サードパーティモジュールインポートのリスク軽減のための独自モジュール生成

信州大学　総合理工学研究科
23W2057B　志良堂泰成

背景：TypeScriptにおけるモジュール使用

Node.jsはnpmから、Denoはdeno.landというパッケージレジストリから
使いたい機能に合ったモジュールをインポートして開発を進める場合が多い

モジュールの使用例

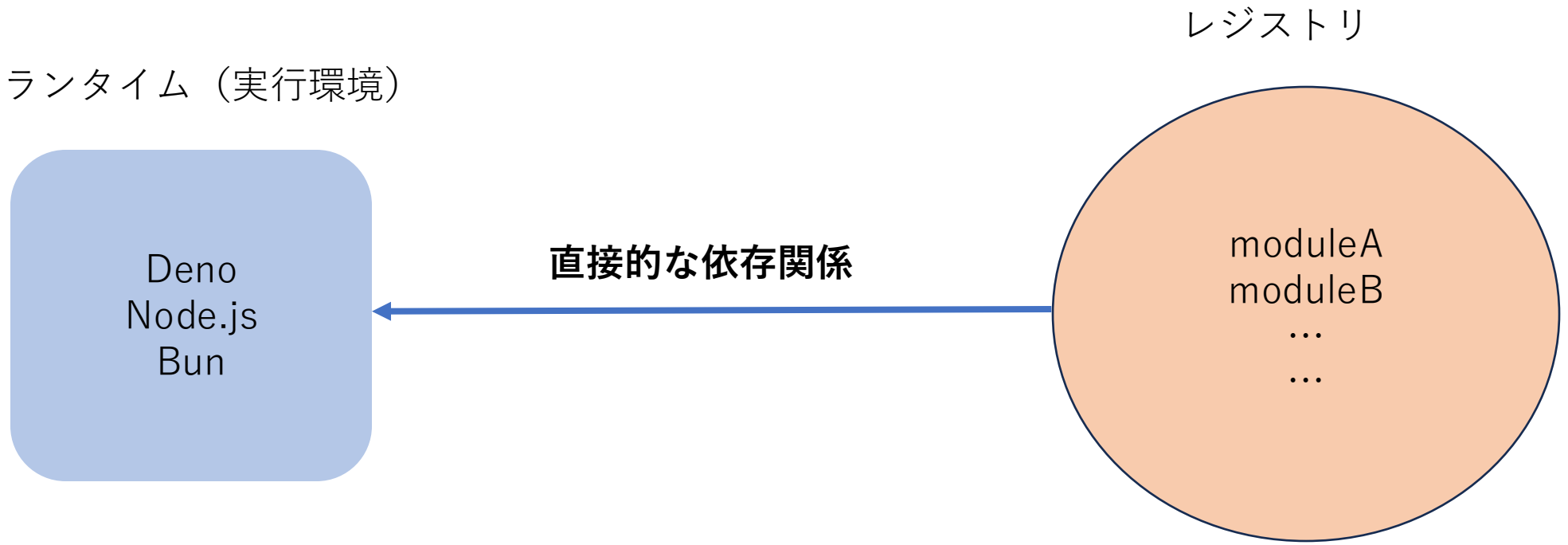
Denoでdeno.land上にあるoakというモジュールの中の
関数であるApplicationを使用する場合

```
import { Application } from "https://deno.land/x/oak/mod.ts";
```

