないこと、そして設計できる人材そのものが不足していることである。

設計が十分に行われない結果、実装中やリリース後に仕様の矛盾やバグが発覚し、手戻りや障害が発生するリスクが高まる。このような実装中やリリース後に判明するバグや仕様ミスは、設計段階で検知できるものが多い。

2.　目的

以上の問題を解決するため、本プロジェクトでは「Design as Code」、すなわちソフトウェア設計のコード化を一般的な開発手法として定着させることを目的とする。これにより次の2点を達成する。一つ目はエンジニアが設計を担う文化の醸成である。設計をコードとして実装と一体的に管理することで、エンジニア自身が設計に主体的に関与できる環境を作る。二つ目は設計の標準化による品質向上と知見の蓄積である。設計をコードとして記述し標準化することで設計知見の再利用が可能になる。さらに、記述した設計の正しさを数学的に検証できる環境を構築することで、属人的な勘に依存しない再現性のある設計が実現できる。

3.　開発の内容

本プロジェクトでは、Design as Codeを実務で実現するためのソフトウェアとしてgoat・goat-cli・YAGIの3つを開発した。

3.1 goat

goatはGo言語を用いてシステムの仕様をコードとして記述・検証できるライブラリである。goatを利用することで以下の3点が可能になる。

・コードを書くような設計体験の実現
・記述した設計の正しさの検証
・設計と実装の一貫性の維持

goatは図3.1のようにシステムを複数のステートマシンがキューを介してイベントをやりとりすることで相互作用するものとモデリングする。例えばシンプルなクライアント・サーバーのアプリケーションを考えると、goatではクライアントやサーバーがステートマシンとなり、それぞれが実行中や停止中といった状態をもつ。クライアント・サーバー間のリクエストとレスポンスがイベントとしてモデリングされる。

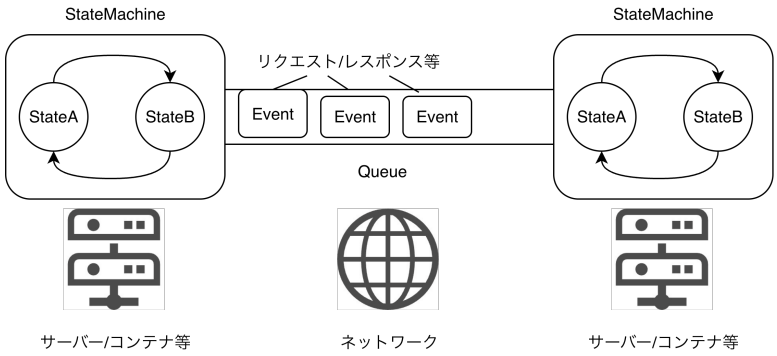

図3.1 goatによるシステムのモデリングのイメージ

goatによる仕様記述は以下のステップで進む。

1. スキーマの定義
2. ルールの定義
3. 振る舞いの定義

スキーマの定義ではシステムの構成要素のステートマシンとその状態、ステートマシン間でやり取りされるイベントの型を定義する。ルールの定義ではシステムの満たすべき性質をルールとして記述する。振る舞いの定義では各ステートマシンがあるイベントを受け取った時にどのように振る舞うかを記述する。これにより、検査可能な仕様が得られる。以下はgoatによる仕様記述の簡易的な例である。

```go
// StateMachineのスキーマを定義
serverSpec := goat.NewStateMachineSpec(&Server{})

// StateMachineに状態を定義
serverSpec.
    DefineStates(serverInit, serverRunning).
    SetInitialState(serverInit)

// Eventのハンドラーを定義
goat.OnEvent(serverSpec, serverRunning,
    func(ctx context.Context, event *eCheckMenuExistenceRequest, server *Server)
{
        goat.SendTo(ctx, event.Sender(), &eCheckMenuExistenceResponse{
            Exists: true,
        })
    },
)

// インスタンスを生成
server, _ := serverSpec.NewInstance()

// ルールを定義
rule := goat.Always(goat.NewCondition("count>0", server, func(s *Server) bool {
    return s.Count > 0
}))

// モデル検査を実行
err := goat.Test(
    goat.WithStateMachines(server),
    goat.WithRules(rule),
)
```

