



Object-Oriented Conference 2024

Object Oriented Conference 2024 公式ガイドブック

2024-03-24 初版第1刷 OOC 2024 実行委員会 発行

はじめに -OOC 2024 開催によせて

OOC2024 にお越しくださり、ありがとうございます。

ariaki さんが何か書く

2024 年 3 月 24 日
Object Oriented Conference 実行委員会
代表 森川晃

免責事項

- 本書の内容は、情報提供のみを目的としています。正確性には留意していますが、必ずしも保証するものではありません。この本の記載内容に基づく結果について、著者・編集者とも一切の責任を負いません。
- 会社名・商品名については、一般に各社の登録商標です。TM 表記等については記載していません。また、特定の会社・製品などについて、不当に貶める意図はありません。
- 本書の一部あるいは全部について、無断での複写・複製はお断りします。

目次

はじめに -OOC 2024 開催によせて	3
第Ⅰ部 ご案内	9
第1章 開催概要	10
1.1 理念と目的	10
1.1.1 OOC 実行委員会	10
1.2 開催概要	11
1.3 参加者向けご案内	12
1.3.1 会場アクセス	12
1.3.2 来場予約について	12
1.3.3 体調管理について	12
1.3.4 スタッフ巡回について	12
1.3.5 撮影・録画	13
1.4 注意事項	14
1.4.1 お子様連れの参加者の方へ	14
1.4.2 参加者の皆様へ	14
1.4.3 緊急時対応	14
第2章 行動規範	15
2.1 はじめに	15
2.2 行動規範	15
2.2.1 概要	15
2.2.2 本則	15
2.3 行動規範違反について	18
2.4 この行動規範の問い合わせ先	19
2.5 この行動規範のライセンス	19
第3章 協賛企業様のご紹介	20
3.1 ゴールドスポンサー	21
3.2 シルバースポンサー	21

第 4 章	OOC2024 前夜祭のご紹介	22
4.1	前夜祭の目的	22
4.2	前夜祭の会場	23
4.3	前夜祭の企画内容	23
4.3.1	本編スピーカーによるランダム対談	23
4.3.2	明日のトークを 1 分紹介（スピーカー PR）	24
4.3.3	歓談タイム	24
4.4	前夜祭のご飯とお酒	24
4.5	GoodLuck!	25
第 II 部	協賛企業様による寄稿記事	26
第 5 章	DDD で迷わぬために、チームの正解を見つける	27
5.1	背景	27
5.2	1."DDD 認識共有会" の開催	28
5.2.1	やること	28
5.2.2	ゴール	28
5.2.3	やってよかったこと	28
5.2.4	工夫したこと	29
5.3	2."DDD 実装ガイドライン" の作成	29
5.3.1	書いたこと	29
5.3.2	やってよかったこと	30
5.4	おわりに	31
第 6 章	SOLID 原則をギュッとまとめた	32
6.1	ギュッとまとめました。	32
6.2	解説	33
第 7 章	レビューガイドラインを導入してみよう！	34
7.1	レビューガイドラインの導入の背景	34
7.2	レビューガイドライン 7 つの観点	34
7.3	コードレビューでの指摘方法	40
7.4	最後に	40
第 8 章	Fortran によるオブジェクト指向プログラミング	42
8.1	はじめに	42
8.2	ベクトル空間	42
8.3	Fortran による n 次元ベクトルの実装	43
8.4	さらなる発展にむけて	45

第 9 章 オブジェクト指向で目指すものと目指さないもの	47
9.1 オブジェクト指向とはなにか	47
9.1.1 設計/分析/プログラミング/言語	47
9.1.2 オブジェクト指向で書くために必要なこと	47
9.2 雰囲気 OO を用いる目的、目指す場所	48
9.2.1 プロジェクト特性と雰囲気 OO	49
9.3 まとめ	49
第 10 章 PHP 製 OSS に見るデザインパターンの具体例 3 選 + α	50
10.1 ①ストラテジパターン@PHPUnit	50
10.2 ②オブザーバパターン@Laravel	52
10.3 ③ビジターパターン@PHP-Parser	54
10.4 α. インタフェース、多態性の活用	57
10.5 おわりに	57
第 11 章 モデリングを育てる前の種蒔きと土作り	59
11.1 これはなんの記事？	59
11.2 種蒔き：輪読会	59
11.3 土作り：ADR(Architecture Decision Records)	60
11.4 さいごに	60
第 12 章 OOUI を関数で考える	62
12.1 UI を関数で考える	62
12.2 モデルを TypeScript の型で記述する	63
12.3 モデルに関心のある関数を書く	64
12.4 モデルの具体としてのデータと UI	65
12.5 おわりに	67
第 13 章 デザインパターンを学ぶ	69
13.1 はじめに	69
13.2 デザインパターンとは	69
13.2.1 デザインパターンへの批判など	69
13.2.2 その他のデザインパターン	70
13.3 デザインパターンを学習する上で大事なこと	70
13.4 さいごに	71
第 14 章 生成 AI の不確実性に立ち向かうソフトウェアアーキテクチャ	72
14.1 生成 AI とは	72
14.2 生成 AI をプロダクトで活用するということ	73

14.3	そんな生成 AI を活用するプロダクトをどう設計するか	74
14.4	コアドメインを互いに分離する	76
14.5	生成 AI のモデルを切り替え可能にする	76
14.6	まとめ	77
第 III 部 イベントを支える技術-スタッフ寄稿—		78
第 15 章 搬入・宅急便関係の準備いろいろ		80
15.1	事前準備	80
15.1.1	事務局の事前準備	80
15.1.2	一般サークルの事前準備	82
15.2	まとめ	82
15.3	次回予告	83
スタッフ紹介		84
コアスタッフ	84
デザイン	85

第1部

ご案内

第 1 章

開催概要

1.1 理念と目的

目的と理念

1.1.1 OOC 実行委員会

私たち Object Oriented Conference は、有志のボランティアによって成り立っています。

少ない人数で試行錯誤を繰り返しながら皆さまにより良い価値を提供できればと考え、日夜精一杯がんばっています。もし至らないところがありましたら、どうかご容赦いただき、改善点を遠慮なくご教示ください。イベントを通じて私たちも成長し、皆さまと共によいエンジニアコミュニティを築けることを願います。

また、私たちと一緒に楽しさを作っていく仲間を募集しています。興味のある方はお近くのスタッフにお声がけいただくか、公式 Twitter アカウントまでご連絡ください。

1.2 開催概要

催事名

Object Oriented Conference

催事略称

OOC 2024

開催日時

2024 年 3 月 24 日（日）11:00～17:00

開催会場

お茶の水女子大

会場住所

〒112-8610 東京都文京区大塚 2-1-1

開催規模

トーク 40 本参加者 1000 人 (想定)

運営団体名

OOC 実行委員会

公式 Web サイト

<https://ooc.dev/>

公式 Twitter アカウント

<https://twitter.com/ooc>

ハッシュタグ

#OOC_2024

1.3 参加者向けご案内

1.3.1 会場アクセス

<https://www.ocha.ac.jp/access/index.html>

- ・ 東京メトロ丸ノ内線「茗荷谷」駅より徒歩 7 分
- ・ 東京メトロ有楽町線「護国寺」駅より徒歩 8 分
- ・ 都営バス「大塚 2 丁目」停留所下車徒歩 1 分

1.3.2 来場予約について

Connpass にて登録をお願いしております。

カンファレンス参加は無料ですが、ご登録をお願いします。

<https://ooc.connpass.com/event/305241/>

ただし、各種ノベルティは先着順で配布します。十分な数を準備しておりますが、遅い時間にご来場された場合にはお渡しできない場合があることをご了承ください。

また、懇親会へのご参加は事前登録が必須です。

1.3.3 体調管理について

新型コロナウイルス、インフルエンザなどの流行が予想されます。体調のすぐれない方は無理をしないようにして下さい。

体調不良時は近くのスタッフまで遠慮なくお声がけください。

1.3.4 スタッフ巡回について

会場内の安全確保のため、スタッフが常時巡回します。会場内では、スタッフの誘導にしたがってください。

不審物、トラブル、迷惑、危険行為、不審な行動をする者等に気づいた場合は、すぐにお近くのスタッフまでご連絡ください。その他にも不明点があれば、事務局・スタッフまで遠慮なくご相談ください。地震・火災・テロなどの際は、スタッフの誘導にしたがって、慌てず避難ください。

1.3.5 撮影・録画

会場内にて撮影・録画を行う場合は、個別に被撮影者の許可を得てください。

会場には不特定多数の方が来場するため、十分な配慮をお願いします。 不審な点がみられる場合、運営事務局が撮影内容を確認いたします。

なお、運営スタッフがイベント開催資料のため、会場内を撮影する場合があります。

1.4 注意事項

1.4.1 お子様連れの参加者の方へ

- ・迷子、ケガなどのトラブルを防止するため、絶対に目を離さないようにしてください。
- ・会場の出入りにあたっては、必ず手をつないで出入りください。
- ・迷子等防止のため出入口スタッフにて確認・声掛けする運用としています。
- ・混雑が予想されます。他の参加者の目線に入らないことがあるため、混雑している区画を避ける、周囲に声掛けをするなど、お子様の安全確保にご留意ください。
- ・3歳以下のお子様連れの参加者のみなさんは、通路でぶつかったりする危険を踏まえ、お子様から目を離さないようお気をつけください。
- ・実行委員会として全力でサポートするよう努めますが、あらゆるトラブルは故意の有無に問わらず、OOC 実行委員会はその責を免れるものとします。

1.4.2 参加者の皆様へ

- ・背の低いお子様が歩いていることがあります！
- ・足元にご注意いただき膝下や荷物などがぶつからないようご注意ください。
- ・ひとりでいるお子様に気付かれた場合は、スタッフにお声がけください。全力で対応します。
- ・その他トラブル、迷子等に気づいた場合、直ちに運営事務局、会場スタッフにご連絡ください。

1.4.3 緊急時対応

迷惑行為・危険行為等に遭遇した場合、お近くのスタッフにご連絡ください。お近くにスタッフが見つからない場合、受付スタッフにお声がけください。

地震・火災・テロなど災害発生時はスタッフが誘導いたします。慌てずスタッフの指示に従ってください。

行動規範

2.1 はじめに

Object-Oriented Conference（以下、「当カンファレンス」といいます）は、すべてのエンジニアが自身の知見を共有することで成長と幸福をもたらすことをコミュニティ理念とし、この使命を実現するためにコミュニティ活動を行います。

当カンファレンスの理念に基づき、すべての関係者（参加者・登壇者・出展者・スタッフ・スポンサーを含む。以下「関係者」といいます）の判断の拠り所や取るべき行動を定めたものが、行動規範です。

すべての関係者がこの行動規範を理解・遵守し、高い倫理観をもって、誠実で公正に行動することを望みます。

この行動規範は、当カンファレンスが主催するイベント会場やイベントに関連する活動、オンラインでのコミュニケーション空間など、当カンファレンスのあらゆる活動に対して適用されます。

当カンファレンスの主催者ならびにスタッフは、関係者に対してこの行動規範の遵守を徹底することで誰にとっても安全かつ安心な環境を確約し、違反者に対しては、警告や、イベント会場からの即時退場の指示等を含むあらゆる処置を当カンファレンスの裁量でもって行います。

2.2 行動規範

2.2.1 概要

関係者は、自らのとった行動や発言に責任をもち、常に他者に敬意と礼節をもって接します。

また、特定の個人や団体に対する攻撃的な発言を慎み、あらゆるハラスメントや差別を排除し、他の関係者が安全かつ安心に過ごすことができるよう徹底します。

当カンファレンスは、悪質な勧誘やなりすまし行為、個人情報の不正利用、性的コンテンツの掲示などの迷惑行為を許さず、反社会的勢力の構成員または協力者を加入させません。

当カンファレンスは、本規範の違反者に対してあらゆる手段を講じて厳正に対処し、円滑なコミュニティ運営に努めます。

2.2.2 本則

敬意を持った行動の徹底

関係者は、コミュニティ参加を通じて、常に他者への敬意と礼節をもって接します。

また、特定の個人または団体に対する暴言や誹謗中傷など貶める発言を慎み、他者への攻撃的な言動を行わないように務めます。

とくにイベントにおいては、運営者・登壇者・出展者などに対して進行を妨げないよう注意するとともに、攻撃的発言を慎み、円滑な運営に協力します。

ハラスメント行為の禁止

関係者は、他者を許容し、他者の人格を尊重する為、あらゆるハラスメント行為をしないよう徹底します。

他者の人格とは、性差・性自認と表現・性指向・障がい・容姿・外見や身体的特徴・年齢・健康状態・人種・民族・出身国・宗教・政治・思想などあらゆることがらを指し、すべての人がもつアイデンティティを尊重します。

また、ここでのハラスメントとは、公的空間での性的な画像や類する表現・ナンパ行為（容姿に関する発言、恋愛・性的興味を目的とした発言）・誹謗・中傷・脅迫・暴力・暴力の助長・威力行為・ストーキングやつきまとい・不適切な身体的接触・写真撮影や録音によるいやがらせ・コミュニケーション運営に対して繰り返し中断や混乱を目的とした行為、および社会通念に照らし嫌がらせと認められる一切の言動を指します。

関係者は、他者に対する威嚇、品位を貶める行為など、敵対的な状況を生み出す言動を控え、自らのとった行動や発言に責任をもちます。

エンジニア尊重主義

当カンファレンスは、すべてのエンジニアがもつスキルや技術、職業や職責、および、その背景と選択を尊重します。

初心者と上級者、知っていることと知らないこと、使用する言語やフレームワークなどの技術的素地によって差別されるべきではありませんし、あらゆる技術や職業に貴賤はありません。

勧誘の禁止

当カンファレンスは、コミュニティを健全に運営するため、一切の勧誘行為を禁止します。

ここでの勧誘とは、営利を目的とした宣伝や営業、エンジニアコミュニティ以外への誘導、政治活動、宗教活動、求人募集など、当カンファレンスの目的と直接合致しない行為を指します。

