第4章 動的アーキテクチャの見直しを行う方法

動的構造のリファクタリング…タスク設計＆優先順位

藤倉 俊幸

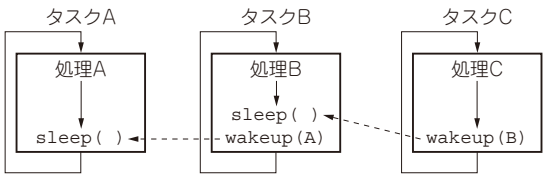

図1　動的アーキテクチャの例
タスク間同期の設計などが含まれる

リファクタリングの重要性

● ソフトウェアは一度作ったら終わりではない

一般に，リファクタリングと言うと，一通りシステムが動いた後，メンテナンスや機能追加のためにソフトウェアを見直すことを指します．一度作ったソフトウェアを面倒見ながら育てていくことは重要で，そのためにリファクタリングを行うと考えるのは合理的です．リファクタリングの過程で多くのことを学べたり，技術を蓄積・継承できたりします．

ソフトウェアは，きちんと面倒を見ないと容易に岩より硬く（修正不能に）なります．アーキテクチャ・レベルの見直しをしないで，ソースコードの修正ばかり繰り返すとif文のネストが深くなったり，分岐の多いソースコードになって修正困難となったりします．そのため，ソフトウェアの状況は適切なメトリックス（測定基準，尺度）を取って監視する必要があります．

● 動的アーキテクチャの見直しは難しい

ここで，動的アーキテクチャというのは，タスク設計などのコーディングを始める前に決めておくべきソフトウェアの構造を指します．具体的には**図1**に示すようなタスク間同期の設計や，タスク数と優先度を決めることです．これに失敗するとデッドロックが起きたり，パフォーマンスが出なかったりします．

最近では，ソースコードの修正などをAIがやってくれるようになりました．しかしアーキテクチャ・レベルの見直しなどはまだできないようです．組み込みシステムで特に重要な動的アーキテクチャとなるとなおさらです．

取りあえず現状把握しておいて破りの発見

● メトリックスを取るのが重要

リファクタリングをするには，ソフトウェアを解析して現状を把握し問題点を明確にし，どう改善するか決めないといけません．そのためにはソフトウェア構造の可視化と，適切なメトリックスによる定量化が重要です．

定量化と言っても，リファクタリングの目標値にするのではなく，バージョンアップとか製品のリリースごとにメトリックスを取っておいて，それらを比較して悪い方向に進んでいるかどうかを確認することが必要です．

● おきて破りの設計がないか

現状を把握して，おきて破りの設計がないか，これでよく動いてるな，ちゃんとテストしたのか，のような部分を見つけて改善するのが最優先です．

このためには，タスク間でグローバル変数を共有するときは排他制御しなければいけない，などのおきてを組織として共有していると役に立ちます．

表1に割り込みハンドラのおきての例を示します．1番目は，設計目標のようなものでこれに従えば2番目，3番目はおおむねクリアされるのではないでしょうか．1番目が不徹底だと割り込みハンドラは複雑化して他のおきて破りが発生します．OSを使っていないシステムだとこのおきてではキツいので，排他制御するためにいきなり6番が破られるパターンが多い気がします．

リエントラント（reentrant）な関数を実現したい場合は，スタック変数のみを使用します．リエントラント関数は，排他制御せずに共有できる優れものです．

● 非機能要求の改善

リファクタリングが必要になるもう1つの切っ掛け