goatで検証できる性質は以下の4種類がある。

- 常に成り立つこと
  - （例）在庫システムにおいて在庫数が常に負にならない
- ある条件が成立したら必ずいつか別の条件が成立すること
  - （例）リクエストを送ったら必ずレスポンスが返る
- 最終的に必ずある条件が成立する
  - （例）分散システムの各ノードが最終的に一致した状態に収束する
- 無限回ある条件が成立する
  - （例）送信待ちキューが何度でも空になる

検査を実行して違反が検出された場合は違反に至るまでの過程が以下のようにトレースとして出力されるため、設計上の問題箇所を特定しやすい。

```
Condition failed. Not Always mut<=1.
Path (length = 5):
  [0]
  StateMachines:
    Name: StateMachine, Detail: {Name:Mut,Type:int,Value:0}, State: {Name:StateType,Type:main.StateType,Value:A}
  QueuedEvents:
    StateMachine: StateMachine, Event: entryEvent, Detail: no fields
  [1]
  StateMachines:
    Name: StateMachine, Detail: {Name:Mut,Type:int,Value:1}, State: {Name:StateType,Type:main.StateType,Value:A}
  QueuedEvents:
    StateMachine: StateMachine, Event: exitEvent, Detail: no fields
    StateMachine: StateMachine, Event: transitionEvent, Detail: {Name:To,Type:goat.AbstractState,Value:&{{0} B}}
    StateMachine: StateMachine, Event: entryEvent, Detail: no fields
  [2]
  StateMachines:
    Name: StateMachine, Detail: {Name:Mut,Type:int,Value:1}, State: {Name:StateType,Type:main.StateType,Value:A}
  QueuedEvents:
    StateMachine: StateMachine, Event: transitionEvent, Detail: {Name:To,Type:goat.AbstractState,Value:&{{0} B}}
    StateMachine: StateMachine, Event: entryEvent, Detail: no fields
  [3]
  StateMachines:
    Name: StateMachine, Detail: {Name:Mut,Type:int,Value:1}, State: {Name:StateType,Type:main.StateType,Value:B}
  QueuedEvents:
    StateMachine: StateMachine, Event: entryEvent, Detail: no fields
  [4] <-- violation here
  StateMachines:
    Name: StateMachine, Detail: {Name:Mut,Type:int,Value:2}, State: {Name:StateType,Type:main.StateType,Value:B}
  QueuedEvents:
    StateMachine: StateMachine, Event: exitEvent, Detail: no fields
    StateMachine: StateMachine, Event: transitionEvent, Detail: {Name:To,Type:goat.AbstractState,Value:&{{0} C}}
    StateMachine: StateMachine, Event: entryEvent, Detail: no fields
```

goatでは記述した仕様からInterface Definition Language（IDL）とE2Eテストコードが生成できる。この機能を活用することで記述した仕様とアプリケーションがインターフェースとE2Eテストで検証した振る舞いレベルでは一致していることが確かめられる。この機能をCI/CDパイプラインに組み込むことで仕様と一致しないインターフェースや振る舞いの変更を自動的に検知できる。これによって設計と実装との乖離を防ぐことができる。

3.2 goat-cli

goat-cliはgoatで記述した仕様を静的解析し、Mermaid記法のシーケンス図を自動生成するCLIツールである。goatによる設計はコードとして記述されるため、システム全体のやりとりの流れを一目で把握することが難しい。goat-cliはこの課題を解決するために開発され、ステートマシン間のイベントの送受信を視覚的に確認できる（図3.2）。