ただし、次の場合はその限りではありません。

1. 相手方が直接望んだ上でなされる、関係者間での情報提供
2. 協賛企業による宣伝告知
3. その他、当カンファレンスの主催者が特別に認めるもの

なりすましの禁止

当カンファレンスは、他者へのなりすましを許容しません。

ここでのなりすましとは、自身以外の特定の人物、自身が所属しない団体、自身が組織内で意思

決定権を持つように見せかける行為など、あらゆる詐称行為を含み、なりすましを用いたあらゆる言動を禁止します。

性的コンテンツの禁止

関係者は、公共空間において性的コンテンツを掲示しません。

ここでの性的コンテンツとは、裸体表現・ポルノグラフィー・わいせつ行為・性行為など、性を連想する画像、映像や文章表現、あるいはその他の物品を指します。

また、性的な関心を引き起こすような服装・制服・コスチュームを使うべきではありませんし、その他の手段で性的な関心を引き起こすような環境を作ることもしません。

個人情報不正利用の禁止

関係者は、他者のプライバシーを侵害せず、個人情報を不正に利用しません。

当カンファレンスへの参加・関与によって得られた個人情報を外部へ共有または持ち出すことは固く禁じます。

当カンファレンスは、当カンファレンスの円滑な運営および業務遂行に必要と判断する場合のみ、外部業者またはサービスに対して個人情報の一部または全部を委託する場合があります。

反社会的勢力の排除

関係者は、自ら反社会勢力の構成員または協力者でない事を表明および確約します。

ここでの反社会勢力とは、暴力団、暴力団関係企業、総会屋若しくはこれらに準ずる者又はその構成員ではないことを指します。

また、あらゆる犯罪や暴力に対して毅然たる態度で臨み、その要求には一切応じません。

撮影や録画などの記録

関係者は、写真撮影・動画録画・音声録音など（以下、記録）において必ず被記録者に許可を取り、原則として私的利用にとどめます。

撮影時は相手の配慮を忘れず、また相手や主催者から撮影中止を命じられた場合には速やかに従います。

なお、当会のイベントは、主催者・スタッフ・スタッフに委託された者によって記録し、参加者に対して同意を得たうえでコンテンツを公開する場合があります。

著作権保護

コミュニティより提供されるすべてのコンテンツは、提供者（主催者・登壇者・出展者など）に著作権が帰属します。

私たちは、著作権者を尊重し、厳格な著作権保護に努めます。

最善の努力

関係者は、行動規範を遵守し、誇り高く誠実で公正な行動に努めます。

当カンファレンスにおけるすべての関係者は、当カンファレンスとともに最善の努力を尽くすことで、誰にとっても安全かつ安心な環境を作りだします。

関係者の行動指針は、常に「Don't be evil (邪悪になるな)」かつ「Do the right thing (正しい行動をとろう)」であり続けます。

2.3 行動規範違反について

行動規範違反者への対処

当カンファレンスは、違反行為を行った者に対する警告・退出指示・参加停止・追放などを含むあらゆる処置を行う権利を有し、適切と判断した範囲で行います。なお、この場合、既に受領したイベント参加費等は返金致しません。

また、必要に応じて法的機関に相談し、早急な解決と被害者の保護に努めます。

行動規範違反を発見した場合

もし関係者が他の関係者の行動規範違反を発見した場合、関係者や他の関係者がハラスマントに遭っている場合、関係者や他の関係者に対して危険または不寛容な態度を示された場合、その他に懸念や質問がある場合は、すぐに当カンファレンスの主催者またはスタッフに連絡してください。

お知らせ頂いた内容は、法的な情報開示の必要がある場合を除いて、連絡者の同意がない限り匿名で扱いますのでご安心ください。

当カンファレンスの主催者またはスタッフは、ハラスマントを経験した人がイベント会場内および往復路において安全かつ安心と感じられるよう、警察への通報連絡、付き添い者の提供、およびその他のあらゆる支援を行います。

行動規範違反を指摘された場合

もし関係者が他の関係者の行動規範違反を発見した場合、関係者や他の関係者がハラスマントに遭っている場合、関係者や他の関係者に対して危険または不寛容な態度を示された場合、その他に懸念や質問がある場合は、すぐに当カンファレンスの主催者またはスタッフに連絡してください。

お知らせ頂いた内容は、法的な情報開示の必要がある場合を除いて、連絡者の同意がない限り匿名で扱いますのでご安心ください。

当カンファレンスの主催者またはスタッフは、ハラスマントを経験した人がイベント会場内および往復路において安全かつ安心と感じられるよう、警察への通報連絡、付き添い者の提供、およびその他のあらゆる支援を行います。

スタッフの識別

当カンファレンスが主催するイベントにおいて、主催者およびスタッフは、自身がスタッフであることを誰もが識別できるよう、専用の名札・Tシャツ・エプロン・バッジなどによって表示します。

2.4 この行動規範の問い合わせ先

この行動規範に関してご質問がある場合は、以下までお問い合わせください。

Object-Oriented Conference 実行委員会

hello@ooc.dev (担当：森川)

2.5 この行動規範のライセンス

この行動規範は、技術書同人誌博覧会の行動規範をもとに作成され、CC-BY-4.0に基づいてライセンスされます。

令和5年9月7日制定

第3章

協賛企業様のご紹介

3.1 ゴールドスポンサー



エンジニアフレンドリーシティ福岡

<https://efc.fukuoka.jp/>

エンジニアフレンドリーシティ福岡は「エンジニアが集まる、活躍する、成長する街、福岡」の実現に向け、エンジニアと福岡市が協力して実施している取組みです。

エンジニアやコミュニティが交流、活動できるエンジニアカフェ (<https://engineercafe.jp/ja/>) の運営をはじめ、エンジニアを取り巻く環境の充実に貢献されたコミュニティや企業の表彰制度など、エンジニアファーストの精神で様々なことに取り組んでいます。

3.2 シルバースポンサー

某社

第4章

OOC2024 前夜祭のご紹介

OOC2024 コアスタッフ前夜祭担当 オーニシ@onishi_feuer

OOC2024 では前夜祭を開催しました。このガイドブックが頒布されるのは OOC 当日のため、前夜祭は既に開催後ではありますが、何を目的としてどんなことをやったのか、OOC 本編参加者の皆様にも知って頂き、次回の開催でも前夜祭があるようであればぜひとも参加して頂ければと思っております。

4.1 前夜祭の目的



前夜祭の目的は「参加することで翌日の OOC 本編がより楽しくなる」ことです。前日からオブジェクト指向設計について話したり、聞いたりしながら

- 明日はこんな話が聞きたいな
- このセッション面白そうだと思うんだけどどう思う？
- 前夜祭でも登壇してたこの人の話をもっと聞いてみたいな
- 美味しいご飯を食べながら設計の話をするのって最高だな！ 明日は1日ガッツリこれがでかけるのか！

そんな風に前日から徐々に気持ちを高めていってもらえるといいよね、というのが目指したゴールです。

4.2 前夜祭の会場



図: docomo R&D OPEN LAB ODAIBA

前夜祭の会場となったのは株式会社 NTT ドコモさんが運営する「docomo R&D OPEN LAB ODAIBA」。

2023 年 6 月にオープンしたばかりのスペースで、最新のドコモのネットワーク設備や次世代の光ネットワーク装置を、開発者向けになんと無料で提供してくれています。

こちらはさまざまな技術やアイデアとドコモの R&D 技術をかけ合わせて新しいサービスの創出を目指すとともに、技術者の新しい学びや発見、出会いの場としても提供されており、充実した設備とスペースを使ったイベントの開催も募集してくれています。

まさに OOC の理念や目的にピッタリ合致した会場！

ということでこちらのラボをお借りして開催することができました。感謝です！

4.3 前夜祭の企画内容

4.3.1 本編スピーカーによるランダム対談

前夜祭の目玉企画！ 少人数のスピーカー同士による対談/座談会です。「オブジェクト指向」に関する話題から運営がテーマを出題し、それについて OOC2024 本編のスピーカー同士で対談してもらうことで、スピーカー・参加者ともに本番に向けて気持ちを高めていただくと共に、アドリブ要素も相まって本番とはまた違った趣向を楽しむエンターテイメント要素も兼ねてみました。

即興対談となるとグダる可能性があるため、ファシリテーターを声優兼エンジニアの祭田絵理さ

んにお願いしました。ラジオドラマ「IT 正常運用促進課」の司会も務められており、司会業と技術両方に通じるまさに適任でした！

4.3.2 明日のトークを1分紹介（スピーカーPR）

もう1つの企画がこちら。その名の通り翌日のトークについて1分でサクッと紹介してもらうコーナーです。参加者には本編でどのトークを聞くか選択する材料になり、スピーカーは本編のPRをするとともに登壇の予行練習にもなる。もちろん1分で話したいことをすべて話しきることはできないため、「もっと聞きたかったー」を引き出すことも狙いです。

4.3.3 歓談タイム

参加者同士で交流する時間も欲しいよねってことで最後は歓談タイム。前夜祭のカジュアルな雰囲気でオブジェクト指向について語り合ったり、明日のセッション内容、注目しているスピーカーやトークについて話してもらうための時間です。やっぱり人と話すのは盛り上がるし、明日これが聞きたい！が見つかるきっかけにもなりますよね。

4.4 前夜祭のご飯とお酒

前夜祭はお祭りなのです！ ということでもちろん食事！ そしてお酒が付いてきます！



図：ピンチョス！



図: 肉!



図: 寿司!

オードブルはもちろん、肉！寿司！！本編懇親会顔負けのラインナップ！！！宴だ宴だー。

4.5 GoodLuck!

そんな前夜祭を経ての本日 OOC2024 本編、みなさん楽しんでいってくださいね！
そしてまだ未定ですが次回があれば、その時にまた前夜祭があれば、その時はぜひ参加してくださいと嬉しいです。

それでは GoodLuck!

第Ⅱ部

協賛企業様による寄稿記事

DDD で迷わないために、チームの正解を見つける

当稿はホワイトプラスのテックブログの記事を、
Object-Oriented Conference 2024 ガイドブック向けに加筆・修正したものです
<https://blog.wh-plus.co.jp/entry/2022/12/09/advent-calendar>

Lenet という宅配クリーニングサービスを展開しているホワイトプラスの CX 開発チーム所属でエンジニアをしている徳廣と申します。

ホワイトプラスに入社し、DDD（ドメイン駆動設計）をチームで「よりスムーズな実装」を目指すために取り組んだ 2 つを紹介します！

5.1 背景

突然ですが、「DDD における "Repository (リポジトリ)" についての説明」を求められた時、チームメンバーと「同じ話」ができるでしょうか？

入社してすぐの私にはできませんでした。

DDD に限らず「前職（前配属先）のチームでは別の解釈をしていました。」「解釈は同じでも、コンテキストによって方針は異なる」というのはよくある話だと思います。

背景や環境（学んできた経緯や実践してきた場）が異なるため、当然ですよね。そしてこの解釈やこれまでの経験との違い～で問題が起きました。

- 起きた問題
 - レビューの指摘で手戻りが発生
 - 既存コードについての確認で、度々チームの時間が奪われる

早くチームに貢献したいという焦りもあった私は「チームの正解（目指す設計・実装）を明確にすれば、新たな参画メンバーでも自分のフレームに気付けるし最速でチーム貢献できるようにのは？」と考えました。

チームの正解（目指す設計・実装）を探すために「チームメンバーそれぞれが考える理想をuzzけるのが早い」と考え "DDD 認識共有会" を開催することにしました！

5.2 1."DDD 認識共有会" の開催

5.2.1 やること

共有会の内容はシンプルです。

- メンバーの一人が自身の解釈や理解していることを、こんな感じで話すだけ。
 - 「そもそも DDD とは？ その目的とは？」
 - 「Aggregate（集約）とは？ その責務は？ 一番理想に近い既存コードは？」
- 誰が話すのか？
 - 徳廣が話し手になりました。（理由は後述）

5.2.2 ゴール

- チームとして
 - チームの正解（目指す設計・実装）を見つける。
- メンバー個人として
 - 普段の疑問や悩みを積極的に発言し、より良い設計・実装を模索する。
 - * 「こういう風に実装しているところを見つけたけど、自分なら別のアプローチがいいと思う。みんなはどう？」
 - * 「この前の実装でこんな問題が出てきたんだけど、みんなはどうしてる？」
- DDD 歴が浅いメンバーは理解を深める機会とし、疑問があれば解消する。
 - 「なるほど、わからん」も積極的に発言してもらう。

5.2.3 やってよかったこと

- 当初の問題点であった「時間が奪われる」ことが減った。
 - チームの正解（目指す設計・実装）を把握しているため、実装時に迷いがなくなった。
 - キャッチアップの速度が早まったと実感しています。
- 思わぬ効果
 - レビュワー間の方向性の違いによる議論が減った。
- DDD 歴が浅いメンバーの理解が深まった。
 - 既存コードに対して「チームの正解（目指す設計・実装）と違うかも？」と、疑問を持てるようになった。
 - 数ヵ月後に「共有会の時にみんなが話してたことがやっとわかりました」などの成長を感じられるようになった。
- より DDD をチームに浸透させる施策のきっかけができた。
 - 後述の「2."DDD 実装ガイドライン" の作成」で説明します

5.2.4 工夫したこと

- 他のメンバーとブツかりやすく意見が出しやすくするため、チーム歴が浅い人を話し手にした。
 - チーム歴が長いメンバー同士だと ズレ は既に解消されているため。
 - 今回は入社 2 ヶ月であった私が担当しました。
- チームの正解を導き出した時の背景が大事なので、質疑応答は議事メモとして残すようにした。
- 質疑応答は DDD 歴が浅いメンバーから話をしてもらうようにした。
 - 話に入れないまま、結果だけを受け入れてしまわないようにしました。

5.3 2."DDD 実装ガイドライン" の作成

「1."DDD 認識共有会"」を開催したことでの、DDD 歴が浅いメンバーがどのような点で躊躇やすいかがわかりました。

次のステップとして「パターン毎の実装ガイドライン作成」の施策を取り組んでいる話になります。

5.3.1 書いたこと

- 「責務と役割」
 - 各パターンの責務・役割における振る舞いや属性、制御について概念をまとめています。
- 「実現する方法」
 - MUST で実装する時に忘れてほしくないことを書いています。
- 「実装のポイント」
 - 実装する時に悩みそうなポイントをまとめています。
- 「実装例」
 - 各パターンの振る舞い毎にまとめています。

実装例

参照系ユースケース

例：リストを表示する

```
php
class UseCase
{
    private DddRepository $dddRepository;

    public function __construct(DddRepository $dddRepository)
    {
        $this->dddRepository = $dddRepository;
    }

    public function execute(int $id): Result
    {
        // 復元の依頼
        $domainItems = $this->dddRepository->of(Id::of($id));
        // DTOに詰め替えて返却
        return new Result(
            array_map(
                fn (DomainItem $item) => $this->domainItemToResultItem($item),
                $domainItems
            )
        );
    }

    private function domainItemToResultItem(DomainItem $item): ResultItem
    {
        return new ResultItem(
            $item->tagNumber()->value(),
            $item->name()->value(),
            $item->note()->value(),
        );
    }
}
```

