Object.create(null)

0506
2022

プログラムの複雑さ・表面積・グラフの構造

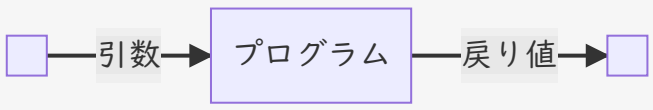
プログラミング 日記

特に何かしらの出典はありません.

プログラムの複雑さに対する大局的で直感的な指標として, 表面積とグラフの構造というのを個人的に意識しているという話. いわゆる code smell をどう嗅ぎつけているか.

表面積

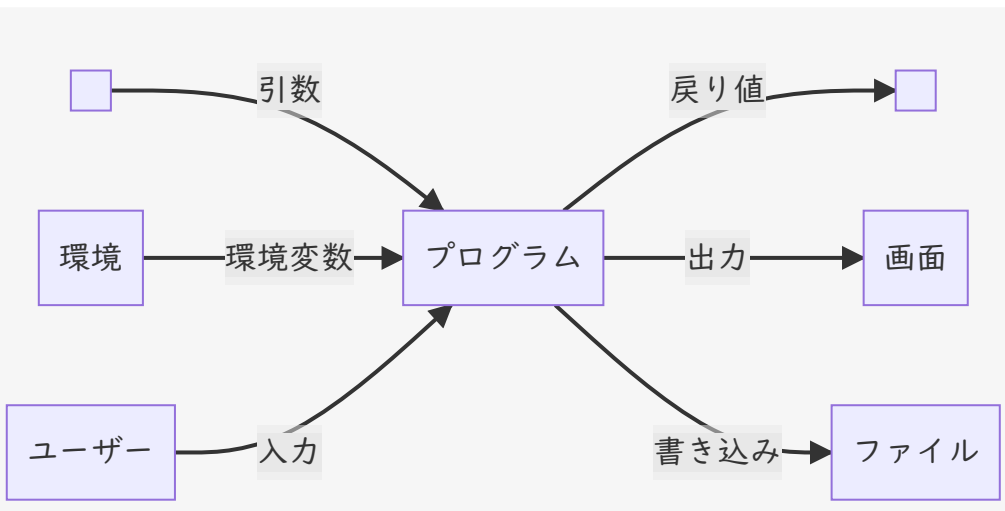
プログラムは最も単純には 1 つの入力チャンネル (引数) と 1 つの出力チャンネル (戻り値) でモデル化できます. 要するに関数ということですが, 関数型プログラミングに限らず大抵は似たような考え方ができます.



一方で現実世界で価値のあるプログラムとなるためには引数と戻り値だけでは不十分で, 実際にはその他の入出力チャンネルも必要になってきます. 例えば,

- 可変な変数の読み書き
- 環境変数の読み取り
- ユーザー入力の読み取り
- 画面への出力
- ファイルの読み書き
- データベースの読み書き
- HTTP リクエスト
- React Hooks

などなど. これらは乱暴にまとめると副作用と呼ばれる類のものです.