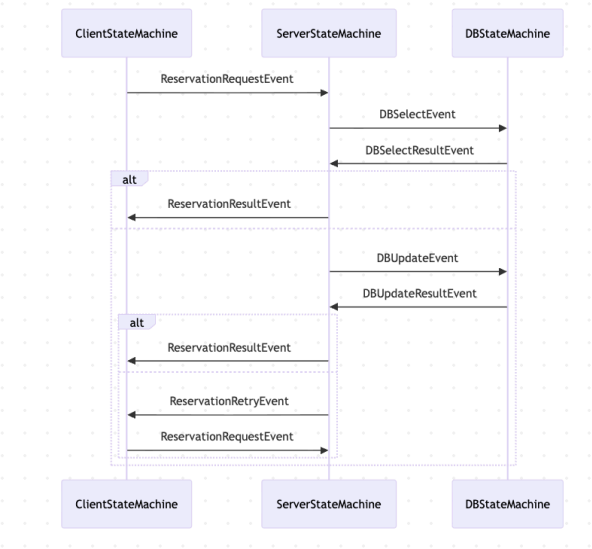

図3.2 生成されるシーケンス図の例

3.3 YAGI

GitHub CopilotやClaude Code等のコーディングエージェントの登場により、AIを活用したvibe codingが一般化しつつある。vibe codingは小規模な開発では有効だが、大規模なコードベースでは適切なコンテキストを与えないまま実装を進めると意図した通りのコードが生成されない問題がある。この問題を背景に仕様駆動開発（SDD）が注目されている。仕様駆動開発とはAIが実装を適切に生成できる詳細度で先に仕様を記述し、それを元に実装を進める開発手法である。このSDDという言葉は流行しているものの、実際にうまく適用できている現場は少ない。これは現状のSDDにおける仕様が自然言語によるドキュメントに過ぎないことが原因であると考えられる。自然言語は曖昧であるため人間同士でもAIとの間でも意味の取り違えが生じうる。また自然言語による仕様は正しさを機械的に検証できず、仕様と実装の一貫性を確認することも困難である。これに対して、自然言語による仕様だけで

なくgoatによる仕様も組み合わせることでより実践的なSDDを実現できないかと考えて開発したのがYAGIである（図3.4）。YAGIは要件定義と設計に特化したCLI型のAIエージェントアプリケーションである。AIとの対話を通じて自然言語による仕様書であるDesign Docsとgoatによる仕様の両方を生成する。YAGIとの対話は5つのフェーズで進行する。Requirements Phaseではユーザーの要件を整理してDesign Docsにまとめる。Schema Phaseではgoatのステートマシンとイベントを定義する。Rule Phaseではシステムが満たすべき性質をgoatのルールとして記述する。Spec Phaseでは定義したルールを満たす振る舞いをgoatで記述し、記述のたびにモデル検査を実行して違反があれば修正するフィードバックループを回す。Documentation Phaseではここまでの内容を自然言語で整理し、Design Docsとして完成させる。最終的にDesign Docsとgoatによる仕様の両方が得られる。goatによる仕様からIDLやE2Eテストコードを生成し、Design Docsとgoatによる仕様を全て合わせてSDDにおける仕様として活用できる。以下はRequirements Phaseの実行例である。Design Docsへ要件をまとめた後、要約をチャットでも表示している。