図：実装例のサンプル：UseCase パターン

5.3.2 やってよかったこと

- 新メンバーが加わった時に説明がしやすくなった。
- 目指すゴールが明確になったので実装しやすくなった。

- いつでも目指しているゴールがわかるようになったので、他人の時間を奪わなくなった。

5.4 おわりに

もし「同じ単語でもあの人人が話すアレは、私の思っているコレと少し違うかも？」と思ったら、一度立ち止まってチームで話し合ってみませんか？

チームと個々が向いている方向が囁み合うことで生産性の向上にもつながりますし、何より迷いなく進められる事でより良い開発体験を得られるかもしれません！

第 6 章

SOLID 原則をギュッとまとめた

シキム@sikimuOji

6.1 ギュッとまとめました。

SOLID 原則を理解してもらうために、誰しも必ず通る Hello World にまとめてみました。

ギュッと Hello World

```
// メイン
public class Main {
    public static void main(String[] args) {
        Printer printer = new SystemPrinter();
        printer.print(new Hello(), new World());// 開放閉鎖の原則
    }
}

// 出力処理（単一責任の原則）
abstract class Printer {
    abstract void print(First first, Second second);// 依存性逆転の原則
}
class SystemPrinter extends Printer{
    public void print(First first, Second second){
        System.out.println(first.create() + " " + second.create());// リスコフの置換原則
    }
}

// 前半出力情報（単一責任の原則）
interface First { String create(); }// インターフェース分離の原則
class Hello implements First {
    public String create(){
        return "Hello";
    }
}

// 後半出力情報（単一責任の原則）
interface Second { String create(); }// インターフェース分離の原則
class World implements Second {
    public String create(){
        return "World";
    }
}
class Japan implements Second {
    public String create(){
        return "Japan";
    }
}
```

6.2 解説

HelloWorld に潜む SOLID 原則について解説していきます。

単一責任の原則

最小の仕様変更をしたときに、同時に変更するものをクラスにまとめましょう。出力方法を変更するなら SystemPrinter、出力情報を変更するなら Hello や World だけになります。

開放閉鎖の原則

インターフェースを利用して、呼び出し側の変更だけで済むようにしましょう。World を Japan に変更するだけで、出力情報を変更することができます。

依存性逆転の原則

変更することが少ない処理は、変更の多い処理をインターフェースで利用するようにしましょう。First や Second を使うことで、変更の少ない出力処理が変更の多い出力処理に影響しなくなります。

リスクの置換原則

インターフェースで決めた仕様だけを実装しましょう。SystemPrinter で追加処理を入れてしまうと、別の場所で使う場合に追加処理が意図せず動いたり、別の Printer に変更した場合に追加処理が動かなくなってしまうなどしてバグの原因になります。

インターフェース分離の原則

インターフェースは、変更したい最小の単位のまとまりにしましょう。FirstSecond というよう統合すると、"Hello World"から "Hello Japan"に変更したときに、変更していない "Hello"も確認する必要が出てきます。



シキム @sikimuOji <https://twitter.com/sikimuOji>

ちょっと Java 出来る静岡のおじさんです ('・ω・ ')

第7章

レビューガイドラインを導入してみよう！

@akkie76

7.1 レビューガイドラインの導入の背景

現在、私のチームは技術育成を目的として、レビューガイドラインを導入しています。特に若手エンジニアが抱える技術的な課題に対処するため、レビュー指摘を類型化し、メンバーの弱点を定量化するガイドラインを作成しコードレビューに臨んでいます。

7.2 レビューガイドライン 7 つの観点

レビューガイドラインの作成にあたっては「Google's Engineering Practices documentation」^{*1}を参考にし、7 つの観点を設けています。

表 7.1: レビューガイドライン 7 つの観点

No	観点	概要
1	Design	アーキテクチャ、設計
2	Simplicity	理解容易性
3	Naming	クラス、メソッド、変数名などの命名
4	Style	コードスタイル
5	Functionality	機能（要件）を充足しているか
6	Test	テストの記述、パターンが適切
7	Document	コメント、ドキュメントに関連

特に、No.1 ~ No.4 は、オブジェクト指向の観点で非常に重要な観点といえます。ここからは具体例を踏まえて、実際の使用例を紹介したいと思います。

Design (設計)

定義

^{*1} <https://google.github.io/eng-practices/>

アプリケーションとして一貫性のあるアーキテクチャ、設計となっているか。また、アプリケーションにとって適切な責務、振舞いになっているか。

Design では、主に以下のケースが指摘の対象になります。アーキテクチャ原則違反はアプリケーションで導入しているアーキテクチャを前提にしており、これに違反している場合は指摘の対象になります。他にも以下のような基本的な SOLID 原則に反するような場合も指摘の対象になります。

- アーキテクチャ原則違反
- 単一責任の原則違反
- 密結合
- 低凝集
- DRY 原則違反 etc.

単一責任の原則違反の例

以下のコードでは、`add()` で 2 つのバリデーションと 2 つのビジネスロジックが実装されており、単純に「商品を追加する」以外の責務を 1 つの関数の責務として行われています。この関数自体複数の責務を持ってしまっているため（責務超過）、Design 指摘の対象となります。

多重責務の関数例

```
class SpecialDiscountManager {
    // 商品の割引に関するビジネスロジッククラス
    lateinit var manager: DiscountManager

    fun add(product: Product): Boolean {
        if (product.id < 0) { // バリデーション 1
            throw IllegalArgumentException()
        }
        if (product.name.isEmpty()) { // バリデーション 2
            throw IllegalArgumentException()
        }
        val tempPrice: Int = if (product.canDiscount) { // 条件分岐 1
            manager.totalPrice + manager.getDiscountPrice(product.price)
        } else {
            manager.totalPrice + product.price
        }
        return if (tempPrice < 3000) { // 条件分岐 2
            manager.totalPrice = manager.discountProducts.add(product)
            true
        } else {
            false
        }
    }
}
```

このように1つの関数に複数の責務が実装されるような場合、なるべく関数の責務を分離することでテスト容易性も高まるため、望ましい設計といえるでしょう。

責務分離の実装例

```
class SpecialDiscountManager {

    /** 商品の割引に関するビジネスロジッククラス */
    lateinit var manager: DiscountManager

    /**
     * 商品を追加する
     *
     * @param product 商品
     */
    fun add(product: Product) {
        // 不正な商品をバリデートする
        validateInvalidProduct(product)
        // 商品を追加する
        if (canAdd(product)) {
            manager.totalPrice = manager.discountProducts.add(product)
        }
    }

    /**
     * 不正な商品をバリデートする
     *
     * @param product 商品
     * @throws IllegalArgumentException 不正な商品である場合の例外
     */
    private fun validateInvalidProduct(product: Product) {
        if (product.id < 0 || product.name.isEmpty()) {
            throw IllegalArgumentException()
        }
    }

    /**
     * 商品が追加可能か
     *
     * @param product 商品
     * @return 追加可能かどうか
     */
    private fun canAdd(product: Product): Boolean {
        val tempPrice = getTempPrice(product)
        return tempPrice >= 3000
    }

    /**
     * 商品の一時金額を取得する
     *
     * @param product 商品
     * @return 商品の一時金額
     */
    private fun getTempPrice(product: Product): Int {
        return if (product.canDiscount) {
            manager.totalPrice + manager.getDiscountPrice(product.price)
        } else {
            manager.totalPrice + product.price
        }
    }
}
```

}

Simplicity (理解容易性)

定義

可読性のあるコーディングになっているか。処理ができるだけシンプルな振る舞いになっているか。

以下のようにコードが複雑になる場合、Simplicity の指摘対象になります。

- ネストが深い if 文
- 複雑な三項演算子文
- 冗長な SQL
- stream, filter, map を多用した Object 整形文 etc.

例えば、以下のような if 分は指摘の対象になります。

if - else 分岐が多い実装例

```
val powerRate: float = member.powerRate / member.maxPowerRate
var currentCondition: Condition = Condition.DEFAULT
if (powerRate == 0) {
    currentCondition = Condition.DEAD
} else if (powerRate < 0.3) {
    currentCondition = Condition.DANGER
} else if (powerRate < 0.5) {
    currentCondition = Condition.WARNING
} else {
    currentCondition = Condition.GOOD
}
return currentCondition
```

実際のレビューコメントでは、以下のようにネストを解消するように指摘をする場合などに使用します。

早期リターンへの改善例

```
val powerRate: float = member.powerRate / member.maxPowerRate
if (powerRate == 0) {
    return Condition.DEAD
}
```

```
if (powerRate < 0.3) {  
    return Condition.DANGER  
}  
if (powerRate < 0.5) {  
    return Condition.WARNING  
}  
return Condition.GOOD
```

Naming（命名）

変数やクラス、メソッドに責務を意図した明確な名前が付けられているか。英語文法に誤りがないか。typo もこれに含まれる。

このような命名に関する指摘をする場合に使用します。

- 振舞いと一致しない変数名、関数名
- 責務と一致しない関数名
- 英文法の誤り etc.

Style（コードスタイル）

定義

コードスタイルが言語仕様に準拠しているか。

コードスタイルは、言語仕様やフレームワークに準拠することが基本です。公式にガイドラインが公開されている場合、これに反することは避けるべきです。

- 静的解析違反
- 不適切なアクセス修飾子
- 表記違反（スネーク、キャメルなど）etc.

Functionality（機能要求）

定義

システムとして外部仕様を充足しているか。作者が意図した通りの振る舞いであるか。また、システムの通信量、パフォーマンスに懸念がないか。

主な観点としては、外部機能など仕様を充足しているかという点が対象になります。また、パフォーマンスやセキュリティといった非機能要件に懸念がある場合も指摘の対象となります。

- 外部機能が充足していない（不具合）
- 概要設計書のフローチャートと異なるフローになっている
- パフォーマンスの低下が考えられる
- 不要データを送信している etc.

Test (テスト)

定義

システムとして適切な自動テストを兼ね備えているか。自動テストの内容で品質を担保できているか。また、システムを担保するパラメータ群を備えているか。

テストコードが期待にならない場合や、テストでのパラメータに考慮漏れがある場合などテストに関連する事項が指摘の対象になります。

- 対象のメソッドがテストされていない
- テストパターンが網羅できていない（パラメータテスト、閾値テストの不足）
- 分岐がパターンが網羅されていない
- 実装上宣言している静的定数値が直接ハードコードされている
- アーキテクトに準じたテストにならない

Document (文章)

定義

ソースコード上に記載されている doc、コメントが適切な内容であるか。また、関連するドキュメントは更新されているか。その内容は適切か。

ソースコードに関連するコメントだけでなく、プロジェクトで管理している関連ドキュメント (README) も対象になります。

- 関連ドキュメントの更新漏れ (README など)
- doc やコメントの内容が不適切、内容が不適切

7.3 コードレビューでの指摘方法

コードレビューでは上記の 7 つの観点に加えて、指摘修正の要否を 4 つの類型に分けてコメントをします。

表 7.2: 修正要否

観点	概要
MUST	PR、MR をマージするためには必ず修正が必要
SHOULD	修正なしにマージすることはできるが、リリースまでには修正が必要
IMO	レビューアー観点の意見。修正不要
NITS	IMO より細かい意見など。修正不要

コードレビューでマージするために必要な修正は MUST 指摘となります。MUST と SHOULD の使い分けは難しい部分もありますが、SHOULD での指摘事項は別途リリースまでに対応するなど、レビューアーを相談して対応を進めます。一方、IMO や NITS は修正は不要ですが、修正しない場合はその旨をコメントに返信してもらい、コメントを閉じてからマージする運用をしています。具体的な運用方法はチームによって異なるので、チームにあった方法で運用するのが良いと思います。

具体的な利用方法

実際にコードレビューをするとき、上記の 7 つの観点と修正の温度感をこのように交えた Prefix を付与してコメントします。

指摘例

MUST(Design): ドメインロジックが Controller クラスに実装されています。 domain 層の対象 package に新しくクラスを作成して実装を移してください。

7.4 最後に

このように、レビューコメントを分析し PDCA サイクルを回すことで、技術的課題へのアプローチ方法も見えてきます。また、コードレビューでオブジェクト指向が学べるといった恩恵を受けることできるので、ぜひチームに合ったレビューガイドラインを作成してみてはいかがでしょうか。



akkie76 <https://twitter.com/akkiee76>

第 8 章

Fortran によるオブジェクト指向プログラミング

暗黙の型宣言@implicit_none

8.1 はじめに

Fortran は、世界最古の高級言語である FORTRAN を祖先に持つプログラミング言語であり、スーパーコンピュータ等の大型計算機で高性能な計算を達成するために用いられています。Fortran は学習の容易さと高速な動作を両立しており、情報工学を専門としない研究者や技術者に愛用されています。

Fortran でオブジェクトを取り扱えるようになってから約 30 年が経過しています。しかし、Fortran が対象とする問題において、オブジェクト指向プログラミングが積極的に適用される事例は多くありません。その理由には色々あるのでしょうか、要因の一つに情報不足があることは確実です。

Fortran ユーザの興味は数値シミュレーションにあるわけですから、オブジェクト指向プログラミングを始めようとしたときに、「イヌもネコも哺乳類で…」と生物学的分類の話をされても、「オブジェクト同士がメッセージを交換しながら相互作用を…」という謎めいた記述^{*1}を見ても、全く理解できないというのが正直な所です。

本稿では、Fortran ユーザがより親しみを持っている問題の実装を例として、Fortran によるオブジェクト指向プログラミングを簡単に紹介していきます。

8.2 ベクトル空間

私たちが日常的に使っている加算

$$1 + 2 = 3$$

やスカラ乗算

^{*1} 数値シミュレーションにおいて、“オブジェクト”はその意味のとおり物体を表します。また、“メッセージ”という用語を用いた場合、異なるプロセス間でデータをやりとりする MPI (Message Passing Interface, 分散メモリ計算機環境で並列化を行う標準的なフレームワーク) が連想され、それが理解を妨げます。

$$2 \cdot 3 = 6$$

を一般化した概念としてベクトル空間があります。 n 次元ベクトル全体の集合を \mathbb{R}^n と書くことにします。 $\mathbf{a} \in \mathbb{R}^n$, $\mathbf{b} \in \mathbb{R}^n$ がそれぞれ縦ベクトル

$$\mathbf{a} = \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix}, \mathbf{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}$$