↑
関数/クラス名

↑
参照するモジュールのURL名

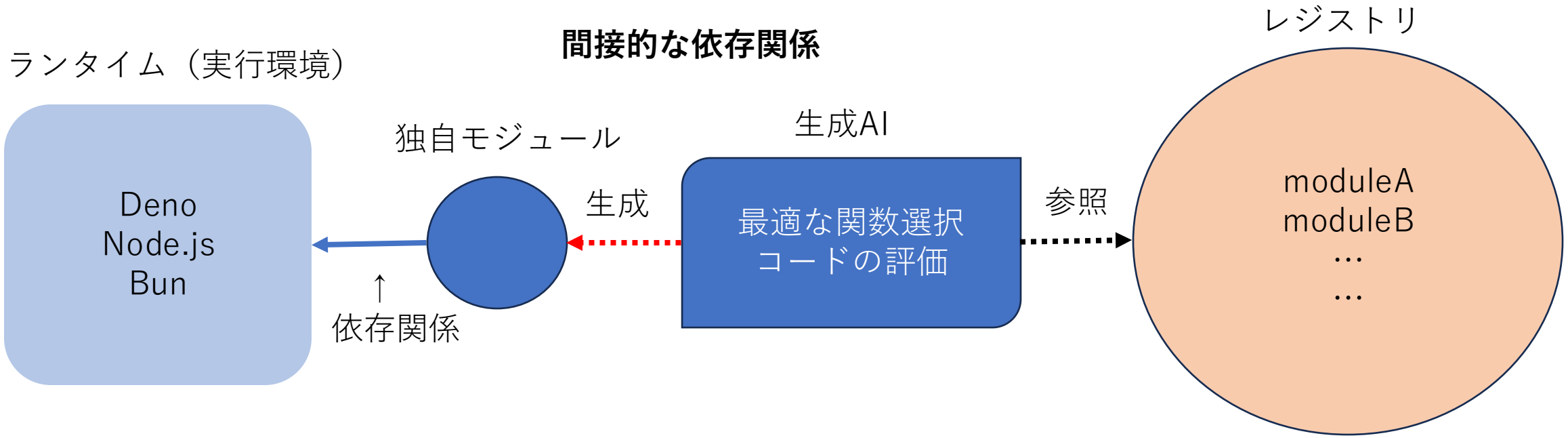
サードパーティモジュールに依存するリスク



リスク

・バージョンアップによる互換性の問題
・モジュールの非推奨化または削除
・セキュリティ上の脆弱性
・必要のない機能の取り込み

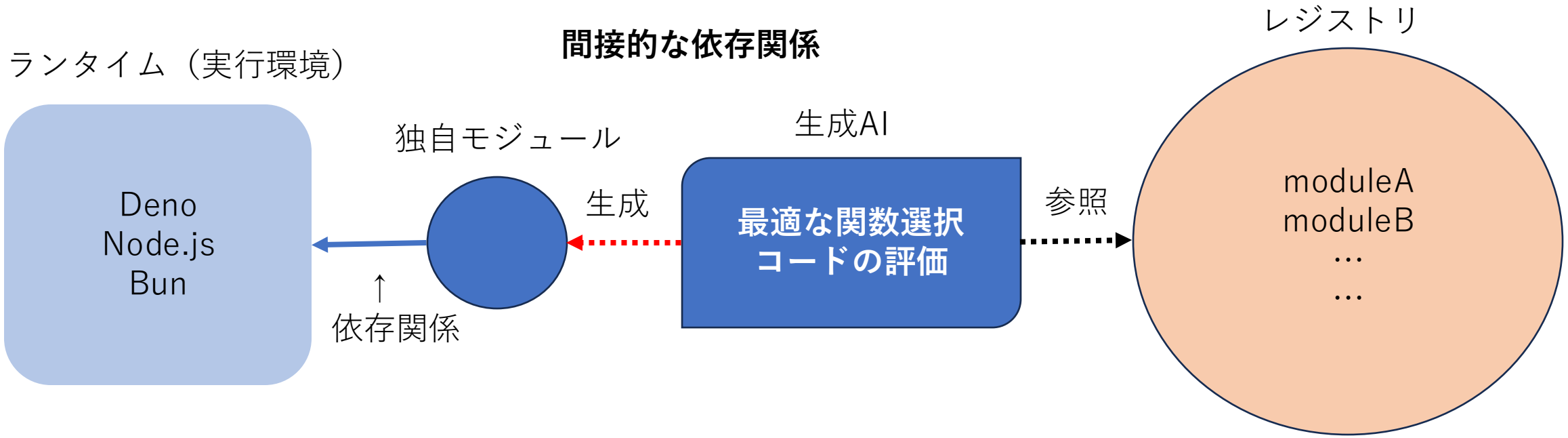
提案：リスク軽減のための独自モジュール



独自モジュール

レジストリ上のサードパーティモジュールを参照し、必要な関数やクラスを生成AIによって開発者の要件に合わせて生成した独自のモジュール。

提案：リスク軽減のための独自モジュール



