組込みソフトウェア向け 設計ガイド

ESDR［事例編］

独立行政法人 情報処理推進機構
技術本部 ソフトウェア・エンジニアリング・センター　編著

本書の内容に関して

・本書を発行するにあたって、内容に誤りのないよう出来る限りの注意を払いましたが、本書の内容を適用した結果生じたこと、また、適用できなかった結果について、著者、発行人は一切の責任を負いませんので、ご了承ください。
・本書の一部あるいは全部について、著者、発行人の許諾を得ずに無断で転載、複写複製、電子データ化することは禁じられています。
・乱丁・落丁本はお取り替えいたします。下記の連絡先までお知らせください。
・本書に記載した情報に関する正誤や追加情報がある場合は、IPA/SEC のウェブサイトに掲載します。下記の URL をご参照ください。

独立行政法人 情報処理推進機構（IPA）
技術本部 ソフトウェア・エンジニアリング・センター（SEC）
http://sec.ipa.go.jp/

はじめに

航空・宇宙、医療をはじめとするミッションクリティカルな分野や、通信、交通、電力、水道などの生活基盤としての社会インフラ分野においては、マイコン組込み製品・システムの占める割合がますます増加するとともに重要性が高まっており、安全・安心が強く求められています。それに呼応して製品/システムの大規模化(高機能、多機能)、高度化・複雑化とそれを実現するソフトウェアの規模が急拡大する中、ソフトウェアを原因とする故障、事故が後を絶たず、ソフトウェアに対する高品質・高信頼性の要求が高まっています。

ソフトウェアの品質を担保するためには、ソフトウェア開発の設計工程で不具合の潜在化を先手を取って防ぐことで効果を出すことができます。しかし、設計工程で不具合の潜在化を防ぐためのノウハウ(定石)は、多くの場合、各開発の場面での文書化が行われておらず、伝承の域に留まってしまっています。また、それらノウハウについても、どの程度の安全・安心要求に対し、どの程度のからくりを入れたらよいかということについて整理されたものもありません。

独立行政法人 情報処理推進機構(IPA)技術本部 ソフトウェア・エンジニアリング・センター(SEC)組込み系プロジェクト/安全ソフトウェア構築技術部会では、組込みソフトウェア開発現場で設計を行う上で求められる質を確保したシステム・ソフトウェアを実現するために現場で行われている設計上の工夫、注意点を事例として収集し、それぞれの事例からノウハウを抽出し、設計作法として整理文書化したガイド(事例集)を編纂しました。事例の分析にあたっては、極力個々の製品分野固有のノウハウとならないよう目的とする品質特性を明確にし、利用する設計の各場面(ソフトウェア設計、ソフトウェア・アーキテクチャ設計、システム設計など、または設計対象)が明確になるよう留意しました。

大規模化するソフトウェアの品質面で大きなネックとなっている設計面、実装面での技術者による出来不出来のばらつきを解消し、ソフトウェア開発上流段階での品質作り込みの一助となることを願っています。

2012 年 11 月

独立行政法人 情報処理推進機構(IPA)
技術本部 ソフトウェア・エンジニアリング・センター(SEC)
組込み系プロジェクト
三原幸博、浜田直樹、十山圭介

目　次

はじめに……3

解説　設計ガイド[事例編]の読み方、作法表　7

1　設計作法について……8
1.1　組込みソフトウェア開発における設計……8
1.2　設計作法の目的と位置付け・想定利用者……10
1.3　作法の読み方……11
2　品質のとらえ方……13
2.1　品質特性……13
2.2　品質特性と作法の考え方……16
3　本書の構成……17

Part A　設計コンセプトを決める　19

A-1　イベントドリブン型のシステムの設計では状態遷移図と状態遷移表を併用する……20
A-2　正常系を設計すると同時に異常系も併せて設計する……22
A-3　ソフトウェア部品は単機能になるように設計する……24
A-4　ソフトウェアの再利用は設計レベルで行う……26
A-5　実装は設計のあとに行う……27
A-6　テストを考慮して設計する……28
A-7　ソフトウェアの要素には機能を表す名前を付ける……29
A-8　COTS（Commercial Off-The-Shelf）製品を安易に利用しない……30
A-9　1関数1責務（要素機能）にする……32
A-10　同期通信ではタイムアウトやタスク生存期間を考慮する……39
A-11　不正値でも補正可能な場合は補正する……42
A-12　イベントを周期的に処理する場合は周期を意識した安易な処理分割をしない……44
A-13　シリーズ製品では違いを意識して設計する……47
A-14　シリーズ製品では エラー処理を統一する……51
A-15　シリーズ製品では 共通点を意識して設計する……54
A-16　ループ処理からの例外脱出の仕様を定義する……57
A-17　時間制約を定義しこれを守ることにより確実な動作を保証する……59
A-18　2次元マトリクスを利用して出力競合を解決する……61
A-19　2次元マトリクスを利用して機能競合を解決する……63
A-20　実行時の不具合の追跡手段を用意する……65
A-21　メモリの動的な扱いを避ける……68

A-22 故障を回避する …… 70
A-23 システム改修・更新時には、先に性能解析をしてから
アーキテクチャとソフトウェアを変更する …… 73
A-24 ハードウェア制御を凝集する …… 76
A-25 入力パラメータチェックはシステム全体で最適化する …… 78
A-26 派生開発でソースコードの再利用性が低ければ設計の見直しを行う …… 80
A-27 ソフトウェアの階層構造の深さは、性能要件と保守性の
トレードオフを考慮して決定する …… 82
A-28 要件定義では、先に性能要件をシステム全体で評価する …… 84

Part B システムレベルの設計の工夫 87

B-1 エラー発生時にも処理の継続性を確保する …… 88
B-2 システムのバージョン変更が円滑に行えるようにする …… 90
B-3 システムの構成に合わせてデータをチェックするメカニズムを用意する …… 91
B-4 システム障害時にも稼働を継続するために縮退運転を検討する …… 93
B-5 単位系を統一する …… 95
B-6 スタックがオーバーフローしないようにする …… 97
B-7 割込みレベルはシステム全体を考慮して設計・保守する …… 99
B-8 扱うデータの特性に応じてシステムの構成要素を分離する …… 101
B-9 メモリの動的確保に留意する …… 103
B-10 状態遷移設計を適度なタイミングで見直す …… 105
B-11 不必要な待ち操作は避ける …… 109
B-12 機能仕様を決める際には例外への対策を網羅的に行う …… 111
B-13 可変長データを扱うシステムではデータ長の異常への対策を行う …… 113
B-14 ログ・トレースに関するルールを定める …… 115
B-15 ハードウェア変更時の動作を保証する …… 116
B-16 メモリ領域を効率良く使用する …… 117
B-17 必要以上の汎用性、拡張性を設計・実装しない …… 119
B-18 複数の箇所に分散している処理はまとめる …… 122
B-19 類似のデータ変換処理は統合する …… 124
B-20 性能改善のための変更は性能ボトルネック解析に基づいて行う …… 127
B-21 サードパーティ製品やオープンソースソフトウェアなどと組み合わせて
開発するときは、事前に十分調査・解析・評価する…… 129

Part C ミドルウェア・ネットワークスタック・ライブラリレベルでの工夫 131

C-1 ポートスキャンによる外部からの攻撃に備える …… 132
C-2 入力データに対し処理を継続的に行う場合に優先度逆転による異常な状態を回避する … 133
C-3 状態遷移を行うモジュール間で状態の一貫性が保たれるように設計する …… 136
C-4 変数の初期化にあたってはシステムの挙動を網羅的に考慮する …… 140
C-5 グローバル変数は極力使用しない …… 143
C-6 不必要な処理はしない …… 145
C-7 データ管理機構を使ってデータの一貫性を保証する …… 146
C-8 パラメータチェックに関するルールをプロジェクトで統一する …… 148
C-9 開発中に動作速度をモニターしプロファイリングを行う …… 150
C-10 処理速度とソースコードの可読性を両立させる …… 152
C-11 複数のデータフォーマットの相互変換は中間データフォーマット経由で実現する …… 154
C-12 既存のソフトウェアを改造する前に仕様をよく理解する …… 156
C-13 事前に構造解析を行うことにより効率的にソフトウェアを再設計する …… 159
C-14 機能の変更・追加のときにはソフトウェアの静的構造を崩さないように注意する …… 161
C-15 効率を意識してテーブル設計を行う …… 164
C-16 データアクセスを局所化しデータ更新処理による不整合を防ぐ …… 167
C-17 モジュールの参照方向を上位から下位へ統一する …… 168
C-18 機能変更を行う前にパフォーマンスに与える影響を十分検討する …… 171

Part D システムで扱う周辺デバイス操作に関する工夫 175

D-1 多重割込みはどうしても必要な場合のみ使用する …… 176
D-2 デバイス初期化時の副作用を回避する …… 177
D-3 周辺デバイスの I/O アクセスは専用関数・マクロを用いる …… 179
D-4 ハードウェアを保護するための機能レイヤを設ける …… 180
D-5 デバイスの寿命に関する特性を考慮して設計する …… 182
D-6 ハードウェアの機能と物理的特性を十分に分析してから設計する …… 185
D-7 ハードウェアの操作を出来るだけ遅らせてパフォーマンスを向上させる …… 187

解説

設計ガイド［事例編］の読み方、作法表

1 設計作法について

1.1 組込みソフトウェア開発における設計

今日、高品質な組込みソフトウェアを時間的にも経済的にも効率良く開発したい、という QCD の要求は加速度的な勢いで高まっています。こういった要求に応えて開発を効率良く進めるには、開発の基礎となる部分で十分な作り込みを行い、分析や認識の不足に起因する手戻りを防止することが重要です。

これを実践するために組込みソフトウェア開発の過程を細かく把握する道具として有用なのが組込みソフトウェア開発のプロセスです。プロセスについて詳しくは本シリーズの「ESPR：組込みソフトウェア向け開発プロセスガイド」をご覧いただくとして、ここでは、設計に関わる部分を中心に概観します。

● V 字モデルと開発プロセス

ソフトウェア開発はシステムに対する要求から出発し段階的な詳細化により実装を得る流れと、段階を逆にたどって実装が要求を満たしていることを確認しながら全体を統合する流れとからなっています。このようなソフトウェアのライフサイクルにおける各段階の関係を図 1-1 の V 字モデルです。V 字の左側が詳細化の、右側が確認・統合の流れに対応し、各段階の対応関係が水平方向の矢印で表現されています(注 1)。

注 1：V 字モデル上の「流れ」はあくまで情報の依存関係にすぎず、作業工程の順番とは必ずしも関係ありません。従って、ここに示された流れの通りにソフトウェア開発を進めるようすすめるものではありませんし、この流れに沿った開発でなければ適用出来ない、という性質のものでもありません。

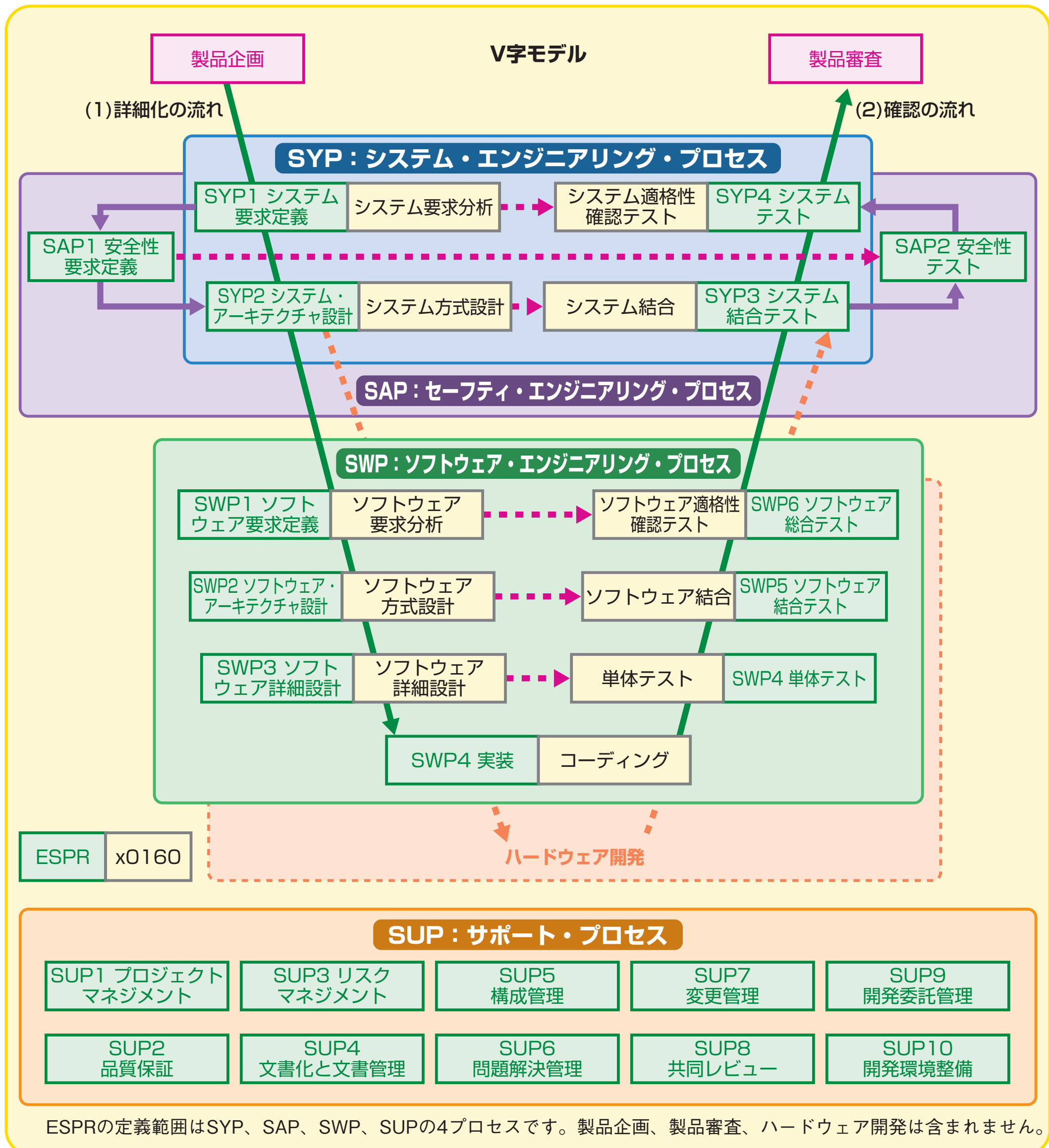


図1-1：V字モデルと開発プロセス

現実の開発は多くの場合、各ステップを行ったり来たりしながら進行します。現在行っている作業がどこに相当するかを常に意識すると、開発で必要になる項目のギャップを防ぎやすくなります。こういった項目の依存関係を整理し、混乱を防止するためのツールがＶ字モデルやプロセスです。

●設計とは

「設計」とは、要求から実装を得るための段階的な作り込みのことです。設計には様々なレベルがあり、実現したいソフトウェアの複雑さや規模に応じて幾つかの段階に分かれるのが一般的です。本書ではソフトウェア開発のV字モデルにおけるシステム・アーキテクチャ設計やソフトウェア・アーキテクチャ設計、ソフトウェア詳細設計に相当するプロセスに注目して、高品質な組込みソフトウェアを効率良く実現するためのノウハウを提供します。

本書ではとくに、最近の組込みソフトウェア開発の現場で顕著になってきた既存のソフトウェアから出発する差分開発や、既存のソフトウェアの仕様を変えずにソフトウェアの構造を整理する改善(注2)にも注意を払い、多様な状況でのノウハウを収集するよう努めました。

● 設計作法とは

ソフトウェアの設計についてはすでに様々な研究や書籍があります。こういったもののほとんどは特定の手法や体系に依存しています。ところが、現実の開発のうち大多数はすでに存在する何らかのソフトウェアの枠組みや要素を利用しています。こういった状況で新たに与えられた手法や体系を適用することは、必ずしも容易ではありません。

本書では、現場の問題意識を反映した実際に利用しやすいガイドブックとするために、様々な企業の方々に委員としての参加をお願いし、各委員から提供された事例について委員会でその背景や特徴、類似の経験や他の事例との関係などを詳細に検討しました。そして、今回は組込みソフトウェア開発の現場の事例から得られた知見を、特定の手法や体系に依存しない形まで抽象化してまとめた74個の「作法」として提供することにしました。

1.2 設計作法の目的と位置付け・想定利用者

本書は組込みソフトウェアの設計を行う人が作業の参考とすることを第一の目標としています。利用しやすいよう、実際の設計事例から抽出したノウハウを定型のフォーマットで分類整理しています。また、豊富な例や図表により、主旨を複数の角度から理解出来るよう配慮しています。

● 想定する利用者と利用方法

本書は経験を積んだ設計者だけでなく、設計に携わるようになって日の浅い若手の技術者にとっても分かりやすいよう心がけました。想定している利用方法は以下の通りです。

◇組込みソフトウェアの設計や設計レビューでの利用

　　目標とする設計と最も関連の深い作法を参考に、設計を行います。

◇設計者の研修、独習のための学習教材

　　問題の発見と解決という具体例を通じた失敗の疑似体験により、個々の技術者が幅広い視野を獲得出来ます。

◇設計要領書作成の雛形

　　本書の作法のフォーマットを参考にすることにより、組織や部門で受け継がれているノウハウ

注2：いわゆる「リファクタリング」も改善の一種と考えることが出来ます。

を明確で利用しやすい形に文書化出来ます。

◇期待出来る効果

本書を以上のように利用することにより、次のような効果が期待出来ます。

- 見通しが良く、目的とする品質を備えたソフトウェアの設計の設計が容易になる。
- ソフトウェアの品質面で大きなネックとなっている、設計における技術者による完成度のばらつきを小さくする。
- 設計上の明らかな誤りを、設計段階やその後のレビューなどで早期に除去出来る。

1.3 作法の読み方

それぞれの作法は、原則として以下のフォーマットに従って整理・記述されています。

①場面による分類
②作法名称
③作法概要
④メリット
⑤留意点
⑥解説
⑦例
⑧関連する品質特性
⑨関連する作法

設計コンセプトを決める
Part A

A-8 COTS(Commercial Off The Shelf)製品を安易に利用しない

作法概要
既製のソフトウェアやオープンソースソフトウェアを利用する場合は、事前に十分な動作検証を行う。

メリット
●市場での想定外の動作や不具合が防止出来る。
●システムテストの効率化が可能となる。
●設計手戻りの防止に繋がる。

留意点
①オープンソースソフトウェアの場合は、ソースコードの静的解析なども有効である。
②ソフトウェアは開発元の必要により変化していくため、バージョンの追従方針をあらかじめ決めておく必要がある。
③将来的な拡張性や保守性も考慮する必要がある。
④オープンソースソフトウェアの使用にあたっては、例外を含む網羅的で十分なストレステストにより品質を担保する。

●解説
既製のソフトウェア製品やオープンソースを利用することは、システム開発のコスト削減と開発期間短縮に有効な手段である。ときには、システムの制約上、他社製品を組み込んでシステムを構成しなければならない場合もある。しかし、それらのソフトウェアは、内部の設計構造や振舞いが完全に把握出来るとは限らない。その結果、利用したシステムにおいて、想定外の動作や不具合を起こすことがある。利用に際しては、次の点なども考慮し、十分な検証が必要である。
①システムに組み込んで使用する場合に、期待通りの動作をするかどうか。潜在化していた条件が新たな制約になったり、期待通りの性能が出ない可能性はないかを検証する。
②他部門での利用実績があれば、その検証結果や不具合事例などを確認する。
③一般向けに提供されているソフトウェア製品などは、第三者による動作検証が実施されている場合があるので、それらの情報を利用する。
④ソフトウェアコードだけではなく、開発環境に関しても、バージョンの違いやツールの組み合わせ、実績のない新機能などに注意する。

例 BDプレーヤのようなAV機器
他社製品で使用している制御用ミドルウェアを購入し、自社製品に実装した。通常の使用状態では問題なく動作したが、外付けHDDを接続すると、ある特定の条件で時々ハングアップすることがあった。
不具合原因を調べたところ、内蔵DVD/BDドライブからの信号遅延の影響で、状態遷移が不正となることが分かった。更に、実績ある他社製品を調べてみると、内蔵DVD/BDドライブのバージョンが異なっていることが分かった。
そこで、タイミングが異なっても正しく状態遷移するようコードを修正し、他に同様なケースがないことを確認するため、制御バスタイミングに重点を置き、網羅的にテストを実施した。

他社製品
S/W
制御用ミドルウェア
Ver.1.2.3
内蔵DVD/BD
I/F
外付けHDD
購入／実装
異なるバージョンでは正しく動作しない
自社製品
S/W
制御用ミドルウェア
Ver.1.3.2
内蔵DVD/BD
I/F
外付けHDD
[現象]
外付けHDDを接続したときだけハングアップ

図A-3：自社製品に他社のCOTS製品を実装

関連する品質特性
●機能性(正確性、相互運用性)
●信頼性(成熟性)
●保守性(変更性、安定性、試験性)

関連する作法
B-21 サードパーティ製品やオープンソースソフトウェアなどと組み合わせて開発するときは、事前に十分調査・解析・評価する(P129参照)

30 Part A ●設計コンセプトを決める
A-8 ● COTS (Commercial Off The Shelf) 製品を安易に利用しない 31

図1-2：作法の読み方

①場面による分類

組込みソフトウェアの設計は開発の様々な場面にまたがっています。本書では設計対象の範囲により場面を4つに分類し(後述)、作法をそれぞれに振り分けました。

②作法名称

作法の内容を端的に表現した名称です。

③作法概要

作法で何を行うかを簡潔にまとめています。

④メリット

作法を実行することにより得られるメリットです。

⑤留意点

作法を適用する上で注意が必要となる点です。デメリットもこの中に含まれます。

⑥解説

作法の背景や基本的な考え方、適用方法などについて説明します。

⑦例

作法の適用例です。出来るだけ「問題の分析・解決を作法を通じて実施する」というストーリーで提示しています。

⑧関連する品質特性

作法との関連が深い品質特性（後述）を挙げました。作法の適用により向上するものだけでなく、逆に低下してしまうようなトレードオフの関係にある特性も含まれていることに注意が必要です。

⑨関連する作法

作法との関連が深い他の作法を挙げました。ソフトウェアの設計においては、異なる設計場面によく似た作法が存在することが珍しくありません。このような共通のパターンを理解することにより、応用の幅が大きく広がります。

2 品質のとらえ方

2.1 品質特性

ソフトウェアの特性が開発の目的にどれだけ適合しているかを表すのが「品質」です。設計はソフトウェアの品質に決定的な影響を及ぼします。開発したソフトウェアが要求を満たすことが出来ない場合、その要因はしばしば「バグ」と呼ばれます。ソフトウェア・エンジニアリングの世界では、ソフトウェアの製品としての品質はより広い概念でとらえられています。このソフトウェア製品の品質概念を整理したものが、ISO/IEC9126-1 であり、これを JIS 化したものが JIS X 0129-1[注3]です。

● JIS X 0129-1 とソフトウェアの品質

JIS X 0129-1 では、ソフトウェア製品の品質に関わる特性(品質特性)に関しては、「外部及び内部品質」として「機能性」「信頼性」「使用性」「効率性」「保守性」「移植性」の 6 つの特性を規定しており、さらにそれぞれ幾つかの副特性に分類されています。

また、利用時の品質として「有効性」「生産性」「安全性」及び「満足性」の 4 つの特性を規定しています(副特性は規定されていません)。利用時の品質はソフトウェアを利用する際の環境全体の中で利用者から見た品質です。従って、ソフトウェアそのものの特徴付けである外部及び内部品質とは視点が異なりますが、やはりソフトウェアの設計に深い関わりがあります。

JIS X 0129-1 の品質特性の定義のうち外部及び内部品質の品質特性及び品質副特性と本書が考える「設計の品質」の関係を表 1-1 に、利用時の品質特性を表 1-2 に示します。

注 3：現在、ISO/IEC 25000(SQuaRE、JIS X 25000)シリーズの JIS X 25010 として、現在改訂作業が進められています。

品質特性 (JIS X 0129-1)		品質副特性 (JIS X 0129-1)		設計の品質
機能性	ソフトウェアが、指定された条件の下で利用されるときに、明示的、及び暗示的必要性に合致する機能を提供するソフトウェア製品の能力。	合目的性	指定された作業、及び利用者の具体的目標に対して適切な機能の集合を提供するソフトウェア製品の能力。	要求を満たす機能が提供されている。
		正確性	必要とされる精度で、正しい結果もしくは正しい効果、または同意出来る結果もしくは同意出来る効果をもたらすソフトウェア製品の能力。	機能の作り込みが正しく行われている。
		相互運用性	1つ以上の指定されたシステムと相互作用するソフトウェア製品の能力。	他システムとのインターフェースが正しく機能している。
		セキュリティ	許可されていない人またはシステムが、情報またはデータを読んだり、修正したりすることが出来ないようにし、同時に許可された人またはシステムが、情報またはデータへのアクセスを拒否されないようにするために、情報またはデータを保護するソフトウェア製品の能力(JIS X 0160 : 1996)。	システムが適切に保護されている。
		機能性標準適合性	機能性に関連する規格、規約または法律上、及び類似の法規上の規則を遵守するソフトウェア製品の能力。	
信頼性	指定された条件下で利用するとき、指定された達成水準を維持するソフトウェア製品の能力。	成熟性	ソフトウェアに潜在する障害の結果として生じる故障を回避するソフトウェア製品の能力。	使い込んだときのバグの少なさ。
		障害許容性	ソフトウェアの障害部分を実行した場合、または仕様化されたインターフェース条件に違反が発生した場合に、指定された達成水準を維持するソフトウェア製品の能力。	バグやインターフェース違反などに対する許容性。
		回復性	故障時に指定された達成水準を再確立し、直接に影響を受けたデータを回復するソフトウェアの能力。	異常時の復旧しやすさ。
		信頼性標準適合性	信頼性に関連する規格または規約を遵守するソフトウェア製品の能力。	
使用性	指定された条件の下で利用するとき、理解、習得、利用出来、利用者にとって魅力的であるソフトウェア製品の能力。	理解性	ソフトウェアが特定の作業に特定の利用条件で適用出来るかどうか、及びどのように利用出来るかを利用者が理解出来るソフトウェア製品の能力。	利用者からみて分かりやすいか。
		習得性	ソフトウェアの適用を利用者が習得出来るソフトウェア製品の能力。	利用者が早く使えるようになるか。
		運用性	利用者がソフトウェアの運用、及び運用管理を行うことが出来るソフトウェア製品の能力。	利用者が容易に使えるか。
		魅力性	利用者にとって魅力的であるためのソフトウェア製品の能力。	利用者が魅力を感じるか。
		使用性標準適合性	使用性に関連する規格、規約、スタイルガイドまたは規則を遵守するソフトウェア製品の能力。	
効率性	明示的な条件の下で、使用する資源の量に対比して適切な性能を提供するソフトウェア製品の能力。	時間効率性	明示的な条件の下で、ソフトウェアの機能を実行する際の、適切な応答時間、処理時間、及び処理能力を提供するソフトウェア製品の能力。	処理時間に関する効率性。
		資源効率性	明示的な条件の下で、ソフトウェア機能を実行する際の、資源の量、及び資源の種類を適切に使用するソフトウェア製品の能力。	資源に関する効率性。
		効率性標準適合性	効率性に関連する規格または規約を遵守するソフトウェア製品の能力。	

品質特性（JIS X 0129-1）		品質副特性（JIS X 0129-1）		設計の品質
保守性	修正のしやすさに関するソフトウェア製品の能力。	解析性	ソフトウェアにある欠陥の診断または故障の原因の追求、及びソフトウェアの修正箇所の識別を行うためのソフトウェア製品の能力。	設計の理解しやすさ。
		変更性	指定された修正を行うことが出来るソフトウェア製品の能力。	設計の修正しやすさ。
		安定性	ソフトウェアの修正による、予期せぬ影響を避けるソフトウェア製品の能力。	修正による影響の少なさ。
		試験性	修正したソフトウェアの妥当性確認が出来るソフトウェア製品の能力。	修正した設計に基づく実装のテスト、デバッグのしやすさ。
		保守性標準適合性	保守性に関連する規格、または規約を遵守するソフトウェア製品の能力。	
移植性	ある環境から他の環境に移すためのソフトウェア製品の能力。	環境適応性	ソフトウェアにあらかじめ用意された以外の付加的な作法、または手段無しに指定された異なる環境にソフトウェアを適応させるためのソフトウェア製品の能力。	異なる環境への適応のしやすさ。 ※標準規格への適合性も含む。
		設置性	指定された環境に設置するためのソフトウェアの能力。	インストールしやすさ。
		共存性	共通の資源を共有する共通の環境の中で、他の独立したソフトウェアと共存するためのソフトウェア製品の能力。	プロトコル等共通の手順の遵守。
		置換性	同じ環境で、同じ目的のために、他の指定されたソフトウェア製品から置き換えて使用することが出来るソフトウェア製品の能力。	明確に定義された外部仕様を遵守。
		移植性標準適合性	移植性に関連する規格または規約を遵守するソフトウェア製品の能力。	

表1-1：外部及び内部品質

品質特性	
有効性	利用者が指定された利用の状況で、正確かつ完全に、指定された目標を達成出来るソフトウェア製品の能力。
生産性	利用者が指定された利用の状況で、達成すべき有効性に対応して、適切な量の資源を使うことが出来るソフトウェア製品の能力。
安全性	利用者が指定された利用の状況で、人、事業、ソフトウェア、財産または環境への害に対して、容認出来るリスクの水準を達成するためのソフトウェアの能力。
満足性	指定された利用の状況で、利用者を満足させるソフトウェア製品の能力。

表1-2：利用時の品質特性

2.2　品質特性と作法の考え方

本書で提供する作法については、前述の品質特性の中からそれぞれの作法に関連の深いものを「関連する品質特性」の項目に記載しています。これにより、目標とする製品の品質特性に合わせて適用する作法を選択する、あるいは作法を利用するにあたって影響の及ぶ品質特性を見極めることが出来ます。

このようにして品質特性との関係を意識しながら設計を進めることにより、目標とする品質をクリアすることが容易になります。

3　本書の構成

導入部に相当する本編（「解説」）の後、74 個の作法を以下の 4 つの設計場面に対応するパートに分類して収録しています。

◆ Part A. 設計コンセプトを決める

設計思想や方針など、設計を始めるにあたって決めておく・意識しておく必要のある作法です。

例えば系列製品のエラー処理について個々の製品の設計に先立ってどのようなエラーが起き得るか分析し、それぞれの扱いを規定しておきます（作法 A-14）。

こういった道具立てを設計の様々な場面で利用することにより、一貫した見通しの良い開発が可能になります。

◆ Part B. システムレベルの設計の工夫

サブシステムへの分割や OS、タスク分割といった組込みソフトウェアを組み上げるための枠組みに関する作法です。適切でバランスの良い枠組みを設定することにより、個々の部品に相当する部分の開発を見通し良く行うことができます。

例えば、システムで扱うデータが性質の違うものに分類できる場合、この分類に合わせて処理を行うサブシステムの割り当てを行うことにより、性質の異なるデータを同時に扱うことによる困難を回避できます（作法 B-8）。

こういった作法は、プロセスでは主に「システム・アーキテクチャ設計」や「ソフトウェア・アーキテクチャ設計」に深い関連があります。

◆ Part C. ミドルウェア・ネットワークスタック・ライブラリレベルでの工夫

枠組みと個々のデバイスの間の様々なサービスを行う部分での作法です。何らかの既存のシステムを利用する場合の工夫なども含まれています。

例えば、データベースを利用する際に効率を意識したテーブル設計を行うことにより、使用するメモリ容量やアクセスに要する時間などのオーバーヘッドを最小限にとどめることができます（C-15）。

こういった作法は、プロセスでは主に「ソフトウェア・アーキテクチャ設計」や「ソフトウェア詳細設計」に深い関連があります。

◆ Part D. システムで扱う周辺デバイス操作に関する工夫

デバイスやプロセッサなどに関わる作法です。ハードウェアに直接関係するためソフトウェア技術者が必ずしも得意としない分野ですが、ひとたび不具合が生じると解決に手こずることが珍しくありません。先手を取って注意事項を網羅しておくことは非常に有効です。

例えば、リセット時のデバイスのポートの状態をあらかじめ調べてからドライバーを設計することにより、予期しない動作を防ぐことが出来ます（D-2）。

こういった作法は、プロセスでは主に「ソフトウェア詳細設計」に深い関連があります。

Part A

設計コンセプトを決める

A-1 イベントドリブン型のシステムの設計では状態遷移図と状態遷移表を併用する

作法概要

●まず最初に、状態遷移図を書いて、システムの主要な動作を検討する。
●次に、状態遷移表に変換し、抜けや漏れをチェックする。
●状態の粒度や階層にばらつきのないようにレビューする。

メリット

●状態遷移図を使うと処理のイメージを把握しやすい。
●状態遷移表を使うと設計の抜けや漏れを発見しやすい。

留意点

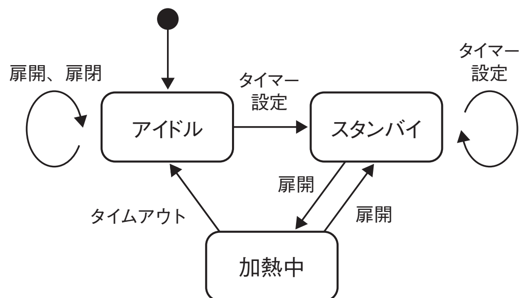
①状態と処理を混同しない(フローチャートとの違いを理解する)。
②図と表は相互に変換可能である。自動的に変換するツールを利用すれば、整合性維持のための手間を減らすことが出来る。
③種々の条件の組み合わせなどにより状態が変化するシステムは、変化のきっかけをイベントとみなして、イベントドリブン型のシステムとして扱うことが出来る。

●解説

注目しているシステムへの外部からの働きかけ(デバイスからの入力など)を、一般に、「イベント」と呼ぶ。イベントをきっかけにしてシステム内部の状態が変化し、それに伴って様々な処理を行う処理方式をイベントドリブン(あるいはイベント駆動)方式と呼ぶ。イベントと状態及び処理との関係に関して図示したものが状態遷移図、表にまとめたものが状態遷移表である。この2つは等価であるが、状態遷移図には全体の動作が把握しやすい、状態遷移表には抜けや漏れを把握しやすいといったメリットがそれぞれあり、両方を併用することが望ましい。

状態遷移図の書き方には様々な流儀があるが、図A-1にはUML 2.2で規定されている表記法によるものを示す。UMLの状態遷移図は、最近では、入れ子状の階層構造などを表現出来るよう拡張されている。

表A-1と表A-2には状態遷移表の代表例を2種類示す。行に状態、列にイベントをとり、ある状態でイベントが起きた場合の遷移先の状態を表に示したもの(表A-1)と、行に現在の状態、列に遷移先の状態をとり、遷移を引き起こすイベントを表に示したもの(表A-2)である。いずれも、設計上、遷移が起こらない組み合わせは“—”で示している。