を表し、 $k \in \mathbb{R}$ とするとき、加法およびスカラ乗法は

$$\mathbf{a} + \mathbf{b} = \begin{pmatrix} a_1 + b_1 \\ \vdots \\ a_n + b_n \end{pmatrix}$$

$$k \cdot \mathbf{a} = \begin{pmatrix} k \cdot a_1 \\ \vdots \\ k \cdot a_n \end{pmatrix}$$

によって定義されます。集合 \mathbb{R}^n に加法とスカラ乗法が定義され、ある性質を満たすものをベクトル空間といいます。

8.3 Fortran による n 次元ベクトルの実装

さて、上記の $\mathbf{a} + \mathbf{b}$ に用いられる加算演算子 $+$ と、要素同士の和 $a_1 + b_1$ に用いられる加算演算子 $+$ は同じものでしょうか？ 同様に、 $k \cdot \mathbf{a}$ と $k \cdot a_1$ に用いられる乗算演算子 \cdot は同じものでしょうか？

私たちは、記号 $+, \cdot$ が同じであっても、使われる箇所によってその振る舞いが異なることを理解しています。この振る舞いを、Fortran で実装してみましょう。

リスト 8.1: Fortran によるベクトル型の定義

```
module type_vector
  use, intrinsic :: iso_fortran_env
  implicit none
  private

  type, public :: vector
    real(real32), public, allocatable :: val(:)
  contains
    procedure, public, pass(1) :: add_vec_vec
    procedure, public, pass(r) :: mul_r32_vec
```

```
generic :: operator(+) => add_vec_vec
generic :: operator(*) => mul_r32_vec
end type vector

contains
  function add_vec_vec(l, r) result(add)
    class(vector), intent(in) :: l ! + 演算子左のベクトル
    class(vector), intent(in) :: r ! + 演算子右のベクトル
    type(vector) :: add

    add%val = l%val + r%val
  end function add_vec_vec

  function mul_r32_vec(factor, r) result(mul)
    real(real32), intent(in) :: factor ! *演算子左のスカラー
    class(vector), intent(in) :: r ! *演算子右のベクトル
    type(vector) :: mul

    mul%val = factor*r%val
  end function mul_r32_vec
end module type_vector
```

module（モジュール）は、プログラム単位（宣言や定義を含み、個別にコンパイルされるスコープ）の一つです。contains文の上部で定数や変数の宣言を行い、contains文の下部で関数やサブルーチンの定義を行います。関数とサブルーチンを総称して、手続とよびます。

ベクトル型は、クラスに相当する利用者定義派生型として定義します。利用者定義派生型の定義はtype文を用いて行われます。派生型の中では成分（メンバ変数、フィールド）を宣言します。成分を宣言するだけであれば構造体と同様ですが、メンバ関数やメソッドに相当する手続は、派生型の中にあるcontains文の下部で定義します。ただし、Fortranの場合は、派生型の定義内で手続も定義するのではなく、同一モジュール内で定義済みの手続と派生型を結びつけることでそれを実現します。このことから、Fortranではメンバ関数やメソッドに相当するものを、type-bound procedure（型に束縛された手続、型束縛手続）とよびます。

一定の条件を満たした型束縛手続は、演算子を介して呼び出せます。それを設定しているのがgeneric :: operator() =>の箇所で、+演算子と*演算子を、加算とスカラー乗算を行う型束縛手続と関連付けています。

このように定義したベクトル型を使ったプログラム例は、リスト8.2のようになります。

リスト8.2: ベクトル型の使用例

```
program main
  use :: type_vector
  implicit none

  type(vector) :: a, b, c
  a = vector([1.0, 2.0])
  b = vector([2.0, 3.0])
  c = a + b
  print *, c%val !   3.00000000      5.00000000
```

```
c = 4.0*a
print *, c%val !    4.00000000      8.00000000
end program main
```

+ 演算子を用いたベクトル \mathbf{a} , \mathbf{b} 同士の加算を行った結果、その要素同士の加算が計算されていることがわかります。要素同士の加算は、型束縛手続の中で同じく + 演算子を用いて行われているわけですが、同じ + でもその役割が異なることを再現できています。

8.4 さらなる発展にむけて

Fortran ユーザにとってはなじみ深い例を用いて、Fortran におけるオブジェクト指向プログラミングの方法を簡単に説明しました。これだけでは、オブジェクト指向プログラミングの利点は理解しにくいと思いますが、手も足も出ない状態から一歩足を踏み出す助けになれば幸いです。ユーザが増えることで様々な試みがなされ、その中から有益な知見が生まれてくることを期待しています。

最後に、著者が行っている試みを一つ紹介しておきます。著者は、アルゴリズムの理解が容易で試行錯誤しやすい、非圧縮性流れの数値シミュレーション用フレームワークを作成しています。

非圧縮性流れの数値シミュレーションにおける最も基本的なアルゴリズムである Fractional Step 法は、以下の 3 本の式を順次計算していくことで、流れの時間変化を計算します。

$$\begin{aligned}\mathbf{u}^* &= \mathbf{u}^n + \Delta t [-(\mathbf{u}^n \cdot \nabla) \mathbf{u}^n + \nu \nabla^2 \mathbf{u}^n] \\ \nabla^2 p^{n+1} &= \frac{\rho}{\Delta t} \nabla \cdot \mathbf{u}^* \\ \mathbf{u}^{n+1} &= \mathbf{u}^* - \frac{\Delta t}{\rho} \nabla p^{n+1}\end{aligned}$$

\mathbf{u} は速度、 p は圧力、 ρ 、 ν は流体の物性値、 Δt は計算アルゴリズムの都合で現れるパラメータです。

著者が開発しているフレームワークを用いると、上記の計算アルゴリズムはリスト 8.3 のように実装されます。数式っぽく書けるところは数式っぽく、そうでないところは自然言語的に記述できるように設計しています。

リスト 8.3: 非圧縮性流れの計算を行うコード例

```
type(staggered_uniform_grid_2d_type), target :: grid !! 空間格子
type(vector_2d_type) :: u !! 速度
type(scalar_2d_type) :: p !! 圧力
type(vector_2d_type) :: u_aux !! 中間速度

type(vector_boundary_condition_type) :: BC_u !! 速度境界条件
type(scalar_boundary_condition_type) :: BC_p !! 圧力境界条件
```

```
! 初期化
u = .init.(u .on. grid)
p = .init.(p .on. grid)

! 中間速度の計算
u_aux = (u + dt*(-((u.dot.nabla)*u) + kvisc*.laplacian.u)) &
        .impose. BC_u !&

! 圧力 Poisson 方程式の求解
p = .inverse.(( &
    (laplacian(p).with.BC_p) == (dens/dt*.div.u_aux)) &
    .using. BiCGStab() &
    .until. below_criterion &
) !&

! 速度の計算
u = (u_aux - dt/dens*.grad.p) .impose. BC_u !&

! ファイル出力
call output(p .as. vtr .to. "p")
call output(u .as. vtr .to. "u")
```

このように書けたから何なんだ？ という意見もあるとは思います。一方で、オブジェクト指向プログラミングができると、実装できるコードの幅が広がります。また、数値計算以外のプログラミングに関する情報は、どうしてもオブジェクト指向プログラミングを前提にしているところもあるため、オブジェクト指向プログラミングに親しむと、それらの情報が理解しやすくなるという利点もあります。少しでも興味があれば取り組んでみてください。



暗黙の型宣言 https://twitter.com/implicit_none

Fortran や数値計算に関する技術書を作成しています。

オブジェクト指向で目指すものと目指さないもの

虎の穴ラボ 河野裕隆@hk_it7

虎の穴ラボでリードエンジニアとマネージメントをやっている河野です。

オブジェクト指向との向き合い方を目指すものと目指さないものという視点で書きます。100% 私見です。

9.1 オブジェクト指向とはなにか

まず前提として、オブジェクト指向について定義する必要があります。今回の話では単純に「必要な各要素をオブジェクト（モノ）として扱い、モデリングする手法」として進めます。一方で「オブジェクト間のメッセージングによりシステムを構築する手法」という視点は少し薄い内容となります。

9.1.1 設計/分析/プログラミング/言語

「オブジェクト指向」は適用するフェーズの接頭語として用いる例があります。具体的には「オブジェクト指向設計」や「オブジェクト指向プログラミング」です。

章のタイトルとして「設計/分析/プログラミング/言語」としましたが、「言語」だけ少し趣きが異なります。「オブジェクト指向言語」は、意味を開くと「オブジェクト指向で書くことができる言語」ということができるかと思います。言葉尻を捉えれば、「オブジェクト指向以外でも書こうと思えば書ける言語」ということです。つまり、Javaなどのオブジェクト指向言語で書くからオブジェクト指向とはならず、オブジェクト指向で書くからオブジェクト指向なわけです。書いて少し混乱してきましたね。

では「オブジェクト指向で書く」とはどういうことでしょうか？これは最初に決めた定義に則り、「必要な各要素をオブジェクト（モノ）として扱い、モデリングする手法を用いてプログラムを書く」ということになります。

9.1.2 オブジェクト指向で書くために必要なこと

本来であれば「オブジェクト指向開発方法論」を学び、「オブジェクト指向分析」、「オブジェクト指向設計」を経て「オブジェクト指向プログラミング」を実施する必要がありますが、そこまで

厳密に進めているところもあるかと思います。つまり、我々は雰囲気でオブジェクト指向をやっているような状態です（しっかり分析、設計を経て開発を行っている皆様、申し訳ありません）。

また時間をかけて設計するケースでも、オブジェクトに落とし込むことが難しいことがあります。例えばWebサービスの開発でUserControllerを作ったり、OrderServiceを作ったり、あるいはDateUtilを作ったりしてしまう場合です。

ControllerやUtilはモノではなく、観測できません。にも関わらず厳密なオブジェクト指向で開発していると主張するのは一種の「雰囲気でオブジェクト指向やっている状態」だと思います（前述のモジュールがオブジェクト指向から外れていることを認識して進めている場合、この限りではありません）。

この雰囲気でオブジェクト指向をやっている状態を雰囲気OOと名付け、その目的を明確にしていきます。

余談：DDDと雰囲気OO

ここ数年で技術イベント等でもよく聞かれるようになったドメイン駆動設計。

概念としてはオブジェクト指向のために限定された設計手法ではありませんが、オブジェクト指向と組み合わせることが多い考え方です。何をもってオブジェクトとするかを「ドメインエキスパート」らによる「ユビキタス言語」を介してモデリングし、設計、開発していくさまは、まさしくオブジェクト指向分析/設計そのものと言えます。

オブジェクトを考えるためにはオブジェクトを観測する主体が必要であり、量子論で交わされた議論をなぞると「月を見ていないとき、月は存在しない」のです。

一方でサービスの利用などオブジェクトと呼び得ないものも登場し、厳密なオブジェクト指向とは乖離します。

DDDが注目されるようになった背景として、オブジェクト指向の分析/設計がやや軽視されてきたカウンターのようなものがあるのではないかと個人的には考えています。前段の言葉を用いると「雰囲気OOとDDDは相容れない」ということです。

9.2 雰囲気OOを用いる目的、目指す場所

まず「お断り」として、雰囲気OOは悪ではありませんし、したがって本記事で処方箋を出すつもりもありません。ではなぜ雰囲気OOは「悪ではない」と言えるのでしょうか？

結論から言うと厳密なオブジェクト指向は難しそうです。それに対して、雰囲気OOはオブジェクト指向開発方法論のエッセンスを適宜活用して、組み合わせて開発していきます。

雰囲気OOで良ければ思想を学ぶ必要もなければ、Utilの代替オブジェクトの命名に悩むこともありません。

各オブジェクトは「ある程度の単一な責務を持つ」、「責務に対してエキスパートとする」、「プロパティ等の可視性は最低限に保つ」あたりを満たしていれば、一定の保守性は担保されます。またどの言語であっても、このくらいの「オブジェクトを扱う機能」は用意されています。

大きいシステムになればなるほど、すべての処理を追うことが困難になります。そのようなとき

に「ざっくり内容を把握する」には雰囲気 OO はぴったりです。各オブジェクトはそれぞれのエキスパートであるため、メソッドとそのシグネチャで何をしているのか予測できるはずです。

逆に言えば、コードを正確に追わないと「ざっくり内容を把握」できなくなった場合は設計に立ち返る必要があります。私がコードレビューするときは「オブジェクト指向的にどうか」という具体性のない指標よりも「ざっくり内容を把握できるか」という点を意識しています。

9.2.1 プロジェクト特性と雰囲気 OO

一方でプロジェクト特性に合わせてオブジェクト指向に対する濃度、粒度を調節して、設計/開発を進める必要があると感じています。具体的には GUI プログラミングでは濃度を若干濃くして、Web（特にサーバーサイド）プログラミングでは少し薄くするようなイメージです。

これは GUI プログラミングは「モノ」を扱うのでオブジェクトの概念が持ち込みやすいためです。オブジェクトが何かを描写するのでカプセル化も意識しやすいかと思います。

画一的にオブジェクト指向を適用すると窮屈になってしまいますが、濃度を調節してオブジェクト指向の考え方を部分的に適用していくことで、開発スピードの維持にもつながると考えます。

9.3 まとめ

オブジェクト指向は難しいと言われますが、オブジェクトを外から見たときに「ざっくりわかる」ようになっていれば問題ないという話でした。

一部エッセンス的に（保身も兼ねて）難しい言葉も使っていますが、言いたかったことは前の 1 文の通りです。雰囲気 OO を通して、コスト、バリュー、クオリティのバランスを取った開発ができると良いな、願っています。



河野裕隆 https://twitter.com/hk_it7 <https://github.com/h-kono-it>

虎の穴ラボで Web サービスの開発、マネジメントを行っています。Java、Ruby、TypeScript/JavaScript、Vue.js とフロントもバックも書きますし、AWS 等をいじったりもします。何でも屋さんです。

第 10 章

PHP 製 OSS に見るデザインパターンの具体例 3 選 + α

hirokinoue

オブジェクト指向、デザインパターン…概念はなんとなくわかった気がするんだけど、どのように使えばいいのだろうか…？ そんな感覚を持ったことが誰しも一度はあるのではないしょうか？ そんな時、具体例を参照することで理解を深められると考えています。そこで、PHP 製の OSS に見られる生きた事例 3 つ + α を紹介したいと思います。

