汎用機・構造化世代向け、オブジェクト指向

目次

はじめに 1
1. 構造化とはなんだったか 1
1.1. 歴史的経緯 2
1.2. 汎用機のメモリ限界とオブジェクト指向 2
2. 構成物の定義と名前 2
3. 設計の遷移 3
3.1. 構造化による設計 4
3.2. オブジェクト指向への展開 6
4. 汎用機・構造化の不都合な点 7
5. どうする？ 7

汎用機・構造化世代向け、オブジェクト指向

はじめに

2023 年の現在 J2EE という言葉を聞く頻度は減りましたが、20 世紀の終わりに出現したこの用語は当時世間を騒がせた『2000 年問題』以上に IT 業界にとってインパクトがありました。汎用機と称されていた構成が技術的な限界を迎え、移行先と目されたクライアント・サーバ方式の開発で四苦八苦しているところに Java/Web 方式が出現して簡便さで一気に流れが向かった記憶があります。

その後 Oracle 社の Java ライセンス方針変更により JDK1.8 の頃からサーバ系のスクリプト言語である Python、JavaScript/TypeScript、Ruby 等に関心が移り、それ用のフレームワークやライブラリがオープンソース・プロジェクトで盛んに開発されるようになって実質一系統の OS とメーカ互換の COBOL や PL/I を使う汎用機の世界は定番が存在しない乱世に移りました。

日本の現状は Java を含む各種開発言語の仕様策定に日本人・企業が参加せず[1]仕様追加・改廃に即応できなくなっています。さらに不運な点はユーザ企業が主体的に IT 技術に取り組むための自前の要員や環境を用意してなかったことで、システムの品質は発注先のスキル頼り、請負側は担当者個人の自助努力頼みになっています。言語仕様の提案や情報交換サイトを利用するための英会話能力の不足による世界からのオイテケボリは日本語による実装が可能な環境[2]を作るか、逆に実装工程の国内実施を諦める（設計工程が終わったら実装は全て海外に発注する）まで解消しないでしょう。

開発言語とその周辺環境は用途等により各種の組合せが並立していますが、オブジェクト指向を標榜する言語は特徴を取込み合って類似の機能が使えるようになり、更に主要なフレームワークが移植されて使用言語に関係なく設計を行うことができるようになりつつあります。

以上を前提に、汎用機・構造化に慣れた人達が今後必要になる考え方を確認します。

1. 構造化とはなんだったか

“構造化”は“構造”（言及先の構成物及び構成物間の配置・関係の状態）に“化”（～のように変える）という接尾辞が付き、日本語として成立しているのか疑問ですが、“構造”は Structured programming の structure（組み立てる）の過去分詞の翻訳で「組み立てたような形態」にしろという意味に解釈し、そのためにまず「構成物に分解」する方法を考えます。

以下 構造化の作業をウォーターフォール的に表すと以下の 3 段階になります。

① 分析：システムの構成要素をデータと機能・処理として抽出する
② 設計：データの項目や相互の関連とバックアップ等の運用、機能・処理の実装内容を決める
③ 実装：データの格納場所やサイズ、機能・処理のコード化

---

[1] Java の仕様を策定する Java Community Process（JCP）には日本企業も参加（会費負担）していますが、仕様策定で主体的に活動している様子はありません https://jcp.org/en/jsr/all
JCP とは... https://jcp.org/aboutJava/communityprocess/JCP_Japan_JUG_visit_July2018.pdf
※Ruby は日本人が作りましたが人気の基のフレームワーク Ruby on Rails は海外発です

[2] 日本語の仕様書から JavaVM(JVM)で動作するバイナリ（class ファイル）を作りだすコンパイラを作れば実現できます。曖昧にならないように定義や式（Algorithm = Logic + Control の Logic 部分）だけを日本語で記述する言語仕様とし、JVM は JCP の成果を活かした最新のものを利用できるようにすれば世界から取り残される心配もありません。
このように JVM 上で動作する Java 以外の言語仕様には Scala、Kotlin、Jython 等があります