図A-1：状態遷移図の例

イベント＼状態	タイマー設定	扉　開	扉　閉	タイムアウト
アイドル	スタンバイ	アイドル	アイドル	—
スタンバイ	スタンバイ	—	加熱中	—
加熱中	—	スタンバイ	—	アイドル

表A-1：状態遷移表の例①

次状態＼現在状態	アイドル	スタンバイ	加熱中
アイドル	扉開、扉閉	タイマー設定	—
スタンバイ	—	タイマー設定	扉閉
加熱中	タイムアウト	扉開	—

表A-2：状態遷移表の例②

例　電子レンジ

主要な機能は、扉の内側にあるタイマーを設定して扉を閉じると、設定した時間だけ加熱することである。この状態遷移図は図A-1に示した通りである。

もし、仮に、スタンバイ状態に関して、タイマー設定イベントによるスタンバイ状態への遷移を書き落としていたとする。この状態遷移図を状態遷移表に変換すると、表A-3になる。この表をチェックすると、スタンバイ状態におけるタイマー設定イベントが無視されていることが分かる。

イベント＼状態	タイマー設定	扉　開	扉　閉	タイムアウト
アイドル	スタンバイ	アイドル	アイドル	—
スタンバイ	—	—	加熱中	—
加熱中	—	スタンバイ	—	アイドル

表A-3：状態遷移表の例③

関連する品質特性

- ●機能性（合目的性、正確性）
- ●保守性（解析性、変更性、安定性、試験性）

A-2 正常系を設計すると同時に異常系も併せて設計する

作法概要

正常系と異常系を同時に視野に入れて設計を進める。

メリット

異常系の処理の抜けや漏れを防止出来る。

留意点

①システムによっては、異なる経路を通る電波同士の干渉などによる電波障害などの物理的な事象も考慮する。
②搭載するハードウェア(CPU)の制約を考慮する。
③ドライバなどのファームウェアの特徴を考慮する。
④過去のトラブル事例を参照し、異常系に漏れがないようにする。
⑤ FMEA(故障モード影響解析)や FTA(フォルトツリー解析)、HAZOP(プロセスに注目したリスク分析)などの手法を利用する。
⑥非同期動作の場合、確実にタイムアウト処理が行われるようにする。
⑦リトライを行う場合はレイヤーごとのリトライ回数を把握し、適切な範囲に収まるよう設計する。

●解説

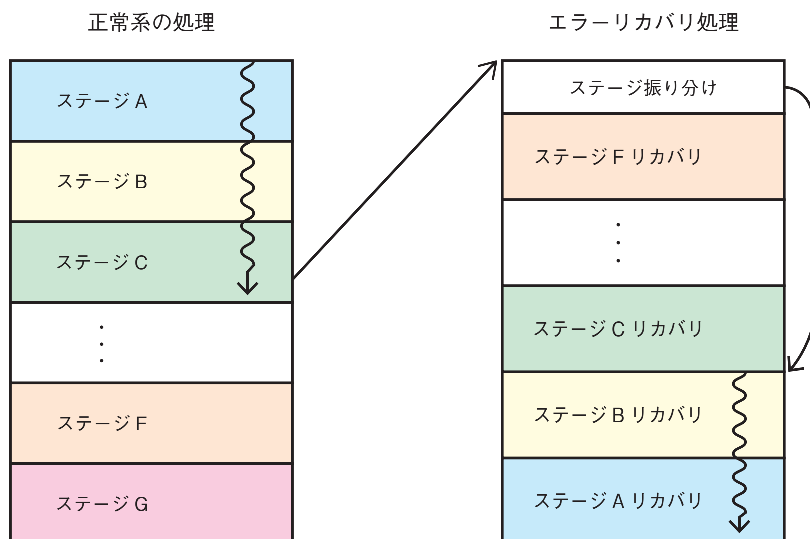
エラー処理を異常系、それ以外の部分を正常系とそれぞれ呼ぶことにする。まず正常系のみに注目して設計し、異常系の処理はあとから追加するという方法をとると、異常系の処理に漏れや抜けが生じやすい。従って、異常系と正常系の両方を同時に視野に入れ、次の事項を考慮しながら設計をすすめる必要がある。

①システムでどのような異常動作があるかをあらかじめ洗い出し、周辺デバイス動作(ハードウェア的、ソフトウェア的、環境的など様々な側面)などへの影響を考える。
②共通化出来る異常処理はまとめる。
③異常処理は、システムレベルとコンポーネントレベルとにレベルを分ける。
④異常発生時のリカバリ方法(戻り先、リカバリの順序など)を考える。
⑤異常の検知(ウォッチドッグなど)と記録方式(ログなど)を考える。

例 電波時計

まず、システムが正常に動作していることを前提とした設計を行った。次に、標準電波が受信できないなど、発生し得る例外的な状況を考えて、その検出手段と対処法を順次追加していくようにした。結果として異常状態を網羅的に洗い出すことが出来ず、動作を安定させることが出来なかった。

そこで設計方針を見直すことにした。まず、正常系の処理ステージを分割するときに、異常時のリカバリ処理の単位と対応するようステージＡからステージＦに分割した。これに沿って正常系を組み立て、エラーを検出すると異常系に実行を移すように設計した。こうすると、正常系の処理ステージＣで異常検出した場合、ステージＢリカバリ、ステージＡリカバリの逆順で逐次的に対応するリカバリ処理を実行出来るなど全体の見通しが良くなり、品質を向上させることが出来た。



図A-2：異常時のリカバリ処理の単位と対応するよう正常系をステージ分割

関連する品質特性

機能性（合目的性、正確性）

関連する作法

A-14　シリーズ製品では エラー処理を統一する（P51 参照）

A-3 ソフトウェア部品は単機能になるように設計する

作法概要

ソフトウェア部品は、粒度をそろえ、単一の機能や責務(役割)を提供するように設計する。

メリット

ソフトウェアの見通しが良くなり、求める機能を実現しやすく、また、変更やテストが容易になる。

留意点

①クラスや構造体に情報を追加するときには、そのクラスや構造体の目的に合致しているか十分チェックする。
②他の粒度や同一の粒度の他の部分など、全体のバランスを取りながら、部品の機能や責務を決める。
③オブジェクト指向設計(OOSE)のロバストネス分析などを活用する。
④部品には安易に内部状態を持たせない(部品間の依存関係の抑制とは、しばしばトレードオフの関係にある)。
⑤再帰的な処理を行う場合は、無限ループなど想定外の振舞いをしないよう十分注意する(原則として再帰呼び出しを使わない)。

●解説

ソフトウェア部品を設計するときは、対象となる粒度に合う、単一の目的(機能、責務)を実現する部品になるように心がける。その理由は、要求される機能を実現しやすくなり、変更やテストが容易になるからである。具体的には、次の事項を心がける。

①機能がひと目で把握出来るようにする。
②複数の機能・責務が含まれてしまった場合は分割する。
③使い回されている関数があれば、共通部分を抽出して部品化する。
④既存のソースコードのモジュール構造を分析し、結合度や凝集度を改善する。

例 携帯電話機のアプリケーション

電卓機能と割り勘計算機能は共通点が多いため共通の関数とし、呼出し時の引数でモードを指定して動作を切り替えるよう設計した。ところが、電卓への機能追加により、割り勘計算に思わぬ副作用が出るなど保守が困難であることが分かった。

そこで、2つの機能を個別のアプリケーションとして再設計を行い、共通点を括り出して部品化した。これにより見通しが向上し、保守が容易になった。

関連する品質特性

- 機能性（合目的性、正確性）
- 保守性（解析性、変更性、安定性、試験性）

A-4 ソフトウェアの再利用は設計レベルで行う

作法概要

●安易にコードだけを再利用しない。
●再利用が可能かどうかを、設計レベルで十分に見極める。

メリット

異常系での抜けや漏れを防ぐことが出来る。

留意点

①設計時には再利用についても考慮し、設計書に再利用に関する記述を行う。
②プロトタイプのコードは、機能や異常系に対する考慮が不完全なので、それを使用して得られた経験を反映して再設計し、実装を行うようにする。
③再利用の際には、あらかじめ制約事項を分析定義しておく。

●解説

安易にコードだけを再利用すると、異常系での抜けや漏れを見落とし、ソフトウェアの品質が低下しがちである。

コードを再利用したい場合は、そのまま利用することが可能かどうかを、設計レベルで十分に見極める必要がある。

例 電子レンジ

ユーザインターフェースを確認するためにプロトタイプを作成した。このプロトタイプのコードに必要な処理を追加して、製品用のコードを作成した。その結果、タイマが壊れた場合の処理が正しく実装されず、加熱が止まらなくなる不具合が発生した。

そこで、プロトタイプを参考に設計書を作成し検討を行い、これに基づいて実装するよう方針を変更した。これにより、プロトタイプの欠点を継承することなく、製品用のコードを作成することができた。

関連する品質特性

機能性(合目的性、正確性)

A-5 実装は設計のあとに行う

作法概要

開発プロセスとしての設計や実装を明示的に分離する。

メリット

ソースコードやドキュメントの品質をコントロールしやすい。

留意点

①実装しているのが製品かプロトタイプかを明確に区別する。
②プロセスの責務(目的、役割)を明確に意識したマネジメントが必要になる。

●解説

ソフトウェア開発において設計と実装を同時に進行させると、ソースコードの品質が保証できない、更には、ドキュメントはソースコードと合わないということになりやすい。従って、ソフトウェアの品質を確保するには、設計や実装という開発プロセスを意識的に分離する必要がある。それぞれのプロセスで何をするかを明確に意識し、次の事項などを心がけながら、開発を進める。

①「とりあえずソースコードに落とす」ことをしない。
②「デバッグ」と称して、なしくずし的に設計内容を変更しない。
③設計変更は設計プロセスに戻り、その結果をもとに実装し、ソースコードが設計に適合していることはソースコードレビューで確認する。

例 時計

時刻の表示部分の「：」が表示されないことが、単体テストで明らかになった。調査したところ、タイミングに起因する不具合であることが分かり、設計プロセスでのミスと判断し、設計書の修正とレビューを行った。その後、修正された設計書に基づいて実装の修正とレビューを行い、単体テストで「：」が表示されることを確認した。

関連する品質特性

機能性(合目的性、正確性)

関連する作法

A-13　シリーズ製品では違いを意識して設計する(P47 参照)

A-6 テストを考慮して設計する

作法概要

●テストを容易に網羅的に行えるようにソフトウェアを設計する。
●テストを複雑にする複数モードの組み合わせや多重割込み、処理パターンなどを避ける。

メリット

●テストの有効性をあらかじめ確保することが出来る。
●整理された見通しの良い設計内容にすることが出来る。

留意点

①階層構造に分割し、インターフェースを明確化する。
②部品化や集約を行う。
③シミュレーションしやすいように抽象化する。

●解説

テストを意識して設計することによって、テストがやりやすくなるだけでなく、設計そのものの見通しがよくなる。テストを意識した設計となるためには、次の事項などに留意して設計を進める。

①依存関係のある複数のモードの組み合わせは避ける。
②多重割込みは避ける。
③割込みタイミングに影響を受けやすい処理パターン(read-modify-write など)は避ける。
④網羅的なテストが難しいパターンは、可能な限り避ける。

例 店員側と顧客側の両方にタッチパネルを備える支払いシステム

それぞれの入力を独立に受け付けるように設計した。同時に入力が行われる場合など、タイミングによってシステムの振舞いが変わる可能性があるので、その分、テスト回数を増やして、検証を実施した。実際に運用が始まると、店員による支払処理と顧客による取消処理が同時に行われた場合に、タイミングによっては顧客が支払わなくても支払ったことになるケースが発生してしまった。

そこで設計を変更することにした。排他制御などの方針を明確にし、これに基づいて入力を一括して処理するモジュールを作成した。これにより、支払処理が確実に行えるようになった。

関連する品質特性

保守性(解析性、変更性、安定性、試験性)

A-7 ソフトウェアの要素には機能を表す名前を付ける

作法概要

- ●ソフトウェアの要素の責務(機能、目的)にふさわしい名前を付ける。
- ●ソフトウェアの要素の機能が名前に対して適切かどうかをチェックする。

メリット

ソフトウェアの見通しが良くなる。

留意点

開発プロジェクトまたはチームごとに命名規則を策定しておく。

●解説

モジュールや、関数、クラスというソフトウェアの要素を設計するとき、各要素にその責務(機能、目的)に対応しない名前を付けると、そのソフトウェアの保守が難しくなる。責務と名前とが互いに対応するように、次の点を心がける。

①保守・拡張していくうちに、名前と機能の関係が崩れがちになるので注意する。
②「○○マネージャ」といったあいまいな名前の要素は、機能が集中し過ぎているようであれば分割して、機能にふさわしい名前を付け直す。
③共用の関数や処理などは、使いやすい名前を付けるようにとくに気を付ける。
④名前に英語を使う、ローマ字を使う、などの方針を決める。
⑤名前の衝突が見つかった場合、設計に問題がないか注意して見直す。

例 ②の例として、マルチメディアプレーヤ

ユーザによる設定を記録し、各モードの再生をコントロールするためのモジュールに設定マネージャという名前を付けたところ、モードごとのコードが混在して、設定マネージャの保守が困難になってしまった。

そこで、モードによって振舞いの変わる部分を抽出してそれぞれを独立のモジュールとし、モードによって呼び分けるように設計と実装を整理した。その結果、ソフトウェアの構造が見やすくなり、保守が容易になった。

関連する品質特性

保守性(解析性、変更性、安定性、試験性)

A-8 COTS（Commercial Off-The-Shelf）製品を安易に利用しない

作法概要

既製のソフトウェアやオープンソースソフトウェアを利用する場合は、事前に十分な動作検証を行う。

メリット

●市場での想定外の動作や不具合が防止出来る。
●システムテストの効率化が可能となる。
●設計手戻りの防止に繋がる。

留意点

①オープンソースソフトウェアの場合は、ソースコードの静的解析なども有効である。
②ソフトウェアは開発元の必要により変化していくため、バージョンの追従方針をあらかじめ決めておく必要がある。
③将来的な拡張性や保守性も考慮する必要がある。
④オープンソースソフトウェアの使用にあたっては、例外を含む網羅的で十分なストレステストにより品質を担保する。

●解説

既製のソフトウェア製品やオープンソースを利用することは、システム開発のコスト削減と開発期間短縮に有効な手段である。ときには、システムの制約上、他社製品を組み込んでシステムを構成しなければならない場合もある。しかし、それらのソフトウェアは、内部の設計構造や振舞いが完全に把握出来るとは限らない。その結果、利用したシステムにおいて、想定外の動作や不具合を起こすことがある。利用に際しては、次の点なども考慮し、十分な検証が必要である。

①システムに組み込んで使用する場合に、期待通りの動作をするかどうか。潜在化していた条件が新たな制約になったり、期待通りの性能が出ない可能性はないかを検証する。
②他部門での利用実績があれば、その検証結果や不具合事例などを確認する。
③一般向けに提供されているソフトウェア製品などは、第三者による動作検証が実施されている場合があるので、それらの情報を利用する。
④ソフトウェアコードだけではなく、開発環境に関しても、バージョンの違いやツールの組み合わせ、実績のない新機能などに注意する。

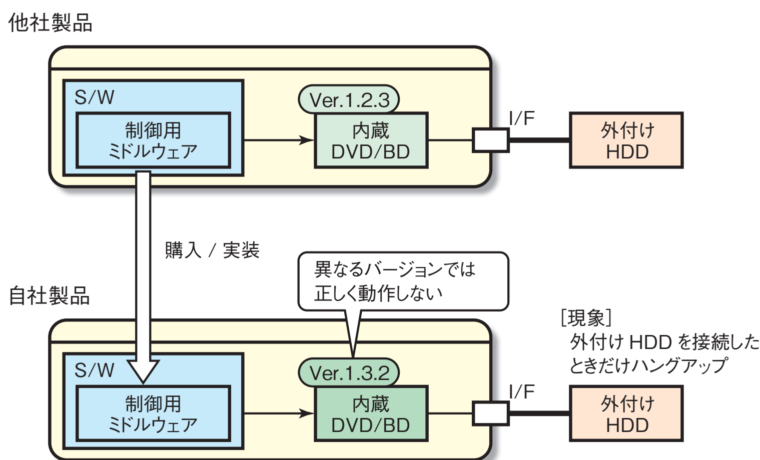
例 BD プレーヤのような AV 機器

他社製品で使用している制御用ミドルウェアを購入し、自社製品に実装した。通常の使用状態では問題なく動作したが、外付け HDD を接続すると、ある特定の条件で時々ハングアップすることがあった。

不具合原因を調べたところ、内蔵 DVD/BD ドライブからの信号遅延の影響で、状態遷移が不正と

なることが分かった。更に、実績ある他社製品を調べてみると、内蔵 DVD/BD ドライブのバージョンが異なっていることが分かった。

そこで、タイミングが異なっても正しく状態遷移するようコードを修正し、他に同様なケースがないことを確認するため、制御バスタイミングに重点を置き、網羅的にテストを実施した。



図A-3：自社製品に他社のCOTS製品を実装

関連する品質特性

●機能性（正確性、相互運用性）
●信頼性（成熟性）
●保守性（変更性、安定性、試験性）

関連する作法

B-21 サードパーティ製品やオープンソースソフトウェアなどと組み合わせて開発するときは、事前に十分調査・解析・評価する（P129 参照）

A-9 1関数1責務(要素機能)にする

作法概要

関数で処理する責務を明確にし、1関数1責務(要素機能)とする。

メリット

不具合対応が容易になり、仕様変更の影響を受けにくくすることが出来る。

留意点

①仕様変更や機能追加に安易に対応し、責務(要素機能)が複数にならないようにする。
②責務(要素機能)の粒度の決め方に関して、プロジェクト内で取り決め、統一を図る。

●解説

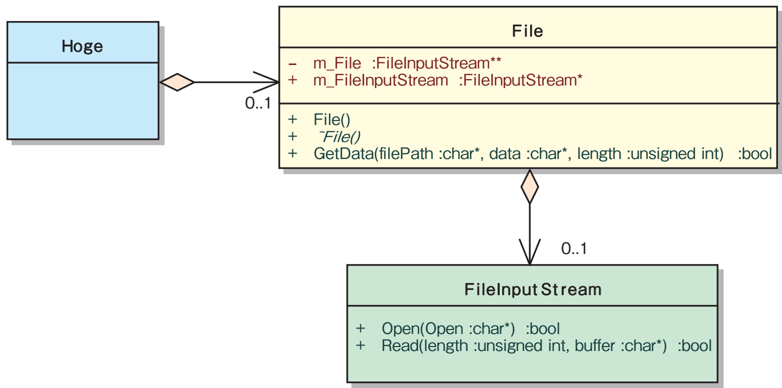
1つの関数に複数の責務(要素機能)を持たせた実装は、よく見受けられる。しかし、このような実装を行うと、コードの可読性が悪くなり、そのために不具合箇所の特定を難しくし、不具合対応によるデグレード(変更が品質を低下させる事態)の可能性も高める。

更に、複数の責務(要素機能)を持った関数は、多くの関数と関連を持つことになるため、仕様変更のときにコード変更が集中する傾向が高い。そのために、変更による影響を大きく受け、保守性を低下させる。

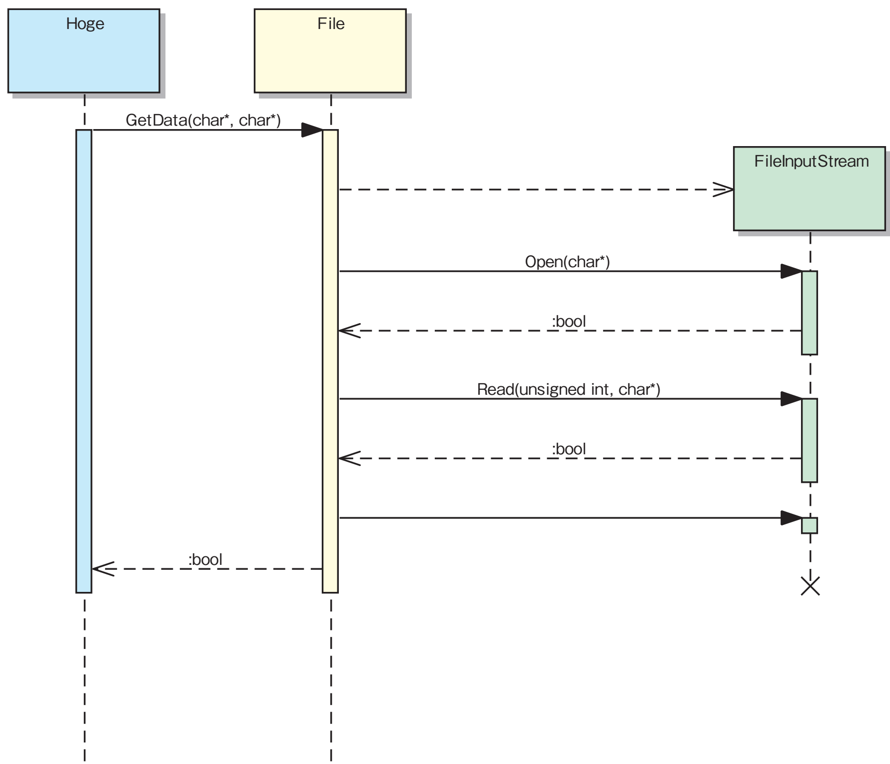
解決策は、1関数1責務(要素機能)にすることで保守性を高め、ソフトウェアの信頼性を向上させることである。

1関数1責務(要素機能)にする手順としては、パッケージ、モジュール・クラス、関数といった階層ごとに責務が存在するため、上の階層から順次、トップダウンで整理していくやり方が有効である。そのときに、関数の責務(要素機能)をすべて関数名に書き出すようにすると、その関数が複数の責務(要素機能)を所有しているかどうかの判別が容易になる。

例1 ①悪い例：GetData と関数を命名し、ファイルのオープンとデータの取得を実装する。



図A-4：悪い例の論理構成

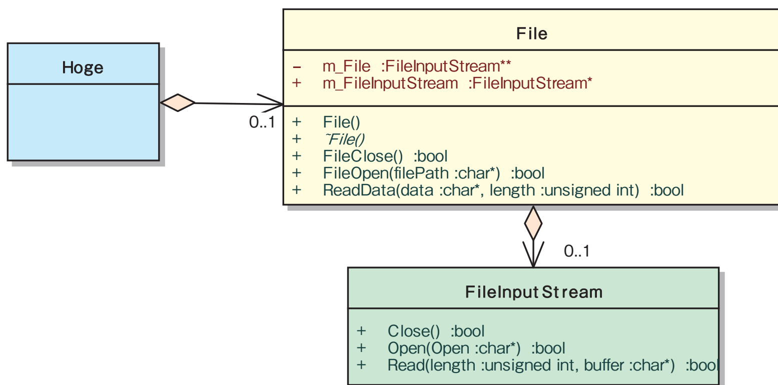


図A-5：悪い例の動的構成

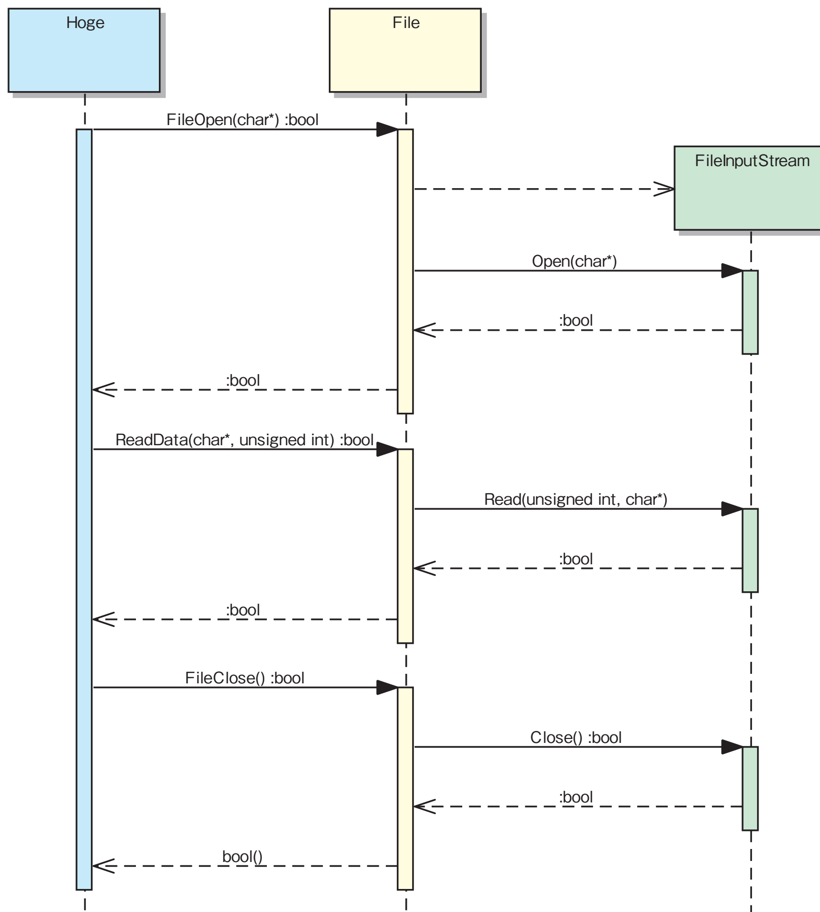
```
bool File::GetData(const char *filePath, char *data, const unsigned int length){
    if(NULL == data || NULL == filePath || 0 == length)
    {
        return false;
    }

    m_File = new FileInputStream();
    if(NULL == m_File)
    {
        return false;
    }

    bool ret = m_File->Open(filePath);
    if(false == openRet)
    {
        delete m_File; m_File = NULL;
        return false;
    }

    ret = m_File->Read(length, data)
    return ret;
}
```

図A-6：悪い例のコード

②良い例：FileOpen、ReadData と関数を命名し、ファイルをオープンする機能とファイルからデータを読み込む機能を分離する。

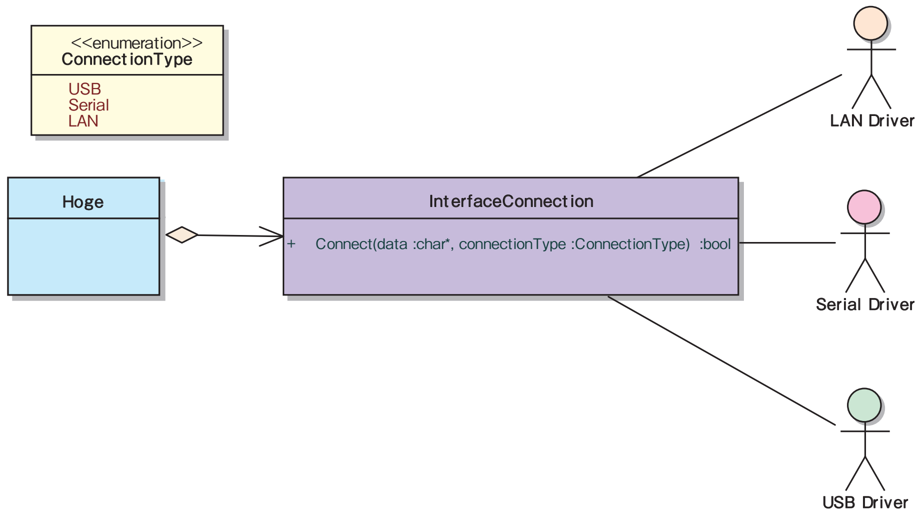


図A-7：良い例の論理構成

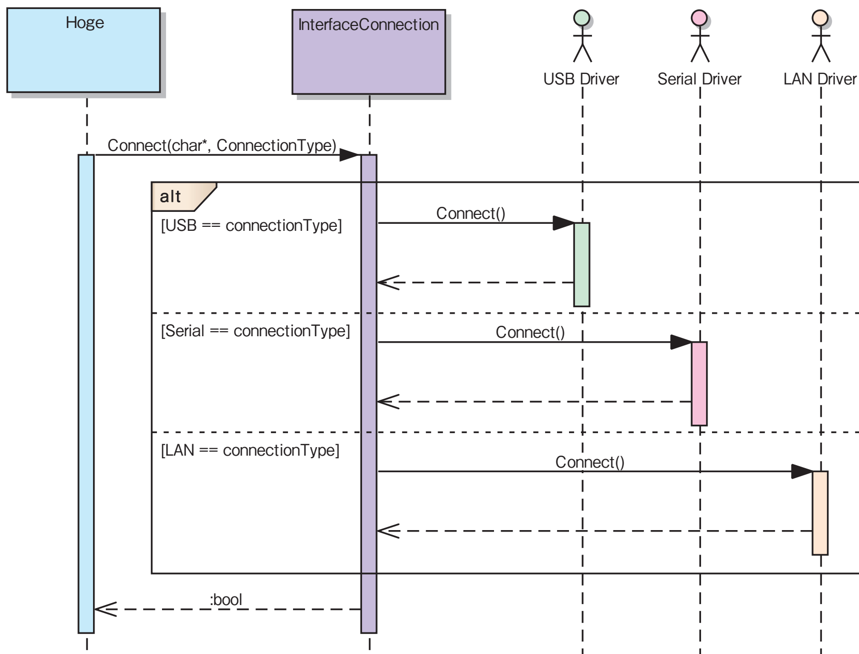


図A-8：良い状態の動的構成

例2 ①悪い例：Connect と関数を命名し、様々な通信(USB、シリアル、LAN など)を 1 つの関数で実装する。

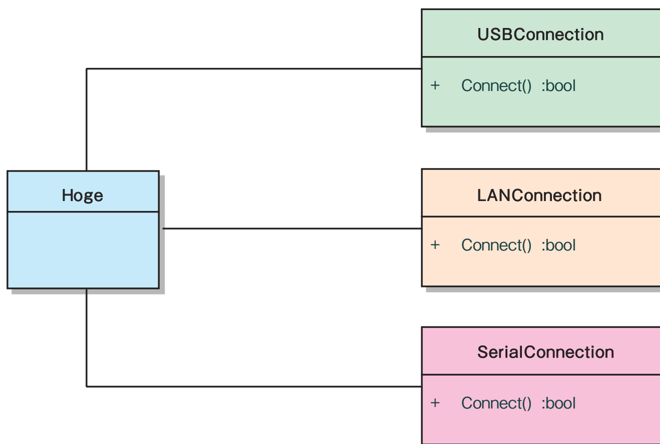


図A-9：悪い例の論理構成



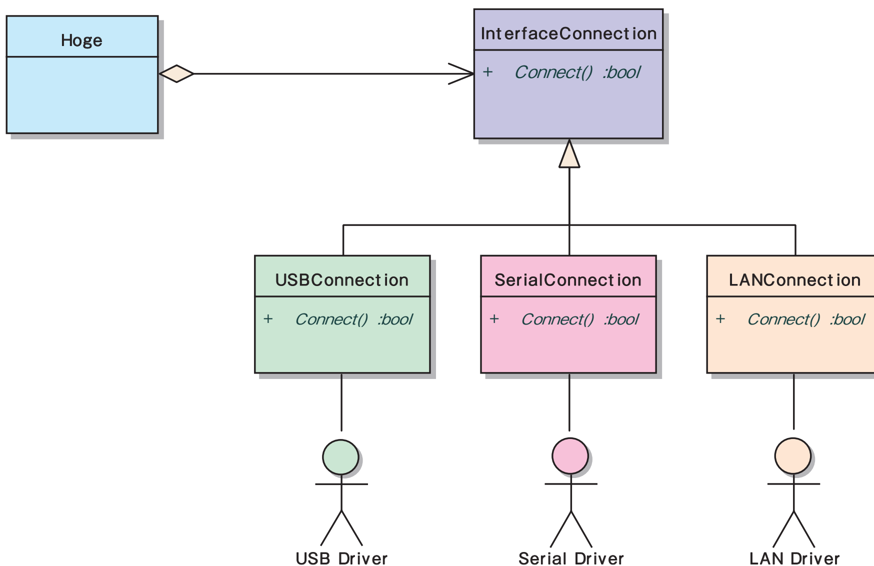
図A-10：悪い例の動的構成

②良い例(1)： 接続方式ごとにファイル / クラスを用意し、各ファイル / クラスで接続形態に適した Connect を実装する。

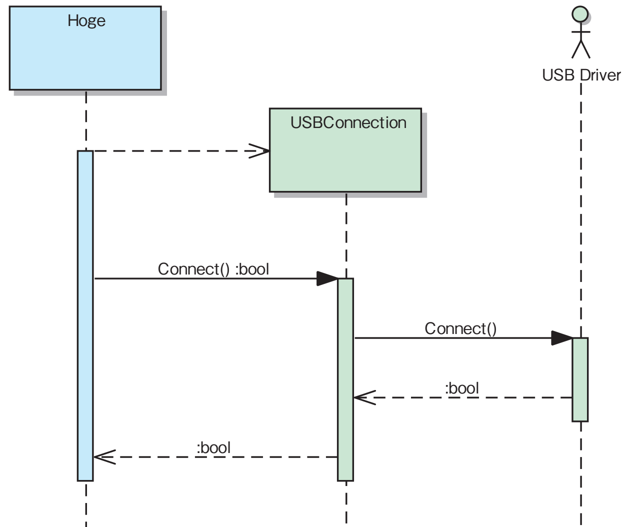


図A-11：良い例の論理構成

③良い例(2)： 共通のインターフェースと通信形態ごとのクラスに分離し、それぞれのサブクラスで接続形態に適した Connect を実装する。



図A-12：良い例の論理構成



図A-13：良い例の動的構成

関連する品質特性

●保守性（解析性、変更性、安定性、試験性）
●安全性

A-10 同期通信ではタイムアウトやタスク生存期間を考慮する

作法概要

- 同期通信では、意図しない動作の停止を避けるためにタイムアウト機能を利用する。
- タイムアウト時には、通信先に通知してから通信要求タスクを終了する。
- タスク間の通信タイミングの分析として、シーケンス図やタイミングチャートを作成し、論理検証を実施する。

メリット

同期通信がエラーのときのメモリアクセス違反を防止することが出来る。

留意点

①ハードウェアが介在するような同期通信を行うときは、ハードウェアなどの仕様の最大値を使うのではなく、最大値＋αをタイムアウト値とするのが望ましい。
②タスクの生成と消滅を行う場合は、依存関係のあるタスクの生存期間も考慮する。

●解説

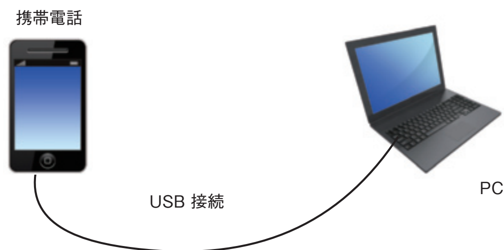
組込みシステムにおいては、限られたリソースを有効活用するために、必要なときのみタスクを生成し、必要がなくなると消滅するように設計することがある。また、同期通信においては、無限に応答を待つとシステムがデッドロックに陥る可能性があるために、タイムアウト処理を考慮することは一般的である。従って、タスク間で同期通信を行う場合、同期通信を要求したタスクが、タイムアウト処理によって、通信先のタスクに通知を行わずに消滅する可能性がある。この場合、通信先のタスクが要求タスクの消滅を知らずに応答を返すと、メモリアクセス違反を引き起こすことになる。