独自モジュールによるメリット

・依存関係の軽減
・セキュリティの向上
・ファイルサイズの最適化
・dependabotやdeno-uddの代替

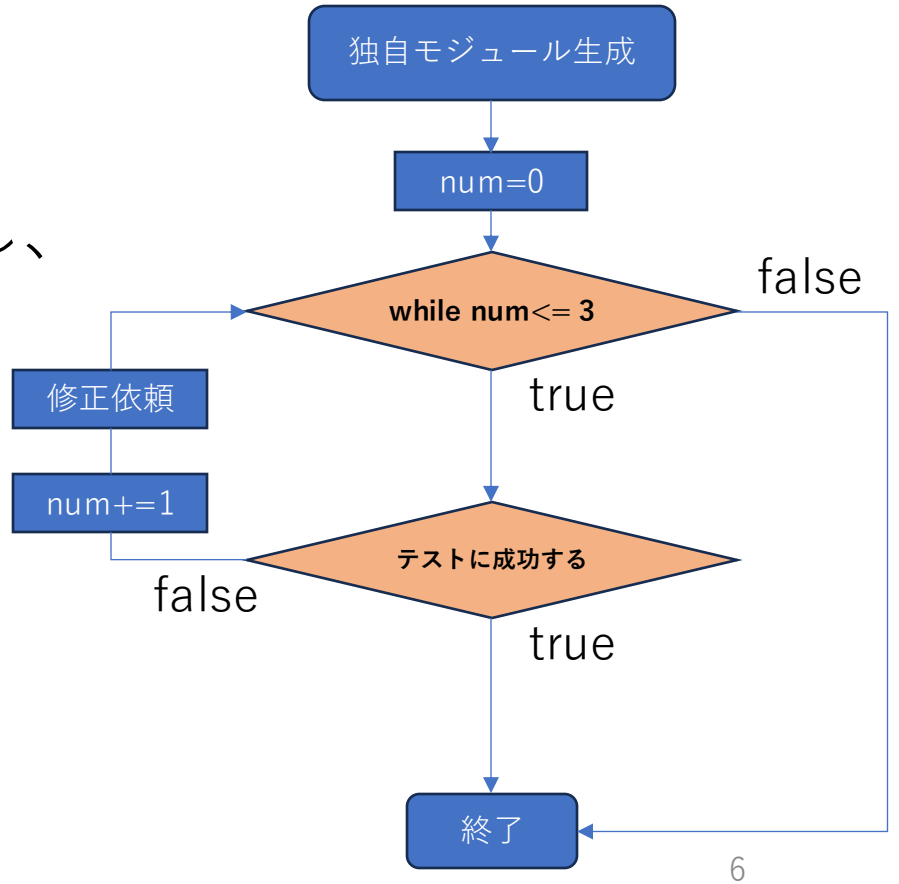
※deno-udd：dependabotのdeno版
モジュールの新バージョンを提案する。

自動修正システムの組み込み

独自モジュール生成の成功＝テストに成功

1回目の生成でテストに通らなかった場合、
そのエラー文と生成したコードをプロンプトに追加し、
再度生成/テストのサイクルを行う。
そのサイクルのうち、独自モジュールのテストを
3回までに合格しなければ失敗と扱う。