汎用機・構造化世代向け、オブジェクト指向

1.1. 歴史的経緯

構造化を 3 段階で表しましたが、日本語では実装にあたる“構造化プログラミング”だけが教育されてきました（“構造化”が題名に付く教本を検索した結果はプログラミング関連が数冊ある程度です)。分析、設計工程の構造化は DFD（データフローダイアグラム）やモジュール構造図、ハイポ図等の図式が普及しましたが図式化する（構造化する）手順を説明した教材は見たことがありません。それでも普及した背景には銀行の第 3 次オンラインや信販会社の伝票処理のリアルタイム化が行われた昭和の終盤辺り、当時は日本企業が潤沢な資金を持っていてスクラップ＆ビルドで新しいシステムを作ろう、そのために設計者の教育から始めようという流れがあり、ユーザによる業務のレクチャと並行してメーカーによる設計技法の指導が行われました。内容はモジュールの結合度を下げて強度を上げるためにどう設計すべきかということで、1970 年代に IBM 社が作った HIPO(Hierarchy input-process-output…ハイポ図)を土台に、一部にオブジェクト指向を加味したような手法もありました。

構造化プログラミングは新人教育で使われ、概略はスパゲティプログラムのアンチとして順処理、分岐、繰返しの制御構造で GOTO を使わずにプログラムを作りましょうという内容です。当初は「そりゃ、なんの条件もなければ順処理、データに種類があれば分岐するだろうし、データに繰返しがあれば繰り返しの処理に自動的になるでしょうよ...」という程度の思いでしたが、後年 アセンブラを得意とする方が作ったプログラムを改修する機会があり、大量のフラグとジャンプ命令でどこを触っても崩れる逆ピラミッドのようなソースを見たとき協働で開発・保守するためには確かに構造化が必要だと納得しました（アセンブラで構造化ができないわけではありません-為念)。

1.2. 汎用機のメモリ限界とオブジェクト指向

銀行の第 3 次オンラインが開発された頃 所謂 汎用機（メインフレーム）は 1 つのジョブに割り当てることができるメモリは仮想記憶の拡張リージョンを使っても 31 ビット 2GB のアドレス空間が最大で、標準の割り当ては数十 MB 程度です。メモリに全体を展開しきれない大きなプログラムは同時に使われることがないモジュールをグルーピングして置き換えながら（オーバレイ）実行します。

このように、汎用機の場合サイズの大きなプログラムはモジュール構造を人間が管理する必要がありましたが、Linux 等のサーバは現在一般的な 64 ビットで当時の汎用機の 2^{33} 倍のアドレス空間を持ち、オブジェクトが必要になった時点でメモリにロードして不要になったら抹消するので予め AP 全体の構造やサイズを確認する必要はありません。Java 等のクラスはロード時に処理（メソッド）が確定（上書き：オーバロード）し、オブジェクトが参照されている間はデータが維持されます。

2. 構成物の定義と名前

構造化は対象を幾つかの構成物に切り分け名前を付けます。名前を付けてＩＯを決めれば構成物の定義になります。例えば“構造化プログラミング”といえば「あぁ、順／分岐／繰返し GOTO レスね」と中身をいちいち説明しなくても協働者間の会話が成立するようになります。

日本語には以心伝心という言葉がありますが、コンピュータシステムは逆に厳密な式と文で表現されていなければなりません。そして、構造化で対象とする「構成物及び構成物間の配置・関係」の各構成物は内容を的確に表す名前付けが必要です。構成物がサブ構成物に分解できるのであれば更に名前を付け、定義していきます。

構成物は通常 名前を使って利用されます。名前で呼び出すたびに実体が変わると困るので一意であることが前提になります。一意な範囲を名前空間と言い、名前空間が異なれば同一名が許されます。

汎用機の場合は区分編成ファイル（ライブラリ）のメンバとして保存し、JCL で当該のファイル名と場合によってはファイルがある DASD を指定します。したがって、DASD＋ファイル名が名前空間になりファイル名＋メンバ名が一意になるようにします。