ここで紹介するデザインパターンの実装は、書籍やネットで紹介されているものと完全一致するとは限りませんが、それぞれのパターンが有るべき特徴を持っていると考えています。紙面の都合で、説明に関係のある部分のみを抜粋します。コードの全量は GitHub を参照して下さい。

10.1 ①ストラテジパターン@PHPUnit

以下の特徴を備えたストラテジパターンの一種と捉えて紹介します。

- 同種の振る舞い（同じシグネチャ）を持ち、詳細の異なる実装がいくつかある。
- 実行時のコンテクストに応じてその実装が選択される。

PHPUnit はユニットテストのライブラリです。以下のようにしてテストを記述します。この後 assertEquals メソッド^{*1}を深掘ってゆきます。

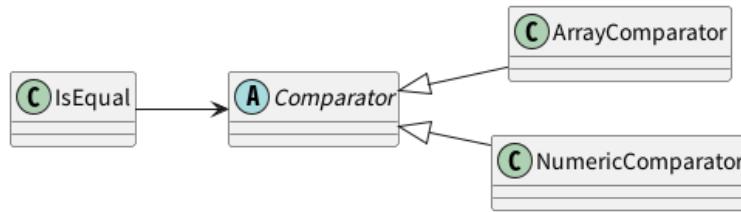
クライアントコード例

```
final class EqualsTest extends TestCase
{
    public function testFailure(): void
    {
        // $this->assertEquals($expected, $actual);
        $this->assertEquals(1, 0);
    }
}
```

ユニットテストにおいて期待値と実行結果が一致することを確認する assertEquals メソッドは

^{*1} <https://docs.phpunit.de/en/9.6/assertions.html#assertequals>

内部で IsEqual クラスを使用しています。ここでストラテジパターンが活用されています。初めにクラスの関係を確認します。



それでは見てゆきます。IsEqual と、そこで使用される Comparator に注目します。Comparator には実装がいくつかあります。IsEqual は実行時のコンテキスト（テストにおける期待値と実行結果の組み合わせ）に応じて実装を選択します。Comparator からのぞいてみます。

Comparator^{*2}

```

abstract public function accepts(/* 略 */): bool;
abstract public function assertEquals(/* 略 */): void;
  
```

Comparator は accepts メソッドや assertEquals を持つ抽象クラスで、その実装は様々あります。例えば配列を比較する ArrayComparator、数値を比較する NumericComparator などです。続いて IsEqual の evaluate メソッドを見てみます。

IsEqual^{*3}

```

public function evaluate(mixed $other, string $description = '', bool $returnResult = false): ?bool
{
    // 略

    // ComparatorFactory は Factory (下記) のエイリアス
    // Comparator の実装の配列を生成する
    $comparatorFactory = ComparatorFactory::getInstance();

    try {
        // Comparator の配列から実装を選択する
        $comparator = $comparatorFactory->getComparatorFor(
            $this->value,
            $other,
        );

        // 選択した実装に応じた assertEquals メソッドが実行され
        // 期待値と実行結果が比較される
        $comparator->assertEquals(
            $this->value,
            $other
        );
    } catch (Exception $e) {
        if ($returnResult) {
            return false;
        }
        throw $e;
    }
}
  
```

^{*2} <https://github.com/sebastianbergmann/comparator/blob/146dc7bb46c39d8d42bb0fd8cfdec588f0814f66/src/Comparator.php>

```
        $other,
);
```

Factory^{*4}

```
// ComparatorFactory::getInstance() からここに辿り着く
private function registerDefaultComparators(): void
{
    // 様々な Comparator の実装を配列に詰めている
    $this->registerDefaultComparator(new MockObjectComparator);
    $this->registerDefaultComparator(new DateTimeComparator);
    // 略

    public function getComparatorFor(mixed $expected, mixed $actual): Comparator
    {
        // $comparator は Comparator 型
        // コンテクストに応じて実装を選択する
        foreach ($this->defaultComparators as $comparator) {
            if ($comparator->accepts($expected, $actual)) {
                return $comparator;
            }
        }
    }
}
```

10.2 ②オブザーバパターン@Laravel

以下の特徴を備えたオブザーバパターンの一種と捉えて紹介します。

- ・ サブジェクト（観察される者）とオブザーバ（観察する者）がいる。
- ・ サブジェクトはオブザーバを保持する（サブジェクトにオブザーバを登録/解除できる）。
- ・ サブジェクトの状態変化（サブジェクトからの通知）を察知して、オブザーバの処理を行う。

Laravel は Web アプリケーションフレームワークです。 Laravel において、イベントの生成・発行からイベントリスナの処理を行う一連の流れがオブザーバパターンを形成しています。

フレームワークが提供する Dispatcher はサブジェクトの役割を果たし、イベントリスナがオブザーバの役割を果たします。 Web アプリケーションはこの仕組みを利用するクライアントになります。サブジェクトはオブザーバを保持しますので Dispatcher にイベントリスナを登録する必要があります。また、イベントの種類に応じて通知を受けるべきリスナが異なりますので、イベントとリスナの関連づけも行います。

まとめると、クライアントがイベントやサブジェクトたる Dispatcher を生成してオブザーバたるイベントリスナに通知させ、イベントリスナはこれを受けて処理を行います。この仕組みにより、例えば、ユーザー登録すると”ユーザー登録済みイベント”を発行し、これを受け取ってメールを

^{*3} <https://github.com/sebastianbergmann/phpunit/blob/57e07eef9de8e016be8da0f53a35d5df1fa72ebd/src/Framework/Co>

^{*4} <https://github.com/sebastianbergmann/comparator/blob/146dc7bb46c39d8d42bb0fd8cfdec588f0814f66/src/Factory.ph>

送信するといった振る舞いを実現できます。

events/Dispatcher^{*5}

```
// イベントとイベントリスナを登録する
public function listen($events, $listener = null) {}

// イベントを発行する
public function dispatch($event, $payload = [], $halt = false)
{
    // 略

    if ($isEventObject &&
        $payload[0] instanceof ShouldDispatchAfterCommit &&
        ! is_null($transactions = $this->resolveTransactionManager())) {
        $transactions->addCallback(
            // リスナを呼び出す
            fn () => $this->invokeListeners($event, $payload, $halt)
        );
    }

    return null;
}

// リスナを呼び出す
return $this->invokeListeners($event, $payload, $halt);
}

protected function invokeListeners($event, $payload, $halt = false)
{
    // 略

    foreach ($this->getListeners($event) as $listener) {
        // リスナの handle() が実行される
        $response = $listener($event, $payload);
    }
}
```

リスナ^{*6}

```
class FooListener
{
    public function __construct() {}

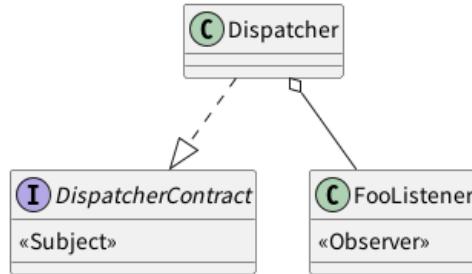
    // Dispatcher の invokeListeners() からここに辿り着く
    public function handle(FooEvent $event): void
    {
        // サブジェクトからの通知（イベント）を受けて行う処理
    }
}
```

※ Laravel では様々なやり方でイベントとリスナの実装が可能ですがほんの一部だけ紹介しました。

^{*5} <https://github.com/illuminate/events/blob/22d6718c5859f2290a8a8ac32b24ed7a4bc1a93f/Dispatcher.php>

^{*6} <https://laravel.com/docs/10.x/events#define-listeners>

クラスの関係は以下のようになっています。



10.3 ③ビジターパターン@PHP-Parser

以下の特徴を備えたビジターパターンの一種と捉えて紹介します。

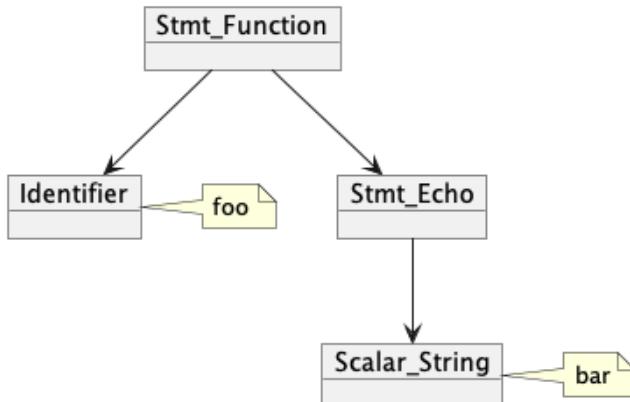
- 構造と操作を分離している。
- 操作の具象を定義することで、構造に対する操作を追加できる。また逆に操作を除くこともできる。

観察対象である PHP-Parser の説明から始めます。PHP-Parser は PHP の静的解析に利用されるライブラリです。PHP のコードを読み込んでノードと呼ばれるオブジェクトの連なりに変換します。これを AST と呼びますがここでは説明を割愛します。例えば、このコードと下記の AST が対応づきます。下図の四角形がノードです。

解析対象コードサンプル

```
<?php
function foo() {
    echo 'bar';
}
```

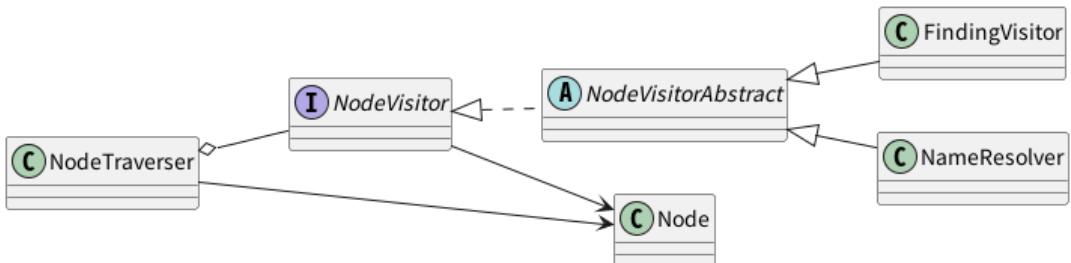
AST



PHP-ParserにおいてTraverserとVisitorによりビズターパターンが成立しています。Traverserが構造側の処理を担い、Visitorが操作側の処理を担います。

Traverserは一つ一つノードを読み込みます。また、TraverserはVisitorを保持しており、Visitorにノードを処理させます。つまりすべてのノードはTraverserにより辿られ、Visitorにより処理されます。Visitorには様々な実装があります。例えば、条件を満たすノードを見つけるVisitorやノードの名前を特定するVisitorがあります。構造と操作を分離するというのは、ノードを辿るという処理とノードに対して行う処理を別々に構築できるということです。

まずクラスの関係を確認します。



それでは実装を見てみます。

NodeTraverser^{*7}

```

class NodeTraverser implements NodeTraverserInterface {
    public function __construct(NodeVisitor ...$visitors) {
        // Traverser は NodeVisitor を保持している
        $this->visitors = $visitors;
    }

    public function traverse(array $nodes): array {
        // 略
    }
}
  
```

```

        foreach ($this->visitors as $visitor) {
            // Traverser は NodeVistor を使用する
            if (null !== $return = $visitor->beforeTraverse($nodes)) {
                $nodes = $return;
            }
        }
        // 略

    protected function traverseNode(Node $node): void {
        // 略

        foreach ($this->visitors as $visitorIndex => $visitor) {
            // Traverser は NodeVistor を使用する
            $return = $visitor->enterNode($subNode);
    }
}

```

NodeVisitor のシグネチャは以下の通りです。

NodeVisitor^{*8}

```

interface NodeVisitor {
    public function beforeTraverse(array $nodes);
    public function enterNode(Node $node);
    public function leaveNode(Node $node);
    public function afterTraverse(array $nodes);
}

```

NodeVisitor には FindingVisitor や NameResolver など様々な実装があります。Visitor の具象クラスは NodeVisitor を実装する抽象クラスである NodeVisitorAbstract を拡張して作ります。FindingVisitor を見てみます。beforeTraverse や enterNode を実装していることがわかります。NameResolver も見ていただくと、これと異なる実装を確認していただけます。

FindingVisitor^{*9}

```

class FindingVisitor extends NodeVisitorAbstract {
    public function beforeTraverse(array $nodes): ?array {
        $this->foundNodes = [];

        return null;
    }

    public function enterNode(Node $node) {
        $filterCallback = $this->filterCallback;
        if ($filterCallback($node)) {
            $this->foundNodes[] = $node;
        }
    }
}

```

^{*7} <https://github.com/nikic/PHP-Parser/blob/ce019e9ad711e31ee87c2c4c72e538b5240970c3/lib/PhpParser/NodeTraverser.php>

^{*8} <https://github.com/nikic/PHP-Parser/blob/ce019e9ad711e31ee87c2c4c72e538b5240970c3/lib/PhpParser/NodeVisitor.php>

```

        return null;
    }
}

```

NodeTraverser は具象ではなく NodeVisitor に依存するので、NodeVisitor を実装しているあらゆる具象を利用できます。ノードに対して行いたい処理を個別の具象クラスとして定義し、Traverser に渡すことで処理を追加できるということです。不要な処理は Traverser から取り除くこともできます。

AST に限らず、順々に読み込んだものに対して処理を行うケース、そしてその処理を着脱したいケースでビジターパターンが適用できます。

10.4 α. インタフェース、多態性の活用

オブジェクト指向を学ぶ過程において、インターフェースの使い道というのはひっかかるポイントの一つではないでしょうか。上述のデザインパターンの中に示唆があります。

ビジターパターンの NodeTraverser を見て下さい。NodeTraverser は Visitor の具象ではなく抽象 (NodeVisitor) に依存しています。beforeTraverse(), enterNode(), leaveNode(), afterTraverse() があれば処理を行えるようになっており、NodeVisitor のシグネチャを持つその実装は要件を満たします。これにより、ノードに対して行いたいことに応じて処理を実装し、用途に応じて処理を着脱できます。

ストラテジパターンの Factory クラス（この事例ではインターフェースではなく抽象クラスへの依存ですが）にも同じような特長が見られます。これらの例の中で、仮に NodeTraverser や Comparator の具象に依存しているとすると、実装が差し替えられず柔軟性を失います。また依存を明示しない（型を示さない）と改修する際にバグを混入させやすくなります。