自動修正のフローチャート



分析

分析項目1：　対象とするモジュールから、
必要な機能を抜き出して独自モジュールにできるか。

分析項目2：　参照先であるレジストリ上の対象モジュールの
アップデートを独自モジュールに反映できるか。

調査するケース：**参照先が正常なモジュールのケース**（**正常時**）
と**エラーを含むモジュールのケース**（**エラー時**）

技術選定

採用するレジストリ

JSR（The JavaScript Registry）

2024年3月にDenoLand社が公開したJavaScript/TypeScriptのモジュールレジストリ。さまざまな実行環境（Deno,Node.js,Bun等）で実行可能であり、npmとの互換性もある。

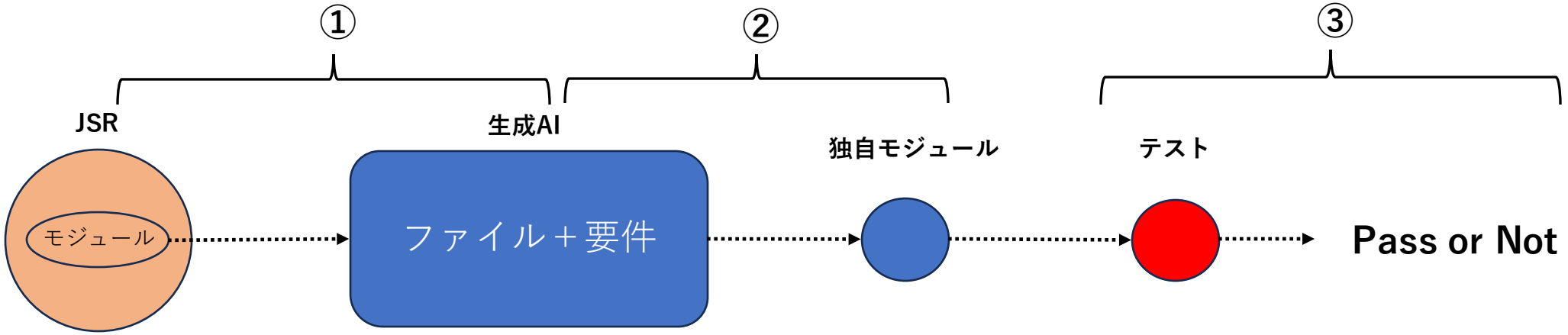
採用する生成AIモデル

OpenAI API：GPT-4o　Pythonで提案手法のシステムを実装

対象モジュール

条件：JSR内のTypeScriptファイルだけで機能が完結するモジュール
　　readme.mdでの使用例やテストファイルが存在するモジュール
理由：手法の初期段階としての実装において、複雑な仕様を避けるため

分析手順



①JSR上の対象モジュールについて、連携先のgithubリポジトリからファイルを取得。

②生成AIにプロンプトとして①で取得した**ファイル**と、
分析項目1→必要な関数だけを抜き出してください。
分析項目2 →アップデートを反映してください。
の文言を同時に与え、独自モジュールを生成。

③独自モジュールが機能を満たしているかテストし、機能するかどうかを評価。
修正できない場合はプロンプトの示唆を増やす。

分析方法　**分析項目1**　正常時の独自モジュール生成

正常時

プロンプト

{対象モジュールのコード}
Aという関数/クラスが機能するようにtypescriptファイルを実装してください。

対象モジュール
他モジュールに**依存関係のない**モジュール→**6**件
@phughesmcr/booleanarray,@cupglassesdev/random
@cross/runtime,@bearz/secrets,@mary/datefns,@enc/core

他モジュールに**依存関係のある**モジュール→**1**件
@cross/test

他モジュールに依存関係のあるモジュールについては、
依存先モジュールのファイルもプロンプトに含める。

評価する観点
・テストをパスするか。
・他への依存関係が存在するとどうなるか。

分析方法　**分析項目1**　エラー時の独自モジュール生成

エラー時

プロンプト

{エラーを含む対象モジュールのコード}
Aという関数/クラスが機能するようにtypescriptファイルを実装してください。

挿入するエラーのパターン
①関数名が変更されている場合（関数名エラー）
②関数の宣言されている型が変更されている場合（型宣言エラー）
③入力の型が変更されている場合（入力値エラー）