これを回避するためには、タスクの生成・消滅に関する設計方針をプロジェクト内で決めるのが望ましい。設計方針としては、例えば次のような考え方がある。

①タスクの生成・消滅を一元管理するタスクを導入する。
②タスクの生成・消滅の手順をルール化する（例えば、タスク間に依存関係があるものについては、同時に削除されるようにしておくなど）。

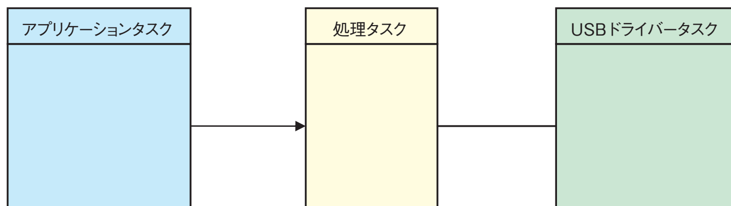
もう１つの回避策として、タスク間で同期通信を行う場合、タスクの生存期間を考慮したシーケンス図やタイミングチャートを作成して、論理検証を行う方法がある。これらの対策を実施することで、ソフトウェアの信頼性・安全性を高めることが出来る。

例 図 A-14 のようなシステム構成



図A-14：システム構成

携帯電話のアプリケーションにおいて、PC との通信を行う場合に、図 A-15 に示すタスク構成をとると仮定する。

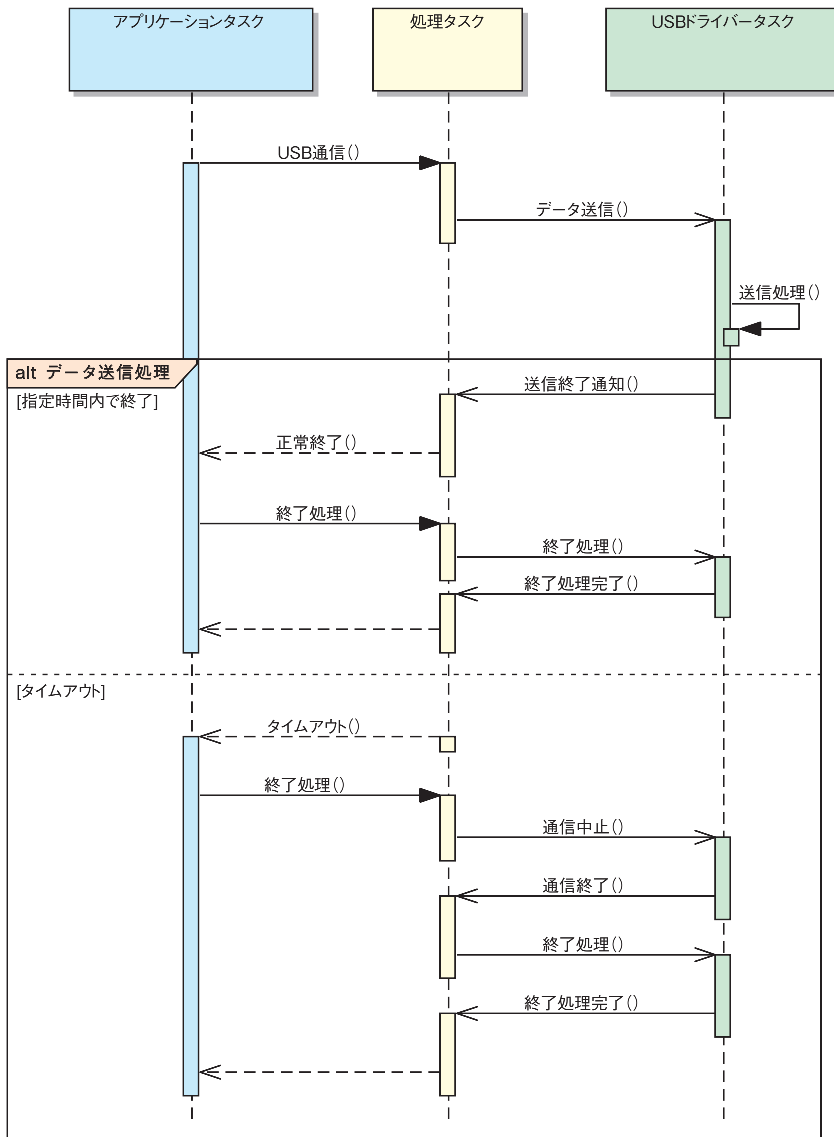


図A-15：タスク構成

アプリケーションタスクは、必要な処理を行うために、処理タスクを生成して同期通信を行う。処理タスクは、PC との USB 通信を開始して必要なデータを入手するために、USB ドライバータスクに通信依頼を行う。その際に、通信応答を無限に待つことによるシステムのデッドロックを避けるために、処理タスクはタイムアウト機能を有効にする。

USB ドライバータスクからの応答がないままタイムアウト時間となると、処理タスクは、アプリケーションタスクにエラーを戻す。アプリケーションタスクは、これによって処理終了を知り、処理タスクを消滅させる。その後に、USB ドライバータスクに PC からの応答が届き、その応答を処理タスクに戻そうとすると、メモリアクセス違反などの不具合が発生する可能性がある。

これを避けるために、図 A-16 に示したシーケンス図やタイミングチャートを作成し、タスクの生存期間やタイムアウト時の処理に起因する不具合が発生しないことを確認しながら設計を進めた。



図A-16：タイミングを考慮したシーケンス図

関連する品質特性

●信頼性（成熟性、障害許容性）
●安全性

A-11 不正値でも補正可能な場合は補正する

作法概要

- 補正可能な正常範囲外のデータを受け取ったときに、正常範囲の標準値に補正し、動作を継続する。
- 正常範囲外の値を受け取ったことを示す内部ログを記録し、問題発生を認識出来るようにする。

メリット

軽微な不具合で、結合テスト時などにおけるシステムダウンや再起動が避けられ、処理を継続することが出来る。

留意点

取得したデータは不正であることは間違いないので、必ずログを分析し、障害として報告する。

●解説

大規模で複数レイヤのサブシステムに分かれており協調動作するようなシステムでは、システムダウンや再起動の影響が大きくなるため、軽微な不具合によって本番稼働時にシステムダウンや再起動を行わないようにする。ここで、軽微な不具合とは、例えば表示上の不具合など、本来の機能要件に影響を与えることの少ない不具合を指す。

システムダウンや再起動を回避するためには、サブシステム間のインターフェースなどで補正可能な正常範囲外のデータを受け取ったときに、正常範囲の標準値に補正し、動作を継続する。ただし、正常範囲外の値を受け取ったことを示す内部ログを記録し、問題発生を認識出来るようにする。

例えば、C 言語による実装を行う場合、if else の最後の処理や switch 文のデフォルトなどで、あり得ないケースに対して一律に abort 処理などのプログラムを異常終了させる処理を行うという方針があり得る。その場合、わずかな表示系のデータの不具合など、軽微な障害でもシステムを終了させてしまい、協調動作する他のサブシステムの評価作業に影響を与えてしまうことがある。もし、不正なデータが補正可能であれば、標準値などに補正してシステムを終了させないようにし、これによって評価作業中止などを回避することが出来る。とくに大型のシステムの場合、一部の表示上の不具合などの理由で評価作業中止が発生すると、評価要員の待機コストが発生し、プロジェクト全体の日程にも影響を与え、大きな問題となる。

例 オフィス機器

大型のオフィス機器においては、各機能の設定を液晶パネルに設定画面を表示して行う。その設定画面の初期値は、初期値を管理するサブシステムから通信で受け取っていた。このサブシステムの不具合により、取得した初期値が正常範囲外となると、設定画面の表示部はこれを検出し、abort 関数を呼び出してシステムを再起動させた。このような処理によって設定画面を表示させると、システムが再起動されてしまい、機器の機能を起動する動作の確認など、その先の作業を一切行うことができなくなった。

取得した初期値が正常範囲外だった場合、これをシステムのデフォルト値に補正しても、システムの動作には全く影響はない。この程度の軽微な不具合がシステム全体の評価作業を止めてしまい、プロジェクト全体の日程に影響を与えた。

関連する品質特性

- ●信頼性（障害許容性）
- ●保守性（解析性）

A-12 イベントを周期的に処理する場合は周期を意識した安易な処理分割をしない

作法概要

- 周期的なイベント処理では、受信したイベントに関連する処理は、ポーリング周期内に完了させる。
- その処理が周期内に完了しない場合でも、安易に処理分割しない。

メリット

保守性(解析性、安定性、試験性)を維持することが出来る。

留意点

ポーリング周期内にどうしても処理が完了しない場合には、設計の見直しを行う。

●解説

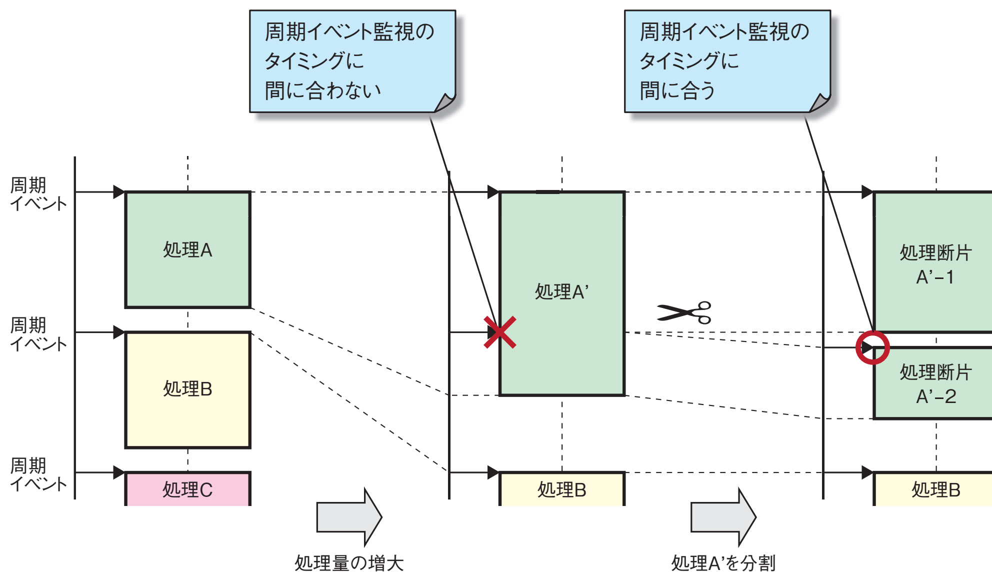
リアルタイムOS(RTOS)を採用しない小規模な組込みソフトウェアにおいては、外部からのイベントを一定周期でポーリングし、受信したイベントに関連する処理を行うことが多い。その実現手段の1つが、メインループである。つまり、1つのプログラムの中でイベントポーリングとイベントに関わる処理を実行し、それらの処理を、ループを構成することによって定期的に繰り返す。このメインループを採用するときには、実行すべき「処理」は、イベントのポーリング周期内に完了するよう設計する。

もし、周期内に「処理」が完了しないケースが発生すると、これは、メインループ方式では、要求される機能の達成が不可能となっていることを意味し、設計の見直しが必要となる。とくに、差分開発を長年行っているソフトウェアに関しては注意が必要である。開発開始当初はソフトウェアの規模が比較的小さく、イベントのポーリング周期内にすべての「処理」の実行が完了していたが、機能が複雑化し、処理量が増大化するにつれ、周期内に「処理」が完了しないケースが発生してくる。この場合には、設計の見直しを行う。例えば、次のような設計変更案が考えられる。

① OSが導入されていなければ、OSを導入し、外部からのイベントを周期的に監視するタスクと、イベントに関連する「処理」を行うタスクを別個のタスクとする。

②タイマ割込みを用いて外部からのイベントを周期的に監視する。

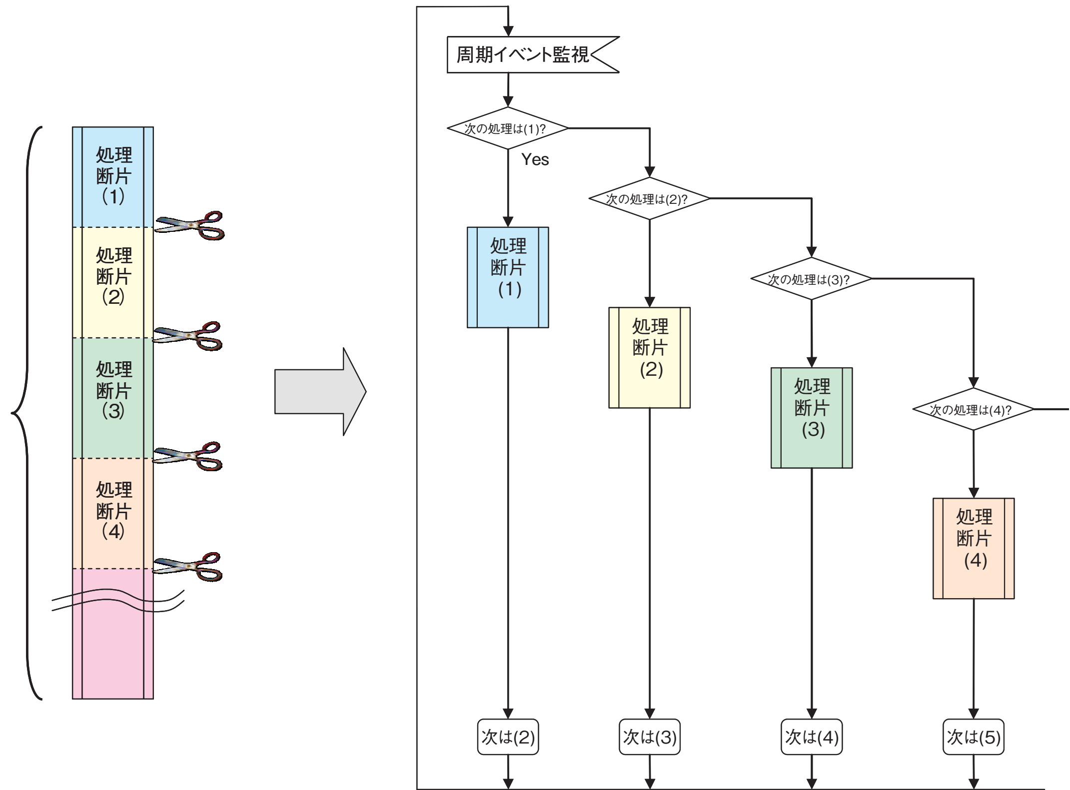
しかし、設計の見直しに踏み切れず、現状の設計思想を変更することなく、現状の設計思想に合わせ込んで対応してしまう傾向が強い。例えば、図A-17のように、完了までに長時間が必要な「処理」を、イベントのポーリング周期に間に合うように複数の処理断片として分割すれば、一時的には解決は可能である。しかし、長期的展望に立つと、決して望ましい対応であるとは言えない。



図A-17：処理量の増大に伴う処理の分割

イベントのポーリング周期に間に合わせるためだけの理由で処理分割が行われているため、各処理断片間の責務境界(機能分担、データや変数の処理分担など)は必ずしも明確ではない。このような処理分割の経緯が開発担当者間で暗黙知として共有されている間は、まだ問題ない。しかし、開発担当者が替わり、これらの暗黙知が失われてしまうと、図 A-18 のように分割された各処理断片に関する設計意図を理解することは大変困難となり、その結果、保守性が低下し、差分開発や流用開発において莫大な工数を要する事態に陥る。

最も確実な対応策は、リアルタイム OS の採用や、MPU のアップグレードなどであるが、製品コスト、開発期間などの制約からやむを得ない場合は、十分にレビューした上で処理分割し、設計意図を明確に文書として残す。



図A-18：処理の分割が繰り返されることによる構造の複雑化

関連する品質特性

保守性（解析性、安定性、試験性）

A-13 シリーズ製品では違いを意識して設計する

作法概要

●系列製品開発を行うときは、系列ごとの変動点を分析する。
●共通部分と変動部分を分離し、変動点を変動部分に凝集(集約)させる。

メリット

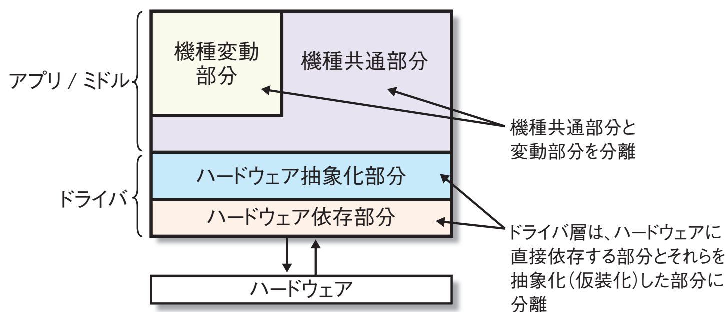
変動点を凝集することによって、系列製品開発における変更が容易になる。

留意点

実行効率の低下やハードウェアリソース不足など、「効率性」とのトレードオフに関する課題が発生する可能性がある。

●解説

数種類の系列製品を開発する場合、各系列によって搭載する機能を変えたり、使用するハードウェアを変えることによって差別化を図ったりすることが一般的である。これらの系列製品を複数種、同時に短期間で開発出来るようにするため、ベースとなるフレームワークを構築し、そのフレームワークを各系列向けにカスタマイズする戦略が広く採用されている。このとき、各系列間で共通になる部分と変動する部分(変動点)とを分析し、図 A-19 のように、共通部分と変動部分を分離してフレームワークを設計する必要がある。

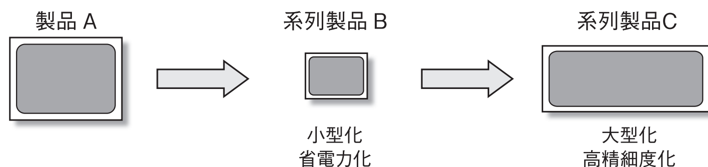


図A-19：系列製品開発を考慮したフレームワーク設計

フレームワークの設計を行う時点で、今後のすべての機能進化を見通すことは難しいが、見通せる範囲については確実に分析をし、それを極力、フレームワーク設計へ反映させる。変動点となり得る要素としては、先に挙げたように、ハードウェアに依存する部分(ドライバ層)や、系列製品ごとの異なる機能(アプリ/ミドル層)などが考えられる。

例 図 A-20 に示すように、ディスプレイを備える製品 A をベースとして、より小型化・省電力化を図った製品 B と、より大型化・高精細度化を図った製品 C を開発する場合

ハードウェアに依存する部分は、異なるディスプレイを使用することに伴い、ハードウェアドライバの変更が必要なケースなどがこれに相当する。他方、系列製品ごとの異なる機能は、各製品で画面サイズが異なることに伴い、画面にダイアログを表示するアプリが用いる座標系の変更が必要なケースなどがこれに相当する。

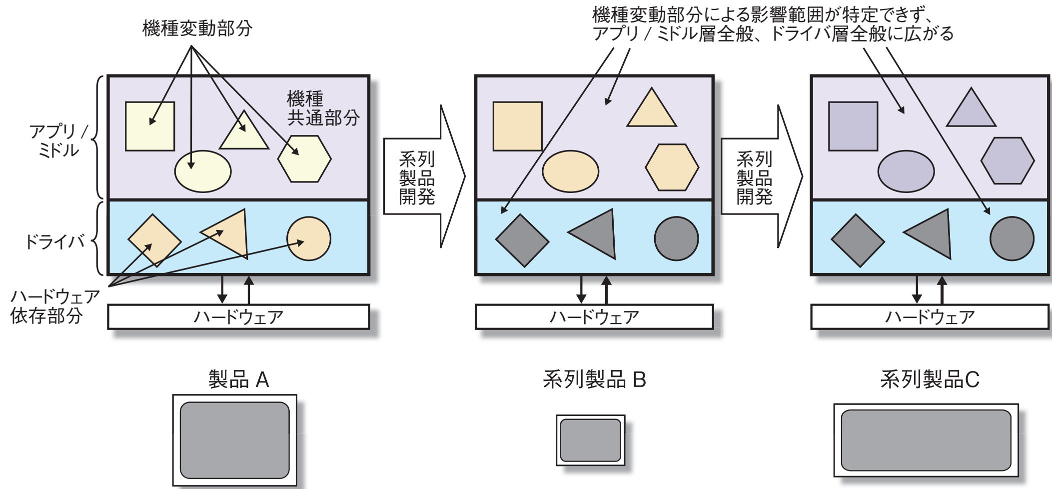


機能の変更に伴い、
・ハードウェアドライバの変更
・アプリの座標系の変更が発生

図A-20：系列製品開発

このようなハードウェアに依存する部分や系列製品ごとの異なる機能など、機種変動部分を図 A-21 のように分散させた設計にすると、機能追加やハードウェア変更が発生したときに、その影響範囲を特定することができず、結局、アプリ/ミドル層やドライバ層の広い範囲に影響が広がってしまう。

逆に、図 A-22 のようにこれらを集約する設計を行うと、機能追加やハードウェア変更が発生したときの影響を局所化することができ、その対応が容易となる。とくに、ハードウェアに依存する部分を集約すれば、ハードウェアに非依存なアプリケーション部分に対する、シミュレータなどを利用したデバッグ環境の構築が容易になり、ソフトウェア開発の初期における品質向上に繋がる可能性が高まる。



図A-21：機種変動部分が分散された設計

図A-22：機種変動部分を集約した設計

なお、この変動部分の集約は系列製品開発に限った話ではなく、例えば、今まで使用していたハードウェアが生産中止となり代替品への変更を余儀なくされるケースなど、その他の組込み機器開発にも適用可能である。

ただし、新たな問題も発生する。実行効率の低下や、ハードウェアリソースの消費量増加に伴うリソース不足などの問題に繋がる可能性がある。一般に、「再利用性・移植性」と「効率性」はトレードオフの関係になることが多く、このような問題が発生しやすい。

この場合、まずは系列製品開発の根幹を担う「共通部分」において、性能などの非機能要件を最大限に確保出来るように設計する。それでも満たせない場合は、システムの非機能要件としてどちらが優先されるかを正しく認識した上で、それに沿った適切な設計を行う。

一方が優先され、他方が無視されるといった単純な選択肢しかないわけではなく、例えば、原則として「再利用性や移植性」を重視して設計を行うが、「効率性」を満たせない箇所については、例外的に逸脱を許容するといった考え方で、バランスをとることも可能である。ただし、このときには、後に第三者が設計意図を理解出来るようにするために、なぜそのような例外的な対応を行ったかを記録として残しておくことが重要となる。

関連する品質特性

- ●保守性（変更性）
- ●移植性（設置性）
- ●効率性（時間効率性、資源効率性）

関連する作法

- ● A-5　実装は設計のあとに行う（P27 参照）
- ● A-15 シリーズ製品では共通点を意識して設計する（P54 参照）

A-14 シリーズ製品ではエラー処理を統一する

作法概要

- 系列製品開発においては、エラーを分類し、エラーコードを体系化する。
- エラー分類の観点でユースケースの例外フローを検討し、エラー処理の共通モジュール化を検討する。
- 上位関数と下位関数において例外処理の実装を統一し、エラー処理に一貫性を持たせる。

メリット

系列製品の相互運用性とソフトウェアの再利用性が向上する。

留意点

上位関数で例外処理を実装する場合は、下位関数が通知するエラーを原則として無視しないようにする。

●解説

系列製品開発では、システム障害に対するエラー処理が製品間で一貫していることが重要である。この一貫性が考慮されず、例えば、同種の障害に対して機種ごとにエラーコードが異なると、フィールドエンジニアやユーザの障害対応などが混乱をきたしたり、系列製品間でのソフトウェアの再利用が困難になるなどの問題が発生する。これを解決するために、系列製品間で統一的なエラー処理が必要である。システム設計段階において、次の手順や観点に従って、エラー処理の統一を検討する。

①エラーの分類とエラーコードの体系化

システム障害に関連して、発生する可能性のあるエラーを列挙し、エラー処理を共通化出来るように、サービスコールやオペレータコール、自己復帰などの対応方法、障害部分を識別するためのシステム構成などの観点で、すべてのエラーを分類する。これらの分類に基づき、ユニークに識別可能なエラーコード体系を決め、エラー処理の方式やエラーメッセージの書式などを検討する。

②ユースケースの例外フローの決定

システムやサブシステムのユースケース記述においては、エラー分類に基づいて例外フローを検討する。既にエラーが体系化されているので、例外フローの単純化と抜け防止が可能である。

③エラー処理の共通モジュール化

ユースケース記述の例外フローにおいて、エラー処理として行われる、共通の機能を洗い出す。これらの機能には、全タスクを適切に中断させるための他タスクへのエラー発生通知や、エラー原因の解析や復帰のためのシステム情報の保存、エラーメッセージの変更容易性を考慮した表示機能などが考えられる。これらの共通機能を共通モジュール化することを検討する。

④例外処理実装方法の統一

例外処理の実装漏れを起こさないように、エラー処理を下位関数で行うか、上位関数で行うかに関する方針を明確にする。上位関数で例外処理を実装する場合は、下位関数が通知するエラーを原則として無視しないようにする。

エラーがどのようなコンテキストで発生したかによって、エラー処理が変わることがあるので、このコンテキストを得るために、エラー時の戻り値や静的変数の使用方法などを統一する必要がある。更に、コンテキストを詳細に把握するために、発生した関数や時刻、ハードウェア状態などの詳細情報を含む構造データを定義することが有効である。

例 系列製品の開発

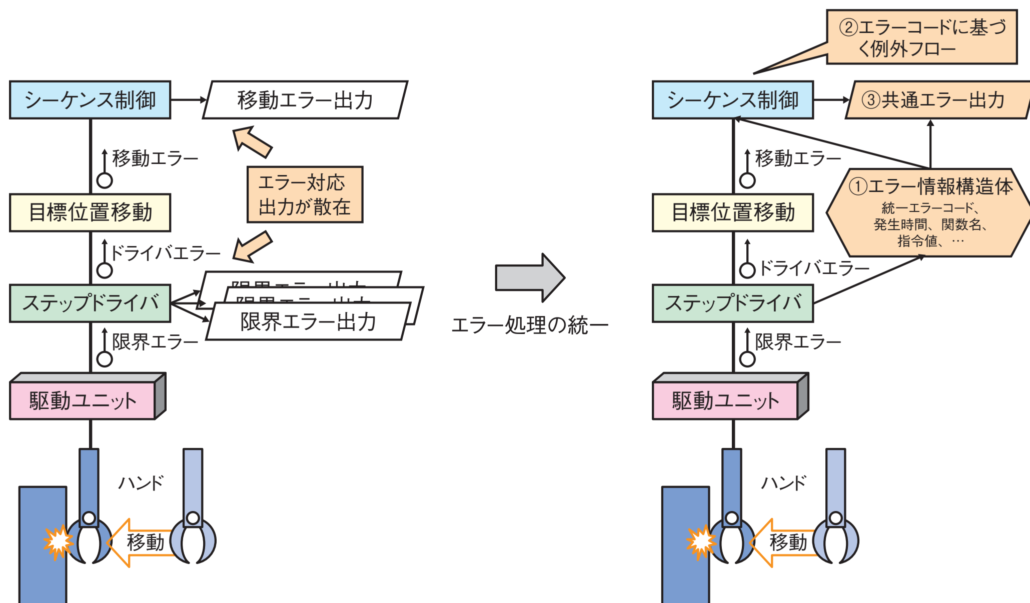
開発規模の大きい製品を複数チームで開発した。図 A-23 の左図に示すように、ロボットハンドに移動指令を出したところ、移動可能範囲を超えたというエラーが発生した。このエラーメッセージは出力されたが、あらかじめエラー処理方式を決めておかなかったため、エラーを検出した関数とその上位関数が、それぞれエラーメッセージを出力し、かつ、そのメッセージ書式が不統一という事態に陥った。

エラー処理を統一するために、次に挙げる改善を実施し、図 A-23 の右図に示すように設計を見直した。その結果、系列製品間で運用を共通化することが出来、ソフトウェアの再利用性を向上させることが出来た。

①エラー種別によるエラーコードの体系化を行い、発生時のコンテキストが分かるようにエラー情報構造体を定義して、エラーと発生場所がユニークに識別出来るようにする。

②ユースケースの検討のときに、エラーコードごとに例外フローを検討し、エラー処理を凝集させ、冗長なエラー処理を省く。

③エラーメッセージ出力関数を共通化して、エラー表示内容の変更や出力先の変更などを容易にする。



図A-23：エラー処理の統一

関連する品質特性

- ●機能性（合目的性、相互運用性）
- ●信頼性（回復性）
- ●使用性（運用性）
- ●保守性（解析性、変更性）
- ●移植性（環境適応性、設置性）

関連する作法

A-2　正常系を設計すると同時に異常系も併せて設計する（P22 参照）

A-15 シリーズ製品では共通点を意識して設計する

作法概要

- 製品の系列化のために、製品固有機能を凝集(集約)させる。
- インターフェースの違いを吸収する共通インターフェースを作成して、ハードウェアやリアルタイムOS(RTOS)などを仮想化する。

メリット

製品の系列化における機能拡張が容易になり、移植性が向上する。

留意点

各モジュールで個別に資源管理せず、資源管理を集中化する。

●解説

系列製品開発において、ソフトウェアの再利用性を向上させるために、系列製品間での機能の可変性(変動点)分析に基づいて、共通機能から成るフレームワークを設計する必要がある。このときに、組込みソフトウェアの特徴として、ハードウェアやRTOSなどが製品の性能とコストの面から相違することも多いため、次の観点を考慮して、製品固有機能の凝集を図る。

①ハードウェアの仮想化

ソフトウェアは、通常、アプリケーション層やシステム制御層、デバイスドライバ層にレイヤ化された構造をとる。系列製品開発においては、デバイスドライバ層でハードウェア仕様を抽象化して、上位レイヤがハードウェア仕様に依存しないようにし、その再利用性を高めることが必要である。そのため、デバイスのモデル化と複合化の観点で、ハードウェアの仮想化を検討する。

デバイスのモデル化の観点では、システム制御層のモデルに適した機能とパラメータを考慮し、デバイスドライバ層の関数インターフェースを定義する。例えば、測距センサのデバイスドライバインターフェースとして、ハードウェアに依存する測距原理(三角測量や反射波の位相差など)にかかわらず、常に距離へ変換する機能を提供する。

デバイスの複合化の観点では、デバイスの抽象度をより高めるために、デバイスドライバ層を多層化し、複数デバイスから構成された仮想デバイスを提供することを検討する。例えば、モータによる可動部と距離センサで移動ステージを構成する場合、システム制御層の再利用性を高めるために、複合デバイスの仮想デバイスドライバを構成し、移動方向と距離をパラメータとするインターフェースを提供する。

②起動・停止処理の集中化

製品の起動・停止処理においては、ハードウェアを制御する各タスクは相互に同期をとりつつ、安全なシーケンスで起動・終了を行う必要がある。各ハードウェアの起動・終了シーケンスをそれぞれのデバイスドライバに行わせると、起動・停止時にタスク間の同期通信が輻輳し、デバッグや機能追加などを難しくする傾向がある。製品の起動・停止シーケンスを集中制御する、1つの仮想的なコントローラを用意し、起動・停止シーケンスに関わるシステムイベントのすべてを処理させると設計の見通しが良くなり、機能拡張や移植性などが向上する。なお、この仮想的コ

ントローラが複雑にならないように、起動・停止処理以外はそれぞれのデバイスドライバへ凝集させる。

③ RTOSの仮想化

RTOSのシステムコールは、関数名や引数、戻り値などのインターフェース仕様がOSによって異なる。このため、システムコールを直接使用した実装を行うと、OSへの依存性が高まり、移植性が損なわれる。基本機能は共通のものが多いので、共通のシステムコールに利用を絞り込めば、ラッパー関数ライブラリを作成することによってOSの置換えを容易にすることが出来る。このラッパー関数がパラメータを変換し、実装されている等価なシステムコールを呼び出す。単体テストなどの都合でPC上でのシミュレーションを実施したい場合では、POSIXなどの汎用OSのシステムコールも意識したラッパー関数インターフェースにすることも可能である。

④マイクロプロセッサ依存コードの凝集

系列製品間で使用するマイクロプロセッサが変わると、プロセッサのワード長や、キャッシュサイズ、バイトオーダの相違によって、同じプログラムでもメモリアクセス回数が増加し、実行時の性能低下を招く場合がある。これに対処するために、データアライメントやキャッシュサイズ、バイトオーダに最適化したデータ構造を定義したり、バイトオーダの違いに伴うビット処理や型変換処理(キャスト)などを行ったりしなければならないが、これらは凝集し、置換可能にする。

⑤リソース管理の凝集

系列製品ごとにメモリ容量や搭載される機能などが変化するため、ハードウェアやタスク、RTOSに最適なリソースを割り当てる必要がある。各モジュールが、動的メモリ確保と同期通信に必要なリソース管理処理を行っている場合は、これらの処理を凝集し共通モジュール化すれば、移植性と保守性が向上する。

例 製品を起動するプログラムの作成

プログラム作成には、一般に手順的もしくは時間的凝集が構造的に望ましくないと言われていることから、ハードウェアユニットごとに制御タスクを設け、対応するハードウェアユニットの初期化シーケンスを各タスクで処理するようにした。しかし、安全な起動シーケンスを実現するためには、各ハードウェアユニットの初期化タイミングを同期させなければならず、メッセージとイベントフラグのシステムコールを用いてタスク間の同期をとる必要があった。

初期製品では問題なく動作したものの、その後の系列製品でハードウェアユニットの追加やタイミングなどの変更が発生した。起動シーケンスの見直しが必要だが、起動シーケンスが複数タスクに分散して処理されているため、製品ごとの起動シーケンスのチューニングが困難になってしまった。この対策として、起動シーケンス全体をコントロールするタスクを設け、各ハードウェアユニット制御タスクは、このタスクのみと同期をとるハブ構造にした。

関連する品質特性

●機能性(相互運用性)
●信頼性(成熟性)
●保守性(解析性、変更性)
●移植性(環境適応性、設置性)

関連する作法

A-13　シリーズ製品では違いを意識して設計する(P47 参照)

A-16 ループ処理からの例外脱出の仕様を定義する

作法概要

- 周辺デバイスまたは周辺サブシステムの外部仕様からの想定可能な例外を漏れなく洗い出す。
- 周辺デバイスまたは周辺サブシステムに対する待ち処理を実施する場合は、状態変化の例外に対してタイムアウトを設ける。

メリット

周辺デバイスまたは周辺サブシステムの例外事象を含めたインターフェース仕様を定義することで、高信頼なソフトウェアを構築することが出来る。

留意点

①まず、正常系の満たすべき仕様を定義する。次に、想定外の事象が発生したときの処理、すなわち異常系の満たすべき仕様を定義する。
②すべての外部インターフェース先(周辺デバイス、周辺サブシステム)について例外時の振舞いを定義する。

●解説

対象MPUと周辺デバイスまたは通信で接続された周辺サブシステムとの間の外部インターフェースに関して、その例外事象への対応が完全に網羅されていない場合、システムが待ち状態から脱出できず、システム障害を発生させてしまう可能性がある。その理由は、インターフェースデータの変化を捉えるためにはシステムは待ち状態になる必要があり、インターフェースデータが正常に変化しなければ、この待ち状態が解除されないからである。