10.5 おわりに

以上、デザインパターン、インターフェースの使い道について実例を見てきました。あなたが見つけたデザインパターン、オブジェクト指向設計技術の生きた事例を紹介していただけるうれしいです。



hirokinoue <https://github.com/hirokinoue>

株式会社ホワイトプラスで EM をしています。かつては COBOL を少々、現在は PHP を少々。

*⁹ <https://github.com/nikic/PHP-Parser/blob/ce019e9ad711e31ee87c2c4c72e538b5240970c3/lib/PhpParser/NodeVisitor.php>

静的解析と仲良くなりたい。

モデリングを育てる前の種蒔きと土作り

浅野正貴@mackey0225

今回『チームでモデリングを育てるうえで考えたこと・気づいたこと』というタイトルで登壇・発表させていただきます。皆様、何卒よろしくお願ひいたします。

11.1 これはなんの記事？

今回の話の前日譚として、きっかけや前提となる私たちのチームでの取り組みを紹介する補足記事になります。喻えるならば、**モデリングを育てる前の準備として必要となる「種蒔き」と「土作り」**について書いています。

なので、オブジェクト指向はこの記事でも一切出てきません！ ご容赦ください！！

今回、この記事で紹介する取り組みは、以下の 2 つです。

- 種蒔き：輪読会
- 土作り：ADR(Architecture Decision Records)

それについて、説明します。

11.2 種蒔き：輪読会

私たちの開発組織では、有志のメンバーで輪読会を行っています。

一般的な輪読会と運用はさほど変わらないと思います。テーマや読みたい書籍をメンバー間で出し合い、メンバーを募り、グループを作ります。各グループ内で、週次で各回の決められた範囲を事前に読んでおき、感想や疑問点、自分たちのプロダクトへどう活かすかといった議論をします。

過去には、以下のような書籍を読んでいきました。

- Clean Architecture 達人に学ぶソフトウェアの構造と設計
- SCRUM BOOT CAMP THE BOOK
- 良いコード／悪いコードで学ぶ設計入門 一保守しやすい成長し続けるコードの書き方
- 現場で役立つシステム設計の原則 などなど

その中でも、今回、モデリングを考えるきっかけとなったのが、以下の2冊です。

- ドメイン駆動設計 モデリング/実装ガイド
- ドメイン駆動設計 サンプルコード&FAQ

この2冊を読み終わった後に、メンバー間でどう活かすか議論しました。その上で上がった課題で、「ユビキタス言語の整備やモデリングの運用がうまくできていない」というものがありました。これが、今回の登壇のきっかけになります。

輪読会という取り組みを通じて、現在抱えている課題を照らすことができただけでなく、課題に対する温度感を共有することができました。ここも喻えるならば、まだ芽が出ていない種の存在を再認識し、まずは種を蒔こうというムードにできました。

11.3 土作り：ADR(Architecture Decision Records)

前節に述べた通り、この時点では、種を蒔くことは決まったが、どこに、どの様に蒔くのかはまだ決まっていませんでした。

無策に種を蒔けばいいものではありません。砂漠やアスファルトの上に蒔いてしまうと、発芽どころか、そのまま腐ってしまう可能性があります。そのために、まずおこなったのが、この節で説明する ADR を用いた土作りをしました。

一般的に ADR はシステムやプロダクトにおけるアーキテクチャー上の設計判断を記録するため用いられますが、私たちはアーキテクチャーのみに限定せず、プロダクト開発における意思決定が伴うものに対して記録するようにしています。

その理由や背景はいくつかあります。例えば、以下のようなことに対応していくためです。

- 意思決定の記録を形式化することで How だけでなく Why も記録するようになる
- 意思決定者が何かしらの理由でいなくなつたとしても判断ができるようになる
- 決定時から、時間が経つと環境や情勢の変化があり、その決定が時代にフィットしているか判断できるようになる

今回、「ユビキタス言語の整備やモデリングの運用がうまくできていない」という課題に対して、この ADR を用いて、何で管理するのか、どんなものを準備すべきか、どんなタイミングで更新するのかを決定してきました。(決定事項の内容については、当日の発表でお話できればと思っております。)

このチームで決めていった定義(つまり、土壌)の上で、チームで育てていっています。

11.4 さいごに

一人でサービスを開発・保守するのであれば、自分の裁量ですすめられますが、現実問題、そのようなことは稀で、基本的には複数人で開発・保守することが一般的です。

複数人で継続的に運用されるものを考えるためには、目線合わせは必要です。私たちは、本記事

で紹介した取り組みや当日の登壇でお話する内容を常に考え続ける必要があると思います。



浅野正貴 @mackey0225 <https://x.com/mackey0225>

BABY JOB 株式会社 Java エンジニア

大阪からの遠征組なので、一人でも多く知り合いを作って帰りたいです！

第 12 章

OOUI を関数で考える

中川孟

はじめまして、株式会社ゆめみ新卒フロントエンドエンジニアの中川孟です。

突然ですがみなさん、OOUI を実践していらっしゃるでしょうか。デザイナーの皆さんには OOUI の実践者が少くないと思いますが、エンジニアで OOUI を活用している方はまだあまりいないかもしれません。

私は、OOUI がエンジニアにとっても優れたメンタルモデルを提供すると気付き、日々の業務で OOUI を実践しています。OOUI はオブジェクトという「共通認識」を通じてデザイナーとエンジニアがより良く協業するための優れた手法です。

そして、デザイナーとエンジニアのより良い協業を模索したいという思いから、この半年、同期のデザイナーとともに OOUI 勉強会を継続して開催してきました。「OOUI を関数で考える」というテーマで、デザイナー向けに、React, TypeScript を用いて OOUI をプログラムで記述してみようという内容の勉強会です。

最近、その勉強会の効果を少しづつ実感しています。折角の機会ですので、本稿ではその勉強会の中でも特に肝心な内容を紹介したいと思います。

12.1 UI を関数で考える

`UI = f(state)` や `UI = f(data)` などで示される通り、多くのフロントエンドエンジニアは「UI とは関数である」というメンタルモデルを共有しています。

そもそも「関数」が多くのデザイナーにとって慣れないものであるようなのですが、ここでは簡単に「何かを入れたら何かが出るもの」をイメージしてください。お金を入れたらカプセルが出てくるガチャガチャなんかは関数のいい例でしょう。

関数で重要なのが「引数」と「返り値」という概念です。簡単にいえば、引数は「関数に入れるもの」、返り値は「関数から出てくるもの」です。先の例で言えば、お金が引数、カプセルが返り値にあたります。

これを踏まえると、`UI = f(data)` は「`data` を引数にとる関数 `f` の返り値として `UI` が存在する」といったような意味になります。`UI` が関数で表現されるのです。

素朴な例を挙げるとすれば、下記のような Greet 関数コンポーネントはいかがでしょう。`name` を引数に取って、`p` タグに囲まれた”Hello, ○○”という文字列を返します。`name` に `hajime` を入れると、`<p>Hello, hajime</p>` が返ってくることになります。

```
const Greet = ({ name }) => {
  return <p>Hello, {name}</p>;
};
```

この勉強会のゴールの1つは、OOUI を「オブジェクトを受け取って UI を返す関数」として理解すること、すなわち $\text{OOUI} = f(\text{object})$ というメンタルモデルを獲得することです。そしてそのために、オブジェクトを引数に取る関数コンポーネントを記述することに挑戦します。

ところで、OOUI の「オブジェクト」とは別にプログラミング言語にも「オブジェクト/object」という機能が存在します。少し紛らわしいので、これ以降「OOUI の文脈でのオブジェクト/object」は「モデル/model」と表現します。

12.2 モデルを TypeScript の型で記述する

$\text{OOUI} = f(\text{model})$ をプログラムで表現していく上で、まず重要なのは「それがどのようなモデルなのか」ということです。

フロントエンドの現場では、しばしばそれを TypeScript の「型/type」という機能で表現します。例えば Todo モデルは下記のように記述されます。

```
type Todo = {
  id: string;
  title: string;
  dueTo: Date;
  isDone: boolean;
};
```

Todo がモデル、その中身の title や isDone がプロパティです。title や isDone の注釈に使われている string や boolean などはそのプロパティのデータ型を表現しています。

勉強会では「ある人に関する情報」が詰まった Profile モデルを作成することにしました。「プロフィールとしてどのような情報があったらいいか」というモデリングから始めて、それがどのようなプロパティ・データ型で表現できるかを考えていきます。

実際に、最初に出来上がったモデルがこちらです。名前やアイコン、お気に入り情報が詰まっています。

```
type Profile = {
  name: string;
  iconUrl: string;
  favorites: {
    foods: string[];
    colors: string[];
  }
};
```

```
        artists: string[];
    };
};

};
```

このモデルに沿ったデータを記述してみると、例えばこのようになります。

```
const profileA: Profile = {
  name: "hajime",
  iconUrl: "https://example.com/hajime.png",
  favorites: {
    foods: ["apple", "banana"],
    colors: ["blue", "green"],
    artists: ["aiko", "spitz"],
  },
};
```

12.3 モデルに関心のある関数を書く

OOUI = f(model)において、モデルは関数の引数に登場します。そして TypeScript を使う上では、関数がどのような引数を受け取るのかもまた、ひとつの型として表現されます。

試しに、ひとつの Profile を表現する ProfileCard コンポーネントを書いてみましょう。

```
interface Props {
  profile: Profile;
}

const ProfileCard: FC<Props> = ({ profile }) => {
  return (
    ...
  );
};
```

これまでよりもぐっと複雑なコードに見えますね。

下の const ProfileCard の方は、: FC<Props> の部分を除けば Greet 関数と構造が同じです。 interface Props の方は、type Todo の書き方と大体同じです。

上記のコードの概要を説明すると、「ProfileCard は引数 profile を受け取る関数コンポーネントです。そして引数 profile は Profile 型です」といった意味になります。

この場ではコードの書き方の話はさておき、少し遠目で見てイメージを喚起することに意識を向けてみてください。

- interface Props に続く { profile: Profile } の部分が、コンポーネント記述の中の FC<Props> に続く { profile } と対応していること

- FC<Props>という ProfileCard コンポーネントに対する型注釈が OOU = f(model) の右辺のイメージとぴったり重なること (FC が f、Props が model にあたります)

などがわかると、「OOUI を関数で考える」のイメージがなんとなく伝わってきます（はい、この説明だけでは中々わかりません、すみません）。

ProfileCard はモデルの単体（あるいは詳細、シングル）に関心のあるコンポーネントですが、モデルのリストに関心のあるコンポーネントももちろん考えられます。その際、インターフェイスにも違いが出ます。

```
interface Props {
  profiles: Profile[];
}

export const ProfileList: FC<Props> = ({ profiles }) => {
  return (
    ...
  );
}
```

要するにモデルの配列を引数に取ることですが（引数名も複数形にしました）、こう見ると interface Props にコンポーネント（関数）の関心が現れることがはっきりわかります。

12.4 モデルの具体としてのデータと UI

さて、関数コンポーネントを定義しただけでは UI が立ち上がりません。UI 表示させたい場所で適当な引数を渡して関数を呼び出す必要があります。ガチャガチャにお金をいれてハンドルを回すのと同じことです。

React では関数コンポーネントの呼び出しは

```
<ProfileCard />
```

のような素朴なタグ表現で完結しますが、ProfileCard には引数が必要なのでした。それを呼び出しのタイミングで渡してあげます。次のようになります。

```
<ProfileCard profile=?>
```

さて、?に入るるのはどのようなデータでしょうか。その説明が先程記述した

```
interface Props {  
  profile: Profile;  
}
```

にあたります。「引数 profile は Profile 型のデータです」ということです。

この Profile 型のデータの一例として、「モデルを TypeScript の型で記述する」の章で書いた profileA が挙げられます。

```
const profileA: Profile = {  
  name: "hajime",  
  iconUrl: "https://example.com/hajime.png",  
  favorites: {  
    foods: ["apple", "banana"],  
    colors: ["blue", "green"],  
    artists: ["aiko", "spitz"],  
  },  
};
```

ということで、この profileA を ProfileCard に渡してあげれば、そのデータに沿った UI が表示されることになります。

```
<ProfileCard profile={profileA}>
```

具体的にどのような UI が表示されるんだ？ ということについては、実は先程「…」で省略していた ProfileCard 関数の返り値に記述されています。例えば以下のようになりますが、詳細な説明はお近くのフロントエンドエンジニアに聞いてみてください。

```
const ProfileCard: FC<Props> = ({ profile }) => {  
  return (  
    <div className="max-w-2xl py-8 flex items-center justify-center rounded-2xl  
          border border-gray-300 bg-white">  
      <div className="max-w-xl flex flex-col gap-6">  
        <h1 className="text-xl text-gray-500">My Profile</h1>  
  
        <div className="flex items-center justify-between">  
          <div>  
            <h2 className="text-4xl font-bold">{profile.name}</h2>  
            <ul>  
              <li className="text-base">{profile.role}</li>  
              <li className="text-base">{profile.catchcopy}</li>  
            </ul>  
          </div>  
          <img />  
        </div>  
      </div>  
    </div>  
  );  
}
```

```

        className="shadow-xl"
        width={136}
        src={profile.iconUrl}
        alt="プロフィール写真"
    />
</div>

<ul className="flex gap-10">
    <li className="w-40 h-auto bg-blue-100 rounded-lg flex flex-col items-start text-sm p-4">
        {profile.favorites.foods.map((food) => (
            <span>{food}です。</span>
        )))
    </li>
    <li className="w-40 h-auto bg-yellow-100 rounded-lg flex flex-col items-start text-sm p-4">
        {profile.favorites.colors.map((color) => (
            <span>{color}</span>
        )))
    </li>
    <li className="w-40 h-auto bg-gray-200 rounded-lg flex flex-col items-start text-sm p-4">
        {profile.favorites.artists.map((artist) => (
            <span>{artist}</span>
        )))
    </li>
</ul>
</div>
</div>
);
};

```

ちなみに上記は勉強会でデザイナー達が実際に書いたコードですが、彼らは勉強会が始まった当初、HTML/CSS も書けませんでした。ここまで書くのにはたくさんの知識が必要で、改めて彼らに敬意を評します。

12.5 おわりに

いかがだったでしょうか。今回は紙幅の都合で説明をかなり端折っており、デザイナーに向けて書いたものの、プログラミングに慣れない方にとってはいささか伝わりづらい内容になってしまったかもしれません。