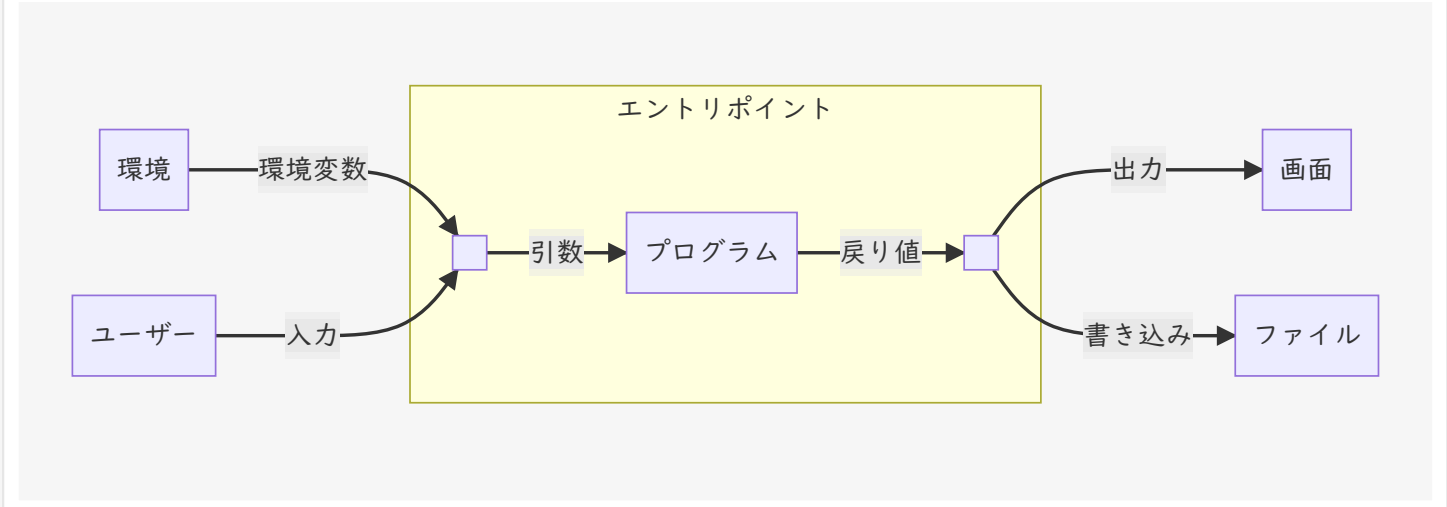
さて私が「表面積」と呼んでいるのはプログラムの持つ入出力チャンネルの数のことで, これが大きいほど複雑であると考えます.

なぜ表面積が多いほど複雑になるのか. 一つは大抵のプログラミング言語において, 関数 (またはそれと同等のもの) のシグネチャには入出力チャンネルのうち引数と戻り値しか明示されないことが多く, 逆に言えばそれ以外の入出力チャンネルについては暗黙的に扱われることが多いためです. プログラムの動作を把握するためにはこれらの暗黙的な入出力についても知っておく必要がありますが, そのためには関数のシグネチャのみならず, コード全体を読まなくてはなりません. これは非常に大変です.

二つ目は, 引数と戻り値以外の入出力チャンネルには, プログラム外部とのやりとりを含むものが多くあることです. これらを扱おうとした場合は全てがプログラム内部で完結している場合に比べて例外処理が多く必要になり, 特にプログラム全体に散らばってしまうと煩雑で漏れも出やすく, また重複も多くなるなど良いことがありません. またプログラムが外部と切り離せなくなってしまうと, テスタビリティの面でもセットアップが煩雑になるといった課題もあります.

三つ目は認知負荷の問題です. 表面積が大きいほどプログラムが一度に関係するものが多くなり, そして関係するものが多ければ多いほど人間が認知することが困難になります.

ではどのように表面積を減らすようにリファクタリングするかというと, 理想的には引数と戻り値以外の入出力チャンネルを, プログラムのエントリポイント (main 関数や HTTP リクエストのハンドラなど) に局所化します. こうすることでプログラムの大部分の入出力を引数と戻り値のみとすることができ, 大幅な単純化をすることができます.



あるいはもしこのようにできなかったとしても, 入出力チャンネル自体を引数を使って注入することで, 入出力の有無を明示してプログラムの動作を把握しやすくしつつ, かつテストでのモックを使った差し替えなどを行いやすくすることもできます. ただしこの場合であっても, 必要以上にプログラムの各部分に入出力チャンネルを渡したりはせず, 局所的にも表面積を小さくすることが重要です.