サーバ OS はファイルをディレクトリ階層の中に保存します。ディレクトリ名は共用のライブラリであれば開発元の組織名や仕様名、それ以外ではシステム名等を付けて細分化してサブディレクトリに落としていきます。例えば、FOCS 社が開発するライブラリのディレクトリ階層は以下になります。

＜*workspace*＞
└ jp
　└ co
　　└ focs
　　　└ lib
　　　　└ example.java

Java では階層・名前空間をパッケージと呼び、上の例のパッケージ名は jp.co.focs.lib になります。

3.　設計の遷移

システム化の対象に対して「構成物及び構成物間の配置・関係」を基に設計する構造化は構成物をオブジェクトと捉えればそのままオブジェクト指向になります。ただし、実装に関しては明らかに異なる（ここでは汎用機で COBOL85 を使う場合に限定[3]します）部分があるのでそれに合わせて設計を変えていかなければなりません。以下、具体的な例を使って説明します。

【ケース例】

・販売店から注文を受け、受注可否チェックを行い問題がなければ受注登録処理を行う。
・納品先となる販売店と代金の請求先となる本社を取引先に混在して管理しており、販売店レコードに設定した本社コードで関係づけている。

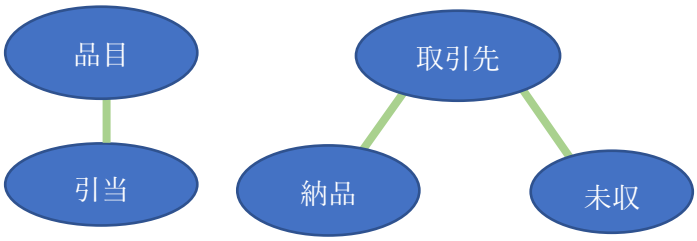
＜受注可否チェック＞以下 OK 条件

・在庫チェック：品目.在庫 ≧ sum(引当.数量) + 注文.数量
・上限チェック：販売店の制限数量 ≧ sum(納品.数量) + 注文.数量
・売掛チェック：本社の売掛限度 ≧ sum(未収.金額) + 注文.金額

＜受注登録処理＞

・引当登録：品目.引当件数に 1 加算して更新し、引当を新規登録（注文.数量）
・納品登録：取引先（本社）の納品予定件数に 1 加算して更新、納品を状態 = 「未済」で新規登録

－ 以上 －



[3] PL/I、COBOL2002 等 局所変数や処理の動的割当てが可能な言語は説明の簡略化のため除外します

汎用機・構造化世代向け、オブジェクト指向

3.1. 構造化による設計

構成物をモジュールとして設計します。更にサブ構成物がある場合は子モジュールまたは実装の段階でセクションになります。モジュール分割は保守性を上げるために以下のような基準が使われます。
①モジュール間の依存度（モジュール結合度）を下げ、扱うデータ機能を混ぜない（モジュール強度）
②リエントラントになる（再呼び出しで結果が変わらない）ように作る
③１モジュールは数百ステップで分割する

※②は、データ領域の確保がモジュール毎に一つだけなので「状態（データ）を処理後に残さない」ようにして副作用を防ぎます。（言語仕様としてデータが自動的にクリアされることはありません）

モジュールの結合度や強度は教科書的には幾つか段階がありますが、１データ×１機能に近づけていくと強度が上がり処理振分用の制御パラメータや共用データが減って結合度が下がります。

【設計例】

注文処理メイン
DATA　DIVISION
01 注文
01 品目
01 引当
01 取引先
01 納品
01 未収
PROC　DIVISION
受注可否チェック
　在庫チェック
　　CALL 品目／引当　USING 品目，引当
　　品目／引当モジュール　LINKAGE 品目，引当
　　・品目読込み
　　・引当-該当品目全件読込み＆数量集計
　上限チェック
　　CALL 取扱店／納品
　　USING 取扱店 CD，取引先，納品
　　取扱店／納品モジュール LINKAGE 取引先，納品
　　・取扱店 CD で取引先を読込み
　　・納品-該当納品全件読込み＆数量集計
　売掛チェック
　　CALL 本社／未収
　　USING 取引先.本社 CD，取引先，未収
　　本社／未収モジュール LINKAGE 取引先，未収
　　・本社 CD で取引先を読込み
　　・未収-該当未収を全件読込み＆金額集計
　（続く⤴）