評価する観点
・テストをパスするか。
・他への依存関係が存在するとどうなるか。
・エラーを生成AIがどう扱うか。

分析方法　**分析項目2**　正常時のアップデート

正常時

プロンプト

{**旧バージョンの独自モジュールのコード**}+{**対象の最新モジュールのコード**}
Aという関数/クラスに関して新しい方に合わせて更新してください。

対象モジュール
特定の関数に、バージョンによる差分が見られるモジュール3件

@mary/datefns,@enc/core
入出力仕様の変わらないアップデート

@cross/runtime
新関数が登場し、新たな依存関数が増えるアップデート

分析方法　**分析項目2**　エラー時のアップデート

エラー時

プロンプト

{**旧バージョンの独自モジュールのコード**}+{**対象の最新モジュールのエラーを含むコード**}
Aという関数/クラスに関して新しい方に合わせて更新してください。

挿入するエラーのパターン
①関数名が変更されている場合（関数名エラー）
②関数の宣言されている型が変更されている場合（型宣言エラー）
③入力の型が変更されている場合（入力値エラー）
※分析項目1と同様の内容

分析結果　**分析項目1**　正常時の独自モジュール生成

表１：7つのモジュールに対して、独自モジュール生成を行なった結果

対象モジュール	他モジュールへの依存関係	入力ファイル数	入力ファイルの総行数	独自モジュールの総行数	テスト
@cupglassdev/random	なし	1	238	66	○
@phughesmcr/booleanarray	なし	1	659	123	○
@cross/runtime	なし	1	442	51	○
@bearz/secrets	なし	3	358	175	○
@mary/datefns	なし	1	531	8	○
@enc/core	なし	4	408	73	○
@cross/test	あり	3	531	-	△

分析結果　**分析項目1**　正常時の独自モジュール生成

表１：7つのモジュールに対して、独自モジュール生成を行なった結果

対象モジュール	他モジュールへの依存関係	入力ファイル数	入力ファイルの総行数	独自モジュールの総行数	テスト
@cupglassdev/random	なし	1	238	66	○
@phughesmcr/booleanarray	なし				○
@cross/runtime	なし				○
@bearz/secrets	なし				○
@mary/datefns	なし	1	531	8	○
@enc/core	なし	4	408	73	○
@cross/test	あり	3	531	-	△

外部モジュールへの依存関係がない
モジュール6件について、テストに成功した。

分析結果　**分析項目1**　正常時の独自モジュール生成

表１：7つのモジュールに対して、独自モジュール生成を行なった結果

対象モジュール	他モジュールへの依存関係	入力ファイル数	入力ファイルの総行数	独自モジュールの総行数	テスト
@cupglassdev/random	なし	1	238	66	○
@ph			659	123	○
@cr			442	51	○
@be			358	175	○
@mary/datefns	なし	1	531	8	○
@enc/core	なし	4	408	73	○
@cross/test	あり	3	531	-	△

入力ファイルの総行数よりも
独自モジュールの総行数の方が少ない。

分析結果　**分析項目1**　正常時の独自モジュール生成

独自モジュール化するための対象の関数/クラスの記述部分だけではなく、
他に必要なプログラム構成要素も同時に抜き出して機能を成立させていた。

例：@cross/runtimeモジュールの
関数getCurrentRutimeの抜き出し

操作前後のプログラム構成要素の比較
（黄色部分は対象関数に必要な構成要素）

対象モジュール

Enum（列挙型）

1.Runtime
2.Product

Functions（関数）

1.getChromeVersion
2.getFirefoxVersion
3.getEdgeVersion
4.getSafariVersion
対象関数→ 5.getCurrentRuntime
6.getCurrentProduct
7.getCurrentVersion