エンジニアがデザイナーにとって伝わりづらい説明をしていて、良い協業ができるわけがありません。ですので本稿のわかりづらさはいたく反省するところではありますが、これは決して意図的にわかりづらくしているわけではないのです。両者の間にはコンテキストの広い溝があり、ただ簡単には飛び越えられないということなのです。これは一朝一夕で埋められるものではなく、たゆまぬ歩み寄りが必要になります。そして OOUI は、デザイナーとエンジニアの間で素敵な架け橋となってくれるのです。

一緒に勉強会を作ってくれたデザイナー2人がその経験を素敵なnoteに残しているので、そちらもぜひ御覧ください。

UIデザイナー1年生がコードを書いてみた！ -コードからGUIを見てみる- | こばやし めい
か https://note.com/meika_1123/n/nc61184d6588f
デザイナーとエンジニアのタンゴ | osamu https://note.com/onaka_pocopoco/n/nfb6ee80f12d7
たった3人で始めた勉強会ですが、徐々に和を広げていけたらなと考えています。OOUIを用いたデザイナーとエンジニアの協業に興味のある皆さん、ぜひお声がけください。



中川孟 @yumemi_hajimism https://twitter.com/yumemi_hajimism

株式会社ゆめみでフロントエンドエンジニアをしています。デザインもちょっとやります。

デザインパターンを学ぶ

虎の穴ラボ 浅見

はじめまして、虎の穴ラボでエンジニアをしている浅見です。

本記事では、Object-Oriented Conference 2024 に寄せて、デザインパターンについて私見を述べていこうと思います。

13.1 はじめに

Java などでオブジェクト指向を学習しようと関連記事を漁って行くと、デザインパターンという語にたどり着くと思います。

そこでデザインパターンについて調べようすると、○○パターン、△△パターンといった具体例が多数出てきます。これらをひとつずつ追っていくと、そのたびにソースコードや UML を読解することになり、

私が初学者だったころは、これらの理解に時間と労力がかかりマイナチ全体感をつかめない状況が続いた印象があります。そこで初学者だったころの自分自身にデザインパターンについて概要を説明するとなったらこんな感じかなという話を書いてみようと思います。

13.2 デザインパターンとは

まずデザインパターンとは、よく利用されるクラス設計の形に名前を付けたものです。

特に「オブジェクト指向における再利用のためのデザインパターン」という本が有名で、デザインパターンといったらこの本で紹介された 23 パターンを指すことが多いです。著者たちの通称から GoF デザインパターンと呼ばれています。昨今はライブラリやフレームワークが充実しているため、自分でこれらのデザインパターンを用いたクラスを実装することは少ない印象ですが、デザインパターンは複雑なクラス設計を一言で言い表せるため、エンジニア同士の会話で意思の疎通がスムーズになったり、既存のライブラリの使い方を名前から容易に想像できるようになるため、学習しておいて損はないです。

13.2.1 デザインパターンへの批判など

よくあるデザインパターンへの批判を挙げてみると、「紹介された 23 パターンは何か明確な基準のもと体系立てて列挙しているわけではなく、重要なパターンとそうでないパターンがある」、「Iterator パターンは言語仕様に組み込まれているため、いまさら学習する意味はない」「Singleton

「パターンは不要である」などなどがあります。というのも、そもそも GoF 本が書かれたのは 1995 年で、Java がリリースされる前でした。そんな時代に提唱されたものなので、現代のプログラミングの現場の実情とはかけ離れたパターンがあつたり、昨今ではほぼ見なくなつたパターンもあるようです。実際、著者への 2009 年のインタビューで、「Singleton は廃止しても良い」とか、「パターンの分類を今ならこうする~」みたいな話が出ていたようです。こういった時代背景的な話は、そのときの言語仕様やパラダイム、流行しているフレームワークなどによって今後も変わるものがあるので、今は何がベストプラクティスなのかということは意識して調べながら学習するのがよさそうです。

ちなみに、Iterator パターンが学習する価値が無いのか？と聞かれたら、学ぶ価値アリと答えます。たしかに、今日の Java の世界では標準ライブラリに Iterator インターフェースが定義されているほか、拡張 for 文などで言語仕様レベルで Iterator の呼び出し部分が組み込まれているため、巷でみられる Iterator パターンのサンプルコード全体を自分で実装することはまずないです。一方で、Iterator パターンに理解があれば、拡張 for 文に独自のクラスを渡したり、Stream#iterate メソッドを使うときにすぐにイメージが湧いたり…といった恩恵を受けることでしょう。

13.2.2 他のデザインパターン

ちなみに、GoF が最初に提唱した 23 パターン以外にも、世の中には色々なデザインパターンがあります。

たとえば Null オブジェクトパターンなんものが典型的です。また、例えば Builder パターンについても、GoF で提唱された形、EffectiveJava で提唱された形など、目的は同じでも実装方法にバリエーションがあるケースもあります。

13.3 デザインパターンを学習する上で大事なこと

私が初学者の時、各パターンのサンプルコードや UML は理解したが、使い時がよくわからず結局身につかない…という状況に陥った記憶があります。

これはおそらく、当時私がデザインパターンを「これさえやっておけば良い最強の設計！」という銀の弾丸のようなイメージを持っていたからだと思います。実際には、まず始めに解決したい問題があり、それを解決する手法としてパターンが出てくるので、前提となる問題を抑えておくことが大事です。デザインパターンを学習すると、早くパターンを使ってみたい！となりがちですが、前提の問題部分をすっ飛ばして無理やりコードを書き始めると、必要以上に複雑なコードを生み出す結果に陥りそうです。(そして YAGNI と突っ込まれる…)

GoF デザインパターンは本のタイトルにもあるように「再利用性」を課題にしており、特に委譲・コンポジションを駆使し、ポリモーフィズムによってコードの再利用を実現するようなパターンが多い印象です。「再利用性」が問題意識として上がってくるのはそれなりの規模の開発現場になるため、適当なサンプルコードだとデザインパターンを使った書き方がむしろ冗長な書き方に見えて、必要性がわからなくなつたりします。このため、コードを写経してもあまり理解が深まらない

い… なんてことも考えられます。理想を言うと、実際にそれなりの規模の開発に取り組んで、機能追加やリファクタリングのタイミングなどで「このパターンを使うとコードを再利用できる」と気付ければ、デザインパターンがスッと理解できるように思います。

13.4 さいごに

「デザインパターンとは」の部分で、デザインパターンはクラス設計の典型例に名前が付くことで、意思の疎通がスムーズになるメリットがあると述べました。そしてこの「名前が付く」という部分が大きなポイントだと感じています。

最近では、GitHub Copilot などの AI によるアシストのあるコーディング環境というのが一般的になってきました。こうした環境で AI によるコード補完の恩恵を最大化するためにも、一般に受け入れられる命名を意識することがより一層重要になってきています。このため、デザインパターンの語を知っておくことは今後も大いに意義があるのでないでしょうか。



あさみ @kawaiiseeker <https://x.com/kawaiiseeker>

虎の穴ラボでエンジニアをしています。Java が推し言語です。

第 14 章

生成 AI の不確実性に立ち向かうソフトウェアアーキテクチャ

Algomatic シゴラク AI カンパニー CTO 菊池琢磨 @_pochi

当稿は生成 AI Conf 「実践 LLM エンジニアリング」での登壇内容を、Object-Oriented Conference 2024 ガイドブック向けに加筆・修正したものです

<https://speakerdeck.com/tkikuchi1002/llm-engineering-architecture>

Algomatic のシゴラク AI カンパニーでは、生成 AI を活用した生産性向上支援 SaaS である「シゴラク AI」というプロダクトを開発・提供しています。本稿では、生成 AI を組み込んだプロダクトを開発していく中での設計の工夫について書いていきます。

14.1 生成 AI とは

そもそも生成 AI (Generative Artificial Intelligence) とは、機械学習の手法を利用して、既存のデータセットを基に新しい情報やパターンを創出する AI 技術です。これらは、トレーニングされたデータから学習した構造を利用して、テキスト、音声、画像などの新しい出力を生成する能力を持っています。わたしたちソフトウェアエンジニアとしては、ChatGPT にコーディングの相談をしたり、GitHub Copilot や Copilot Chat などを通じて触れことが多いかもしれません。

生成 AI が世間に認知されるようになったのはここ数年の出来事であり、GitHub Copilot は 2021 年、ChatGPT は 2022 年末にリリースされました。さらに 2023 年初頭には ChatGPT で GPT-4 というモデルが利用可能となり、その性能の高さが大きな話題となったことは記憶に新しいです。

生成 AI の利用シーンは大変幅広いのですが、わたしとしては「非構造化データを構造化データに変換することが非常に得意」だという点が非常に気に入っています。

電気通信大学在学中から、セキュリティソリューションを手がけるスタートアップにてソフトウェアエンジニアとしてインターンシップに従事。その後複数のスタートアップ企業を経て、フィンテック系スタートアップで VPoE に就任。その後、飲食店向け SaaS プロダクトを手掛ける株式会社 Showcase Gig では VPoT として技術統括および開発組織運営を行う。2023 年 8 月に Algomatic にシゴラク AI 事業部 CTO として参画

上記は私のプロフィール文章ですが、これを生成 AI を使って構造化データに変換したのが以下の JSON です。

```
{  
    "出身大学": "電気通信大学",  
    "経験した職種": [  
        "ソフトウェアエンジニア",  
        "VPoE (Vice President of Engineering)",  
        "VPoT (Vice President of Technology)",  
        "CTO (Chief Technology Officer)"  
    ],  
    "現在の肩書き": "シゴラク AI 事業部 CTO"  
}
```

このような自然言語から構造化データ変換を、事前のモデル準備なども必要なく簡単なプロンプト 1 つで行えてしまうのです。生成 AI によって、これまで諦めていたようなプロダクトが、あれもこれも作れるのではないか... と、ワクワクしませんか？

14.2 生成 AI をプロダクトで活用するということ

そんな生成 AI をプロダクトに組み込む際にわたしたちソフトウェアエンジニアが頭を悩ます要素として、代表的なものは「利用コスト」「回答速度」「コンテキスト長」などがあります。利用コストは生成 AI を呼び出す際の API 利用料であり、回答速度はその API によるレスポンスが全て得られるまでの時間です。コンテキスト長は、「生成 AI に一度に渡せる文字列の長さ」のことで、その長さに一定の制限があるのが現状です。

それらの問題に対して、以下のような創意工夫をもってプロダクトに落とし込んでいきます。

- 利用コスト: コストを下げるために安価なモデルを利用する、自前で LLM を構築する、
- 回答速度: キャッシュ処理を工夫したり、遅さが気にならないように体験を工夫したり
- コンテキスト長: テキストを分割したり、要約をしたり

また、生成 AI のモデルによってこれらの特徴は大きく異なります。現時点の自然言語の取り扱いにおいて精度が最も高いのは GPT-4 というモデルですが、このモデルは比較的コストは高く、回答速度は遅めです。ちなみに、その発展系である GPT-4-Turbo はかなり回答速度は早くなりましたが、GPT-3.5-Turbo に比べるとまだ遅さが感じられます。

14.3 そんな生成AIを活用するプロダクトをどう設計するか

AlgomaticのシゴラクAIカンパニーで開発・運用しているシゴラクAIは、いわゆる「法人向けChatGPT」と呼ばれるカテゴリの製品です。顧客への提供価値の中心に「生成AIそのもの」を据えたサービスであり、生成AIなしには成立しないプロダクトです。



図: シゴラクAIのスクリーンショット

どういうプロダクトかといえば、本家ChatGPTを思い浮かべてもらえばわかりやすいです。「チャットUIでAIに話しかけると返事がもらえる」というシンプルなものです。みなさんなら、どのような設計をしますか？今後の機能拡充を見越して、クリーンアーキテクチャのようにレイヤーを分けるでしょうか。それとも、階層構造は設けず、フラットな構成とする…あるいは、さらに異なる考え方で設計するでしょうか。

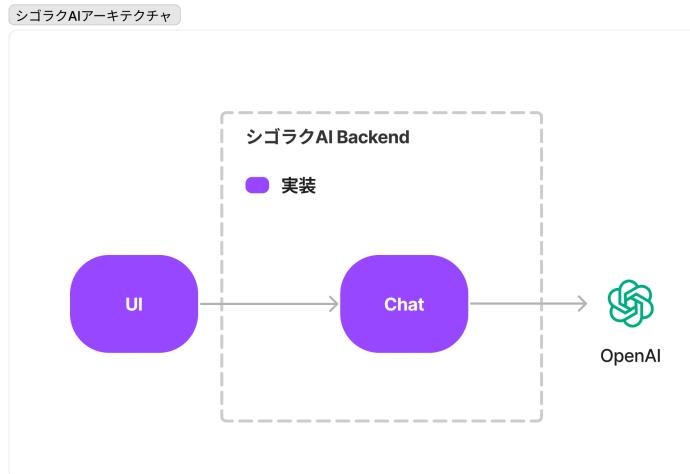


図: 初期アーキテクチャ

シゴラクAIの最初期は、抽象化なども挟まず、レイヤーの定義などもしない非常にシンプルな

設計でした。

少人数で開発していましたし、事業としての不確実性も高く、クリーンアーキテクチャのような今後のスケールを見越した手厚い設計は適さないと判断しました。当然機能としてもそれほど多くなく、ユーザーがログインできて、生成 AI に話しかけることができて、その返事をもらえる。それ以外には、ユーザー管理やコスト管理、請求管理の機能がある、というくらいでした。

ただしその後、事業が進むにつれて「社内ドキュメントに基づいて回答をする機能」、「Web ページの内容を自動で取得して回答する機能」といった AI による回答アルゴリズムを拡張するような機能開発や、「Slack で問い合わせられる機能」、UI リニューアルなど、ユーザー体験を新たにするような改修などを行うことになります。その過程で「これは何か秩序を設けないとまずいな」となり、シゴラク AI に必要なアーキテクチャ上の観点を考えました。