グラフの構造

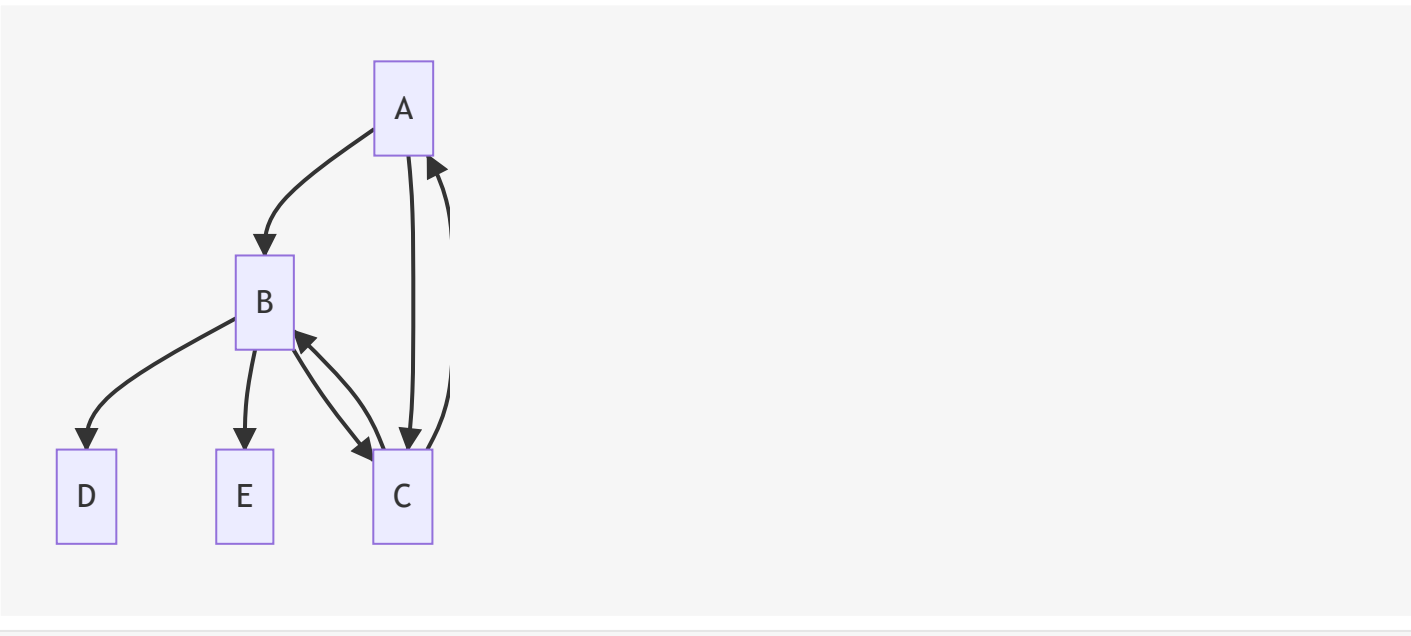
プログラムにおけるモジュール間の依存関係やデータフローなどに対して, それらを表現する (有向) グラフを考えることができます.

グラフの構造にはいくつか種類がありますが, 個人的に特に意識しているのは,

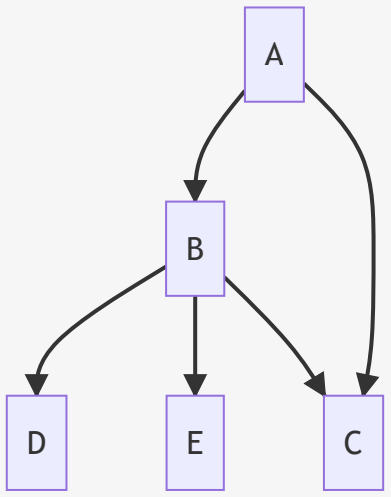
- 木
- 非巡回グラフ
- 一般のグラフ

の 3 種類です. これらは 木 < 非巡回グラフ < 一般のグラフ の順に一般性が高くなり, そしてこの順番にプログラムが複雑になる傾向があると考えます.