Variables（変数）

1.CurrentRuntime
2.CurrentProduct
3.CurrentVersion

独自モジュール

Enum（列挙型）

1.Runtime

Functions（関数）

5.getCurrentRuntime

Variables（変数）

1.CurrentRuntime

分析結果　**分析項目1**　正常時の独自モジュール生成

対象モジュール	他モジュールへの依存関係	入力ファイル数	入力ファイルの総行数	独自モジュールの総行数	テスト
@cupglassdev/random	なし	1	238	66	○
@phu					○
@cros					○
@bea					○
@mary/datefns	なし	1	531	8	○
@enc/core	なし	4	408	73	○
@cross/test	あり	3	531	-	△

唯一、外部に依存している@cross/testは
実行ごとに、テストに**成功する場合**と**エラーが起こる場合**が発生した。

分析結果　　**分析項目1**　外に依存関係があるモジュールについて

表２：@cross/testモジュールに対して
独自モジュール化の試行を5回行ったテスト結果

試行回数	テスト結果	修正回数
1	×	-
2	○	1回
3	×	-
4	×	-
5	○	2回

独自モジュール化成功率40%

生成AIのみで自動修正システムによる
関数の自動修正を行うことは可能であるが、安定性に欠ける結果となった。

分析結果　　**分析項目1**　エラー時の独自モジュール生成

表３：エラーコードが挿入されたモジュールを対象とした独自モジュール生成のテスト結果

対象モジュール	関数名エラー		型宣言エラー		入力値エラー	
	テスト	修正	テスト	修正	テスト	修正
@cupglassesdev/random	○	0回	○	0回	○	0回
@phughesmcr/booleanarray	○	1回	○	0回	○	0回
@cross/runtime	○	0回	○	0回	○	0回
@bearz/secrets	○	0回	○	0回	○	0回
@enc/core	○	0回	○	0回	○	0回
@mary/datefns	○	0回	○	0回	○	0回
@cross/test	×	-	△	-	×	-

@cross/testモジュール（依存関係が外にあるモジュール）以外は
エラーを修正し、テストを合格することに成功。

分析結果　　**分析項目1**　エラー時の独自モジュール生成

エラーコードを挿入されたモジュールを対象とした独自モジュール生成

対象モジュール	関数名エラー		型宣言エラー		入力値エラー	
	テスト	修正	テスト	修正	テスト	修正
@cupglassesdev/random	○	0回	○	0回	○	0回
@phughesmcr/booleanarray	○	1回	○	0回	○	0回
@cross/runtime	○	0回	○	0回	○	0回
@bearz/secrets	○	0回	○	0回	○	0回
@enc/core	○	0回	○	0回	○	0回
@mary/datefns	○	0回	○	0回	○	0回
@cross/test	×	-	△	-	×	-

成功数/試行回数　　0回/5回　　1回/5回　　0回/5回

1回/15回　→独自モジュール化成功率：約**6.7%**

正常時の独自モジュール生成の成功率**40%**

エラー時の独自モジュール生成の成功率**6.7%**

正常時安定性が低い
＋
エラーの挿入
＝　さらなる安定性の低下

分析結果　**分析項目2** 独自モジュールのアップデート

3つの独自モジュールに対してアップデートを行なった。

@enc/core, @mary/datefnsに関して

入出力仕様の変わらないアップデートであるため、**正常時・エラー時どちらも成功した。**

@cross/runtimeに関して

新関数が登場し、新たな依存関数が増える複雑なアップデートであるため、
正常時はアップデートに失敗。プロンプトの改善により成功した。
エラー時は、「入力値エラー」に関してのみ失敗し、プロンプトの改善により成功した。

分析結果　**分析項目2**　複雑な仕様のアップデート（正常時）

@cross/runtimeの複雑なアップデートに関して

通常のアップデートプロンプト

{**旧バージョンの独自モジュールのコード**}+{**最新モジュールのエラーを含むコード**}
Aという関数/クラスに関して新しい方に合わせて更新してください。

→**更新されなかった**。テストは成功するが、独自モジュールから変更がなかった。

改善プロンプト

{**通常のアップデートプロンプト**}+
最新のモジュールでは対象関数のために新たに関数を作って適応している様子が見られます。
その内容を反映して、対象関数が機能するように関数を更新してください。