インターフェースの例外としては、来る、来ない、長い、短い、大きい、小さい、多い、少ないなどの例外が発生する可能性がある。周辺デバイスや周辺サブシステムは、対象MPUとは非同期で動作しており、ハングアップやリセットなどの例外事象を対象MPU側で検出することが出来ない。従って、対策として例外事象を想定したインターフェース仕様を、次の手順で定義する必要がある。

①対象MPUとの外部インターフェース先(周辺デバイスや周辺サブシステム)を外部インターフェースリストの縦軸に列挙する。
②外部インターフェースリストの横軸に例外事象を並べて、例外事象が発生したときに待ち状態になる可能性がある外部インターフェース先を特定する。
③ソフトウェア機能要件リストの各機能項目に関して、対象ソフトウェアの例外発生を考慮した仕様を定義する。

例 1つのMPUに搭載される対象ソフトウェアが、デバイスAとデバイスBにアクセスし、デバイスAはサブシステムXに接続されているシステム(図A-24)

まず、外部インターフェースリストの各行に対象MPUとの外部インターフェース先としてデバイスA、デバイスB、サブシステムXを記入する。

デバイスAは、ファームウェアが内蔵されていてリセット端子を持つため、例外としてリセット

が考えられる。また、内部ファームウェアの特性で遅延する可能性があるため、遅延の例外もあり得る。

デバイスBは、ファームウェアが内蔵されていないため、例外の要因はない。

サブシステムXは、MPUが搭載されているため、例外としてリセットと遅延があり、更に通信ケーブルで接続されているため、断線という例外が発生し得る。

これらの検討結果をまとめて、外部インターフェースリストを完成させる。

想定される例外への対処策としては、次のものが考えられる。

①タイムアウトの設定

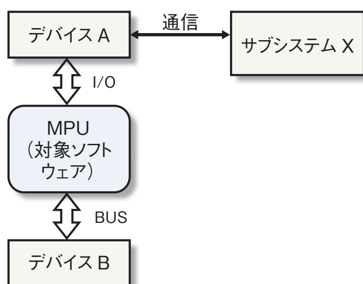
MPUに搭載する対象ソフトウェアが、サブシステムに通信コマンドを送信して、サブシステムから応答の通信コマンドを待つ場合、通信線が断線したり、サブシステムのリセットが発生したりという例外の発生に対して、対象ソフトウェアが待ち状態から解放されるようにタイムアウトを設ける。

②通信遅延の考慮

通信インターフェースの場合には、送信がノイズなどによって正確に伝わらないことがあり、それに伴う再送による遅延や、多くの機器が同じ通信線に大量の情報をアクセスすることでトラフィックが増加し、その影響による遅延が想定される。このような再送やトラフィックによる遅延を考慮した機能仕様を定義する必要がある。

③機能追加や変更の考慮

MPU側の機能量に関しては、その増加に伴う遅延も発生する可能性がある。開発当初は問題がなくても、段階的な機能追加や変更でサブシステムのMPU処理負荷が増加して、通信応答の遅延が発生する可能性がある。また、機能追加や変更により通信量が増加するため、それによる遅延やトラフィックの増加も考慮した仕様を定義する必要がある。



図A-24：システム構成

インターフェース	リセット	遅延	断線
デバイスA	○	○	ー
デバイスB	ー	ー	ー
サブシステムX	○	○	○

○：例外発生の影響あり
ー：例外発生の影響なし

表A-4：外部インターフェースリスト

関連する品質特性

信頼性

関連する作法

● B-12　機能仕様を決める際には例外への対策を網羅的に行う（P111 参照）
● B-16　メモリ領域を効率良く使用する（P117 参照）

A-17 時間制約を定義しこれを守ることにより確実な動作を保証する

作法概要

●各ソフトウェア機能項目に時間制約(注1)を定義する。
●同じ時間制約を持つ機能項目を同じタスク(注2)に割り付ける。

メリット

タスク設計において処理性能のバラツキを抑制させ、保守性や信頼性、効率性を確保することが出来る。

留意点

①時間制約は、物理的な特性や制御対象物の特性から定義し、タスクを周期的に起動することで時間制約を満足させるようにする。
②周期を定義出来ない場合は、イベント発生から起動開始までの時間制約を定義する。

●解説

タスク設計のやり方が開発者によってばらついてしまうと、タスクを分割し過ぎたり、並行性がない機能をわざわざタスクに分割するという事態に陥り、タスク設計がソフトウェアを複雑にする要因となる。

本来、タスクとは、異なる時間制約を1つのMPUで達成するために作り出された技術であると言える。異なる時間制約を持つ機能項目を同じ時間周期で実行しようとすると、待ち処理が入ってしまい、CPUが無駄に使用される。そこで、タスクという概念が生まれ、ソフトウェアを異なる時間制約で並行動作する複数のタスクに分割するようになった。

時間制約に基づくソフトウェア開発で解決しなければならない課題は、2つ考えられる。1つ目の課題は、タスク分割が開発者に依存して行われることによって、並行して動作するタスク間の起動関係とインターフェースを解析仕切れなくなり、その結果、作業効率が悪化し、品質確保が困難な状況になることである。もう1つの課題は、タスクに分割する必要がない部分に対してもタスク分割を実施してしまうことによって、CPU資源(CPU、メモリ)を浪費して性能上の問題を引き起こすことである。

これらの課題を解決するためには、必要最小限のタスクで構成することが求められ、タスク分割の根拠を明確に定義する必要がある。その1つの方法として、同じ時間制約を持つ機能項目を同じタスクに割り付けるというタスク分割の指針が考えられる。その手順は次の通りである。

①1つの機能項目に時間制約が複数ある場合は、機能項目を分解する。
②ソフトウェア要求分析で作成するソフトウェア機能要件リストに時間制約という項目を追加し、機能項目に対して時間制約を定義する。
③ソフトウェア機能要件リストに記載されている機能項目と時間制約を縦軸に記載し、横軸にタスクを定義して、タスクマッピングリストを作成し、同じ時間制約を持つ機能項目を同じタスクへ割り付ける。

注1：時間制約とは、一般的に時間に関する制約条件を意味する。ここでは、繰り返し行わなければならない処理に対する時間間隔(周期時間)の意味で使っている。
注2：ここでのタスクは、並行して動作するプログラムで、割込みプログラムやリアルタイムOSのタスク(スレッド)を示す。

例1 同じ時間制約なのに、異なるタスクに割り付けている場合

この例では、機能項目 1 と機能項目 2 が、同じ時間制約にもかかわらず別タスク(タスク A とタスク C)に割り付けられている。同じ時間制約を持つ機能項目は、1 つのタスクに割り付ける指針を用いると、機能項目 1 と機能項目 2 は、タスク A に割り付けられ、タスク C が不要になる。

機能項目	時間制約	タスク A	タスク B	タスク C
機能項目 1	10msec	○		
機能項目 2	200msec		○	
機能項目 3	10msec			○
機能項目 4	200msec		○	
機能項目 5	200msec		○	

改善

機能項目	時間制約	タスク A	タスク B
機能項目 1	10msec	○	
機能項目 2	200msec		○
機能項目 3	10msec	○	
機能項目 4	200msec		○
機能項目 5	200msec		○

表A-5：タスクマッピングリスト①

例2 異なる時間制約なのに、同じタスクに割り付けている場合

この例では、機能項目 1 と機能項目 2 が、異なる時間制約であるにもかかわらず 1 つのタスク A に割り付けられている。同じ時間制約を持つ機能項目は、1 つのタスクに割り付ける指針を用いると、機能項目 2 はタスク C に割り付けられる。

機能項目	時間制約	タスク A	タスク B
機能項目 1	10msec	○	
機能項目 2	200msec	○	
機能項目 3	50msec		○

改善

機能項目	時間制約	タスク A	タスク B	タスク C
機能項目 1	10msec	○		
機能項目 2	200msec			○
機能項目 3	50msec		○	

表A-6：タスクマッピングリスト②

関連する品質特性

保守性、信頼性、効率性

A-18 2次元マトリクスを利用して出力競合を解決する

作法概要

- 要求を機能ブロックに分解する。
- 並行して動作し、かつ同じ出力を操作する機能ブロックを抜き出す。
- 出力操作対象ごとに2次元マトリクスを作成し、出力操作の競合仕様(競合時の振舞い)を定義する。

メリット

並行動作する機能ブロックが引き起こす、出力競合を解決出来る。

留意点

①機能競合の分析を実施した上で、出力競合の分析を実施する。
②優先度の低い機能ブロックが出力を確保した場合、優先度の高い機能ブロックが待ち状態にならないように設計する。

●解説

複数機能ブロック間のインターフェースの仕様があいまいである、または想定出来ていない場合に、異なる出力操作を同じ出力対象に実施すると、システム障害に繋がる可能性がある。

これを解決するため、ソフトウェア要求分析で出力操作の競合の分析を実施する。具体的には、次の手順に従い、設計時にOSの利用の有無にかかわらず、出力競合を解決する。

①要求を機能ブロックに、1機能ブロック1目的となるように割り当て、ソフトウェア機能要求リストを作成する。
②機能ブロックを目的別に分類整理し、機能ブロックに優先レベルを定義する。
③同じ出力を操作する同じ優先度の機能ブロックを定義する。
④現在実行中の機能ブロックを横軸に、新たに実行を開始する機能ブロックを縦軸に配置する2次元マトリクスを作成する。
⑤このマトリクスの交点が出力競合に対応し、出力操作時の矛盾を考慮して、ここに該当する機能ブロック群の競合に関する仕様を定義する。

例 扇風機に搭載される機能

機能ブロックをF1～Fnとして、ソフトウェア機能要件リストは表A-7に示す通り。ここで、機能ブロックF1は自動運転、機能ブロックF2はモータ温度異常制御、機能ブロックF3はモータ異常制御に対する機能とする。自動運転では、扇風機が室内温度に応じて風量を変化させて送風する。モータ温度異常制御は、モータ温度が高い場合にモータを保護するために風量を最低風量に落とす。モータ異常制御は、モータ異常を検知してモータを停止させる。自動運転とモータ温度異常制御は、同じ優先レベルであるため、並行に動作する可能性があり、同じモータを操作すると出力が競合してしまう。

同じ出力を操作し、かつ同じ優先レベルを持つ機能ブロック群に対して、出力操作対象ごとに表A-7のようなマトリクスを作成する。このマトリクスの交点に出力競合の仕様を定義して、出力競合によるシステム障害を抑制した。

ソフトウェア機能要件リスト

機能ブロック	優先レベル
F1（自動運転）	低い
F2（モータ温度異常制御）	低い
F3（モータ異常制御）	高い
F4	
F5	
Fn	

出力マトリクス（モータ）

	F1	F2	F3
F1	＼	☆	—
F2	☆	＼	—
F3	—	—	＼

☆：競合仕様定義
—：競合無し

表A-7：ソフトウェア機能要件リストと出力マトリクス

関連する品質特性

信頼性

A-19 2次元マトリクスを利用して機能競合を解決する

作法概要

●要求を機能ブロックに分解する。
●機能ブロックを縦軸と横軸に置く2次元マトリクスを作成し、機能ブロックの競合仕様（競合時の振舞い）を定義する。

メリット

●機能ブロックの増加に伴う複雑化を抑制出来る。
●複数の機能ブロックが並行動作する場合、相反する目的を持つ機能ブロック群による誤動作を防止出来る。

留意点

①1つの機能ブロックには1つの目的しか割り当てない。
②機能ブロック間に依存関係を持たせないように機能ブロックを定義する。
③機能ブロックを同じ目的で分類して機能要求リストに定義する。
④機能競合の検討では、機能ブロックが持つ状態間の競合も分析する。

●解説

複数機能ブロック間のインターフェース仕様があいまいである、または想定出来ていない場合に、相反する目的を持つ機能ブロック群が連続または同時並行して動くことにより、システム障害を引き起こす可能性がある。ここで、機能ブロックはタスクよりも粒度が小さい、タスクの中に共存する処理単位と考える。

システムの大規模・複雑化に伴い、機能の量は増加する一方である。このように大規模・複雑化するシステムを新規開発する場合や、機能追加または機能変更する場合などに、すべての機能の競合をあらかじめ想定出来ないと、システム障害を引き起こしてしまう可能性がある。

これを解決するため、ソフトウェア要求分析で機能競合の分析を実施する。具体的には、次の手順に従い、2次元マトリクスでチェックすることにより、機能競合によるシステム障害を抑制する。

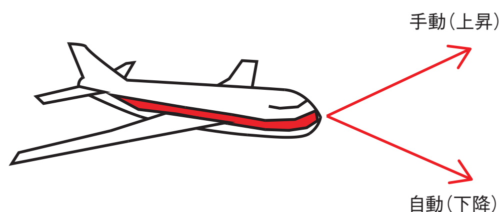
①要求を機能ブロックに、1機能ブロック1目的となるように割り当て、ソフトウェア機能要求リストを作成する。
②機能ブロックを目的で分類整理し、機能ブロックに優先レベルを定義する。
③現在実行中の機能ブロックを横軸に、新たに実行を開始する機能ブロックを縦軸に配置する2次元マトリクスを作成する。
④このマトリクスの交点が機能競合に対応し、ここに該当する機能ブロック群の競合に関する仕様を定義する。

ここまでは、タスクより粒度が小さい機能ブロックを取り扱ってきたが、機能ブロックの代わりにタスクを定義すれば、更に、タスク間のインターフェースの整理などにも応用出来る。

例 航空機に搭載される運転機能

機能ブロックを F1～Fn とする。機能ブロック F1 は自動操縦運転、機能ブロック F2 は手動操縦運転に対する機能とする。機能ブロックの優先レベルを検討すると、自動操縦運転よりも手動操縦運転の方が優先度が高いことが分かる。

こうして作成したソフトウェア機能要件リストを基に機能要件マトリクスを作成する。このマトリクスの交点に機能競合時の仕様を定義して、機能競合によるシステム障害の課題を抑制した。



自動操縦運転で下降中に手動操縦運転で上昇させる。
自動操縦運転では、相反する動作を手動操縦運転で与えることで、更に下降してしまうような機能競合による重大なシステム障害を引き起こしてしまう。

図A-25：航空機の運転機能

ソフトウェア機能要件リスト

機能ブロック	優先レベル
F1（自動操縦運転）	低い
F2（手動操縦運転）	高い
F3	
F4	
F5	
Fn	

機能要件マトリクス

	F1	F2	F3	F4	F5		Fn
F1		☆					
F2	☆						
F3							
F4							
F5							
Fn							

競合仕様を定義する

表A-8：ソフトウェア機能要件リストと機能要件マトリクス

関連する品質特性

信頼性、保守性

A-20 実行時の不具合の追跡手段を用意する

作法概要

システムまたはサブシステムの境界で受け渡しするデータに不正を見つけた場合、異常発生を示すログなどを出力する。

メリット

障害の原因解明や、障害担当者の早期割当てなどが可能となる。

留意点

①ログの手段や情報量などに関してシステム全体の方針を開発早期に決定し、開発や保守などの関係者に周知しておく。
②異常時の情報収集の仕組み作りが機能要件に含まれることは少ないので、品質保証の観点から有用性(対費用効果など)を説明し、開発計画に織り込んでおく。

●解説

複数のサブシステムから構成されていて、それらが協調動作するシステムにおいて、サブシステム間のインターフェースで不正なデータを受け取ったときに、その不正を表す戻り値を返したとしても、呼び元のエラーチェックが十分ではないケースがある。エラーが正しく通知されても、呼び元が戻り値チェックを行わず、処理を継続すると、エラーの原因から直接には関係のない処理で、後になって障害が発生する可能性がある。このようなケースの原因究明は、非常に時間がかかり、難しい作業となる。これを改善するために、不正値を受け取った側が異常が発生したことを示すログなどを出力して、異常が発生していることを認識出来るようにする。

ログの種類と目的は次の通りである。

①インターフェースログ(サブシステムを守るログ)

同じ組織内の別のチーム(サブシステム)や、他社接続のときのように違う組織との間でやりとりを行うときに、発生している事象や受け渡しのデータが異常か正常かを記録し、問題発生時の責任の切り分けを速やかに行うことを目的とする。この作法で扱うログはこれに該当する。

②製品自体を守るログ

製品自体の市場での動きを記録しておいて、問題発生時のエビデンス(ユーザの操作ミスなのか、製品の不具合なのかなど)として使用する。

③動作記録ログ(サービスログ)

製品の動作に異常が発生したとき、サービスセンタが取得して保守サービスに利用する。異常の通知や、それに関して製品で起きている事柄を保守員が調査する。

④仕様改善のためのログ

ユーザが製品をどのように使用しているかを記録し、後の製品開発に使用する。

⑤デバッグ・テスト用のログ

デバッグ・テスト時に動作状態などを記録し、想定通り動作しているかなどを確認する。

更に、ログの取得の仕組みを、次の事項にも留意して、組み込む製品の非機能要件に応じて決定しなければならない。

①ログ取得情報量

すべて取らなければ意味がないのか、あるいは問題発生直前の一定期間でよいのかによって、記録する情報量やログ記録領域が決まる。

②記録スピード

システムが、シビアなタイミングで動作することを要求されているのか、いないのかによって、記録スピードが決まる。

表 A-9 では、代表的なログの出力先を取り上げ、ログ取得情報量と記録スピードに関して、その特徴を示す。

ログの出力先	ログ取得情報量	記録スピード	説明
メモリ記憶	小	高	記録スピードは高速なので、タイミングがシビアな組込み系に向く。記録出来る量は少量のため、リングバッファなどを使用する。問題発生からさかのぼって検証することが可能となる。記録した情報は、コンソールなどへ表示したり、異常発生時にファイルへ出力するなどして取得する。
ファイル記録	大	低	メモリに比べてより大量のデータを記録出来る。アクセススピードは低速のため、タイミングがシビアな製品には向かない。ファイルシステムにもよるが、ログを出力した後、flush や endnl で確実に書き出す処理をしないと、肝心の最後のログがファイルに記録されないので注意が必要となる。
シリアル（コンソール）出力	小～大	低	製品のシリアルポートにログを出力する。シリアルポートを受信、表示可能な端末を用意する必要がある。動作中に、発生している事象をリアルタイムで表示可能であり、動作状況をリアルタイムに確認する必要がある場合に適する。通常、画面上を流れて消えてしまうので記録情報量は少ないが、表示内容をファイルに記録する仕組みを利用すれば、記録情報量は大幅に向上する。
ネットワーク出力	大	低～中	出力先にネットワークを使用することで、動作状況の遠隔地での確認が可能である。書き込み速度はネットワークの品質に左右される。

表A-9：ログの出力先

例 オフィス機器

液晶パネルが付いた大型のオフィス機器において、ユーザは、ある機能の設定を行い、その実行を指示した。しかし、設定画面の処理に不具合があり、あり得ない組み合わせでアプリケーションロジック側へ実行のコマンドを発行した。不正な設定内容の実行指示コマンドを受け取ったアプリケーションロジックは、内容をチェックし、設定値に不具合があることを示すエラーを返したが、設定画面側は、このエラーをチェックしておらず、実行中の状態へ移行し、本体動作ステータス変更通知待ちとなった。実行中のステータスが来るまで、ユーザ操作を受け入れないようにしており、いわゆるハング状態となった。

アプリケーションロジックはエラーを返すように実装しているが、設定画面側がエラーを見ていないとは思っていないため、不具合調査時にはエラーは起きていないと判断した。また、実行を指示するコマンドは発行されていることから、アプリケーションロジック側から調査することになった。設定値の不具合に関心が向かず、エンジンがなぜ起動しないのかを中心に、よりハードウェアに近いドメインで調査を進めてしまった。

ドメイン間のやりとりで異常が発生した際にログに出力しておけば問題の原因はすぐに解明されたのだが、このような仕組みを組み入れておらず、関係のないドメインを巻き込み、多大な労力を消費した。調査コストの増大、ドメイン間の不信感の高まりなど、マイナス要因は枚挙にいとまがない。

関連する品質特性

保守性（解析性）

A-21 メモリの動的な扱いを避ける

作法概要

- システム起動時に、必要な変数はすべて割り付ける。
- 割り付けた領域のみ使用するように設計する。
- ヒープ領域のメモリを動的に確保したり、解放したりしない。

メリット

- メモリの解放漏れによるメモリリークを回避出来る。
- 解放済み領域への不正アクセスに起因する不具合を回避出来る。
- 断片化による効率低下とリソース不足を回避出来る。

留意点

①静的に確保して構わないものについては、システムの初期化時にメモリを確保しておく。
②静的に確保して確実に動作する必要があるものと、ベストエフォートでよいものとを区別して設計する。
③確保するメモリ領域の大きさは、必要なデータ量とスループットを考慮して決める必要がある。
④アクセスしているデータ領域の有効性を保証するため、アクセス関数などの機構を用意する。

●解説

動的なメモリ確保・解放を行っていると、メモリが確保出来ない、メモリを確保するために想定以上の時間がかかる、メモリの使用効率やアクセス速度が低下するなど、予期しないトラブルが様々な要因によって発生しがちである。従って、動的なメモリ確保は出来るだけ使用しないのが望ましい。通信などのときのように、必要とする領域の大きさが動的に決まる場合でも、あらかじめ数種類の大きさの領域を確保しておくなど、静的なメモリ確保による対処を出来るだけ検討する。

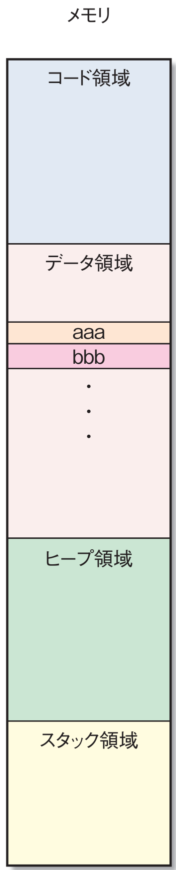
どうしても動的なメモリ確保を使用する必要がある場合は、確保と解放の回数を減らす、単純な実行パターンで使用するなど、メモリに対する不正操作を発見しやすい工夫を考慮する必要がある。

また、動的なメモリの確保が前提となっているJavaのような言語処理系では、ごみ集め(ガベージコレクション)を行う機構を自動的に実行するため、開放漏れによるメモリリークを回避出来る。しかし、ごみ集めの時間的な性質を予測するのは難しいため、リアルタイム性が必要な用途には向かない。

例 静的なメモリ確保による対処

プログラムや関数が使用する変数は、あらかじめ静的変数（static）としてメモリに確保しておく。

静的変数は、一度確保されると解放されない。プログラム走行中は常に使用出来る。



図A-26：静的なメモリ確保による対処

関連する品質特性

●信頼性（成熟性）
●保守性（解析性、変更性、安定性、試験性）

関連する作法

B-9　メモリの動的確保に留意する（P103 参照）

A-22 故障を回避する

作法概要

●機器の故障を回避するメンテナンス方法を決める。
●メンテナンスに最適なソフトウェア機能を決める。

メリット

システムの可用性が向上する。

留意点

①ハードウェア障害が引き起こす機器の故障を体系的に抽出する。
②費用対効果を考慮し、適切なメンテナンス方法の組み合わせを検討する。
③メンテナンス機能の観点で、システム設計を検討する。

●解説

ハードウェアの初期故障や摩耗などの経年劣化をも考慮したメンテナンス機能が、組込みソフトウェアには高可用化のために必要とされる。このメンテナンス機能を実現するためには、付加的ハードウェアとソフトウェア基本構造への考慮が必要な場合も多いため、システム設計段階で次の検討を行う。

①ハードウェア障害に起因して、どのようなシステム故障が発生するかを把握する。

ハードウェアユニットの障害がどのようなシステム故障を引き起こすかを把握するために、フォールトツリー分析(FTA)や故障モード影響解析(FMEA)などの手法を用いて、漏れの少ないシステム故障の列挙を行い、ソフトウェアによるメンテナンス機能が要求されるハードウェア障害を識別する。

②故障を回避するために、適切なメンテナンス方法を決める。

製品の特性(コンシューマ機器か、産業機器かなど)と要求される費用対効果によって適切なメンテナンス方法が変わり、それに対応するソフトウェア機能も異なる。故障が緊急を要さない場合では故障後の修理を円滑にするための機能が、緊急性がありかつ寿命が予測出来る場合では予防的な機能が、それぞれ要求される。メンテナンス方法とそれに対応するソフトウェア機能としては、次の(1)から(3)がある。

(1)ブレークダウンメンテナンス

故障後に修理するメンテナンス方法である。予定外の停止を短期化するために、故障状態を迅速に把握し修理を行う必要がある。ソフトウェア機能としては、サービスコール自動化、故障発生状況とハードウェア障害に関する分析ツール、機器情報と設定のバックアップ、ユニット別マニュアル操作機能などが該当する。

(2)予防メンテナンス

機器状態にかかわらず、一定の時間間隔でメンテナンスする方法である。予定された停止を短

縮し、機器の稼働率向上を図るために、定められたメンテナンスのワークフローを支援する機能が必要となる。サブシステムごとのメンテナンスを実現するためには、システムからの一部ハードウェアユニットの切り離しなど、ハードウェアやソフトウェアの基本構造に対する考慮も必要となる。ソフトウェア機能としては、ハードウェア再調整、ユニット別マニュアル操作または検査機能などが該当する。

(3)予知メンテナンス

ハードウェア障害を予測し、事前にメンテナンスすることで稼働率の向上を図る方法である。性能データや操作ログ、保守ログなどのモニタリングデータと、それに基づく予測モデルによって、ハードウェア障害の発生を予測する機能が必要となる。性能データの取得などのために、センサなどの付加的ハードウェアなどが必要な場合がある。ソフトウェア機能としては、稼働状況のモニタリング、モデルによる障害発生時期予測、障害発生予測時の警告またはサービスコールの自動化、保守交換部品の情報を提供する機能などが該当する。

実際には、機器への重大な故障を引き起こすハードウェア障害には予防メンテナンスを、消耗部品の不足には事前発注を可能とする予知メンテナンスをそれぞれ適用するというように、費用対効果の観点でメンテナンス方法の適切な組み合わせを検討する。1つの機能でも複数のメンテナンスに有効なものがあり、効果的な機能の組み合わせを検討することは、開発効率を向上することにも繋がる。更に、オンサイトだけでなくオンラインで提供出来るメンテナンス機能を考えると、可用性も保守効率も向上することが出来る。

例 オンライン化も考慮したメンテナンス機能

地域分散した機器の可用性と保守性の向上を図るため、次に列記するオンライン化も考慮したメンテナンス機能を検討した。これらのメンテナンス機能の実現には、ハードウェアとして遠隔通信とログ収集のためのネットワークインターフェースとハードディスクの追加、ソフトウェアではハードウェア仮想化のためのレイヤ化が、それぞれ必要であった。

①機器設定の確認と変更のためのリモートコンソール機能
②ソフトウェアバージョンアップのための仮想ドライブ機能
③システムとサブシステムに対して再リセットとパワー ON/OFF を行うリモート制御機能
④サービスに必要な機器構成と定期交換部品に対するリモート情報収集機能
⑤障害分析のためのイベントとセンサ値のログに対するリモート情報収集機能
⑥システム異常検出時の管理者レポートのための稼働状況監視機能
⑦障害とその情報をサービス拠点へ自動通報するサービスコール機能

関連する品質特性

- 機能性(合目的性)
- 信頼性(回復性)
- 使用性(運用性)
- 保守性(解析性、変更性)

関連する作法

D-5　デバイスの寿命に関する特性を考慮して設計する（P182 参照）

A-23 システム改修・更新時には、先に性能解析をしてからアーキテクチャとソフトウェアを変更する

作法概要

- 現行システムを性能評価・解析する。
- 次期システムの要件分析に基づいて性能ボトルネックを解析する。
- 性能ボトルネックを解消出来るシステムアーキテクチャを設計する。

メリット

性能要件の未充足が判明して、手戻りすることによる開発遅延、複雑化によるソフトウェア品質の劣化など、下流工程での問題の出現を予防出来る。

留意点

①性能評価は、開発が完了している現行システムを対象とするため、動作条件や機能仕様などがこれから開発する次期システムとは異なる場合がある。従って、現行システムの性能評価結果を用いて次期システムの性能ボトルネックを解析するときには、それらの相違点を考慮して補正するなどの対処が必要になる。

②システムアーキテクチャレベルの変更には、説得力のあるデータが必要になる。現行システムの性能評価結果を補正して用いて机上計算やモデルシミュレーションを行っただけでは、説得力に欠ける可能性がある。そのような場合には、次期システムとして想定するシステムアーキテクチャのハードウェア部分を、FPGA ボードなどを用いてプロトタイプとして作成し、その上で計測するなどの工夫が必要となる。

③計測によって求めることが困難な性能データがシステム中に存在することもあるため、そのような場合は、仕様などから概算する。そのときには、標準ケースや最悪ケースなど、幾つかの条件・シナリオを想定して算出する。

④システムアーキテクチャの設計に立ち返ってやり直すほうが、全体として開発をより良くすることを納得させるために、プロジェクトマネージャは説得力のあるデータを示しつつ、開発チームの意思を統一する必要がある。

⑤モデルシミュレーションによって性能ボトルネックを解析する場合は、システムアーキテクチャレベルの粗いモデリングが必要になる。モデリングのときには個々の構成要素がシステム性能に与える影響度を考慮して、どのような粒度の、どの構成要素をモデル要素とするかを取捨選択し、統合・簡略化したモデルを作成する。

●解説

システムの性能に対して何が大きな影響を与えるか、何がボトルネックとなるかはシステムの性格によって異なり、同じシステムであっても用途・シナリオによって異なる。システムを構成するそれぞれの機能単位でも、このことは同様である。高いプロセッサ性能を必要とするシステムまたは機能単位もあれば、広いメモリバス帯域を必要とするシステムまたは機能単位もある。このため、現行システムでは性能面で問題がなかった(性能要件を充足していた)としても、それをもとに一部を変更しただけの次期システムが、性能面に問題を生じることは起こり得る。

組込みシステムでは、現行システムの機能仕様の大半を踏襲して、次期システムが開発される場合が多い。そのときに次期システムは、現行システムに対して一定の比率の性能向上を目標としていることも多い。また、ソースコードレベルでの再利用を繰り返してきたレガシーソフトウェアを可能な限り再利用した上で、性能向上を果たすためにプロセッサ、キャッシュ、バスなどのハードウェア構成要素の単体性能を上げることで対応することもよくある。

しかし、どのハードウェア構成要素の単体性能向上がシステム全体の性能に対して効果的となるかどうかは、前述のように、システムまたは機能単位の性格・用途・シナリオによって異なる。時には、過去の開発による経験則から、どのハードウェア構成要素の性能をどの程度上げれば良いのかを判断する場合もある。そのような経験則ベースでの判断は、場合によっては、性能向上に関係の低い要素を選んだり、不要なオーバースペックを生じる可能性がある。とくに、ある構成要素が現行システムの性能ボトルネックとなっているときには、性能向上に関して誤った判断を起こす危険性が高まる。

そのような誤った判断に基づいて次期システムの開発を進めた場合、性能要件が充足しないことが判明するのは、ハードウェア開発もソフトウェア開発も終盤にかかった時点になる。システムアーキテクチャの再検討が必要になるため、開発工程の手戻りによる影響が増大する。それによって開発コストの増大や、計画した出荷時期・運用開始時期の遅延によるビジネス上の利益逸失を招くことになる。このときに、手戻りすることを躊躇して小手先の実装レベルで対処しようとすると、無用の複雑化を招いた上に、結局、性能要件を満たせずに開発が迷走し、更に影響を増大させることになる。

本質的には、システム性能を評価し、性能ボトルネックとなっている構成要素を特定して、その単位性能を向上して、しかるべくシステムアーキテクチャを再設計する必要がある。

まず、ソフトウェアとハードウェア双方の構成要素の利用頻度、占有度などを解析するために、現行システムの性能評価を行う。次に、この現行システムの性能解析結果を用いて次期システムの要件を分析し、次期システムの性能ボトルネックを解析する。システム性能ボトルネックに関しては、机上計算やモデルシミュレーションなどを用い、想定している各条件・シナリオについて解析する。

次期システムの性能要件を充足出来ない条件・シナリオについて、性能ボトルネックとなるハードウェア構成要素の代わりに、置換可能な、単体性能がより高い代替品・代替要素を選択する。必要に応じて、性能面で余裕があるハードウェア構成要素の代わりに、置換可能な、単体性能がより低い代替品・代替要素を選択し、製造コストと消費電力に対する要件を総合的に充足出来るようにする。

ハードウェア構成要素の置換が困難な場合や、あるいは、ハードウェア構成要素の置換によっても解消出来ない性能ボトルネックがある場合は、対応するソフトウェア構成要素のアルゴリズム改善や並列化などによる高速化を検討する。

例 現行システムとして、操作パネルがついている情報処理機器

操作パネルはタッチセンサ式のLCDから構成される。機器本体とユーザーとの対話は、機器の状態についての表示や、機器の処理におけるエラーメッセージなどの表示、機器に対する操作などの入出力であり、すべて操作パネルを通してなされる。

機器のハードウェアは、デュアルコアプロセッサ、キャッシュ、メインメモリ、バス、ビデオメモリ、LCDコントローラなどから構成されている。機器本体の処理はメインプロセッサコアで、操作パネルのUIはサブプロセッサコアで、それぞれ動作する。機器本体の処理対象のデータは、メインメモリに格納されている。操作パネルUIは、パネルの画面に表示する描画データを作成してメインメモリに格納し、タッチパネル上のユーザ操作を処理して機器本体の諸機能を呼び出す。LCDコントローラは、パネルのリフレッシュレートのタイミングに合わせて、メインメモリ上の描画データを読み出してビデオメモリへ転送する。

次期システムは、現行システムの性能向上版として開発されることになった。機器本体の性能向上を企図して、プロセッサの単体性能を30%向上し、かつ、キャッシュサイズも2倍にすることで、与えられた性能要件を充足すると判断した。

しかし、開発を進めてテスト工程に入ると、多くのシナリオで性能要件が満たせないことが判明した。そのため、プロセッサ性能を更に上げて、キャッシュサイズも増やしたものを試作して実験したが、これも満たせなかった。この時点でようやく、システム性能を評価し、ボトルネックを解析したところ、実はプロセッサ性能には余裕があることが判明した。向上したプロセッサ性能によって機器本体の処理能力が上がった結果、メインメモリの単位時間あたりのデータ入出力量が増えていた。もともと、バス帯域は、操作パネルとLCDコントローラによる描画データ入出力によって大半が占有されていた状況にあり、機器本体の処理能力向上で増大した転送データが更に加わり、輻輳が生じていた。これが性能が向上しない原因であった。

ボトルネックとなっていたバス帯域を倍増し、代わりに、余裕があったプロセッサ性能とキャッシュサイズを現行システムより下げて、総合的な製造コストの向上を抑制したシステムアーキテクチャを再設計した。ボトルネックが解消されたため、次期システムは与えられた性能要件を満たすことが出来た。