1. コアドメインを互いに分離したい
2. 生成 AI のモデルを切り替え可能にしたい

図にするとこのようになります。

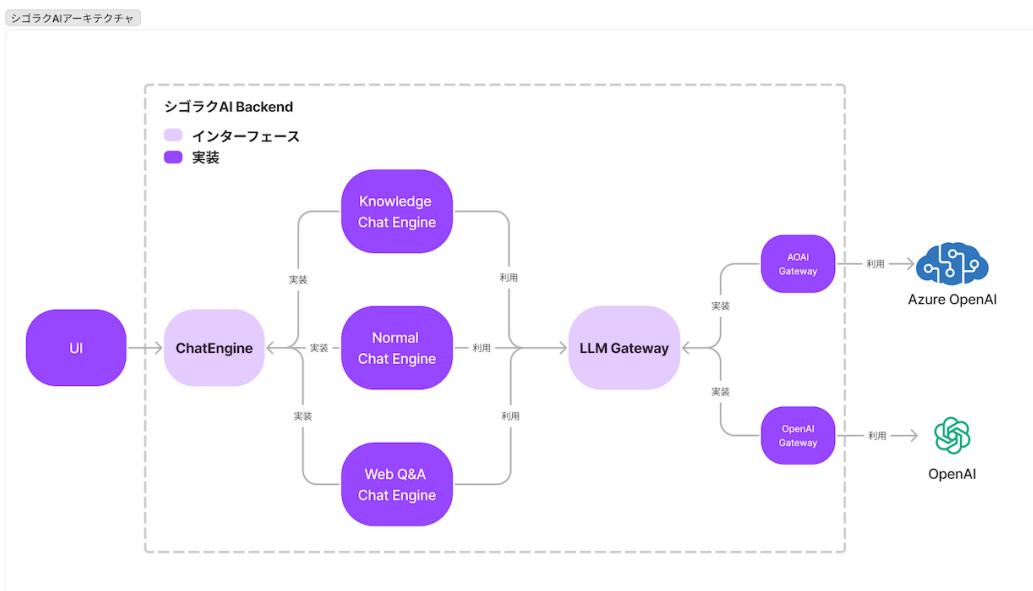


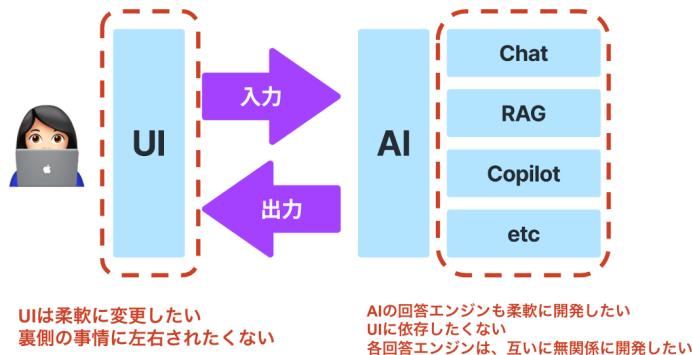
図: 現在のアーキテクチャ

CharEngine と LLM Gateway というインターフェースを用意し、抽象化を行なっています。それ以外にレイヤー定義などは行わず「必要な箇所に、必要な抽象化のみを行なった」状態としました。それぞれの観点について解説します。

14.4 コアドメインを互いに分離する

1つ目の観点は「コアドメインを互いに分離する」ことです。シゴラクAIの価値は、UIも含めた「ユーザーとAIアシスタントとの対話」の体験です。コアドメインは「ユーザー体験」と「AIによる回答品質」であり、これらの品質を高めていくことが大切だと考えています。

そのために、UIは柔軟に変更し、価値検証を繰り返せるようにしたい。裏側のロジックや事情に左右されることは避けたい。一方で、AIの回答品質はUIに関係なく柔軟に開発をしたい。さらに、「社内ドキュメントに基づく回答をする」、「Webページの内容に基づいて回答する」など、ユースケースごとにAIの回答アルゴリズムも複数存在できる必要がありました。



図：コアドメインを互いに分離する

それらを実現するため、AIとユーザーのやり取りを「ChatEngine」として抽象化し、UIと各エンジンは互いに依存せず、抽象にのみ依存する設計としました。UIはChatEngineのインターフェースのみ意識して開発すればよく、回答アルゴリズムも、ChatEngineのインターフェースに適合してさえいれば、裏側では自由に開発できます。これにより、「試験的に全く新しい回答アルゴリズムを試す」といったことも容易に可能になりました。

14.5 生成AIのモデルを切り替え可能にする

2つ目の観点は「生成AIのモデルを切り替え可能にする」ことです。経験的に、特定の生成AIサービスやモデルに依存することは避け、柔軟な切り替えを可能とすることの価値が高いと考えています。

新しいモデルが突然登場したり、過去の制約条件が突然取扱われたりするなどのポジティブな変化に素早く対応できることが事業価値につながりますし、反対に、突然特定のモデルが利用不能になった際の事業リスクを緩和したい、という意図もあります。

また少し具体的な話になってしまいますが、呼び出し先がAzure OpenAI Serviceであれば、Quota制限緩和のために複数エンドポイントの呼び分けをする必要があったり、「生成AIの回答

を JSON 形式に固定する」ような、特定のモデルにしか存在しない機能などもあり、これらの「生成 AI の呼び出しだけに関わる事情」を外に染み出させたくない。さらに、コストや速度面を意識したチューニングにおいては、呼び出し先のモデルを切り替えて動作テストを行いたくなることもあります。そこで、LLM の呼び出し層を「LLM Gateway」として抽象化しました。



各回答エンジンは特定のLLMに依存せず切り替え可能とする

図：生成 AI のモデルを切り替え可能にする

14.6 まとめ

以上が、シゴラク AI の現時点での設計、およびその考え方です。繰り返しになりますが、必要な抽象化のみを行なっただけで、クリーンアーキテクチャなどのように明確なレイヤーの定義やルール決めなどはしていません。ただし、「依存の方向を制限してコアドメインを依存関係の頂点に置く」という考え方はクリーンアーキテクチャと同様であり、部分的にエッセンスとして活用している状態です。こういった設計は今後も継続的に見直し、改善を続けていくことになります。現時点での正解が半年後も正解とは限りません。

ですから、「これは現状の姿であり、今後変わりゆく可能性が高い」とチーム全体で認識しておくことが何よりも大事だと考えています。そして、こういった話は生成 AI に限らず、すべてのプロダクトに通じるところです。

このシゴラク AI の事例がどなたかの役に立てば幸いです。



菊池琢磨 @__pochi https://twitter.com/_pochi

株式会社 Algomatic でシゴラク AI カンパニー CTO を務めています。型とドット絵が好きです。

第 III 部

イベントを支える技術-スタッフ寄稿—

カンファレンスを実現するには、様々な準備があります。スタッフのワーク、考えたことの一端をまとめました。

興味を持った方は、ぜひスタッフの扉をたたいてみてください。

第 15 章

搬入・宅急便関係の準備いろいろ

おやかた@oyakata2438

搬入関係を（も）担当します、おやかたです。

ここでは、搬入に関わるいろいろを述べてみたいと思います。裏側を知っていただけすると幸いです。

搬入搬出としてはサークル対応と、事務局の荷物の対応があります。また、事前準備、当日の搬入、当日の搬出と 3 つありますから、ぜんぶで 6 パターンになりますね。

15.1 事前準備

事前準備編です。

15.1.1 事務局の事前準備

事務局の荷物は、一般参加者から見える部分と見えない部分とありますが、それなりの物量があります。もちろん会場で借りるものもありますが、会場に「直接」届くもの、そして手持ちで搬入するものがあります。

会場で借りるものと手配する

会場で借りるものと手配する代表格は、机といいます。会場に並んでいる机や椅子は、当たり前ですが会場から借ります。もちろんコミケなどのように超でかい規模であれば外部のレンタル業者に手配する必要があるでしょうが、今のところの規模では会場備え付けの備品で事足ります。

会場を予約する段階でおおよその数を申請書に書きます。100 サークル前後を想定していれば、机 100 本、椅子 200 本、など。予約の段階では記入する枠がない場合は、後日「使用計画書」的な書類を提出することになります。ある程度サークル数などが固まっているはずですから、それなりの精度で書くことができるでしょう。

それ以外に会場から借りるものとしては、音響設備、電源、パーティションや掲示関係のいろいろ。会場ごとに所定の手続き書類がありますので、そのテンプレートに記入しましょう。

音響設備は、有線マイク、無線マイク、など。BGM をかけるための有線放送の設備があつたりすることもあります。いわゆる PA 卓のようなものが備え付けのこともありますね。いずれにせよ、有償なのか無償なのかは確認しておきましょう。

電源は容量計算や配線計画なども必要になるのでこれ以上は触れません。基本的に同人誌即売会

ではあまり使わないため、サークルが自由に使ってよいというシチュエーションにはならないかと思いますが、ハードウエア系のデモをするサークルに電源を提供する場合などはそれなりに準備が必要です。これはまた別途、「同人誌即売会を作ろう」の方で書くかもしれません。

会場に届くものを手配する。

会場に直接届くものもいろいろあります。

事務局のもので会場に直接届くものの代表は、ガイドブック、そしてポスターなどの会場掲示用の印刷物です。

ガイドブックは入場者全員に配布するものですから、それなりの部数があります。技書博の場合、無料配布ですが、即売会に醉っては入場料の代わりとして有償配布されたりしますね。

ガイドブックなどは、印刷所から直接搬入されます。印刷所の人（またはそこが手配した運送業者さん）が持ってきてくれます。ガイドブックを注文した印刷所さんと、搬入について調整しておきましょう。

具体的な調整事項は、搬入希望時間と、駐車場/荷捌き所の利用可否、です。

11時開会として、11時に来てては遅いわけです。一方で、会場引き渡し前に来てもらっても対応できない/待たせてしまうので、準備スケジュールをもとに希望時間帯を連絡しておきましょう。

以下は技書博の例です

8:00 会場引き渡し/開錠

9:00～10:00 搬入希望

9:30 サークル入場開始

11:00 一般入場開始

こんな感じで、スケジュールと希望時間を伝えましょう。他の印刷所さんも同じです。

直接搬入するもの

事務局の資材には、スタッフが手持ちで搬入するものがあります。受付用の資材、測量用のメジャーやマーキング用のテープ、文房具、最近だと体温計や消毒キット、などがあります。

測量用のメジャーやマーキング用テープなどは、準備の最初の段階で必要なものですから、会場引き渡しの直後に必要となるものです。ですから、車で搬入する場合、その搬入を担当する人の到着スケジュールをきちんと押さえておきましょう。

それ以外にも、来場者に配布する入場パスなどで、スタッフ作業を経由するもの、あるいは運送業者の都合で会場直送ができなかったものが含まれます。また、来場者に配布するものとして、過去のガイドブックの残部なども、ここに含まれます。基本的に過去のガイドブックはもう使うことはないのですが、ゴミとしてはもったいないです。初めて技書博に参加した人、あるいは前回参加できなかった人がもらってくれることも少なくありませんので、受付などに置いておくと無駄にななりません。これらは、スタッフが手持ち搬入する必要があります。あるいは、宅配便等で搬入するなど、荷物の振り分けを考えておいてください。

さて、これらのものをどうやって搬入するかですが、基本的には車（自家用車またはレンタカー）で搬入できると便利です。手持ち搬入には物量的にも限界がありますしね。

たいていの会場には駐車場があります。ですから、予約の段階で「搬入用に2台」といった形で申請をしておきます。一般サークルまで車搬入できるようにするかは議論がありますが、事務局枠は押さえておきましょう。場合によっては、他の催事との調整が必要となる場合もあります。しかもその日程は1-2か月前であることがざらです。事務局荷物を車搬入する場合は、この事前調整が必須であると考えておきましょう。

また、事務局資材は時々見直しましょう。使わないものを持って行くのは作業効率の点でもよくありません。

15.1.2 一般サークルの事前準備

一般サークル向けには、宅急便の業者さんの選定、搬入予定時刻の調整、伝票記入要領の決定と、サークルへの案内です。

業者選定と搬入予定時刻の調整は前節で完了していますので、伝票記入要領の決定についてです。

大量(おおむねサークル数×1.5箱程度)の荷物が届くので、どの荷物がどのサークルのものなのかはっきりさせる必要があります。サークル数が超多い場合は、搬入票を作ることがあります。搬入票は、箱の側面に貼り付ける紙で、サークル名や配置番号を記載したものです。箱の側面に張り付けることで、箱を積み上げたままでもわかりやすいですし、より大きな文字で表示することができます。

サークル数が少ない場合、伝票の備考欄でこれに代えることもできます。備考欄に、イベント名、内容物、サークル名、配置を記入してもらいます。

書籍 技書博8

あ-01 親方 Project

そして、これらの情報を各サークルに案内しましょう。どんなに遅くても10日前には案内を出してください。

週末に準備、発送手配をする、なんていうスケジュールはあるあるですよね。ですからこの時に案内がないとサークルは困ってしまいます。

案内の中には、送り先住所、催事名、到着指定日時(当日、時間指定なし、など)、備考欄の記載要領を明記します。備考欄(あるいは搬入票)の記載がない場合、荷物の紛失や遅配などのリスクが上がります。また、不着などのトラブルに備え、伝票番号は必ずメモしておく、あるいは当日伝票控えを持っておくように依頼して下さい。

それにあわせて、戻りの荷物の発送の案内などもできるといいですね。

Web等に記載するとともに、サークル、スポンサー、その他関係者にメールをするとよいでしょう。

15.2 まとめ

荷物の対応は地味ですが、重要度が高い調整項目です。

特に技術同人誌は一般にページ数が多く、かなりの重量になります。ということは、手搬入はあまり現実的ではありませんから、おのずと宅配便での搬入が増えます。ここでの準備をミスると当日配布するもの届かないなどの大きな事故につながります。事前調整を忘れないようにしましょうね！

15.3 次回予告

思ったより分量が多くなったので、当日の対応、撤収の対応については次回の記事に回します。
おたのしみに！

スタッフ紹介

コアスタッフ

ariaki @ariaki4dev



スタートアップの開発責任者を拝命し、刺激的な仕事で毎日楽しく駆けめぐり回っています。

おやかた（親方 Project） @oyakata2438



技術書を生やすお兄さんとして、LT 登壇や合同誌主催で新しい著者の背中を押すのが楽しい毎日です。技術同人誌を書くことはメリットしかない！ 楽しいぞ！ 本を書く場所を作れるということで、コアスタッフとして参画しました。スタッフもたのしー。

オーニシ @onishi_feuer



Web サイト制作・運営事業。IT 系ノウハウサイト「電腦世界」、動画掲示板「ゆっくり村」をはじめ、自分で制作した数十個の Web サイトを現在も運営・管理しています。そのほかイベント主催・運営、EC サイトでの輸出販売、輸入販売、動画実況・編集、同人ゲーム制作・販売など個人でできそうなプチ事業を色々やってきたり、新たに始めたりしています。

デザイン

OOC2024 公式ガイドブック

2024年3月24日 OOC 2024

デザイン hoge

編 集 おやかた (@oyakata2438)、ありあき (@ariaki4dev)

発行所 OOC2024 実行委員会

印刷所 K-9