→**更新された**。新関数が新たに作られ、対象関数に適応されていた。

実験結果　**分析項目2**　複雑な仕様のアップデート（エラー時）

入力値エラーを施したアップデート

本来の入力値

```
export function getCurrentRuntime(): Runtime {
  if (verifyGlobal("Deno", "object")) return Runtime.Deno;
```

変更を加えた入力値

```
export function getCurrentRuntime(): Runtime {
  if (verifyGlobal("Deno", 123)) return Runtime.Deno;
```

対象関数	getCurrentRuntime()
新関数	verifyGlobal()

プロンプト	結果
通常のアップデートを行うプロンプト。	**エラーコードのまま 独自モジュールに更新・適応された。**
エラーがあればその箇所を うまく動作するように変更してください。	**エラーコードのまま 独自モジュールに更新・適応された。**
VerifyGlobalという関数に関して、一つだけ入力値エラーが起きています。回避するように実装してください。	**エラーを回避するように 更新された。123→”object”**

実験結果　**分析項目2**　複雑な仕様のアップデート（エラー時）

入力値エラーを施したアップデート

本来の入力値

```
export function getCurrentRuntime(): Runtime {
  if (verifyGlobal("Deno", "object")) return Runtime.Deno;
```

変更を加えた入力値

```
export function getCurrentRuntime(): Runtime {
  if (verifyGlobal("Deno", 123)) return Runtime.Deno;
```

対象関数	getCurrentRuntime()
新関数	verifyGlobal()

プロンプト

通常のアップデートを行うプロンプト

エラーがあればその箇所を
うまく動作するように変更してください。

VerifyGlobalという関数に関して、一つだけ入力値エラーが起きています。回避するように実装してください。

結果

エラーコードのまま
独自モジュールに更新・適応された。

エラーコードのまま
独自モジュールに更新・適応された。

**エラーを回避するように
更新された。123→”object”**

プロンプトの改善により、エラー修正を行いながらアップデートすることができた。

失敗した原因：新関数の入力値の変更を行っていた。

実験結果まとめ

分析項目1　**対象モジュールから、必要な関数を抜き出して独自モジュールにできるか。**

・ファイル構造が単純なモジュールに関しての
関数/クラスを抜き出すことやエラーの修正は比較的容易であった。

・対象の関数/クラスに依存する他のプログラム構成要素も抜き出していた。

・依存関係が他にあるモジュールは、試行毎に正常に抜き出せる場合とそうでない場合が発生した。
→難易度が上がると生成AIの安定性には期待できない。

分析項目2　**参照先であるレジストリ上の対象モジュールのアップデートを独自モジュールに反映できるか。**

・ファイル構造が単純なモジュールに関してのアップデートは比較的容易であった。
・特に指示をしないで新仕様にアップデートを行わせようとすると、何も変更をされなかった。
・新しいバージョンで生まれた新関数についての入力値エラーは検知することができなかった。

→ プロンプトによる示唆で改善が見られた。

課題

生成AI・プロンプトの精度
分析項目1においては、同じプロンプトでもテスト結果が異なる場合があった。

→**妥当性の確保されているプロンプトの作成、生成AIの温度パラメータの調整が必要**

サンプル数の不足
JSRが歴史の浅いレジストリであるため、対象とできるモジュールが少なかった。

→**許容する依存関係を広げることが必要**

サンプルに使用する独自モジュールのオリジナル化
分析項目2において、独自モジュール化したものをそのままアップデート適応した。
実際の開発では、独自モジュールをカスタマイズするケースも多いと考えられる。

→**独自モジュールをオリジナルに加工したものを対象とすることが必要**

まとめ

独自モジュール生成・アップデートに関して

構造が単純なモジュールに関して、独自モジュールの手法を実装することができた。
依存関係等の要因で複雑になると、**生成・アップデートともにプロンプトによる示唆の追加が必要**であった。

実際に使用されやすい（人気のある）モジュールは規模・依存関係ともに複雑な傾向があるため一般開発者が使用する段階に至るには大きなハードルがある。