関連する品質特性

- ●効率性（時間効率性、資源効率性）
- ●保守性（解析性、変更性、安定性、試験性）

A-24 ハードウェア制御を凝集する

作法概要

複数の制御モジュールが排他的に同期をとりながらハードウェア制御を行っている場合は、タイミングを制御するモジュールにハードウェア制御を凝集する。

メリット

●ソフトウェアの効率性が向上する(スループット向上)。
●ソフトウェアの保守性が向上する。

留意点

ハードウェア制御をシーケンシャルでしか行えない場合は、無理に凝集する必要はない。

●解説

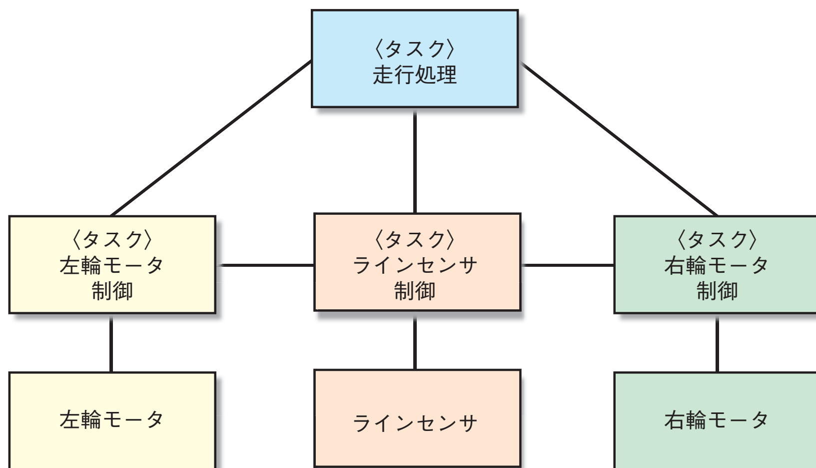
ハードウェア制御モジュールの開発では、シーケンスチャートによる検証が一般的に行われ、各ハードウェアに対し複数のモジュールが制御を行うことも多い。モジュール数が少ない場合、または相互の関連が薄い場合には、問題は起きないが、そうでなければ同期処理の複雑化によって、パフォーマンスの向上が難しいケースも生じ得る。

このように、多くのハードウェア制御が関連性を持つような場合は、ハードウェア制御の同期処理をコントロールするモジュールに、ハードウェア制御を凝集する。これによって、このモジュールだけを改善することで、パフォーマンス向上を図ることが可能となる。

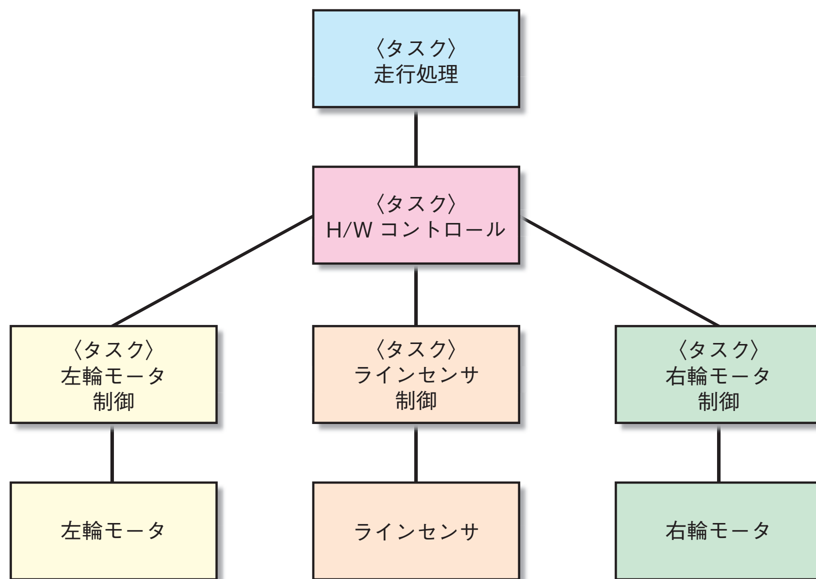
例 ラインセンサや右輪モータ、左輪モータの各ハードウェアを制御しながら、ラインに沿って進むライントレースカー

素朴な設計例の図 A-27 では、制御モジュール間の同期制御が複雑化しやすく、目標のパフォーマンスを達成することが難しい。

そこで、図 A-28 に示すように、各ハードウェア制御をコントロールするタスクを設け、制御モジュール間の同期制御を仲介する役割を与えると、これによって、制御モジュール間での同期制御の複雑さが低減すると共に、目標のパフォーマンスに向けてのタイミング調整に柔軟に対応することが可能となる。



図A-27：悪いモジュール構成



図A-28：良いモジュール構成

関連する品質特性

●効率性（時間効率性）
●保守性（解析性、変更性、安定性、試験性）

A-25 入力パラメータチェックはシステム全体で最適化する

作法概要

関数またはモジュールの入力パラメータが入力許容範囲に収まっているかチェックする機構は、システム全体を考慮したときに最適となるように、その導入に関するポリシーを決定する。

メリット

パラメータチェックを最小限にとどめて、実行速度低下を抑制することが出来る。

留意点

①複数のシステムからモジュールを流用する場合、それぞれのシステムにおけるパラメータチェックのポリシーが異なることに注意する必要がある。
②再利用が想定される場合、入力許容範囲に関する仕様を文書化しておくことが重要となる。

●解説

一般に、各関数に入力として与えられるパラメータは、その関数が想定外の動作を引き起こさないように入力許容範囲が設計時に仕様として決定され、その許容範囲内に収まっているかのチェックをチェック機構によって実行時に行う。このとき、チェック機構の導入に関するポリシーは、システム全体を見通して決定することが重要となる。

例えば、外部インターフェースに相当するモジュールにチェックを凝集し、それ以外の部分については渡されるパラメータは既にチェック済みと認識し、チェック機構を含めないポリシーなどが考えられる。

このポリシーがシステム全体として統一されていないと、設計者は自己防衛的にチェック機構を導入する傾向にある。しかし、システム全体の観点に立てば、本来は入力として渡されることがあり得ない範囲に対しても、過剰にチェックし、実行速度の低下に繋がる可能性もある。

とくに、複数の既存システムからモジュールを流用するケースでは、それぞれのシステムにおける異なったパラメータチェックのポリシーに基づいて設計されている可能性が高いので、注意が必要である。

また、再利用がある程度想定されるモジュールや関数については、チェックが冗長になる可能性を考慮して、設計段階で想定している入力許容範囲に関する情報を文書化するなど、設計情報として第三者が理解出来る形に残しておくことが重要となる。

関連する品質特性

- ●効率性(時間効率性)
- ●保守性(変更性、安定性、試験性)
- ●移植性(環境適応性、置換性)

関連する作法

C-8　パラメータチェックに関するルールをプロジェクトで統一する(P148 参照)

A-26 派生開発でソースコードの再利用性が低ければ設計の見直しを行う

作法概要

派生開発のときの機能追加または変更による影響範囲が、ソースコードの複雑化によって見積ることが出来ない場合は、まず設計の見直し(リファクタリング)を行う。

メリット

●ソフトウェアの再利用性が向上する。
●機能追加または変更による品質劣化が発生しにくくなる。

留意点

①固定部と変動部を明確にしたフレームワークを設計するとき、変動部については、機能、ドライバー、切替機構などに分けて設計すると、機能追加または変更による影響範囲を更に小さくすることが出来る。
②同様に、モジュール、クラス、関数などを責務で分割すると、影響範囲を小さくすることが可能となる。

●解説

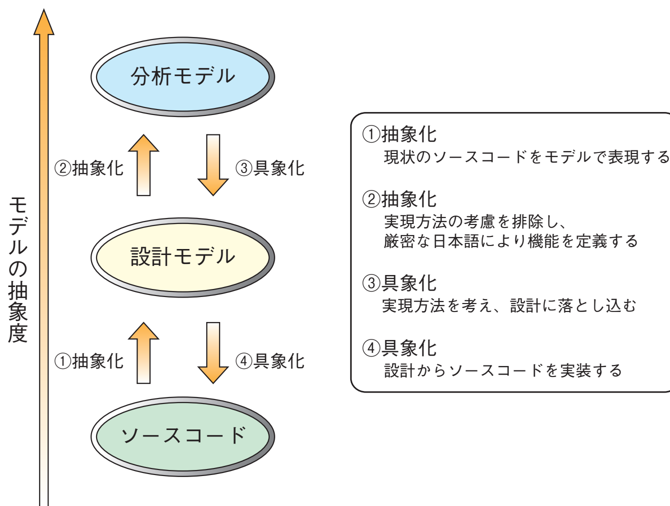
プロトタイプ開発など、機能の実現を優先した開発を実施すると、ソースコードが固定部と変動部を意識せずに作成されるため、派生機種の開発を伴う製品開発においては複雑化しやすくなる。複雑化したソースコードをそのままにして、追加機能または仕様変更を行おうとすると、影響範囲が見極められなくなることがある。

このような状況が発生したときには、設計の見直しを行い、ソースコードの再利用性向上と品質の安定化(機能追加・変更による影響範囲が判断出来る品質の確保)を図る。その結果、品質向上と生産性の向上に繋がる。

再利用性を高めるために設計の見直しを行う場合には、固定部と変動部を明確にしたフレームワークを設計する。これによって、派生開発の機能追加または変更部分が変動部のみになり、影響範囲の特定が容易になる。

例 設計の見直しの手順

既存のソースコードを用いてリバースモデリングを行う方法があり、図 A-29 にその手順を図示化して示す。



図A-29：リバースモデリングの手順

ここで、リバースモデリングを行うときの注意点を次に示す。

①抽象化においては、責務をすべて把握するためにすべてのソースコードをモデル化する。

②抽象化においては、モジュール、クラス、関数などで責務が1つとなるようにする。

③抽象化においては、名前を日本語で書き出し、名前から役割(例えば「〜を変換する」は責務ではなく役割である)を排除した責務名にする。

④具象化においては、固定部と変動部のフレームワークに分けてまとめる。

関連する品質特性

- 信頼性(成熟性、障害許容性、回復性)
- 保守性(解析性、変更性、安定性、試験性)
- 移植性(環境適応性、設置性、共存性、置換性)
- 生産性

関連する作法

C-13　事前に構造解析を行うことにより効率的にソフトウェアを再設計する(P159 参照)

A-27 ソフトウェアの階層構造の深さは、性能要件と保守性のトレードオフを考慮して決定する

作法概要

- 保守性を考慮してソフトウェアアーキテクチャは、階層構造(layered architecture)として定義する。
- 階層構造による実行効率上のオーバーヘッドを見積もり、性能要件が満たせるかどうか解析する。
- 性能要件が満たせない場合、オーバーヘッドを削減出来るよう階層構造を調整する。

メリット

性能要件を満たすためだけに階層構造を崩し、保守性が犠牲にされることを回避出来る。

留意点

①オプションパーツや上位のハードウェアが製品系列に加わると、当初想定していなかった機能を実現するために、ソフトウェアアーキテクチャ全体を見直す必要が生じることがある。
②保守性をある程度損ねたソフトウェアアーキテクチャになりがちなため、それによるリスクを許容出来るかどうかをよく比較検討する。

●解説

ソフトウェアアーキテクチャに階層構造を導入することは、一般的に行われている。図 A-30 のように個々の層は隣接する上位層と下位層とのみ相互作用するように設計することで、抽象化や仮想化、情報隠蔽によるメリットが生じ、これによって保守性が向上する。

図A-30：ソフトウェアの階層構造

一方で、実行効率の面からは、階層構造はオーバーヘッドとなる。従って、性能要件が厳しい場合、階層構造の取決めを破り、中間層を経由せずに下位層を直接利用しがちになる。このため、当初は整然とした階層構造を持つソフトウェアアーキテクチャであっても、開発が進むにつれて随所にバイパス用のコンポーネントが作成されて、複雑化していくことが往々にして起こる。

図A-31：崩れた階層構造

いったん、このように複雑化したソフトウェアアーキテクチャになってしまうと、以降はそれを前提とした設計になるため、開発が進むほど保守性が悪化していくことになる。その結果、ソフトウェア品質は次第に劣化し、新たな製品を開発するたびにバグの発生頻度が上昇していく。

このような問題を引き起こす状況として、開発者自身に階層構造を維持し続けようという意識が欠落している場合が多くあるが、一方で、性能要件を十分に考慮していない階層構造になっている場合も往々にしてある。行き過ぎた抽象化や仮想化などは、1つの処理に必要な階層の数を増大させ、多大な実行オーバーヘッドをもたらす。従って、階層の深さは、性能要件に照らして適正なものにしておくべきである。バイパスせずに性能要件を満たせるのであれば、あえて階層構造を崩す行動をとろうと開発者は考えない。

また、階層構造を崩す改変が加えられた場合、適当なタイミングでソフトウェアアーキテクチャの見直しとリファクタリングなどによる再構造化・再階層化がなされるべきである。

関連する品質特性

●効率性（時間効率性、資源効率性）
●保守性（解析性、変更性、安定性、試験性）
●移植性（環境適応性、設置性、共存性、置換性）

A-28 要件定義では、先に性能要件をシステム全体で評価する

作法概要

システム全体の性能評価をもとに、関連する他のモジュールの振舞いと制約を考慮した上で個々のモジュールの仕様を決定する。

メリット

性能不足による大きな手戻りを未然に防ぐことが出来る。

留意点

ハードウェア開発チームとソフトウェア開発チームが、それぞれ孤立せず、十分な意思の疎通を行えるように留意する。

●解説

システム開発においては、上流工程において機能分割などを行い、個々のハードウェアとソフトウェアモジュールの機能要件及び仕様を定義したあとは、個々のモジュールの開発チームが独立して作業を進めることが多い。しかし、機能要件はモジュール単位に分割できても、非機能要件、とくに性能要件は関連する複数のモジュールにまたがることが多い。このため、関連するモジュールの開発チーム間で、性能要件に関する情報共有が十分になされていないまま各チームが開発作業を進行させると、モジュールの統合時点で性能に関する問題が顕在化することになる。

開発途中でこのような性能問題が生じた場合、進行中の作業をいったん停止してチーム間でコミュニケートすることは、現実的には困難である。また、モジュールの統合時点で、個々のモジュールを調整してアドホックに解決するのも限界がある。従って、このような問題を根本的に避けるためには、モジュール分割して個々の担当チームが開発作業に入る前の時点で、関連するチーム間で緊密にコミュニケートし、性能要件に関する情報共有を十分に行う必要がある。

情報共有が十分になされたあとに、システム全体の性能評価を行う。これも個々のチームの開発作業の開始前に行う必要がある。性能評価結果から、担当するモジュールが受ける負荷などを見積もり、そのモジュールの振舞いと制約条件を考慮して、モジュールの仕様を決定する。

例 操作パネルがついている情報処理機器の開発

操作パネルはタッチセンサ式の LCD から構成される。機器本体とユーザーとの対話は、機器の状態についての表示や機器の処理におけるエラーメッセージなどの表示、機器に対する操作などの入出力であり、すべて操作パネルを通してなされる。

操作パネル用 LCD のサイズ拡大に伴い、ハードウェアをシングルプロセッサ構成からデュアルプロセッサによる AMP 構成に変更し、1 つのプロセッサを操作パネル専用とする必要が生じた。その構成におけるシステム性能を事前に確認するために、サンプルプログラムを作成し、システムの応答時間と画面の描画時間を測定した。ハードウェアの仕様は、その結果を基にして確定した。

ハードウェア開発とソフトウェア開発は並行して進行し、下流工程で統合された。しかし、統合時のテストで期待していた性能が達成出来ないことが判明した。具体的には、パネル上の操作の応答速度が要件を満たしていなかった。サンプルプログラムはGUI、すなわち画面生成のソフトウェアだけで構成されており、それ以外の制御などに必要なコードを一切含んでいなかったため、このサンプルコードを用いた性能測定結果をもとに、ハードウェア仕様を決定することに問題があった。実際は、他の制御やステータス表示などの相互作用が常にある。そのような部分を考慮せずに開発を進行させたのが、要件未達の原因である。ハードウェア開発チームとソフトウェア開発チームがそれぞれ孤立しがちで、十分な意思の疎通を欠いていた。サンプルコードのそのような問題について、ハードウェア開発チームは一切知らされていなかったことが、統合時点まで問題が顕在化しなかった根本の原因であった。

改めて、実機を用いた性能評価を実施した。次期製品は開発中であって存在しないため、現行製品を代替として用いた。現行製品と次期製品で機能要件に大きな違いはなかったため、データ量などの見積もりも現行製品のものを踏襲可能と判断し、次期製品の性能要件に見合うハードウェア仕様を決定した。

関連する品質特性

- **効率性（時間効率性、資源効率性）**
- **保守性（解析性、変更性、安定性、試験性）**

Part B

システムレベルの設計の工夫

B-1 エラー発生時にも処理の継続性を確保する

作法概要

- エラー(障害)発生時にユーザに通知するデータ量を考慮し、エラーメッセージを選別する。
- エラーからの復旧や、エラー発生時の処理継続を可能にする仕組みを用意する。

メリット

エラー発生時にユーザがエラーの状況を把握し、適切に対応がとれる。また、エラーからのリカバリが容易である。更に、エラーが発生した場合でも処理を継続出来る。

留意点

①エラーのレベルを考慮して設計する。
②検出頻度に比べて影響度の小さいエラーは、検出の感度を下げる。
③エラー処理による輻輳の発生に注意する。

●解説

システムに不具合が発生した場合、状況によっては様々な箇所で同時に異常を検出する可能性がある。こういう場合に、異常を検出した箇所がそれぞれ異常をユーザに通知しても、ユーザは有効な対処をとることが出来ず、その上、ユーザインターフェースが輻輳して、操作性が低下してしまう。

このような事態を回避し、ユーザの状況把握と適切な対処やエラーからの復旧、エラー発生時の処理継続を可能にするために、次の事項を検討する。

①エラー履歴情報表示などのデータ量が過大にならないよう考慮する。
②エラー発生時に表示するエラーメッセージを選別する。
③エラーや異常状態を止めるためのアクションを受け付ける仕組みを用意する。
④異常状態から正常状態に戻すためのリカバリ手段を用意する。
⑤リカバリの際にどこまで戻るか決めておく。
⑥リカバリの際に何(データ、処理)をあきらめるかを決めておく。
⑦リカバリに必要なデータを決めておく。
⑧異常発生時に機能縮退などによる処理継続方法を考える。

例 携帯電話

操作をしているときに、SIMカードの接触不良が原因で、回線が切断された。アプリケーションからの「サーバにアクセスできません」、システムからの「SIMカードがありません」という数種類のメッセージが一度に表示された。それでも、表示されるメッセージが邪魔して、ユーザは、アプリケーションをすぐに停止することが出来なかった。

このような事態を回避するため、過去の故障のデータの統計処理を行い、頻度の高いものについてパターン化を行い、それぞれについてユーザーが適切に対応出来るメッセージの表示方法と表示内容を決定した。このシステムに上記のパターンマッチングとメッセージ機構を組み込んだところ、SIMカードの接点の確認を促すメッセージが表示されるようになった。

関連する品質特性

信頼性（障害許容性、回復性）

B-2 システムのバージョン変更が円滑に行えるようにする

作法概要

システム設計の段階で、バージョン変更のための手順とツールを準備する。

メリット

●ハードウェアやソフトウェアのバージョンアップを円滑に行うことが出来る。
●バージョンを戻す必要が生じた場合でも、容易にもとに戻すことが出来る。

留意点

①バージョンアップ・バージョンダウンを繰り返しても、全体として一貫性が保たれるように配慮する。
②バージョン変更はソフトウェアとハードウェアの両方について行われる可能性を考慮する。

●解説

ハードウェア及びソフトウェアのバージョン変更については、システム設計の段階であらかじめ考慮し、出来るだけ円滑にバージョン変更が行えるようにする。そのためには、次の事項などを考慮して、その手順やツールを準備する。

①システムのバージョンアップ・バージョンダウンに対応出来るデータコンバータを用意する。
②状態に不整合が生じても復旧出来るように、プログラムとデータのバックアップを作成する。
③バックアップを保持するためのハードウェアなどを用意する。
④入力は通常通り受け付けるが、出力を行わないというテストモードを利用して、問題がないことを確認してからバージョン変更を行う。

例 炊飯器

節電モードに対応するためソフトウェアのバージョンアップの際、当初はセーブされているデータの途中に節電モードの値を挿入するよう設計した。設計レビューで「節電モード以降のデータのアドレスがずれるため、既存の設定が想定外の値として扱われる不具合が生じ得る」という指摘があった。

そこで、節電モードの値をデータの末尾に保存するよう設計を変更し、これに基づいて実装を行ったところ、バージョンアップに際して既存の値が変わることなく正常動作が確認できた。

関連する品質特性

移植性(環境適応性、設置性、置換性)

B-3 システムの構成に合わせてデータをチェックするメカニズムを用意する

作法概要

信頼境界を考慮しデータチェックを行う。

メリット

データによって、システムが異常な状態に陥ることを回避することが出来る。

留意点

①データが、システムの許容範囲を超えないようにチェックする。
②データ間の依存関係により他のデータが許容範囲を超えることのないようにチェックする。
③入力処理が複数にわたる場合は、その入力順序をチェックする。

●解説

データが許容範囲を超えたなどの原因によって、システムが異常な状態に陥ることを防止するため、データチェックを確実に実施する。このデータチェックでは、次の点などを確認する。

①数値であるか(ニューメリックチェック)
②限界値を超えていないか(リミットチェック)
③書式が整っているか(フォーマットチェック)

データチェックが終わったあとには、許容範囲がデータ値と設定済みデータ値との関連で変化することがあるので、更に次の確認も必要となる。

①関連データとの整合性チェック(バランスチェック)
②入力処理が複数回にわたる場合には、それらの順序チェック(シーケンスチェック)

ここで、シーケンスチェックに関しては、データの抜けやデータ順の逆転、後続するデータの遅延による問題も考慮する。

更に、ソフトウェア設計時においてもデータチェックを効果的に行うために、次の①から③の考慮が必要である。

①チェック関数のエントリのメタデータ化

データの許容範囲及び他データとの関連を確認するチェック関数のエントリを、プログラムへ直接記述せずメタデータ化する(エントリをリストにまとめる)ことは、ハードウェア変更などに伴うプログラム修正回数の削減や、入力処理に連動した関連データの許容値超過の警告または自動補正などを容易にする。

②信頼境界を意識したモジュール分割

信頼境界は、データに誤りがなく、信頼して使える領域の境界を意味し、これを意識してモジュールを分割すれば、冗長なデータチェックを回避することが出来、プログラムの可読性や実行効率を向上させることが出来る。

③データチェックのマクロ化

デバッグなどの利便性のために、各モジュールで防衛的に冗長なチェックを行う場合では、データチェックをマクロ機能で記述し、それによって開発時には有効化し、運用時には無効化することも効果的である。

例 計測機器

計測機器においては、計測条件を設定するデータ入力モジュールと、その設定データに基づきセンサのデータを処理する処理モジュールから構成されることが多い。モジュールごとに開発担当を分けたが、互いに防衛的なプログラムを作成したため、多くの冗長なデータチェックが両モジュールで実施された。

これを改善するため、信頼境界を明確にする再設計を実施した。保守性を上げるために、両モジュールが行うチェック機能を明確にした。具体的には、データ入力モジュールは、リミットチェックを行うことにより設定可能範囲内のデータ入力を保証し、処理モジュールは、内部データと整合性を保証するためのバランスチェックのみを実施するようにした。

関連する品質特性

- 機能性（正確性、セキュリティ）
- 信頼性（成熟性）
- 効率性
- 保守性

関連する作法

C-8　パラメータチェックに関するルールをプロジェクトで統一する（P148 参照）

B-4 システム障害時にも稼働を継続するために縮退運転を検討する

作法概要

●長期連続稼働に必要な資源量(メモリ容量など)を把握する。
●障害レベルに応じて縮退運転方式を設計し、稼働を継続させる。

メリット

長期間の連続稼働が求められるシステムで障害が発生しても、サービスレベルを可能な限り維持することが出来る。

留意点

①システム停止回避のため、資源不足やハードウェア障害に対する代替機能を確保する。
②すべての障害に対する検査、診断を検討する。
③発生した障害に対する隔離、回復機能を検討する。

●解説

製造装置や化学プラントなどでは、一部の障害でシステムが停止すると、生産停止による経済的損失や機器の破壊などが発生することがある。これを回避するために、次の2つのモードに分けて設計することが必要となる。

①通常運転モード
②縮退運転モード

通常運転モードでは、CPU高負荷状態、メモリの不足、フラグメンテーション、ネットワークの通信量超過などが長期稼働においても発生せず、安定した性能を維持する。また、OSやミドルウェアなどのCOTS(Commercial Off-The-Shelf)による想定外の資源使用などが原因で万一の資源不足に陥る場合や、部分的なハードウェア障害の場合には、縮退運転モードに移行してシステムの機能を制限しつつ、可能なかぎり稼働を継続する。縮退運転には様々なケースが想定される。そのため、あらかじめ障害レベルの観点から縮退運転方式を検討し、システムに冗長性を持たせて縮退運転を設計する。

稼働継続が可能な障害レベルに対しては、次の方式が考えられる。

①メモリ不足の場合、省メモリ処理アルゴリズムへの切替えを行い、性能低下を許容しつつ稼働させる。
②ハードウェア障害では、障害部分のハードウェアの再初期化や、代替ハードウェアまたは等価機能のソフトウェア処理への切替えを行い、性能低下を許容しつつ稼働させる。

稼働継続が困難な障害レベルに対しては、代替機能がなく稼働の継続は困難であるが、保守サービスに必要な機能のみを維持し、再稼働の時間を短期化することが考えられる。

それぞれのケースで縮退運転を実現するためには、次の事項の検討が必要である。

①検査：障害部分をどのように検出するか。
②診断：障害はどのような状態か。
③隔離：システムへ影響を及ぼさないために、障害機能をどのように切り離すか。

④回復：障害前の状態からどのように再開するか。

⑤代替：代替機能をどのように動作させるか。

検査は、ハードウェア制御のための指令値を送るたびに、シミュレーションによる内部モデルの予測結果とハードウェアの動作結果を示すセンサ値などを比較する収束テストを実施し、その相違などに基づいて障害の有無を判断する。

診断は、障害直前からの復帰が出来るように、障害直前の内部モデルのデータを保持し、その障害から定められた適切な次の状態(ハードウェア初期化による通常運転、代替機能による縮退運転、サービス機能に制限したユーザメンテナンス運転など)への移行を決定する。

隔離は、システムから障害部分をハードウェア的かつソフトウェア的に切り離し、システム全体への障害の波及を防止する。回復は、必要に応じて仕掛中の生産物を排出するなどのハードウェアの回復動作などを経て、実際の外部状態を内部モデルと一致させる。代替は、障害直前の内部モデルに合うように代替機能を設定し、再稼働を行う。

例 組立てロボットセル

工場のライン上に、ダブルハンドを持った組立てロボットセルがある。片ハンドの障害で１つのロボットセルが停止すると、その影響がライン全体に波及し、全ラインが停止してしまう。全ラインの停止を回避するために、ロボットセルに縮退運転の機能を次のように追加した。

①ハンド位置センサの読取り値が移動指令値と同じではないとの異常を検知し(検査)、

②組立てが片ハンドで可能と判断した場合に(診断)、

③組立てに支障を起こさないように障害ハンドをメンテナンス位置へ安全に退避させ(隔離)、

④正常ハンドのみで作業が出来る組立てシーケンスに切り替え(回復)、

⑤中断した工程から片ハンドによる作業の再開を行う(代替)ことにした。

関連する品質特性

●機能性(合目的性)
●信頼性(障害許容性、回復性)

B-5 単位系を統一する

作法概要

- 内部処理で取り扱うデータの単位系を統一する。
- 入出力時にユーザの指定した単位系へ変換する。

メリット

- ユーザによるデータの誤入力や誤認識を防止し、内部処理を単純化し、信頼性を向上することが出来る。
- ユーザが使いやすい単位系への切替えを可能にすることが出来る。

留意点

内部処理は精度を維持しやすい単位系に統一する。

●解説

産業機器などでは、ユーザや開発者の観点を理由にして、異なる単位系が使用される場合がある。ユーザの観点を重視すれば、ユーザの使用方法に適した単位系を提供する必要がある。これは、機器で提供している単位系がユーザが使用している単位系と異なると、ユーザが機器設定する際にデータ入力を誤る、あるいは機器の出力値の誤認識に繋がるからである。

一方、開発者の観点では、特定ユーザ向けの機能には、そのユーザが使用する単位系や、より単純な計算処理が出来る単位系を採用するなど、開発モジュールの単位系が恣意的になりやすい。

この結果、異なる単位系を採用するモジュールがシステム内に多数存在し、それらの間の相互変換処理も点在することとなり、次のような信頼性や保守性に関わる問題が発生しやすくなる。

①データの相互変換処理に伴って精度が低下し、設計が複雑になる。

②値は異なるが意味は同じというデータが多数存在し、依存関係の判断が困難になる。

これらの問題を回避するために、機器内で使用する単位系をあらかじめ統一し、データの入出力部だけでデータ変換処理を行う。アーキテクチャは、統一単位系の計算処理を中心としたハブ構造となり、相互変換に伴う設計の複雑さが解消される。

ここで重要な点は、単位系の統一にあたっては、力学単位や熱学単位の計算をするのであれば、SI単位(国際単位系)で、かつユーザが多く使用する単位系を選定することである。こうすることで単位変換処理が少なくなり、計算精度を高め、プログラミングにおける勘違いを少なくすることが出来る。また、これとは別に、産業機器ではユーザが想定している外界モデルと、そのための座標系や原点位置といった考慮も重要である。

例 GPS（Global Positioning System）の応用機器

GPSは、複数のGPS衛星からの電波を使用することで、ある地点の座標を知る機能である。地球が完全な球体ではないため、その地点の座標は楕円体や準楕円体の近似モデルへの変換によって得られる。このモデルには、その地域で定められた複数の地域測地系とグローバルな世界測地系があり、同じ座標値であっても、モデルによって地点は変わる。

当初の製品では販売地域が限られていたため、地域測地系を採用した。その後、グローバル展開することとなり、各測地系に合わせた正確な地点を求めるために、座標値を経緯度だけでなく地域ごとの単位系への変換処理が各所に必要となり、ソフトウェア構造が複雑化する可能性が出た。

その対策として、次の再設計を実施し、その結果、ソフトウェア構造は単純化され、ローカライズのための拡張性が向上した。

①内部処理を世界測地系へ統一する。
②世界測地系から地域測地系への変換などを共通モジュール化する。
③入出力時のみ、この共通モジュールを使用する。

関連する品質特性

- 機能性（正確性、相互運用性）
- 信頼性（成熟性）
- 使用性
- 保守性

B-6 スタックがオーバーフローしないようにする

作法概要

- 設計段階で各モジュールが必要とするスタック量を見積もる。
- スタックを不必要に大量に使用しないように、コーディング規約を整備する。
- スタック領域にあらかじめ特定コードを書いておくなどの工夫を講じて、実際の使用量を確認する。

メリット

スタックオーバーフローを防止することが出来る。

留意点

①呼出しスレッド、割込みタイミングを考慮して、スタック領域に適切な余裕を持たせる。
②データ使用量が利用側モジュールにより変動するような設計の場合、適切なスタック量の計算が困難な場合がある。
③再帰呼出しは行わない。どうしても使用しなければならないときは、再帰呼出し回数を厳しく制限する。
④データサイズが大きい自動変数を宣言するなど、スタックを大量に使用するようなコーディングを行わない。

●解説

システムで使用するスタック量は、設計段階で見積もることが原則である。この見積もり方法としては、次のように簡易的に算出する方法がよく用いられる。

①関数呼出しのネストが最も深くなる処理に着目する。
②あるいは、自動変数の合計サイズが比較的に大きい関数に着目する。
③関数内の一連の関数呼出しにおいて、呼び出される関数内で宣言される自動変数と、呼び出すときの引数のデータサイズを求め、それらを積算する。

ただし、実際に使用されるスタックサイズは、ターゲットとなるCPUのアーキテクチャやコンパイラの最適化オプションによって前後するため、多少のマージンを取っておくことが望ましい。

コーディング面では、次のような点について、コーディング規約などで規制すべきである。

①データサイズの大きい自動変数を使用せず、動的メモリを利用する。
②あるいは、呼出し元が、必要なメモリ領域をヒープなどに確保し引数で渡す。
③再帰呼出しは使用しない。

また、スタックオーバーフローの検証のためには、実行中にスタックがどれだけ消費されたかを確認出来る仕組みが有効である。例えば、スタック領域の全体、あるいは最後尾に特定のデータを書いておき、システムを起動してから任意のタイミングでそのデータが変更されたかどうかをチェックする方法がある。

例 スタック使用量の見積もり方法

次のプログラムに示すような呼出し関係になっている場合、関数test()を実行するとtest()→getArea()→func0()→func1()→func3()の経路が最もネストが深くなる。関数test()ではint型の自動変数が6つ定義されており、また関数getArea()を呼び出すためにint型の引数が5つ渡されている。これよりint型のデータサイズを4バイトとして、関数test()の階層では4×(6+5)=44バイトのスタックが使用されると算出する。以後同様に、getArea()、func0()、func1()、func3()の階層で使用されるスタック量を算出するとそれぞれ24バイト、16バイト、4バイト、0バイトとなり、合計した44+24+16+4+0=88バイトが関数test()を呼び出したときに使用される最大のスタック量と見積もることが出来る。

実際は、更に関数呼出しごとにレジスタやリターンアドレスなどの退避領域が必要になるが、その量はCPUのアーキテクチャやABI(関数やシステムコールのインターフェース規約)、使用するコンパイラなどによって異なる。

```
int func3(int x) {                                          /* 0byte */
    return = x * x;
}

int func2(int c, int x) {
    return = x + c;
}

int func1(int a, int b, int x) {
    return = a * func3(x) + b;                              /* 4*1=4byte */
}

int func0(int a, int b, int c, int x) {
    int value;                                              /* 4*1=4byte */
    value = func1(a, b, x) + func2(c, x);                   /* 4*3=12byte */
    if (value < 0) {
       value *= -1;
    }
    return value;
}

int getArea(int a, int b, int c, int start, int end) {
    int x, sum = 0;                                         /* 4*2=8byte */
    for (x = start, x < end; x++) {
        sum += func0(a, b, c, x);                           /* 4*4=16byte */
    }
    return sum;
}

int test() {
    int a = 3, b = -2, c = -6, start = 0; end = 30, area;   /* 4*6=24byte  */
    area = getArea(a, b, c, start, end);                    /* 4*5=20byte  */
    return area;
}
```

図B-1：見積もるコード例

関連する品質特性

安全性

B-7 割込みレベルはシステム全体を考慮して設計・保守する

作法概要

- 機能の追加や複雑化に対応するためであっても、割込みレベルは安易に増やさない。
- アーキテクトが「アーキテクチャー変更べからず集」などのガイドライン文書を作成して、開発者に設計思想を周知しておく。

メリット

優先度逆転などの予期しない問題を未然に防ぐことが出来る。

留意点

①割込みレベルの増加や変更は、アーキテクトのみがその権限を持つようにする。
②開発者自らが行う場合には、アーキテクトの設計思想をよく理解した上で、それと矛盾しないように設計変更する。