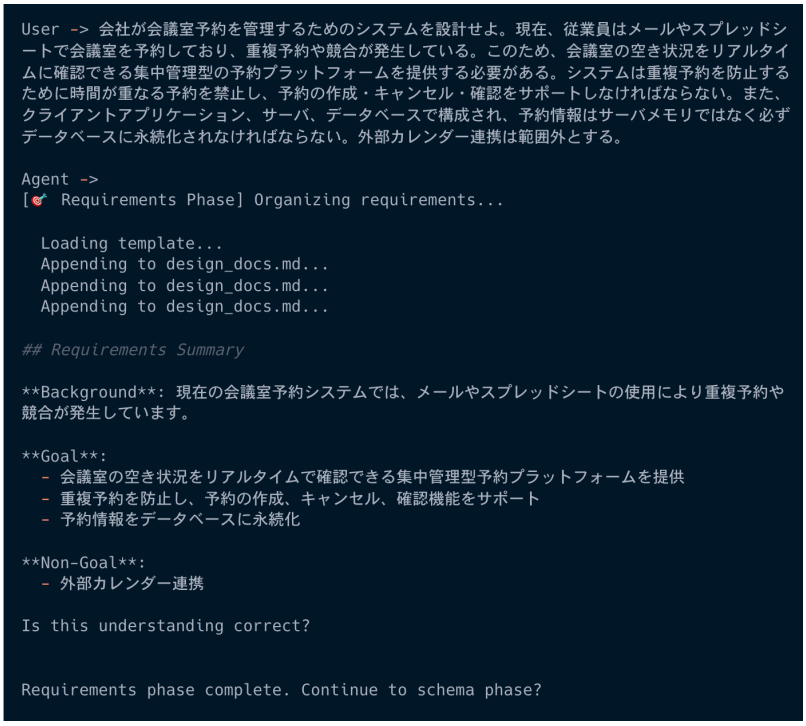

図3.4 YAGIのRequirements Phaseの実行例

4. 従来の技術（または機能）との相違

goatと類似するものとして既存の形式手法のツールが挙げられる。これらは独自のDSLを持ち数学的な専門知識が必要なため学習コストが高く、特殊な実行環境が必要なため既存の開発フローへの組み込みも難しい。goatはGo言語で記述できる上、数学的な専門知識を要求しないため学習コストが低い。Go言語の実行環境があれば動作するため既存の開発フローへの組み込みが容易である。また、goatはIDLやE2Eテストコードの生成により設計と実装を接続できる点も優位である。

YAGIと類似するものとしてKiroやSpeckitのようなSDDを支援するツールが挙げられる。これらで扱われる仕様は全て自然言語によるものに限られている。YAGIはコードによる仕様も生成できるので仕様作成段階でAIとより正確な意思疎通が可能になり、仕様の正しさをモデル検査により機械的に検証できる点で優れている。

5. 期待される効果

本プロジェクトの成果物により、Design as Codeを実務で実現することが可能になる。

Design as Codeの実現により、従来は設計段階では検証できず実装中やリリース後に発覚していた設計ミスを、設計段階で発見できるようになる。その結果、実装中の手戻りやリリース後の障害を減らすことが期待される。さらに、Design as CodeとAIエージェントを組み合わせることで、設計工程からAIを活用できるようになる。これまで生成AIの活用は主に実装工程に限られていたため、これによりソフトウェア開発プロセス全体の効率化が期待できる。またAIはコードとして記述された仕様を参照できるため仕様をより正確に理解でき、SDDによる実装の効率化や品質向上につながる。

さらに、Design as Codeが普及すると設計手法の標準化が進み、設計知識をコードとして蓄積し、共有し、再利用できるようになる。これにより、経験や勘に依存していた設計を再現性のある形で扱いやすくなる。また、エンジニア自身が設計に関与しやすくなるため、設計できる人材の育成にもつながる。その結果、ソフトウェア開発における品質と生産性の向上が期待される。

6. 普及（または活用）の見通し

現時点ではgoatもYAGIも利用しているユーザーがいないため、まずは1人以上のユーザーに使ってもらうことを目指す。そのためにgoatのv1.0.0のリリースとYAGIのデモアプリケーションの公開に取り組む。公開後は開発者コミュニティへの投稿などを通して認知を獲得する。

7. クリエータ名（所属）

戸田 朋花（株式会社サイバーエージェント・株式会社AbemaTV）

（参考）関連URL

goat
https://github.com/goatx/goat

goat-cli
https://github.com/goatx/goat-cli