※業務データに限定したモジュール間の共有イメージです。ＤＢデータの実体がどこに確保されるかやインタフェースに必要な制御データはDBMS やその他のミドルウェアで異なります

●保存が必要なデータはメインモジュールに確保し、サブモジュールにパラメータで渡します

●データを参照するモジュールを限定して強度を上げます

（↘続き）
受注登録処理
　引当登録
　　CALL 品目／引当　USING 品目，引当
　　引当登録モジュール　LINKAGE 品目，引当
　　・品目更新
　　・引当登録
　納品登録
　　CALL 取扱店／納品　USING 取引先，納品
　　納品登録モジュール LINKAGE 取引先，納品
　　・取引先更新
　　・納品登録

汎用機・構造化世代向け、オブジェクト指向

【設計例-改】

注文処理メイン
DATA　DIVISION
01 注文
PROC　DIVISION
　CALL 品目／引当　USING 品目 CD，注文数

　品目／引当モジュール　LINKAGE 品目 CD，数量
　DATA　DIVISION
　01 品目
　01 引当
　PROC　DIVISION
　　在庫チェック
　・品目読込み
　・引当-該当品目全件読込み＆数量集計
　　引当登録
　・品目更新
　・引当登録

　CALL 取扱店／納品
　USING 取扱店 CD，取扱店 CD，注文数

　取扱店／納品モジュール
　　LINKAGE 取引先 CD，納品
　DATA　DIVISION
　01 取引先
　01 納品
　PROC　DIVISION
　　上限チェック
　・取扱店 CD で取引先を読込み
　・納品-該当納品全件読込み＆数量集計
　　納品登録
　・取引先更新
　・納品登録

　CALL 本社／未収 USING 本社 CD,注文金額

　本社／未収モジュール LINKAGE 取引先 CD，金額
　DATA　DIVISION
　01 取引先
　01 未収
　PROC　DIVISION
　　売掛チェック
　・本社 CD で取引先を読込み
　・未収-該当未収を全件読込み＆金額集計
　・売掛金の総額チェック…*モジュール内で処理を完結*

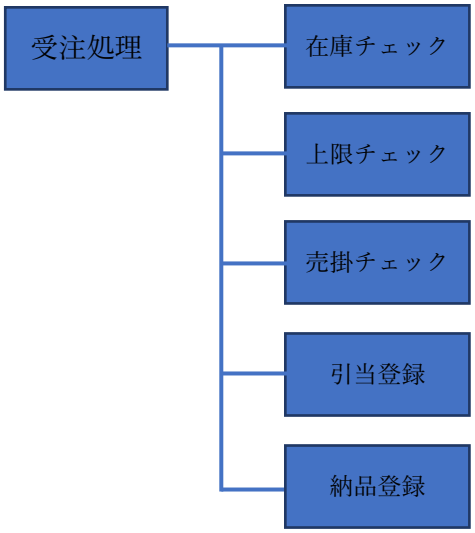
IF　ERROR
　THEN ROLLBACK
　ELSE COMMIT
END-IF

※設計例の、データチェックと登録処理を１つのモジュールで行うように改修した例
【特徴】モジュール毎に担当のデータチェックでエラーが無かったら即更新処理を行います

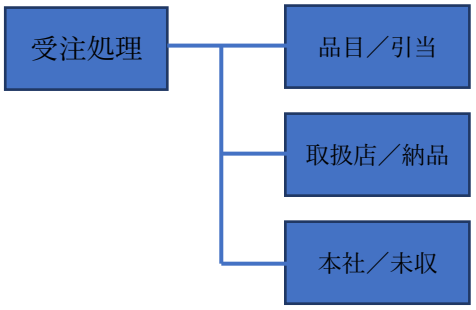
●1 データ＝1 モジュールなのでモジュール間でデータを共有する必要がなく、パラメータは少数の基本項目だけになって結合度が下がります