●解説

タスク優先度が適切に設計されていないと、優先度逆転と呼ばれる現象が生じやすくなる。この現象は、優先度の低いタスクがある共有資源を使用している間に、優先度の高いタスクが同じ資源を要求すると、資源競合により待ち状態に入り、結果的に、優先度の低いタスクの実行が、優先度の高いタスクより優先される現象を意味する。多くの場合、優先度の高いタスクは厳しい時間制約を与えられており、その実行が遅れると、システムのリアルタイム性に大きな影響を及ぼす。従って、優先度逆転が頻繁に生じると、リアルタイム性に対する要求を満たせなくなる可能性が強くなる。

優先度逆転を回避するためには、ハードウェアの制約に対応するように割込みレベルを設計し、それに合わせてタスク優先度を設計すればよい。しかし、長期にわたって機能追加や変更要求が繰り返されると、当初の設計を逸脱することがある。とくに、納期が逼迫しているなど開発スケジュールが厳しい状況にあるとき、開発者はアドホックな変更で対処しがちである。つまり、不必要に割込みレベルを増やし、タスク優先度の設計を複雑にしてしまう。そのようなアドホックな変更は本来の設計思想を考慮せずになされるため、繰り返されると予想外の影響を引き起こす遠因となる。しかも、そのように変更され複雑化したソフトウェアは、優先度逆転が発生するとその原因箇所の特定が困難となる。

割込みレベルの設計変更はシステム全体に影響を与えるため、統制が不可欠であり、システムの性質を深く洞察し将来の要件も想定した上で、適切な割込みレベル設計をしておく必要がある。

また、設計思想を正しく継承させるためには、「アーキテクチャー変更べからず集」などのガイドライン文書を用意しておくのも有効である。

例 「アーキテクチャー変更べからず集」への記載

①新規の割込みレベルを安易に追加すべからず。既存の割込みレベルで目的が果たせないかをよく考慮する。

②タスク依存関係分析や処理遅延に関する影響分析が不十分なままで、新規の割込みレベルを追加すべからず。

新規追加の割込みレベルに対応するタスクの依存関係を分析し、かつ、他のタスクに与える処理遅延などの影響が許容範囲に収まるかを、シミュレーションなどで分析する。

③大きな影響を受けるタスクをそのままにすべからず。

影響が大きいタスクを複数の単位に分割するよう再設計する。

関連する品質特性

- **効率性(時間効率性、資源効率性)**
- **保守性(変更性、安定性)**

B-8 扱うデータの特性に応じてシステムの構成要素を分離する

作法概要

- システムまたはソフトウェアの構成要素を、それらが取り扱うデータの特性に基づいて分類する。
- 分類結果に基づいて、アーキテクチャ設計ではデータの配置と構成要素間の連携を定める。

メリット

重要度の高いデータと処理が、重要度が低いデータと処理によって影響を受けるという不都合を回避することが出来る。

留意点

データ特性は多種あり、その程度も多様であるため、一方の性質を考慮すると他方を考慮できないという場合があり得る。最終的なアーキテクチャは、データに求められる性質のみではなく、責務の粒度や処理の独立性、保守性なども考慮して決定する必要がある。

●解説

組込みシステムの高機能化・高集約化に伴い、組込みソフトウェアの機能・処理が多岐にわたるようになっている。また、ネットワーク化によって、組込みシステムまたはソフトウェアが扱うデータが多種多様になっている。このような状況によって、機能性・信頼性・保守性を高めるために、システムアーキテクチャ及びソフトウェアアーキテクチャの重要性が増している。

アーキテクチャ設計では、機能・処理の分担に基づく検討が主に行われているが、多種多様なデータを扱う組込みシステムまたはソフトウェアに対しては、データの特性に応じた検討も重要である。このときには、次のデータの特性を考慮すべきである。

①QoS（Quality of Sevice　信頼性・応答性）レベル
②データ鮮度（更新・参照の頻度・周期）
③情報セキュリティポリシ

ここで注意すべきことは、特性が異なるデータが混在すると、機能性・信頼性・保守性が損なわれる要因となることである。

特性が異なるデータを分離するためには、システムアーキテクチャ設計の段階では次の選択肢が考えられ、ハードウェア面の制約などに応じて適切なものを選択する。

①サブシステムとして分離
②ボードやSoC（System-on-a-Chip）レベルで分離
③メモリチップで分離

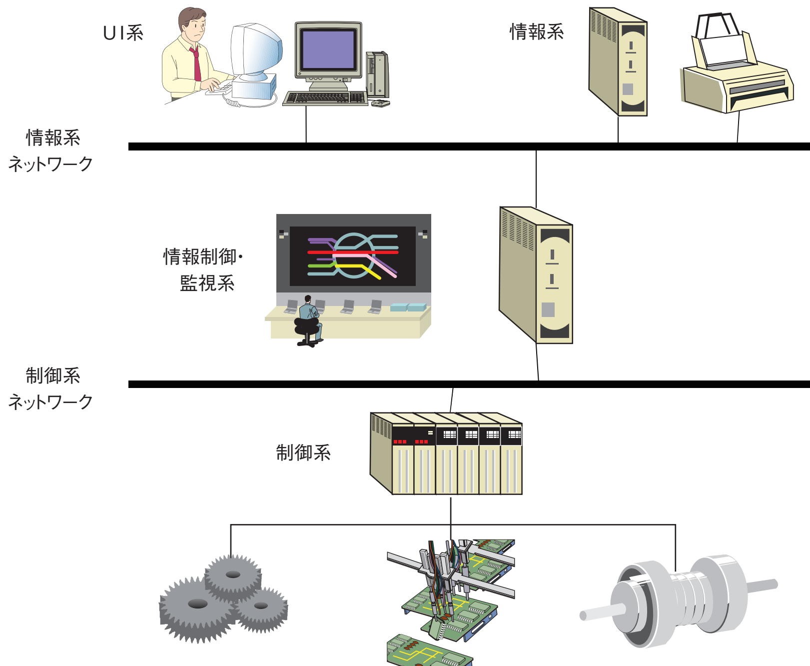
ソフトウェアアーキテクチャ設計の段階では、データ管理に関するソフトウェア機能（OSのメモリ保護機能やDBMSなどのミドルウェア）を用いることが出来る。

例 プラントや機器などの制御・監視・操作を行うシステムのように、制御系に情報系・UI系を追加しているシステム

プラントや機器などの制御と監視を行う部分では、その制御・監視の周期に対応して、制御量と状態値を示すデータを更新・通信することが必要となる。一方で、操作員などのユーザに対応する部分では、ユーザの操作に合わせた頻度でデータ更新が行われる。また、情報系でバッチ処理を行う場合には、データ量が通常はほとんどないが、バッチ実行時に集中して増加することになる。

これらのデータが干渉し合うように配置してしまうと、例えば、バッチ実行時には、制御に必要な周期を保つことが難しくなるというような事態が発生し得る。更には、ユーザ操作部分に不具合が起きた場合には、制御データを破壊してしまう可能性があり、引いては、誤制御を引き起こしてしまうかも知れない。

このような可能性を考慮して、制御系と情報系・UI(User Interface)系は別のサブシステムとして構成するのが好ましい。



図B-2：制御系に情報系・UI系を追加しているシステム

関連する品質特性

- ●機能性（正確性、セキュリティ）
- ●信頼性（成熟性、障害許容性、回復性）
- ●保守性（変更性、安定性）

B-9 メモリの動的確保に留意する

作法概要

- **動的にメモリを確保することを、安易に行わない。**
- **少ないメモリを有効に活用したい場合、固定長領域のメモリ管理機構を用意する。**

メリット

長時間稼働時においてもメモリのフラグメンテーションを回避し、メモリ不足による動作不良を発生させない。

留意点

固定長領域のメモリ管理を行う場合、確保したいデータの使用状況の統計を取って、領域のサイズと個数を決めるようにする。

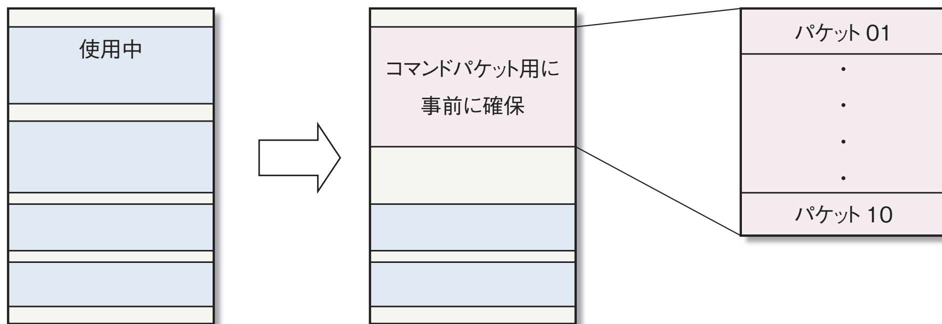
●解説

長時間稼働が要求される組込み機器において、malloc/free 関数などを使用して不定長のメモリ領域を動的に確保することは、非常に危険である。長時間稼働しているうちに、解放された領域と使用中の領域が細かく混在して、フラグメンテーション(断片化)という現象が発生し、全体的には空き容量はあっても、希望するサイズのメモリ領域が確保出来なくなってしまうことがある。

フラグメンテーションを発生させないためには、動的なメモリ確保・開放処理を行わないのが一番の近道である。しかし、少ないメモリを有効に活用したい場合も多々あり、そのような場合には、確保したいデータの大きさや頻度などの統計を取って、あらかじめサイズと個数を決め、固定長のメモリ領域を確保・開放するためのメモリ管理機構を自作する。

例 コマンドパケット

ある OA 機器内においてタスク間でメッセージを交換するときに、そのためのコマンドパケットを不定長サイズでヒープ領域に確保していた。連続稼働試験を行ったところ、メモリの残容量は十分あるのに、コマンドパケットのためのメモリ領域が確保できず、処理が起動できなくなった。コマンドパケットの使用状況の統計を取ったところ、ほとんどが 64 バイト以下で、同時使用は最大で 10 個であることが判明した。そこで、64 バイト固定長のメモリ領域を事前にピーク個数を上回る数だけ確保して、タスクからの確保･開放依頼に応えるメモリ管理機構を内製した。この結果、フラグメンテーションを回避し長時間稼働を達成した。



図B-3：コマンドパケット用領域を確保

関連する品質特性

●信頼性
●使用性（運用性）

関連する作法

A-21　メモリの動的な扱いを避ける（P68 参照）

B-10 状態遷移設計を適度なタイミングで見直す

作法概要

●状態または遷移の数が増加してきたら、状態遷移設計を見直す。
●重複する状態と不要な状態を減らし、階層化によって見やすくする。

メリット

第三者に対する設計の理解容易性が向上する。

●解説

状態遷移設計は、ソフトウェアの動的振舞いを分析し、記述するための設計手法の1つである。状態遷移設計の結果は、状態遷移表または状態遷移図によって可視化され、第三者に対する設計の理解容易性が向上する。しかし、一般にソフトウェアが大規模化するにつれ、状態の数または遷移の数が増加してしまう傾向にある。それに伴い、状態遷移図も、多数の状態の中に多数の遷移が交錯するような複雑な図になる傾向にあり、本来の目的とは逆行し、設計の理解容易性が悪化する恐れがある。

このような事態を防ぐためには、状態遷移設計を適度なタイミングで見直す必要がある。具体的には、次のような観点を考慮して、見直しを行う。

①類似した状態が多数定義されていないか

例えば、対象機器に「モードA」と「モードB」といった2つのモードが定義されているとする。あるSという状態について、「モードA時のS状態」と「モードB時のS状態」として2種類定義されているような場合は、工夫をすれば1つの「状態S」として定義可能であるかも知れない(このようなケースは、「モードA」のみを先に設計し、あとで「モードB」を機能追加したときに発生しやすい)。

②本来は状態として定義しなくてもよい状態が定義されていないか

一般に、状態は、入力されたイベントに対する振舞いを規定するために定義される。複数の状態について、あるイベントに対する振舞いが全く同じであれば、これらの状態は1つの状態として定義すべきであると言える。このような観点で、本来は状態として定義しなくてもよい状態が、定義されていないかを調べる。

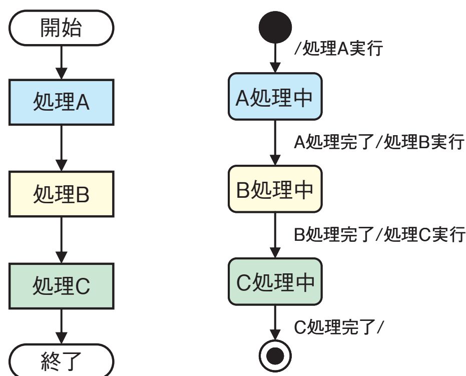
③状態遷移を階層化できないか

状態または状態遷移が増加するにつれて、それらの管理は困難さを増していく。これを改善するには階層化が有効である。通常、複数の状態間には、親子関係をはじめとした、何らかの関連付けが可能であることが多い。ここに着目し、複数の状態をある観点でグルーピングし、それらを階層的に管理すれば、設計の見通しの改善が期待出来る。

まずは、処理または機能として関連性の深い状態を1つのグループとしてまとめてみる。次に、各グループ間はなるべく独立した形でまとめることを意識する。例えば、自グループと他グループの間の遷移が、なるべく1つの状態に集約されている方が、両グループ間の独立性が高いと言

える。仮に1状態に集約できない場合にも、なるべく集約する形でまとめられないかを検討する。
また、別の観点に状態の粒度がある。多数の状態が定義されていると、様々な粒度の状態が存在する可能性が高い。同一階層の状態には、なるべく同じ粒度の状態を集約するように整理し、階層化する。
一般に、1枚の状態遷移図として全容を把握出来る状態数は最大7程度とも言われており、これも整理のときの1つの目安になる。

例1 本来は状態として定義しなくてもよい例

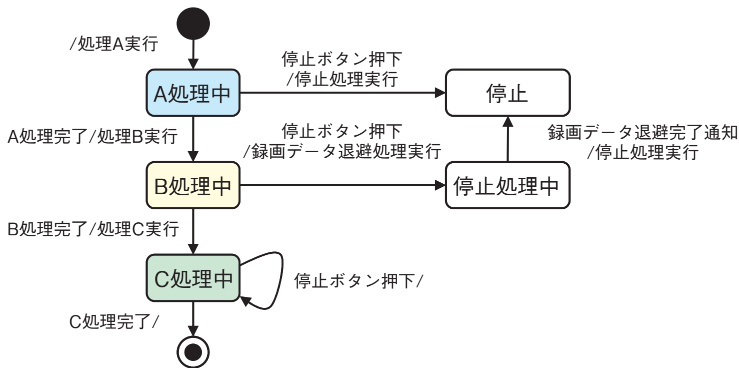


図B-4：逐次的処理実行

ある「処理A」「処理B」「処理C」を逐次的に実行するケースについて説明する。フローチャートで記載すると、図B-4の左側のようなフローとなるが、各処理にそれぞれ1つの状態を対応させて、図B-4の右側に示すように状態遷移設計することも可能である。処理Aを実行中は「A処理中状態」にあり、処理Aが完了すると、自身に対して「A処理完了イベント」を送り、それを契機に、「処理B実行」アクションを行い、「B処理中状態」へ遷移する。処理Bが完了したときには、同様の仕組みで「C処理中状態」へと遷移し、最終的に終了状態へと遷移する。

しかし、これらの逐次的処理を実行している間には、外部からあるイベントを受信したときにとるべき振舞いが同じであれば、状態としては1つの状態とすべきである。例えば、BDレコーダのようなAV再生機器を例にとり、逐次的処理として録画処理(処理A、処理B、処理Cを順に実行することにより映像の録画が行われる)を想定する。この録画処理実行中には、「処理A、処理B、処理Cのいずれを実行していても、ユーザによる停止キー押下イベントが入力されれば、即座に停止処理を行う」という振舞いをとるのであれば、状態としては1つの「録画状態」として定義すればよい。

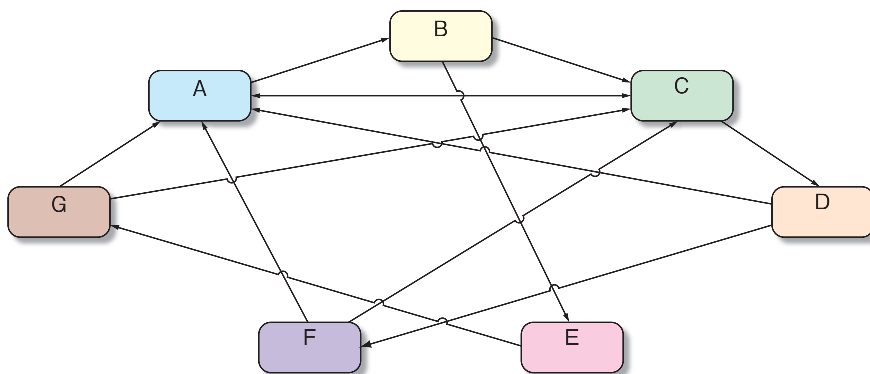
ただし、別状態として扱わなければならないケースもある。例えば、「処理A実行中は即座に停止処理を行うが、処理B実行中は現在録画中のデータの退避処理を行ってから停止処理を行う。処理C実行中は何も変化しない」というように、それぞれの処理実行中にとるべき振舞いが異なる場合は、それぞれを別状態として定義する必要がある。この場合の状態遷移図は図B-5のようになり、元の図B-4とは違ったものになる。



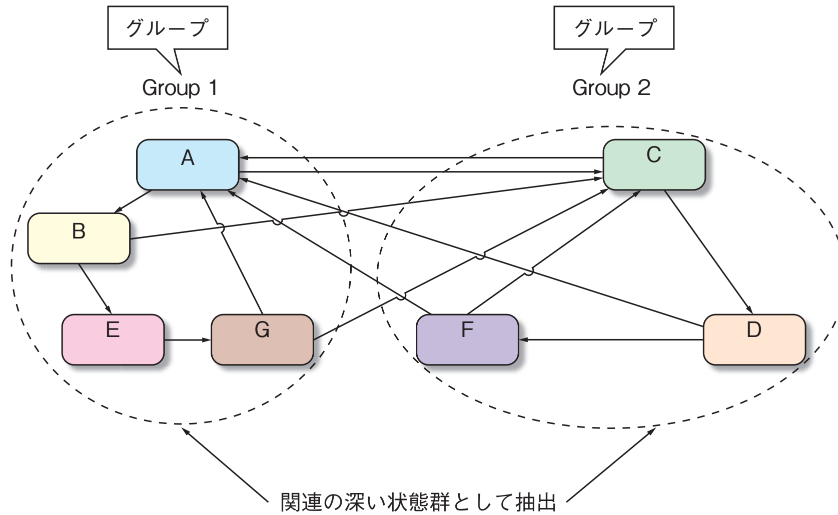
図B-5：本来あるべき状態遷移図

例2 階層化の例

図 B-6 のように A から G の状態が定義された状態遷移図を例に説明する。まずは、これらの状態のうち、1 つの処理または機能として関連の深い状態群を抽出、分類できないか検討をする。ここでは、状態 A、状態 B、状態 E、及び状態 G は 1 つの機能として関連の深い状態群であり、状態 C、状態 D、及び状態 F は別の 1 つの機能として関連の深い状態であると分析できたとして、各状態群をそれぞれ 1 つのグループとしてまとめる（仮に前者をグループ 1、後者をグループ 2 と命名する）。

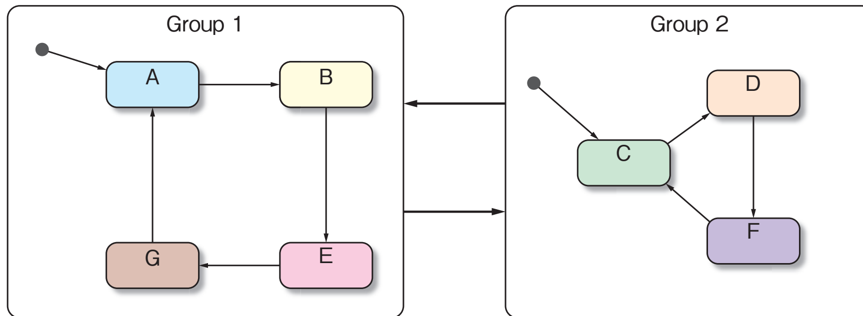


図B-6：整理前の状態遷移図



図B-7：関連の深い状態遷移群を抽出、整理したあとの状態遷移図

次に、状態遷移について着目すると、グループ1に属する状態は、グループ2へ遷移するときには必ず状態Cへ遷移していることが分かる。同様にグループ2に属する状態は、グループ1へ遷移するときには必ず状態Aへ遷移していることが分かる。これらの考察をもとにグルーピングし、階層的に整理した結果が、図B-8に示す状態遷移図である。ここでは、グループ1とグループ2に相当する親状態を定義し、各親状態への入口となる状態を、それぞれ状態A、状態Cとしている。



図B-8：整理後の状態遷移図

関連する品質特性

保守性（解析性、変更性、試験性）

B-11 不必要な待ち操作は避ける

作法概要

終了時刻の異なる複数のモジュールを同時に動かす場合に、遅いモジュールの完了を待たずに後続の処理を進める。

メリット

ユーザが体感する処理時間を短縮することが出来る。

留意点

遅いモジュールの処理は完了していないため、それに伴う不都合を避けるために、処理が完了するまで制限をかけるなどの対処策を講ずる必要がある。

●解説

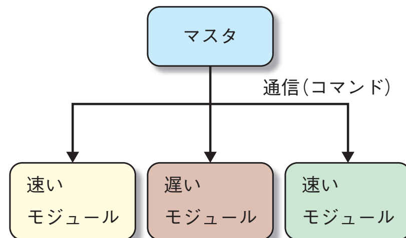
例えば、デバイス起動時の初期化処理を考える。モジュールごとに初期化を指示するが、一部のモジュールの初期化に時間がかかる場合がある。このような場合、すべてのモジュールの初期化完了を待っていると、ユーザに対するサービス提供が遅くなってしまう。出来るだけ早くサービスを開始するには、遅いモジュールの初期化も完了したとみなす必要がある。初期化が完了したことで、サービスを開始し、初期化が完了している部分のサービスを行うことが出来る。

ここで考慮すべきことは、ユーザが初期化が完了していないサービスを要求した場合への対処である。遅いモジュールの初期化はまだバックグラウンドで進行中であるため、リモートからの要求であればサービス要求をキューイングしたり、デバイスの直接の操作者からの要求であれば、初期化が完了していない旨を通知したりする対処が必要となる。

例 モジュールの動作

図 B-9 のような構成で、マスタからのコマンドを受けて、各モジュールが動作するとする。

通常、各モジュールはマスタからのコマンドを受信し、処理が完了するとマスタへ完了通知を行う。マスタは完了通知を受け取ると、後続の動作を指示する。



図B-9：各モジュールの動作

このとき、すべてのモジュールを一斉に動作させるコマンド（例えば初期化）を実施すると、処理が速いモジュールは、処理が完了しているにもかかわらず、遅いモジュールの処理が終わるまで後続のコマンドを受け取れない。

この対策として、遅いモジュールはコマンドを受信すると、すぐに完了通知を行うようにする。その後、本来の処理を実施する。マスタは後続の処理を実施するための準備をする必要があるため、その準備が完了するまでの間に、遅いモジュールの処理を完了させることが出来る。

関連する品質特性

- **効率性（時間効率性）**
- **使用性（魅力性）**

システムレベルの設計の工夫

Part B

B-12 機能仕様を決める際には例外への対策を網羅的に行う

作法概要

MPUの例外とMPUの外部インターフェース（周辺デバイス、周辺サブシステム）の例外を想定して、対策を機能仕様に盛り込む。

メリット

高信頼なソフトウェアを構築することが出来る。

留意点

①正常系をまず定義して、想定可能な例外系の対応仕様を追加定義するようにする。
②対象のソフトウェアを実行する MPU 自身と MPU の外部インターフェース（周辺デバイス、周辺サブシステム）に分けて例外を漏れなく洗い出す。

●解説

対象 MPU とその MPU の外部インターフェース（周辺デバイス、周辺サブシステム）の例外を考慮仕切れない場合、入力データが想定以上の値になったり、インターフェースが不整合になって値が取得出来なかったりなどの事態が発生する。その結果、外部インターフェースから取得するはずの初期値が不定となり、想定外の動作を引き起こす可能性がある。また、外部割込みが頻繁に発生して、想定以上に割込み処理プログラムが動作してしまい、他のプログラムが動作出来ずに、システム障害に繋がってしまうこともある。

例外としては、MPU に関して暴走、リセット、停電などの例外があり得る。周辺デバイスまたはサブシステムに関しては、断線、故障、暴走、リセット、来る、来ない、長い、短い、大きい、小さい、多い、少ないなど、様々な例外が想定出来る。

これらの想定可能な例外を洗い出し、次の手順に従って、例外発生がシステム障害に繋がらないように対策を定義する必要がある。

① MPU とその MPU の外部インターフェース（デバイス、サブシステム）を例外リストに列記する。
②例外リストに列記した各項目に対して、発生する例外を定義する。
③ MPU または外部インターフェース（デバイス、サブシステム）が、ソフトウェア機能要件リストに記載されているいずれかの機能項目に使用されていて、かつ、例外が定義されている場合、例外が発生してもシステム障害に繋がらないように機能仕様を見直す。

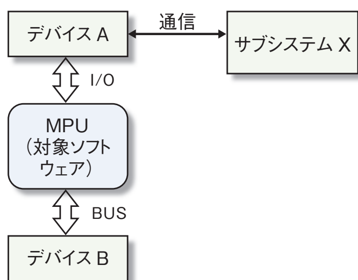
例　例外発生によって入力データが取得できない、または正常にアクセスできない場合

この場合、次のような対策が考えられる。

①入力デバイス A の例外(故障)で入力データが取得出来ない場合に備えて、システム障害にならないように安全に動作可能な初期値をあらかじめ設定しておく。

②デバイス A を経由するサブシステム X から通信データを取りこぼす場合、デバイス B とのインターフェースが不整合になり正確な入力データを確立出来ない場合、それぞれの場合で前回のデータを使用してシステム障害に繋がらないようにする。

③デバイス B とのインターフェースが不整合になり、デバイス B を正常にアクセス出来ない場合、デバイス B の初期化を試みても復旧しなければ、デバイス B のリセットを実施して復旧処理を実施する。



図B-10：例外が発生したときの対策

インターフェース	リセット	遅延	断線
デバイス A	○	○	―
デバイス B	―	―	―
サブシステム X	○	○	○
MPU	○	―	―

○：例外発生の影響あり
―：例外発生の影響なし

表B-1：例外リスト

関連する品質特性

信頼性

関連する作法

A-16　ループ処理からの例外脱出の仕様を定義する(P57 参照)

B-13 可変長データを扱うシステムではデータ長の異常への対策を行う

作法概要

- 可変長データを扱うシステムを設計するときには、データ長の異常を速やかに検知し復旧出来るように、データ長の制限や再送要求のプロトコル定義などを行う。
- 可変長データの処理を設計するときには、データ長に異常があっても予期しない動作とならないように網羅的な対策を行う。

メリット

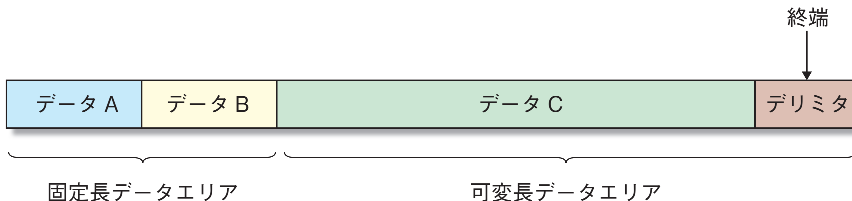
- ソフトウェア自身(ロジック)の信頼性を高めることが出来る。
- 予想外の動作を防止出来る。

●解説

可変長データのデータ通信においては、何らかの要因(例えば無線通信での電波状況)によって、受信したデータがデータ化けすることがある。この場合に、通信データのデータ構造設計として、データの終端をデリミタ(データの区切りを表す記号または特殊文字)のみで判断する設計を行うと、デリミタ部分がデータ化けしたときには、終端が判断出来ずに延々とメモリ領域を検索し続けることになり、その結果、システムが停止するなどの不具合が発生する可能性がある。

従って、可変長データによるデータ通信の設計を行う場合は、データ化けが起きても不正アクセスを引き起こさないデータ構造設計を行ったり、通信プロトコルとしては、データ化けやデータ抜けなどが発生した場合に再送を行う仕組みを設計するなどして、システムの安全性を確保することが重要である。

例 可変長データのデータ通信において、データの終端をデリミタのみで行う、図 B-11 に示すデータ構造設計



図B-11：通信データの構造

この場合、デリミタ部分がデータ化けすると、デリミタが現れるまで、本来、デリミタが存在した領域以降のメモリ領域を延々と検索することとなる。

このような状況を回避する方法として、ここでは２つの方法を説明する。

1つの方法としては、データ通信するデータ領域の最大データ長を設計仕様で取り決め、最大データ長までにデータの終端が検出出来なければ、データが壊れていると判断する。

もう1つの方法としては、図B-12に示すように可変長データエリアの先頭にデータ長の領域を設けることで、データCのデータ終端を検索しなくても判断出来る構造にする。

図B-12：通信データの構造（改善例）

更に、図B-12のデータ長の値に最大値を設計仕様で取り決めると、データ長自体がデータ化けしても、設計仕様上の最大値を超えている場合にはデータが壊れていると判断することも可能になる。

データ通信を行うシステムにおいては、更に、受信したデータの信頼性を確保する必要もある。受信したデータが壊れていると判断した場合に、正しいデータを受信する仕組みを考慮すれば、さらなる信頼性を確保することが可能となる。

正しいデータを確保する方法として、その例を次に挙げる。

①同じ通信データを2つ送信し、2つの受信データの内容が一致していることを確認する。
②送信データにサムチェックを付加し、それを受信側で検証する。
③送信番号を付加し、それを検査することによって、パケットデータの抜けを検出する。
④エラーが発生した場合には、再送を要求出来るようにする。

関連する品質特性

●信頼性（成熟性、障害許容性、回復性）
●安全性

B-14 ログ・トレースに関するルールを定める

作法概要

- ●ログ・トレースのフォーマットと取得レベルを規定する。
- ●ログ・トレースを取得するための関数またはマクロを設ける。

メリット

●規定のフォーマットに基づいたログ・トレースのビュアーソフトウェアの開発・利用によって、作業効率を向上させることが出来る。
●取得レベルを定めることによって、限定されたメモリ領域の中で注目している情報を効果的に取得することが出来る。

留意点

ログ・トレースを取得することは、性能・動作タイミングにある程度､影響を及ぼす。また、保存先が実システム上のメモリ領域であった場合、保存領域が限定されるため保存可能な時間が限られる。これらのことを考慮に入れ、状況に応じて取得範囲やレベルを調整する必要がある。

●解説

シミュレータなどのクロス開発環境の普及・利用が進んでいるが、実際のシステム上でのトラブルシューティングや性能チューニングなどを行わなければならない局面はまだ多い。このときにソフトウェアの挙動を把握・分析するための情報として、ログとトレースが使われている。

しかし、これらのログ・トレースを無秩序に取得すると、性能や動作タイミングなどに大きな支障を及ぼして、ソフトウェアの挙動が大きく変わってしまったり、限られたリソースを使い果たして(上書きすることによって取得出来る時間が短くなって)しまったりということが問題になる。

このため、全体としてログ・トレースのフォーマットと取得レベルを規定しておくことが有用である。取得レベルの規定では、次の事項を考慮する。

①取得の範囲
②取得データの詳細度
③取得の優先度
④取得の条件

更に、これらを統一・徹底する手段として、ログ・トレースを取得するための関数またはマクロを用意し、システムのどの場所でもこれを用いてログ・トレースを取得するようにする。

関連する品質特性

●保守性(解析性、試験性)
●生産性
●効率性(時間効率性、資源効率性)

B-15 ハードウェア変更時の動作を保証する

作法概要

ハードウェアのバージョンの違いを自動的に吸収出来るようにソフトウェアを設計する。

メリット

ハードウェアのバージョン・構成が変化しても、ソフトウェアが正しく動作することが出来る。

留意点

将来のハードウェアの仕様変化を適切に予見することは出来ないため、過剰にハードウェアの変化を見込んで設計またはコーディングしても、活用されない場合がある。変化を見込んだ設計またはコーディングを行うかどうかを適切に見極める必要がある。

●解説

ハードウェアは様々な状況で仕様が変化する。その変化に対して、ソフトウェアは正しく動作する必要がある。そのために、ハードウェアのバージョンを起動時に確認し、それに合わせてソフトウェアの動作を自動的に変化させる。

ハードウェアの仕様変化や違いなどは、次のような状況が想定される。

①機能追加によるバージョンアップ。
②不具合対応によるバージョンアップ。
③試作版と製品版における違い。
④顧客の違いによるハードウェア構成の違い。

ソフトウェア構成としては、アーキテクチャ設計の段階でハードウェアの変化を考慮し、フレームワークになる固定部とハードウェア仕様に依存する可変部分とを、あらかじめ切り分けた構造が向いている。

例 ソフトウェアによるハードウェアの初期化処理

ソフトウェアを搭載するハードウェアが開発中である場合、ハードウェアの小さな仕様変更が発生することは十分に考えられる。

例えば、ハードウェア起動時にソフトウェアによってハードウェアの初期化を行っているとする。その初期化手順と設定すべき値は、ハードウェアのバージョンによって異なってくる。そのために、ハードウェアの初期化処理をハードウェアのバージョンの違いを吸収出来る構造にしておく。

関連する品質特性

●信頼性（障害許容性）
●保守性（変更性）

B-16 メモリ領域を効率良く使用する

作法概要

- ●無駄にメモリ領域を使用しない。
- ●メモリ領域を効率良く利用するための仕組みを検討する。

メリット

メモリ不足に陥らない。

留意点

効率よく利用する工夫として、ダウンキャストまたは多重継承を利用する場合は、その危険性に注意する。

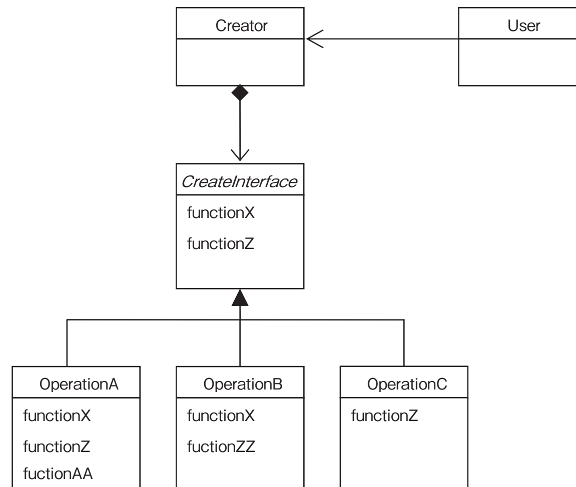
●解説

組込みシステムにおいては、その構成によっては余裕のあるメモリ領域を使用出来ない場合もある。そのためメモリについては十分注意する必要がある。

十分なメモリ領域がない場合は、ソフトウェアに工夫が必要となる。

例 オブジェクト指向設計による組込みシステム

通常は、起動時に必要なすべてのオブジェクトを生成し動作させる。十分なメモリ領域がない場合は、制御対象となるハードウェア構成やバージョン、その他の環境に従って、生成するオブジェクトの種類または数を変更して、該当するオブジェクトだけを起動させることを検討する。こうすることで実際には使用しないオブジェクトを生成せず、メモリ領域を有効に活用することが出来る。