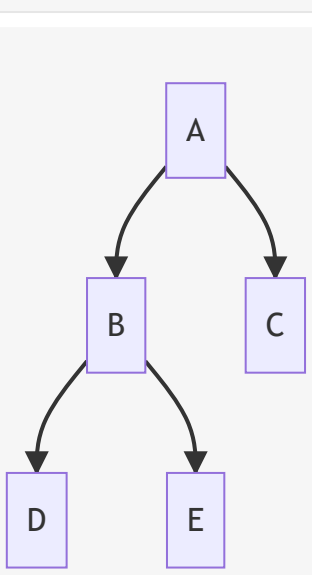
一般のグラフには閉路 (巡回路) があります. 閉路はモジュールの依存関係であれば相互依存している状態であり, 強く関連するもの同士が不当に複数のモジュールに分かれてしまっている可能性があります. またデータフローであれば双方向のデータフローであり, 例えば React コンポーネントであれば状態の同期というアンチパターンに陥ってしまっています.



非巡回グラフは, モジュールの依存関係であれば相互依存がない状態, データフローであれば単方向のデータフローに対応します. それぞれ一般の閉路のあるグラフと比べると, より健全かつ扱いが簡単になっています.

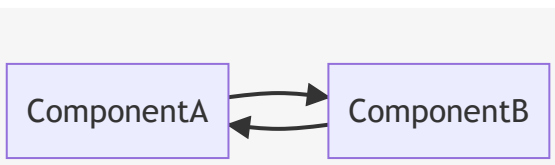


木構造はさらに枝ごとに分割統治を考えることができるため, プログラム的にも人間の認知的にもより簡単に扱うことができます.

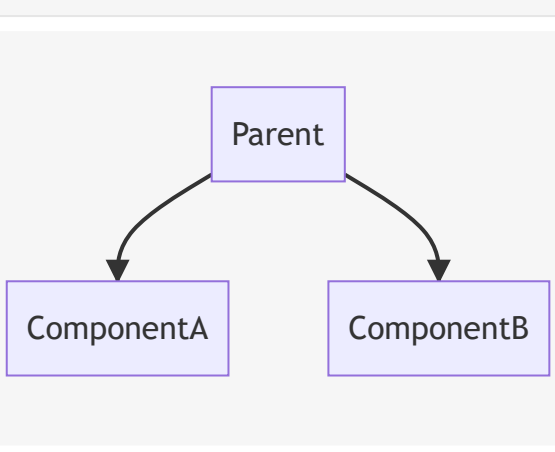


リファクタリングは, 過剰に一般的で複雑な構造になっているグラフを, より制約の強い簡単な構造に変形するようにして行います.

例えば途中でも例に挙げた React コンポーネントの状態管理であれば, 複数のコンポーネント間で状態を同期している状態 = 一般のグラフから,



親コンポーネントなどの一箇所で状態を管理している状態 = 木になるようにリファクタリングします.



非巡回グラフよりも木の方が良いという話は以前記事を書きました.

まとめ

- 表面積を減らせ
- グラフの構造を簡単にしろ

どちらも上のような図を想像してみたり描いてみたりするのがよいです.

by.Susisu (id:susisu) 80日前

関連記事

2021-11-03
TypeScript をより安全に使うために その 2: オブジェクトの具体的な形にアクセスするのを…
前回はこちら. susisu.hatenablog.com 引き続き環境は以下を前…

2020-08-16
TypeScript で次元つきの物理量を安全に扱う
キーワード: 型レベル整数, 幽霊型 前回の記事の予告通り, Type…

2019-12-15
Tagged Templates でたのしい Router & Reverse Router
この記事は はてなエンジニア Advent Calendar 2019 15 日目の…

2019-04-18
subtype 多相, never, union を有効活用していこうな
まずは MonadPlus ばんじゃーいとこういう感じに mzero と mplu…


2017-10-09
差分検出アルゴリズム三種盛り
こんばんは. 気がつけばもうずいぶんと涼しくなってきました. …

« Optional Variance Annotations の挙動メモ
TypeScript をもっと安全にする ESLint プ… »