●全ての取引条件がチェック OK になる前に更新を始めてしまうので、処理中にエラーを検出したら更新データのロールバックを行います

設計例 モジュール構造図



設計例-改 モジュール構造図

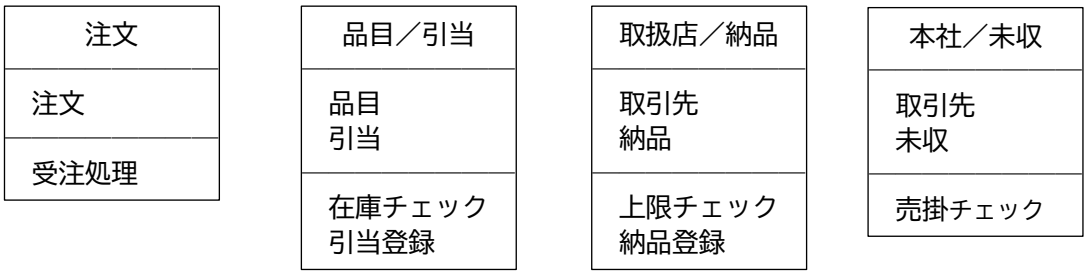


※ 設計例-改はチェックと更新がデータ種類別に入れ子になるので更新ログの出力に注意が必要ですが、データと処理をカプセル化したオブジェクト指向に繋がる構造になっています。