図B-13：クラス構成の例

図 B-13 において個々の機能クラス OperationA から OperationC は、生成用のインターフェースクラス CreateInterface を継承している。Creator はハードウェアの構成など、ソフトウェアが動作する環境を認識し、起動時に必要な機能クラスのオブジェクトを生成する。利用者クラス(User)は Creator を通して機能クラスのオブジェクトを入手し使用する。

こうすることで、すべてのオブジェクトを生成する必要がなく、また、機能拡張や変更があった場合、「新たなクラスの追加」または「生成オブジェクトの組み合わせ変更」をすることで柔軟に対応出来る。

別の方法として、CreateInterface と OpationA から OperationC をテンプレートメソッドパターンとして実装することが考えられる。この場合、利用者は機能クラスの違いを気にしなくて済むが、実際には、基底クラスとなる CreateInterface で定義されるメソッドだけでは不足するため、個々の機能クラス固有のメソッド(functionAA、functionZZ)をそれぞれに定義、実装することになる。

そうなると、利用者クラスは CreateInterface の関数を利用せず、機能クラスのレベルまでダウンキャストして利用しなければならない。(C++ 言語の dynamic_cast 演算子や Java 言語の instanceof 演算子など、言語仕様上可能であれば)ダウンキャスト可能か確認するなどして、注意して利用する必要がある。

ダウンキャストは、基底クラスから派生クラスへ型変換する操作である。基底クラス CreateInterface のインスタンスは、必ずしも派生クラスである各機能クラスのインスタンスとは限らないので、この変換は一般に安全ではなく、エラーが発生する可能性があるため、使用する場合は、特段の注意が必要である。

関連する品質特性

- ●保守性(変更性)
- ●効率性(資源効率性)
- ●機能性(合目的性)

関連する作法

A-16　ループ処理からの例外脱出の仕様を定義する(P57 参照)

B-17 必要以上の汎用性、拡張性を設計・実装しない

作法概要

ソフトウェア全体または一部分の構造に、必要以上の汎用性または拡張性を設計・実装しない。

メリット

●複雑さを減らし、見通しの良い構造とすることが出来る。
●動作速度の低下を抑え、効率性を向上することが出来る。

留意点

①製品の目的や種類、シリーズ展開などを検討し、汎用性または拡張性が必要な場合は、それらを外すことは出来ない。
②ある一部分の構造から汎用性または拡張性を取り払うと、そこだけ他の部分と異なる構造となる。そういう場合は、ドキュメントなどに構造が異なることを明記しておく。

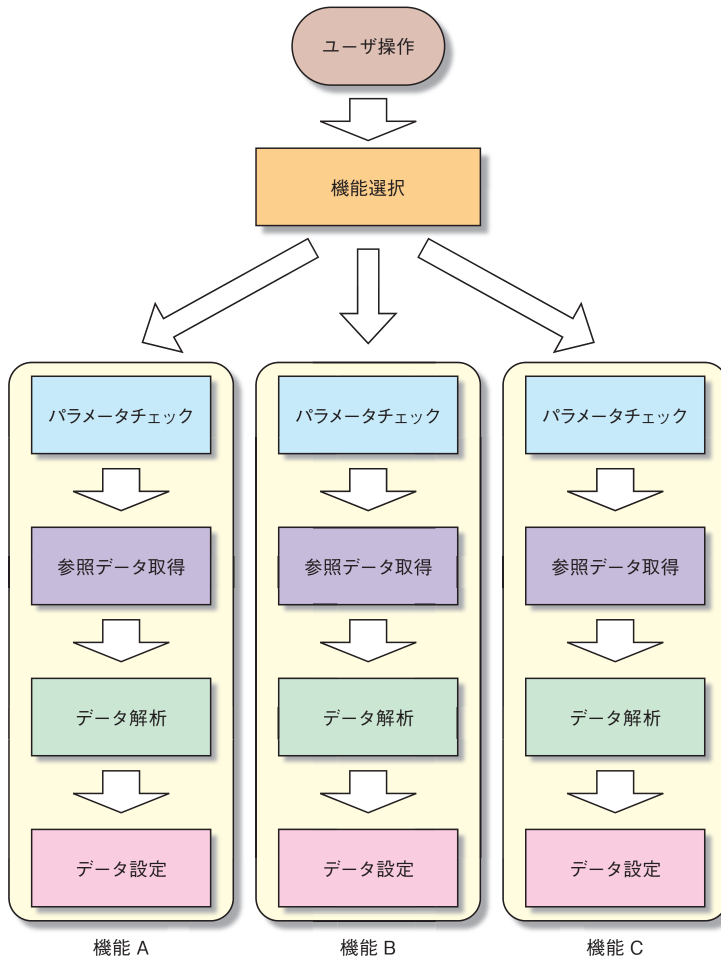
●解説

ソフトウェアの構造に汎用性または拡張性を持たせることは、将来の機能追加と変更に対して有効である。とくに、組込みシステムにおいては、ソフトウェアの再利用が多い、ソフトウェアが様々な製品で長期間利用され続けるなどの理由によって、汎用性または拡張性を持たせておきたい。しかし、本来、必要のない箇所（将来、機能追加や拡張などが発生しない箇所）にそのような機能を持たせると、逆に効率性を低下させる要因となってしまう。

従って、汎用性または拡張性を持たせることと、効率性のどちらが重要であるかを十分検討し、必要のない汎用性または拡張性は設計・実装せず、最もシンプルな構造にとどめるようにする。

例 拡張性のある構造

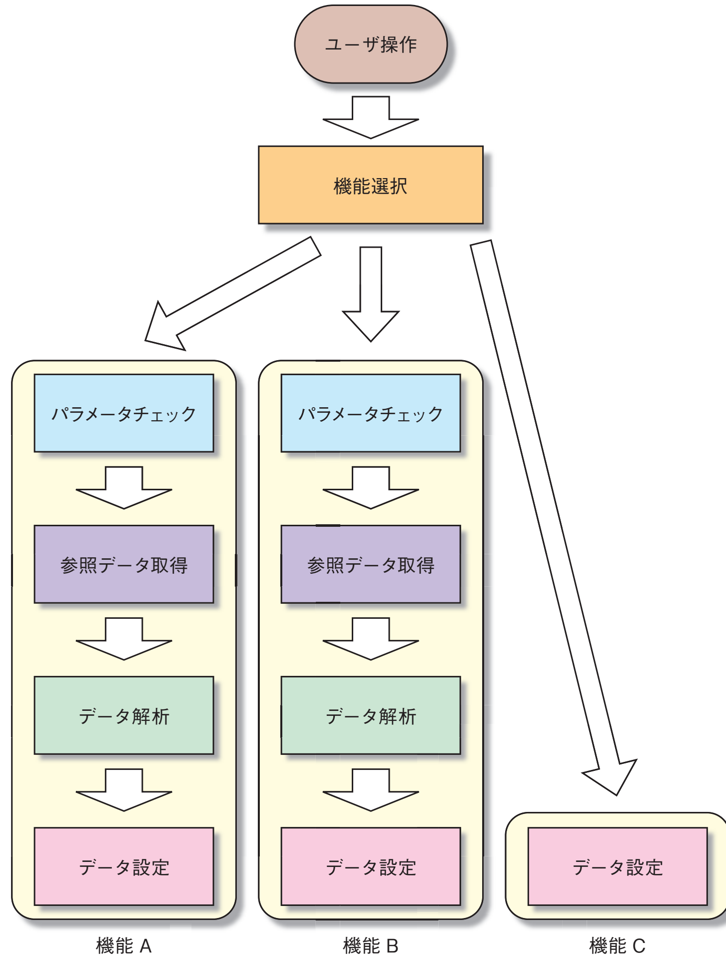
図 B-14 は拡張性のある構造の例を示している。



図B-14：拡張性のある構造

すべての機能が同じ構造となるため、同じ手順または 1 つのクラスを異なるクラスとして生成することで、容易に機能拡張を行うことが出来る。しかし、すべての機能がすべての要素（パラメータチェック、参照データ取得など）を必要とするとは限らない。例えば、機能 C はユーザ操作に対して固定値の設定を行うだけであれば、データ設定以外の要素は不要となる。

図 B-15 は機能 C のみを異なる構造としたときの図である。こうすることで、余計な処理を挟まず、機能 C のレスポンスは改善する。



図B-15：機能Cのみ拡張性を取り払った構造

関連する品質特性

●保守性（変更性、解析性）
●効率性（時間効率性）

B-18 複数の箇所に分散している処理はまとめる

作法概要

同じような処理が複数の箇所に分散し、処理結果を相互に参照する必要がある場合は、関連する項目を1つにまとめて処理をする。

メリット

●簡潔で処理しやすいソースコードとなり、不具合の混入を防止出来、保守性も向上する。
●データの相互参照がなくなり、処理速度が向上する。

留意点

①汎用性を考慮して処理を分散している場合には、処理をまとめることで汎用性を失うことがあるので、注意が必要である。
②クローンコードの検出には、CCFinder (http://www.ccfinder.net/)のようなツールの活用も有効である。

●解説

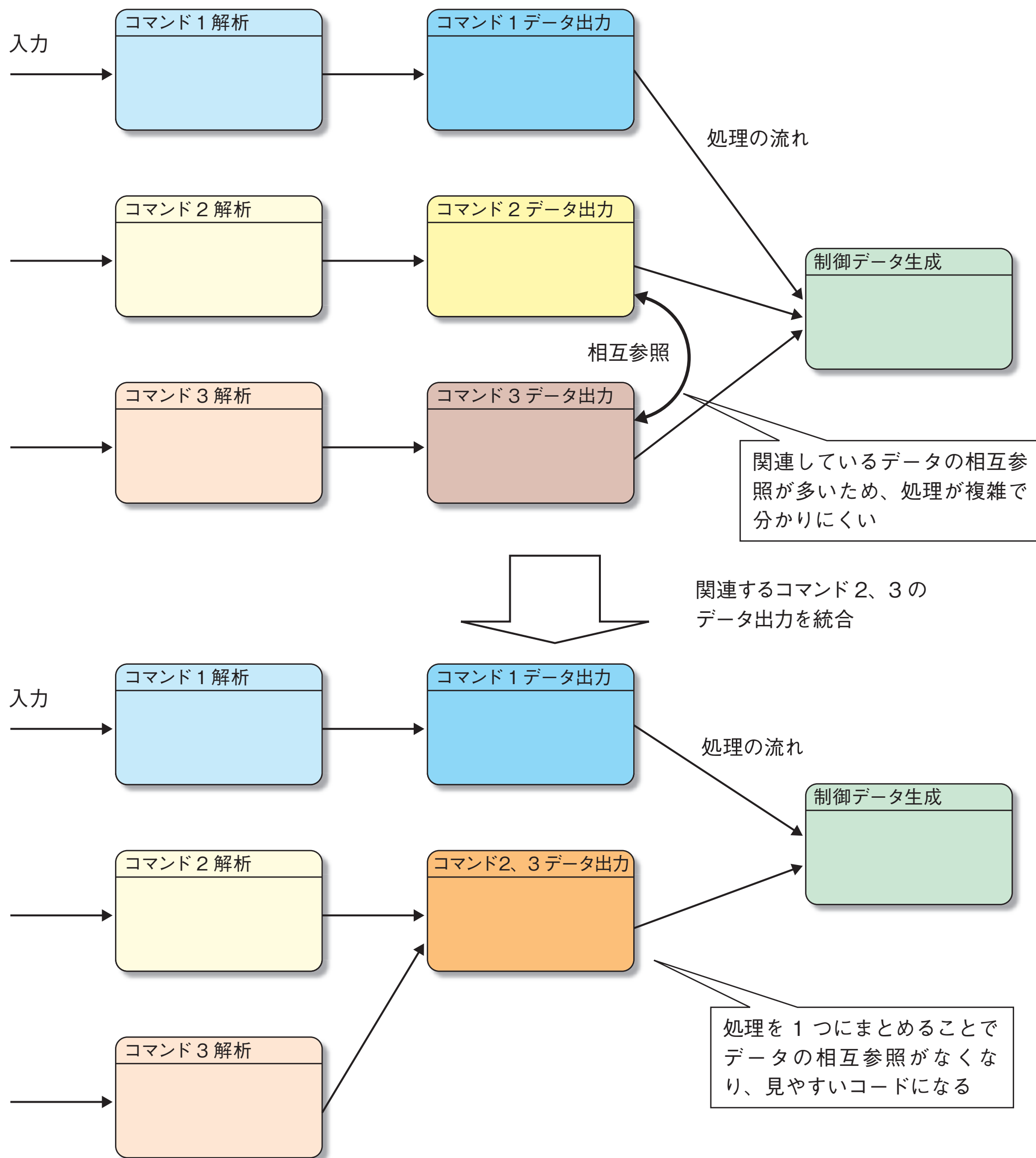
ハードウェアに対する種々のコマンドを解析して制御を行うシステムでは、コマンドの増減に対応しやすいように、それぞれのコマンドに対応した処理(クラス、オブジェクト)を別々に作成して、それぞれのコマンド解析結果から最終的に制御データを生成し出力するような設計にすることがある。

しかし、それぞれのコマンドに依存関係がある場合は、コマンドごとの処理の関連が見えにくかったり、処理速度が低下したりする問題がある。

このような場合には、関連する処理を１つにまとめる。これによって、簡潔で処理しやすいソースコードとなり、処理速度の向上も期待出来る。

例 コマンド 1〜3 の解析とデータ出力処理を、それぞれ実施する場合

関連しているデータの相互参照が多いため、処理が複雑で分かりにくいが、処理を１つにまとめることでデータの相互参照がなくなり、見やすいコードになる。



図B-16：分散している処理を統合

関連する品質特性

- ●信頼性（成熟性）
- ●保守性（解析性、変更性、試験性）
- ●移植性（環境適応性）

B-19 類似のデータ変換処理は統合する

作法概要

類似のデータ変換処理が多数発生する場合は、類似または同一コードを散在させず、基本処理フローに統合する。

メリット

ソフトウェアの保守性が向上する。

留意点

①アーキテクチャ設計時にデータ変換処理は拡張性を考慮しておく。
②責務が異なる類似コードは統合せず、共通部分を関数などに括り出すことを検討する。

●解説

データ変換処理は、基本的な処理フローは同一であるが、パラメータに依存して部分的に詳細処理が異なることがある。そのために、データの圧縮・伸張処理、画像データの変換などでは、類似コードが散在することがよく見受けられる。

類似コードが散在すると、基本的な処理フローに不具合が発生した場合に、修正個所が多岐にわたってしまう。このため、修正漏れなどが発生する可能性が高くなり、ソフトウェアの保守性が悪くなる。従って、保守性を高めるためには、散在する類似コードを統合する設計を行うことが望ましい。

例 ビットマップ画像を加工し、各画像フォーマットに変換する場合

画像フォーマットの変換方式(jpeg、png など)ごとに基本処理フロー（フルカラー、グレースケールなど)がある場合、データ変換モジュールとしては、変換方式と基本処理フローの組み合わせの数だけ必要なことになる。このような場合に変換方式をパラメータとした分岐制御を行うと、図 B-17 に示すように基本処理フローが何回も現われ、類似コードが散在することになる。

```
if( 変換方式 == “jpeg”) {
    if( 処理フロー == ‶ フルカラー ″) {
        基本処理フロー 1
    } else if( 処理フロー == ‶ グレースケール ″) {
        基本処理フロー 2
    } else if(・・・・){
        ・・・
    } else {
        ・・・
    }
} else if( 変換方式 == “png”) {
    if( 処理フロー == ‶ フルカラー ″) {
        基本処理フロー 1
    } else if( 処理フロー == ‶ グレースケール ″) {
        基本処理フロー 2
    } else if(・・・・){
        ・・・
    } else {
        ・・・
    }
} else if(・・・) {
    ・・・・
}
```

図B-17：問題がある実施例

この対策として、次の改善を図る。

①基本処理フロー 1、2 の各内容を統合する（処理の統合）

②基本処理フロー 1、2 に関する処理を同一の引数リストで関数化し、関数ポインタが利用出来るようにする（関数インターフェースの統合）

この結果、類似または同一コードをなくし、ソフトウェアの保守性を高めることが出来る。

ここで、関数ポインタの代わりに、特定のパターンの関数名を生成するマクロを使うことも選択肢の１つとして考えられるが、データ変換などでは、規模的にマクロでは処理仕切れないため、関数ポインタを用いることが望ましい。

```
int DataConverter1((int param1, int param2) {
    /* フルカラー処理フローの統合 */
    基本処理フロー 1
    return nRet;
}

int DataConverter2((int param1, int param2) {
    /* グレースケール処理フローの統合 */
    基本処理フロー 2
    return nRet;
}
 ・・・・

main() {
    int (*pFunc)(int param1, int param2);

    if(変換方式 == "jpeg") {
        if(処理フロー == "フルカラー") {
            pFunc = DataConverter1;
        } else if(処理フロー == "グレースケール") {
            pFunc = DataConverter2;
        } else if(・・・・){
         ・・・
        } else {
         ・・・
        }
    } else if(変換方式 == "png") {
     ・・・
    } else if(・・・) {
     ・・・・
    }
    /* 変換処理 */
    Int nRet = (*pFunc)(param1, param2); /* 関数インターフェースの統合 */
    if (nRet != OK) {
    ・・・
```

図B-18：良い実施例

関連する品質特性

保守性(解析性、変更性、安定性、試験性)

B-20 性能改善のための変更は性能ボトルネック解析に基づいて行う

作法概要

- 性能改善の前にシステムの性能ボトルネックを解析する。
- 性能ボトルネックとなっている処理または関数を対象として高速化の改修を行う。
- コンパイラの最適化オプションを適切に指定する。

メリット

性能ボトルネックを実際に解析するため、性能改善実施後に性能要件の未充足が判明して手戻りすることによる開発遅延を防止することが出来る。

留意点

①呼出頻度の高い処理または関数は性能改善対象の候補ではあるが、必ずしもそれが性能ボトルネックになっているとは限らない。頻度が高くても 1 回あたりの処理時間が短ければ、全体に与える影響は相対的に小さいからである。従って、性能改善対象を特定するためには、処理時間を計測した上で性能ボトルネック解析をする必要がある。
②性能改善への貢献が低いと見込まれる処理または関数を改修して不要な変更を加えてしまうことで、ソースコードの理解容易性を損なうなどの品質劣化を及ぼさないように注意する。

●解説

性能改善を行うときに、上流工程まで戻ってアーキテクチャを再設計しなくても、特定の処理の高速化で対応可能な場合もある。そのような場合は、改善すべき処理または関数の特定と対応策を正しく決定することが重要である。

性能改善対象の処理または関数の特定のためには、システムの各要素の動作時間に対して、モニター（計測）しプロファイリング（分析）を行う性能ボトルネック解析が不可欠である。

特定した処理または関数の高速化のための対応策としては、アセンブラなどの低水準言語による書き換えや、データ構造またはアルゴリズムの変更などがある。

また、コンパイラの最適化オプションの利用も検討する価値がある。なぜなら、人手によるプログラムの書き換えの際に等価性を維持するのは容易ではないからである。また、高速化のために書き換えたソースコードは、理解容易性が低下しやすい。

例 製品の計算処理の速度

開発中の製品が、以前の製品よりも速度が遅いと評価チームから指摘を受け、性能改善に取りかかった。

以前の製品では、ある計算処理が固定小数点で実装されており、開発中の製品では、その処理が浮

動小数点で実装されていた。この処理は高い頻度で呼び出されているものであったため、この処理の高速化によって性能改善が出来ると判断し、浮動小数点から固定小数点に戻した上で、アセンブラ記述による実装に変更した。

しかし、変更後のプログラムの実行時間を計測したところ、性能改善はほとんど見られなかった。改めてプロファイリングを行って性能ボトルネック解析をしたところ、ボトルネック部分は、その計算処理の前後の部分であることが判明した。そこで、その前後の部分の実装を変更して高速化し、更に、その部分の実装に適した最適化オプションをコンパイラに指定したところ、目標の性能改善が達成出来た。

関連する品質特性

- 効率性（時間効率性、資源効率性）
- 保守性（解析性、変更性、安定性、試験性）

関連する作法

C-9　開発中に動作速度をモニターしプロファイリングを行う（P150 参照）

B-21 サードパーティ製品やオープンソースソフトウェアなどと組み合わせて開発するときは、事前に十分調査・解析・評価する

作法概要

- サードパーティ製品やオープンソースソフトウェアなどの機能仕様と非機能仕様を詳細に把握する。
- ソースコードや開発関連文書などの情報取得を十分に行う。
- 特定の条件またはシナリオでテストしてタイミング解析と性能評価を行い、自分の作成する部分の設計の判断材料とする。

メリット

全体の設計の見直しなどの大きな手戻りを未然に防ぐことが出来る。

留意点

①サードパーティ製品の場合は、ライセンス上、ソースコードと開発関連文書の十分な情報開示を受けることが困難な場合があるため、一部開示など、妥協点を交渉する。
②サードパーティ製品またはオープンソースソフトウェアには、そのバージョン改訂によって互換性が損なわれるリスクがあることを留意した上で、それらソフトウェア部品の再利用をすべきかどうかを検討する。

●解説

サードパーティ製品やオープンソースソフトウェアなど、他者が作成したソフトウェアを部品として組み込み、自分たちが作成する部分と連携して、1つのソフトウェア製品として開発するケースが往々にしてある。このようなCOTS（Commercial Off-The-Shelf）と呼ばれる製品に基づく開発においては、オープンソースソフトウェアまたはサードパーティ製品を十分に把握する作業が、自分たちが作成する部分の設計よりも前に必要となる。

しかし、他者が開発するソフトウェアを部品として再利用する場合、その内部設計や想定シナリオ、制約条件などを十分に把握するのは難しく、予期しない問題が発生し得る。

例えば、共有資源排他制御の詳細なタイミングや順序、特定シナリオにおけるスループットなどが把握できていないと、それらについて一方的な仮定を置いたまま、自分たちが作成する部分の開発を進めることになり、統合時点になってから問題が発覚する事態に陥りやすい。その場合の問題は非機能要件の充足不能として出現するため、その問題解決には細かな箇所の修正では不可能で、全体の設計の見直しなどが必要となり、大幅な手戻りによる開発遅延を引き起こす。

そのような事態を未然に防ぐためには、自分たちが作成する部分の設計を確定させる前に、他者が開発するソフトウェア部品の解析と評価を十分かつ詳細に行う必要がある。そのために、可能な限り、当該ソフトウェア部品のソースコードと開発関連文書を入手しておく。

オープンソースソフトウェアであればソースコードはすべて入手可能だが、一方で、十分な質と量の設計関連文書が存在しているかどうかはまちまちである。

サードパーティ製品であれば、ソースコードと設計関連文書は十分に揃っていることが期待出来る

が、ライセンスの問題で完全な開示は望めない可能性がある。

いずれにせよ、仮に完全なソースコードの開示と入手が可能だったとしても、それらのソフトウェア部品の規模が大きく複雑なものであった場合、ソースコードに基づく解析は多大なコストを必要とすることになる。そもそも、COTS に基づく開発の方針を選択した動機は、すべてを内製するよりも開発コストを引き下げることにある場合が多いため、ソフトウェア部品の理解・解析コストが多大なものとなるのであれば、COTS に基づく開発自体が選択されないことになる。

設計関連文書が参照可能ならば、ソースコードのみの場合よりも解析コストは低減出来ると期待出来る。また、ソフトウェア部品のすべてについて理解・解析を必要とせず、特定の条件またはシナリオで解析出来れば十分である場合はままある。そのような場合には、条件またはシナリオに対応するテストケースとテストドライバーを用意し、それを用いて実行した結果をもとに、タイミングと処理順序の解析や性能評価などを行って、自分たちが作成する部分の設計の判断材料とする。

関連する品質特性

- **機能性(合目的性、正確性、相互作用性)**
- **信頼性(成熟性)**
- **効率性(時間効率性、資源効率性)**

関連する作法

A-8　COTS(Commercial Off-The-Shelf)製品を安易に利用しない(P30 参照)

Part C

ミドルウェア・ネットワークスタック・ライブラリレベルでの工夫

C-1 ポートスキャンによる外部からの攻撃に備える

作法概要

●接続相手以外に対してはネットワークのポートを隠蔽する。
●想定外のデータを受信した場合の対策を講じる。

メリット

ポートスキャンによる攻撃によって、予想外の動作を防止することが出来る。

留意点

Web アプリケーションの分野では、様々な団体がネットワーク攻撃に対する脆弱性を回避するノウハウを「セキュアプログラミング」や「セキュアコーディング」などの名称で発行しているため、これらを参考にする。

●解説

ネットワーク経由で制御を乗っ取る、あるいは様々な障害を引き起こす可能性のあるデータを送り付ける攻撃が、インターネットでは日常的に行われている。こういう攻撃では、ネットワークポートに対し無差別にアクセスする、ポートスキャンと呼ばれる手法がよく使われる。

ネットワークインターフェースを備える機器では、このようなポートスキャンによる攻撃を防ぐために、ネットワーク上のデータを不用意に受信しないよう、また受信した場合でも処理に支障が生じないよう、次の対策を行う必要がある。

①接続相手以外に対しては、フィルタリングなどによりネットワークのポートを隠蔽する。
②想定外のデータを受信した場合には、無視するなどの対策を講じる。
③入力データに関して、その異常値検出を行う。

例 自宅の監視カメラ映像を、インターネット経由で外部から確認出来るシステム

ポートスキャンの不正なアクセスにより送られたデータを、正常なデータと同様にスタックに取り込み、解釈しようとしたが、その結果、スタックを破壊し、データ領域に不整合を生じさせ、不正な映像が表示されるようになった。

そこで、次のような対策を行った。

①自宅システムのゲートウェイのパケットフィルタの設定により、あらかじめ登録したアドレス以外からのアクセスを拒否する。
②ネットワークから送られたデータをチェックし、不整合がある場合はログに記録し処理を中止する。

その結果、以前のようなトラブルが起きなくなった。

関連する品質特性

機能性（セキュリティ）

C-2 入力データに対し処理を継続的に行う場合に優先度逆転による異常な状態を回避する

作法概要

- 外部からシステムに対しデータを入力し、このデータに対する処理を継続的に行う場合はデータ入力と処理を分離する。
- データ入力とデータ処理を別タスクに分離した場合の優先度を考慮する。
- データ入力と処理を分離する場合は、セマフォなどによる排他制御を行う。
- 入力と処理のタスクの優先度を変える場合、割込みなどによる優先度逆転に注意する。

メリット

システムが異常な状態に陥ることを回避することが出来る。

留意点

①データ処理の遅延が起きても、時間制約が満足されることを確認する。
②入力処理と資源(メモリなど)を共有する他処理の時間制約をチェックする。

●解説

入力データに対し負荷の高い処理を継続的に行う場合、マルチタスク環境を利用して、データの入力とその処理を別タスクに分離し、データ受渡し用の共有メモリをセマフォなどで排他制御することも多い。このような場合には、両タスクの優先度の違いによって優先度逆転という現象が生じることに留意する。この現象による処理遅延などに対する考慮が、時間制約を満足させるために必要となる。優先度逆転を避けるためには、例えば、次の方法などを検討する。

①クリチカルセクションの間を割込み禁止にする。
②優先度継承機能のあるリアルタイムOS(RTOS)を使用する。

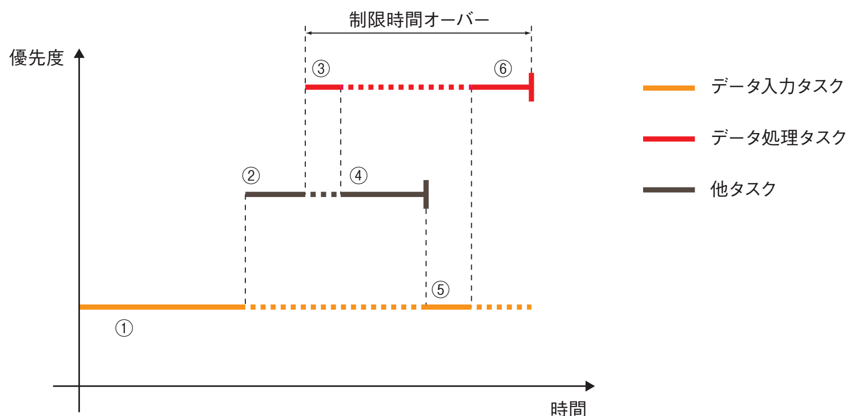
例 計測機器での優先度逆転

設計にあたっては、次のように行った。低優先度のデータ入力タスクは、データチェック後にそのデータ群を共有メモリへ書き込む。高優先度のデータ処理タスクは周期的に起動され、共有メモリにあるデータ群を読み出して、そのデータ処理を行う。データ入力タスクが共有メモリへデータを書き込む区間と、データ処理タスクがデータを読み出す区間がクリチカルセクションとなり、セマフォを使用して排他制御する。

しかし、実行すると、予期しない状態が発生した。例えば、次のような経緯で優先度の逆転が起き、データ処理タスクが遅延させられ(④から⑤まで)、制限時間内に処理ができなかった(図C-1)。

①データ入力タスクがセマフォを獲得し、共有メモリへのデータ書込みを始めた。
②この時点で割込みが発生し、中優先度の他タスクへタスクスイッチし、データ入力タスクはクリチカルセクションに入ったまま待機状態となった。
③その後、より高い優先度のデータ処理タスクが周期起動されたが、データ入力タスクのセマフォ

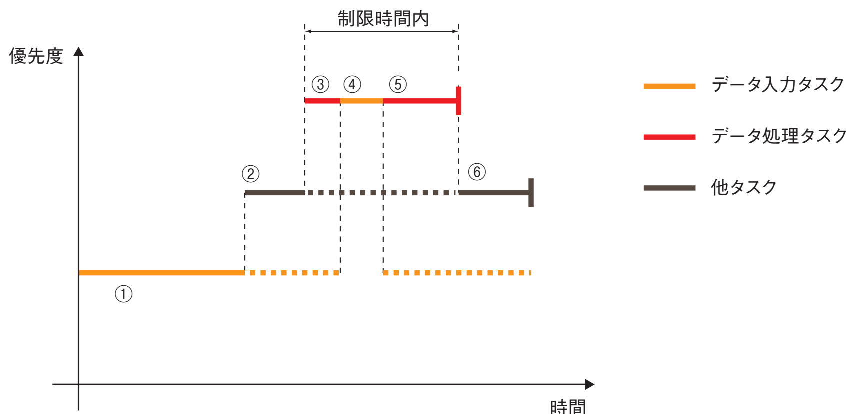
解放待ちとなった。
④中優先度の他タスクが処理を再開した(優先度逆転)。
⑤中優先度の他タスクが終了後、データ入力タスクが実行を再開した。
⑥データ入力タスクがセマフォを開放し、データ処理タスクが実行を再開した。



図C-1：優先度逆転による制限時間オーバー

この優先度逆転を回避するため、優先度継承機能のあるRTOSを使用し、信頼性を向上させた(図C-2)。

①データ入力タスクがセマフォを獲得し、共有メモリへのデータ書込みを始めた。
②この時点で割込みが発生し、中優先度の他タスクへタスクスイッチし、データ入力タスクはクリチカルセクションに入ったまま待機状態となった。
③その後、より高い優先度のデータ処理タスクが周期起動されたが、データ入力タスクのセマフォ解放待ちとなった。
④データ入力タスクがデータ処理タスクの優先度で処理を再開した(優先度継承)。
⑤データ入力タスクがセマフォを開放し、データ処理タスクが実行を再開した。
⑥データ処理タスクが終了後、中優先度の他タスクが実行を再開した。



図C-2：優先度継承による信頼性の向上

この機能を使用することによって、クリチカルセクションに入ったデータ入力タスクの優先度を、クリチカルセクションに入ろうとしているデータ処理タスクと同一の優先度に一時的に引き上げることが出来、優先度逆転を回避出来る。

関連する品質特性

- ●機能性（正確性、セキュリティ）
- ●信頼性（成熟性）
- ●効率性
- ●保守性

C-3 状態遷移を行うモジュール間で状態の一貫性が保たれるように設計する

作法概要

- システム全体で共通の状態を、複数の状態遷移を行うモジュールでそれぞれ定義する場合には、状態の一貫性を保証する。
- システム全体で共通の状態を一元管理すれば、その解決策となる。

メリット

システムの正確性を維持し（矛盾の発生を防ぎ）、システムの破綻を防ぐことが出来る。

留意点

一元管理によって状態数が爆発的に増加する危険性がある。この場合には、別の状態管理の方法を検討する必要がある。

●解説

複数のモジュールで状態遷移設計を行う際、システム全体で共通の状態が、場合によっては各モジュールで定義されることがある。この場合には、各モジュールで定義した状態がシステム全体で「同じ状態」になることを保証する必要がある。しかし、この保証を完全に行うことは事実上困難であり、何らかの原因でこれらの状態の同期が取れなくなるような不具合が発生する場合、テストによる検出や再現は難しい。

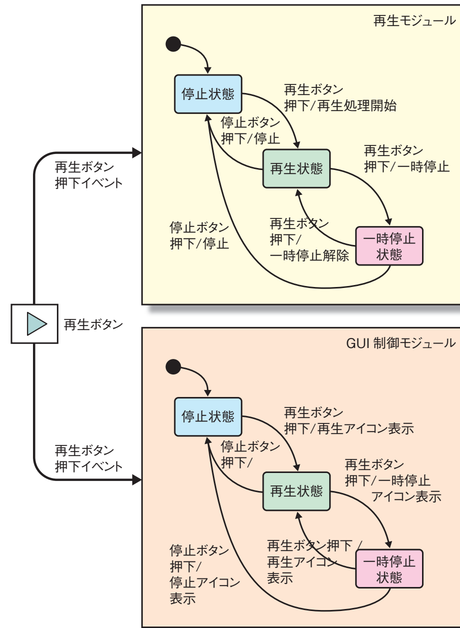
これを回避する1つの方策として、システム全体で共通の状態を管理するモジュールを決定し、状態管理をそのモジュールに一元化することが挙げられる。状態の一元管理を行うことにより、状態遷移の保守性向上を図ることが出来る。また、設計者にとっては複数の状態遷移の同期を意識する必要がなくなる。

ただし、状態の一元管理によって、管理すべき状態数が爆発的に増加する危険性も含んでいるので、そのような場合は状態管理の方法について別の工夫が必要になる。

例 BDプレーヤのようなAV機器

このシステムには、BDコンテンツの再生を担う「再生モジュール」と、GUI（Graphical User Interface）制御を担う「GUI制御モジュール」が存在する。ユーザが再生ボタンを押下すると、画面に再生開始アイコンを表示し、コンテンツの再生を開始する。このとき、もう一度ユーザが再生ボタンを押下すると、一時停止機能が働き、画面に一時停止アイコンを表示し、コンテンツの再生が一時停止する。同様にもう一度ユーザが再生ボタンを押下すると、一時停止機能が解除され、画面に再度、再生開始アイコンを表示し、コンテンツの再生を再開する。また、ユーザが停止ボタンを押下した際は、画面に停止アイコンを表示し、コンテンツの再生が停止する。