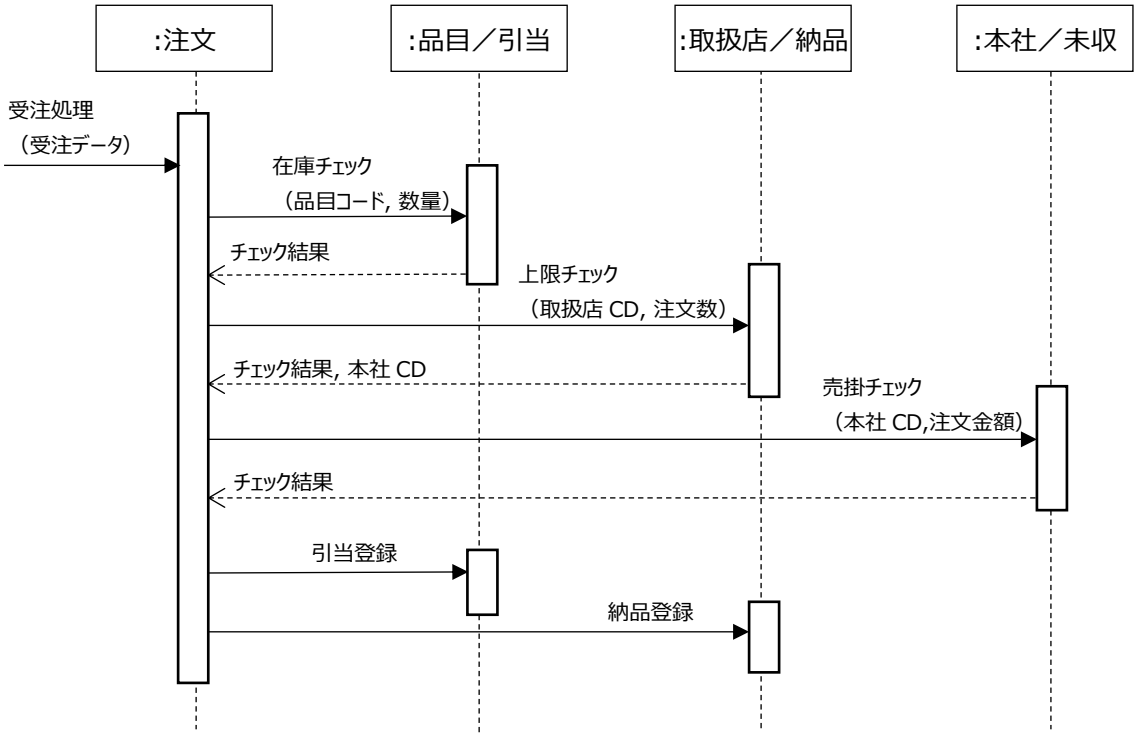
3.2. オブジェクト指向への展開

構造化の設計例-改の構成をクラス化します。

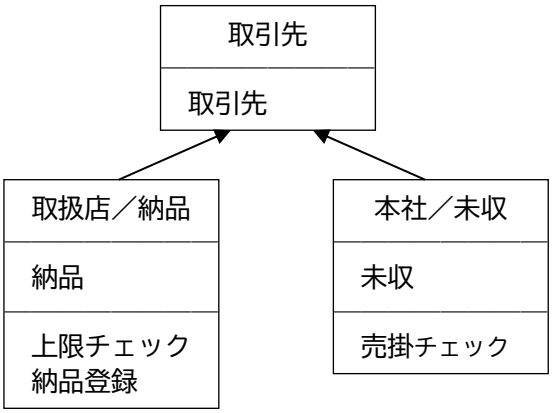
（1）クラス図（初期）



（2）シーケンス図



（3）オブジェクト指向による追加



取扱店と本社はどちらも「取引先」で管理しているので、取引先クラスを継承する型にすることでモジュール強度が上がります。

（4）追加の改善-できることやできないこと

取扱店が持っている本社コードを[取扱店／納品]⇒[注文]⇒[本社／未収]と持ち回って結合度が高くなっています。また、顧客の組織が多段階（県、関東、関西、…）になる場合があるので実際は[取扱店／納品]から[本社／未収]を回帰的に呼び出す方が現実的な構造かもしれません。

後付けで取引先どうしが階層を持つようになった場合は、オブジェクト指向の言語であれば回帰処理で対応が可能ですが、そうでない場合は分析の工程からやり直してデータベースの構造を変えるかリアルタイムの売掛チェックを諦めなければなりません。

4. 汎用機・構造化の不都合な点

COBOL を使った場合データ定義が同一モジュール内の全ての処理から参照できてしまうので、モジュール分割以外に強度を上げる方法がありませんでした。これにより起こる不具合は、簡単な例では添え字を使ってループ（外PERFORM）した先または更にその先で同じ名前の添え字をループに使っていて不可解な動きをする等はありがちです。

また、モジュール分割しても保存が必要なデータはメインモジュールに定義が必要で、モジュール分割しても引数でデータを共有していると最終的な出力がどうなるかは全てのモジュールを調査しないと分かりません。

アジャイルという言葉と共にジャストインタイムでの開発が求められるようになってきて、調査の工程に掛けられる時間は少なくなっています。同じ処理があちこちに書かれていれば改修の手間や見落とし、試験範囲の拡大が課題になり、パラメータの受け渡しで闇雲にデータの受け渡しをしていると改修に関係ない（と判定していた）処理にデグレが発生したりします。

5. どうする？

設計の原則はモジュールやクラスに分割するときは更新・参照するデータだけを処理対象にし、同じ意味の処理ブロックがあったら共通化します。システム開発時の分析が現状に合わなくなってくることはしばしばあります。例えば、事例では「品目と引当」や「取扱店と納品」のように必ず同時に使う複数のエンティティを 1 クラスで扱うようにしましたが、まず最初はエンティティ単位でクラスを考えた方が仕様変更への耐久性があがります。

また、オブジェクト指向言語を使っても作業用のデータをインスタンスやクラスレベルで宣言したり内部のデータを外部からアクセスできるようにしていては、前項で挙げた不具合の全てが引き継がれる上に解析は更に困難になります。

以上