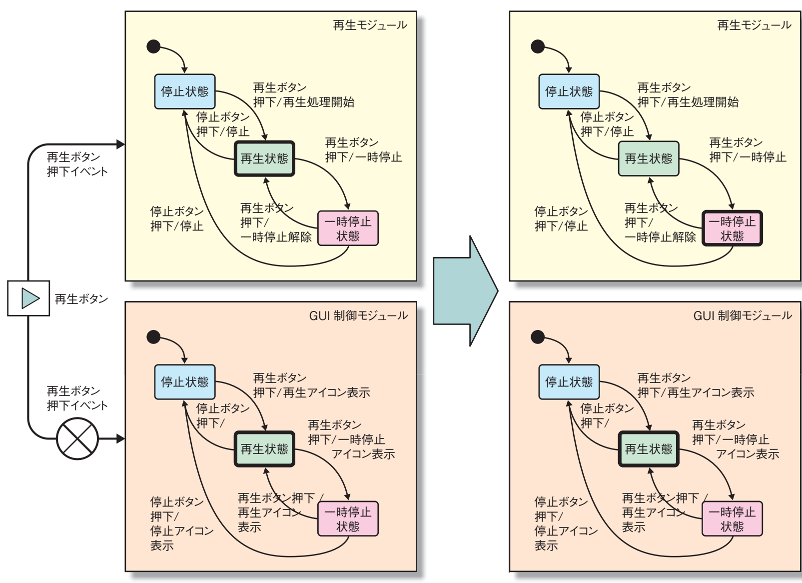
以上のような動作仕様に対して、図C-3のような「再生モジュール」と「GUI制御モジュール」の状態遷移設計が可能である。



図C-3：簡易BDプレーヤの状態遷移設計

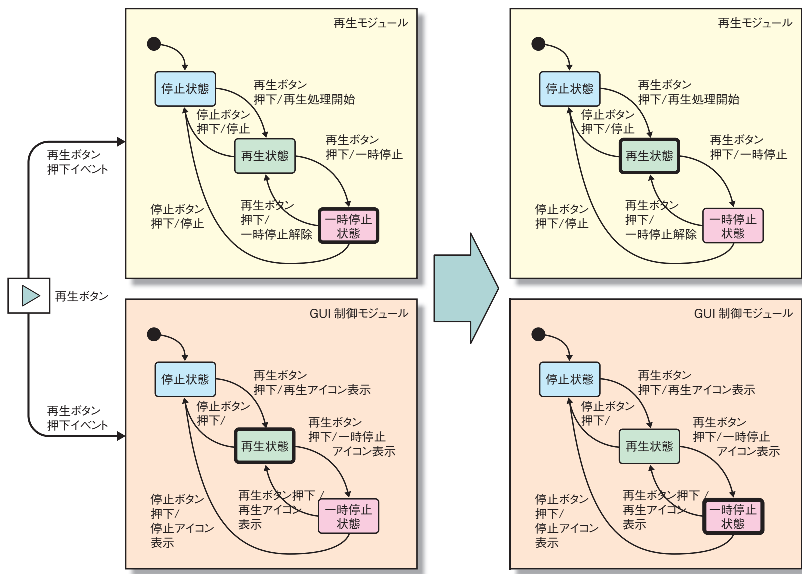
ここで、両モジュールで定義された「停止状態」「再生状態」「一時停止状態」はそれぞれ、「同じ状態」としてシステム全体での一貫性を保証しなければならない状態である。それを実現するために、これらの状態遷移を発生させる「ユーザの再生ボタンまたは停止ボタン押下」というキーイベントは、両モジュールへ配送するように設計する。

このとき、もし何らかの原因によって、キーイベントが再生モジュールのみに配送され、GUI 制御モジュールに配送されなかった場合、図 C-4 のように、GUI 制御モジュールは「再生状態」のままと認識し、再生モジュールは「一時停止状態」へ遷移したと認識することとなり、コンテンツは一時停止されたにもかかわらず、GUI 画面は何も表示されないといった、正しくない動作をすることとなる。



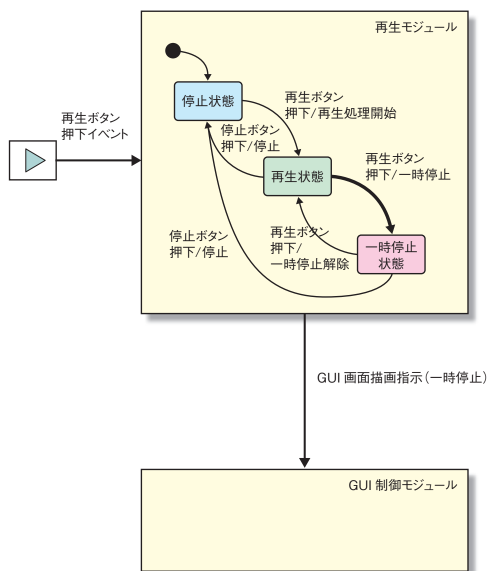
図C-4：キーイベントの配送が正しく行われなかった場合の状態遷移

更にここで、図 C-5 のように再度ユーザによって再生ボタンが押下され、今度は正しく再生モジュールと GUI 制御モジュールにキーイベントが配送されたとすると、両モジュールはそれぞれ「一時停止状態」、「再生状態」へと遷移されたと認識し、一時停止が解除されてコンテンツの再生が再開したにもかかわらず、画面に一時停止の GUI 画面が表示されるという正しくない動作をして、モジュール間の状態の一貫性が維持出来なくなる。



図C-5：更にイベント入力が発生した際の状態遷移

この例では、システム全体で共通の状態を再生モジュールが一元管理し、状態遷移が発生してGUI画面の更新が必要となるたびに再生モジュールがGUI制御モジュールへ再描画を指示するように変更することで、前述の事態を回避している。再描画指示が何らかの原因でGUI制御モジュールに届かなかった場合、一時的に表示に不具合が生じるが、次の指示が到達すれば不具合は解消する。



図C-6：状態管理を一元化した設計

ただし、GUI処理に特化した状態については、GUI制御モジュールが管理する方が望ましい。例えば、あるダイアログ上に表示されるGUIボタンが有効か無効か（ボタンを薄く表示するかしないか）を表す状態などは、この種の状態に相当する。

関連する品質特性

●保守性（解析性、変更性、安定性、試験性）
●機能性（正確性）

C-4 変数の初期化にあたっては システムの挙動を網羅的に考慮する

作法概要

変数の初期値はプログラム起動時に設定されるが、リセットや異常などシステムのとり得る挙動をもれなく考慮して、どのような値を取るべきかを決定する。

メリット

再現性が不確実な不具合を低減させることが出来る。

留意点

検出困難な不具合に繋がりやすいポイントは、以下のように変数の種類により異なる。
- ●静的変数はプログラム終了まで値が保持される。
- ●自動変数はスタック領域上に確保されるため、一般に初期値が不定である。

●解説

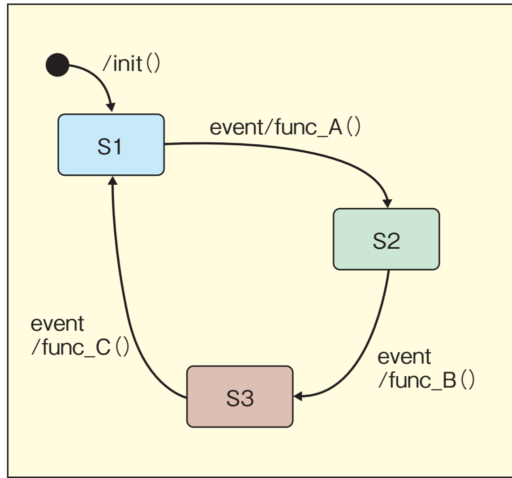
変数は、必ず明示的に初期化を行ってから利用する。変数の宣言時には、その初期値はプログラミング言語の処理系によって異なり、一般に不定である。この不定値をそのまま利用すると、あとの処理で設計者の意図しない振舞いを引き起こし、不具合へ繋がる可能性がある。

とくに、C 言語の static 宣言など、静的変数を利用するときには、注意が必要である。静的変数は、プログラム開始時に 1 度だけ初期化されるのみである。自動復旧などの初期化処理をプログラム実行中に行うシステムでは、静的変数に対して明示的に初期化を行わないと、以前の処理結果として残されていた変数値を利用して処理を継続することになり、本来期待していた挙動とは異なる動きをする不具合へと繋がる恐れがある。

また、この場合、偶然、その変数に適切な初期値が残されていれば、正常に動作する可能性が高いが、不具合の発見を難しくしてしまう。つまり、この種の不具合は、場合によって発生したりしなかったりすることとなり、原因の特定が困難な不具合となる点にも注意が必要である。

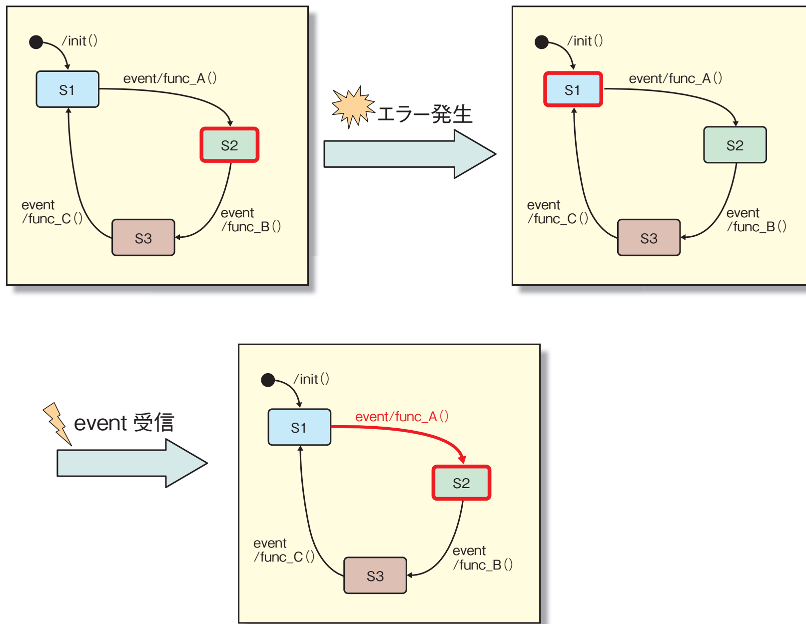
例 プログラム実行時のシステム

図 C-7 に示す状態遷移図に従って動作するシステムにおいて、その状態を *state* という静的変数に記録する。また、実行中に何らかのエラーが発生したときには、自動復旧が行われるものとする。



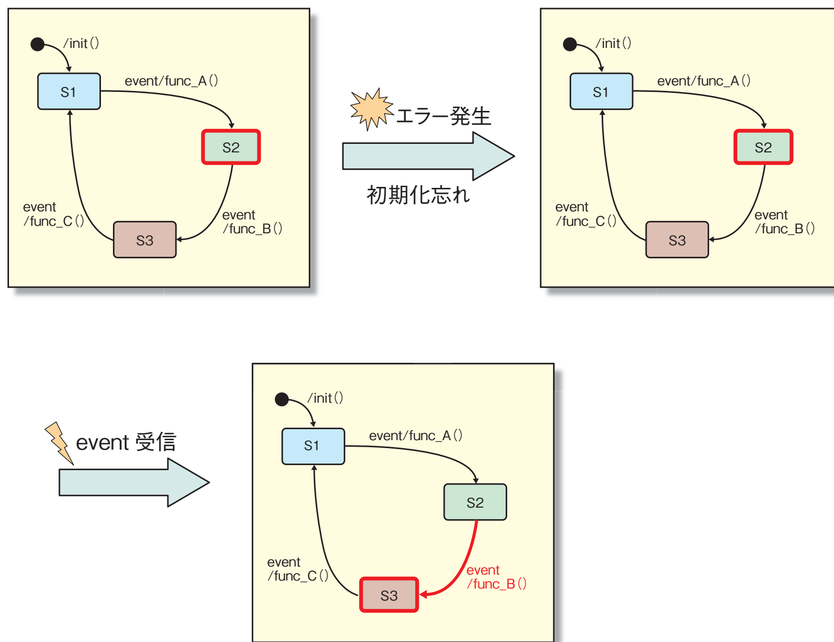
図C-7：想定する状態遷移図

今、仮に状態 S2 でエラーが発生したとする。このとき、図 C-8 のように自動復旧が行われ、状態は初期状態に戻り、初期化処理 init()が実行され、状態 S1 へ遷移する。その後に、イベント *event* を受信すれば、本来は、そのアクションとして func_A()が実行され、状態 S2 へ遷移しなければならない。



図C-8：自動復旧処理に対して本来求められる挙動

ところが、図C-9のように、自動復旧のときに変数*state*の初期化を行わないと、初期化処理が完了した後も、状態はS2にあるものと解釈される。この状態でイベント*event*を受信すると、そのアクションとしてfunc_B()が実行され、状態S3へ遷移することとなり、本来とは異なる動きをすることとなる。



図C-9：状態変数の初期化を行わなかった際の挙動

関連する品質特性

- **機能性（正確性）**
- **信頼性（回復性）**
- **保守性（解析性、変更性、試験性）**

C-5 グローバル変数は極力使用しない

作法概要

変数をグローバル変数とする前に、スコープ(参照可能な範囲)の限定を検討し、極力、グローバル変数とはしない。

メリット

保守が困難なプログラムとなることを回避出来る。

留意点

やむを得ずグローバル変数を使用する場合は、アクセス関数を用意したり、グローバル変数の命名ルールを設けたり、細心の注意を払って使用する。

●解説

グローバル変数は、プログラム上のすべての場所から読出しと書換えが可能で、実装に対する自由度が高まるため、開発者にとっては一見、大変便利であるように思われる。しかし、その反面、どのタイミングで読出しまたは書換えが発生するかといった、すべての挙動を正しく把握することは大変困難であり、このことは、プログラムが大規模化・複雑化するに伴い、一層、悪化する。プログラムを保守、機能拡張するフェーズでは、タイミングなどに依存する挙動を正しく把握出来ずに修正を加えると、開発者の意図しないようなバグを潜在させてしまう危険性がある。

従って、長期的な視点を踏まえ、保守と機能拡張までも考慮すると、グローバル変数は安易に使用しない方が望ましい。グローバル変数とする前に、まずは、該当の変数が必要となる範囲を正しく見極め、その範囲を制限出来ないかを検討する方が先決である。

その上で、どうしてもグローバル変数の使用が必要な場合、次のような考慮を払って使用する。

①アクセス関数

変数に対するアクセス関数(setter/getter 関数)を用意し、必ずこの関数を介して該当変数へアクセスするというルールを設ける。この運用によって、変数へのアクセスを局所化することができ、グローバル変数の挙動に対する見通しが良好になる可能性がある。また、一般には、グローバル変数を単純に読み書きするだけの処理は少なく、その前後に何らかの定形的な処理を行うことが多い(例えば、マルチスレッド環境で、共通領域に定義したグローバル変数へアクセスする場合は、その前後に排他制御を行うのが一般的である)。 これらの処理に仕様変更が発生したときには、その修正に伴う影響範囲をアクセス関数内に局所化出来る効果も生まれる。

②グローバル変数の命名ルール

ハードウェアリソースが限られているような小規模なシステムでは、アクセス関数を使用すると、オーバーヘッドの増大や実行速度の低下などを招くため、その使用が難しいこともあり得る。その場合は、例えばC 言語の場合、次のようなグローバル変数の命名ルールを設ける。

グローバル変数名＝クラス名(あるいはモジュール名)_変数名

命名ルールによって、グローバル変数を参照するときに参照元から参照先のクラス(あるいはモ

ミドルウェア・ネットワークスタック・ライブラリレベルでの工夫

Part C

ジュール）を判別出来るようになり、グローバル変数の挙動に対する見通しを良好に出来る。

関連する品質特性

- ●機能性（正確性）
- ●保守性（解析性、変更性、試験性）

C-6 不必要な処理はしない

作法概要

本来の目的と機能を実現するためには最低限何が必要かを常に考えて、過度のエラーチェックや、形式的で意味のない無駄なコードなどを入れ込まない。

メリット

- 本来の処理の流れが分かりやすくなり、ソフトウェアの見通しが良くなる。
- 流用または移植のときに不具合が入り込みにくくなる。

留意点

形式にとらわれ過ぎると、本来必要のないエラーのチェックなどをしてしまうことがあるので注意が必要である。

●解説

ソフトウェアを設計するときには、シンプルで見通しの良い設計をすることが非常に大切である。設計をしていくうちに、本来は必要ではないのに形式的な設計にとらわれ過ぎて、実際は起こり得ないエラーをチェックしたり、別の箇所でも同じようなチェックをするなど不要なソースコードを生成することがあり、これらは設計を複雑化し、見通しを悪くすることに繋がる。

設計が複雑化すると不具合を見落としやすくなり、流用時や他のシステムへの移植時などに、他の不具合を作り込んだりする危険性が高まる。

このようなことがないように、常にプログラムの目的は何か、最低限必要な機能は何かを考えて設計することが重要である。

例 画像データをメモリへ展開するプログラムについて新規に開発した場合

この開発の一環として、メモリ書込み関数を定義した。メモリ書込み関数ではアクセスのパラメータを事前にチェックし、メモリが実装されていない領域に書き込もうとした場合はエラーを返すように設計した。更に、メモリ書込み関数を呼び出すプログラムは、あらかじめ必要なメモリの確保を行い、また、メモリが実装されていない領域へのアクセスが行われた場合はメモリアクセス例外が発生するよう設計した。これにより、メモリアクセスに関して過度なチェック機構が実装された。

このプログラムを別の開発で再利用した際、画像データは確保しておいたメモリ領域に収まるにもかかわらずエラーが発生した。メモリ書込み関数にはメモリ領域の物理アドレスが埋め込まれていた。再利用する際に、これを変更する必要を見落としていたことが原因だった。その特定には多くの時間を費やした。

関連する品質特性

- 保守性（解析性、変更性、試験性）
- 移植性（環境適応性）

C-7 データ管理機構を使ってデータの一貫性を保証する

作法概要

データ更新が複数の部位で必要な場合には、データの更新を取りまとめて行うデータ管理機構を設ける。

メリット

- 競合回避（排他制御）や優先度判定などの調整を行うことが容易になる。
- データの更新箇所を特定することが容易になる。
- データ更新処理の詳細を、データの利用者から隠蔽することが出来る。

留意点

性能面でクリティカルな部位がデータ更新のために排他制御を行うと、必要な効率が得られない可能性がある。その場合は、排他制御の対象となるデータの単位（構造体などの大きさ）を見直すことによって、必要のない排他制御の回避やクリティカルセクションの縮小などによる効率の向上を検討する。

●解説

複数の部位（タスクなど）が同一のデータに対して更新処理を行う構造にすると、それぞれの部位で更新処理に伴う競合回避や優先度判定などの調整を行う必要が生じる。これに漏れまたは誤りがあった場合には、データの破壊または破損を引き起こすこととなり、不具合の要因となってしまう。この漏れと誤りは、データの更新処理を行う箇所が増えた場合、ソフトウェアの改変を行う場合などに起こりやすい。

共有するデータに対して複数の部位からデータ更新の必要が生じる場合には、データの更新を取りまとめて行うデータ管理機構を設けると良い。データ管理機構を用いると、データの更新処理は次のように行われる。

①データ更新の必要性が生じたときには、データ管理機構にリクエストを出す。

②データ管理機構は、リクエスト間の調整などを行った上で、更新処理を行う。

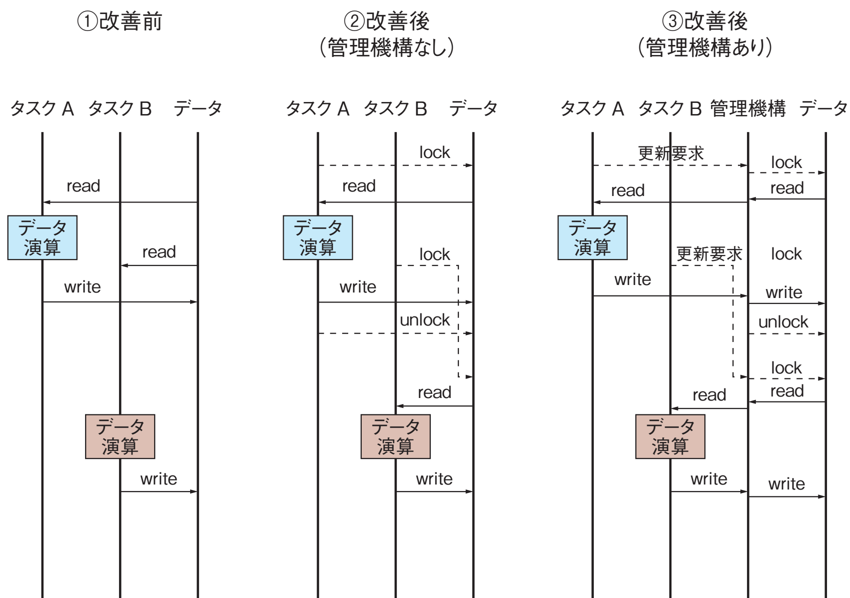
このように、データの更新処理を１つの部位で行う構造とすることによって、更新処理に伴う競合回避や優先度判定などの調整が容易になり、データ更新処理の詳細を隠蔽することが出来るようになり、保守性・再利用性を向上させることが出来る。

例 2つのタスク（タスクA、タスクB）が同一のデータを共有している場合

図C-10において、①ではタスクAの演算結果を更新する前にタスクBが前の値を参照している。意図する動作がタスクAでの演算結果に基づいてタスクBの演算を行うことであれば、これは不具合となる。

この解決策として、一般的には②のようにlock/unlock機能を用いて排他制御を行う。しかし、このようなタスクが多数あったり、各タスクでのデータ参照または更新箇所が散在していたりすると、

排他制御を漏らしてしまう危険性がある。
　そこで③のようにデータ管理機構を設け、ここで排他制御を集中的に行うようにする。



図C-10：2つのタスクが同一データを共有

関連する品質特性

●信頼性（成熟性）
●保守性（解析性、変更性、安定性）
●効率性（時間効率性）

C-8 パラメータチェックに関するルールをプロジェクトで統一する

作法概要

ソフトウェア部品のパラメータチェックの方法に関するルールを定め、徹底する。

メリット

- チェック漏れを防ぎつつ、実行性能とプログラムサイズへの影響を抑えることが出来る。
- チェック済みのパラメータ(引数など)であるのかそうでないのかを明らかにしておくことによって、ソフトウェア部品の再利用を安全に行うことが出来る。

留意点

「すべての箇所で入力パラメータをチェックする」以外のルールとした場合には、再利用されることを想定して、ヘッダ部のコメントとして「パラメータ値がチェック済みであることを前提とする」と明記するように心がける。

●解説

関数、メソッド、モジュール、コンポーネントなどのソフトウェア部品に与えるパラメータには、その部品が想定している値域がある。

一般的には、与えられたパラメータ値がその値域内にあるかどうかのチェックは、それぞれの部品が行うのが最も安全である。しかし、プログラムサイズや実行速度などに関する制約が厳しくなると、パラメータ値のチェックを重複して行うことは極力避けたいという要望が生じる場合がある。

このような場合には、パラメータ値のチェックに関するルールを策定して、ソフトウェアまたは開発組織全体でこれを遵守することが肝要である。

パラメータチェックに関する具体的なルールとしては、次のようなものがある。

①呼ばれた側で必ずチェックする。

②呼ぶ側で範囲内であることを保証した上で呼ぶ(範囲内であることが保証されている場合にはチェックしなくてもよいが、そうでなければチェックしてから呼び出す)。

③呼ぶ側と呼ばれる側の双方でチェックを行う。ただし、コンパイルスイッチなどを利用してどちらかのチェックを外せるようにすることで、実ハードウェア搭載時または製品出荷時に、プログラムサイズと実行速度への影響を軽減する。

関連する品質特性

- 信頼性(成熟性)
- 使用性(理解性、習得性)
- 効率性(時間効率性)
- 保守性(解析性、変更性、安定性)

関連する作法

● A-25　入力パラメータチェックはシステム全体で最適化する（P78 参照）
● B-3　システムの構成に合わせてデータをチェックするメカニズムを用意する（P91 参照）

C-9 開発中に動作速度をモニターしプロファイリングを行う

作法概要

開発中に動作速度をモニター（計測）し、速度の低下が見られた場合はプロファイリング(分析)によって原因を究明し改善する。

メリット

スループットを向上させることが出来る。

留意点

①ソフトウェアの改善を行うときは、事前に調査を行い、動作させて、実際に改善効果が現れる箇所を特定する必要がある。修正した部分だけを見ると改善されているかも知れないが、ソフトウェア全体から見ると、全く改善効果が見られない場合があるので注意する。
②動作速度の改善だけに注目してソースコードの修正を行うと、コーディングルールから逸脱した記述や、他の部分とは異なる例外的な構造などになってしまう。そういう場合、著しく可読性を悪化させ、保守性を低下させる。速度改善要求の強さの度合いを考慮し、改善を行うかどうかを決める必要がある。

●解説

開発の対象によっては、動作速度がより高速であることが、強い要求仕様となる場合がある。例えば、工業製品を生産する工場などで稼動する製造装置は長期間にわたって連続運転するため、その内部で動作するソフトウェアが数マイクロ秒の速度改善を実現すれば、その生産性が長期的に見れば大きな違いとなって表れる。

一般に、機能追加やバグ修正などを行うと、それにつれて処理速度が低下していく。そのため、指標となるソフトウェアの一連の動作と目標動作時間を定め、常に指標となる一連の動作の動作時間を測定して低下していないかを確認し、低下している場合は改善を行う必要がある。

改善を行うにあたり、まずは、ソフトウェア全体の設計の見直しを検討する。例えば、ハードウェアは動作しているがソフトウェアが動作していないタイミングがあれば、そこに処理を移動させるなどの工夫を行う。

次に、部分的な改善を実施する。部分的な改善を行うときは、速度的に問題のある箇所を特定する必要がある。このため、あらかじめソフトウェア内の処理区間の時間を測定(モニター)する仕組みを設けておく。その結果をプロファイリング(分析)することにより速度を低下させている部分を絞り込んでいく。

部分的な改善で気を付けなければならないことは、改善効果の高い部分をいかに特定するかである。部分的には効果の小さい改善であっても、一連の処理の中で複数回動作する箇所であれば全体として大きな改善となる。逆に、部分的には大きな改善であっても、一連の処理であまり動作しない箇所で

あれば効果は小さくなる。別の方法としては、ソースコードの静的解析や実行回数をカウントする仕組みなどを利用して、改善効果の高い部分を特定することも出来る。

更に、コンパイラのオプションによる高度な最適化を利用することも可能である。ただし、作成されたソフトウェアが予期せぬ動作をする場合もあるため、コンパイルのたびに網羅的にテストするなど、注意して利用する必要がある。

例 部分的な改善の例

①組み合わせて呼び出される関数を１つの関数にまとめ、関数呼出しの回数を減らす。
②何層にもネストしている関数呼出しの、各関数のロジックを１つの関数にまとめて、関数呼び出しのネストを少なくする。
③複数の機能を持ち、サイズの大きな関数を小さな関数に分割して、呼出し側が必要な関数のみを呼び出す。
④関数のパラメータチェックを呼出し側で実施し、パラメータがエラーの場合には、関数呼出しを行わない。
⑤実際にはエラーになる可能性がない箇所のエラーチェックを削除する。
⑥呼び出される回数の多い関数を特定し、ロジックの改善を行う。
⑦実装している機能に対して、想定される処理時間に比べて実際の処理時間が長くなる場合がある。ソースコード上ではステップ数が少なくても、コンパイルされたアセンブラ記述で見るとステップ数が多い場合があるため、アセンブラ記述を参考にロジックの改善を行う。

関連する品質特性

●効率性（時間効率性）
●機能性（正確性）

関連する作法

B-20 性能改善のための変更は性能ボトルネック解析に基づいて行う（P127 参照）

C-10 処理速度とソースコードの可読性を両立させる

作法概要

- 処理速度向上を意識し過ぎて分かりにくいソースコードにならないよう注意する。
- 効率性と保守性は相反することを理解しておく。

メリット

ソースコードの可読性が向上する。

留意点

機能実装後に処理速度の改善を行う場合は、実装済みの機能が正しく動作していることを確認出来るテストを、修正を行う前に準備しておく。このテストを速度改善後に行うことによって、機能が損なわれていないことを確認する。

●解説

処理速度を意識し過ぎたソースコードは、理解が困難になる可能性がある。また、必要以上に工夫をこらしたソースコードは、他者にとっては設計意図がつかめないものになってしまう。このようなソースコードは、将来のメンテナンスや機能拡張などのための解析性と変更性を著しく損なってしまう。

効率性を高めるために、処理速度を向上させる工夫を設計またはロジックに組み入れる場合は、保守性とのトレードオフを認識し、どちらを重視するべきか、またはどうバランスをとるべきか、要求仕様を十分に考慮する必要がある。

実際に、処理速度向上のための工夫を行うときには、ソースコードにその意図を十分なコメントとして書いておく、修正であれば修正前のソースコードをコメントとして残すなどの工夫をして、あとでソースコードの解析をするときの助けとなるようにする。

もし、コンパイラのオプションによる最適化を利用することが出来れば、ソースコードに修正を行うことなく、処理速度向上が可能である。ただし、作成されたソフトウェアが予期せぬ動作をする場合もあるため、注意して利用する必要がある。

例 理解を難しくするソースコード

①変数の各ビットをフラグとして利用するロジックは、各ビットが何を示しているか、直感的に伝わらない。

②ビットシフトを多用した演算は、元の演算の意味がくみ取れない。

③変数を局所化してセッター（設定用関数）やゲッター（取得用関数）を使うなどの工夫をせず、グローバル変数でデータのやり取りをしていると、変数の管理が行うことが出来ず、どこで誰が操作、参照しているのか分からない。

④幾つかに分かれていた演算を１つにまとめてしまうと、演算の意図がつかみにくくなる。

関連する品質特性

●保守性（解析性）
●効率性（時間効率性）

C-11 複数のデータフォーマットの相互変換は中間データフォーマット経由で実現する

作法概要

- 個別のデータフォーマットから、独立した中間データフォーマットを定義する。
- 入力データフォーマットから中間データフォーマットへの変換モジュールと、中間データフォーマットから出力データフォーマットへの変換モジュールを作成する。

メリット

必要な変換モジュール数が入力及び出力データフォーマット数の積にならないため、開発量を最小に抑えることが出来る。また、使用するメモリも少なく出来る。

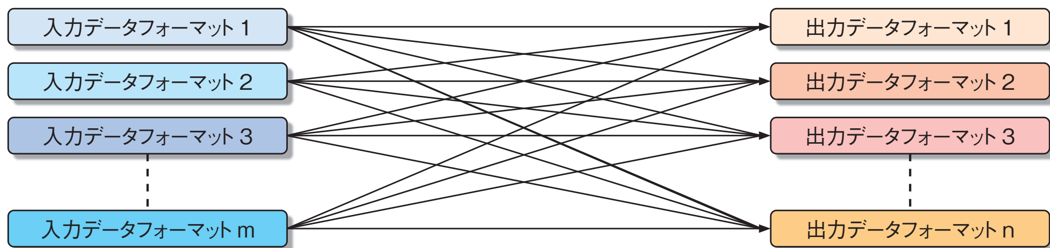
留意点

①中間データフォーマットは、対象とするすべてのデータフォーマットを包含出来るように定義する。将来において対象データフォーマットが増えることもあり得るので、それらも包含出来るように定義する。

②中間データフォーマットを経由する変換方式は、入力データフォーマットを直接に出力データフォーマットに変換する方式に比べて、実行速度が遅くなりがちである。従って、変換に要する時間に制約がある場合は、直接変換方式を選択せざるを得ないことがある。

●解説

入力データフォーマットを直接に出力データフォーマットに変換する方式だと、変換モジュールの数は入力データフォーマット数mと出力データフォーマット数nの積(m＊n)になる。



図C-11：直接変換方式

中間データフォーマットを定義し、すべての変換が中間データフォーマット経由になる方式だと、変換モジュールの数は入力データフォーマット数mと出力データフォーマット数nの和(m+n)になる。このため、入力または出力データフォーマット数が多い場合、直接変換方式に比べて作成する変換モジュールの数が少なくて済む。また、入力または出力データフォーマットが1つ増えた場合でも、追加作成する変換モジュールの数は、直接変換方式では出力データフォーマット数nまたは入力データフォーマット数mだが、中間変換方式では1になる。

図C-12：中間データフォーマットを経由する変換方式

関連する品質特性

●保守性（解析性、変更性、試験性）
●効率性（時間効率性、資源効率性）

C-12 既存のソフトウェアを改造する前に仕様をよく理解する

作法概要

- 既存のソフトウェアを改造して新しいソフトウェアを作成する場合は、改造元の設計仕様をよく確認し､改造による影響範囲を見極めた上で設計を行う。
- 改造元ソフトウェアの設計情報が乏しい場合は、ツールによる構造解析の結果も参考にする。

メリット

- 改造開発時の手戻り防止、開発効率化に繋がる。
- 保守性が向上する。

留意点

①既存ソフトウェアの再利用では、ソースコードが無変更で再利用出来る場合を除き、ソースコードではなく、設計を再利用する。

②改造元ソフトウェアの設計が悪いと、改造が非常に困難な場合がある。新たに作り直すのか、手を入れずにブラックボックスとして流用するのか、改造するのかなど、どの方法が効率的なのかを判断することも必要である。

③改造をする場合は、制御などのタイミングの変化や、応答性能低下などに注意する必要がある。無理な流用または複雑な改造は不具合の原因となり、構造も複雑化して保守性が悪くなる。

④改造時に新たに明確にした仕様や改造の方針、意図、根拠、留意点などは、文書化しておけばテストや今後の保守、次回の改造に有用である。

●解説

以前に作成したソフトウェアを再利用(注)して、変更または機能追加を行って新しいソフトウェアを作る場合、改造元ソフトウェアの仕様(機能面だけでなく、非機能面、とくに性能を含む)をよく理解していないと、しばしば改造ミスによる不具合や予期せぬ動作不良などが発生する。

これを避けるためには、改造元ソフトウェアの構造と動作タイミングを明確にし、改造による影響を見極めることが大切である。

とくに、古いソフトウェアを改造するときにしばしば起こることとして、当時の設計仕様書がない、誰が設計したかも分からず設計情報が十分ではないなどの場合がある。このような場合でも、やみくもに再利用するのではなく、ソフトウェアの静的解析や構造分析、コードクローンの検出などを行い、改造による影響範囲を見極めることが必要である。

改造を実施したあとには、新しいソフトウェアとして仕様を満足しているかをテストするが、併せて、改造による影響が本来出てはいけない部分の動作に影響がないかどうかを確認するため、回帰テ

注：ここでは、再利用の形態を流用と改造に分け、元のソフトウェアをそのまま使うときを「流用」、変更を加えるときを「改造」と呼んでいる。

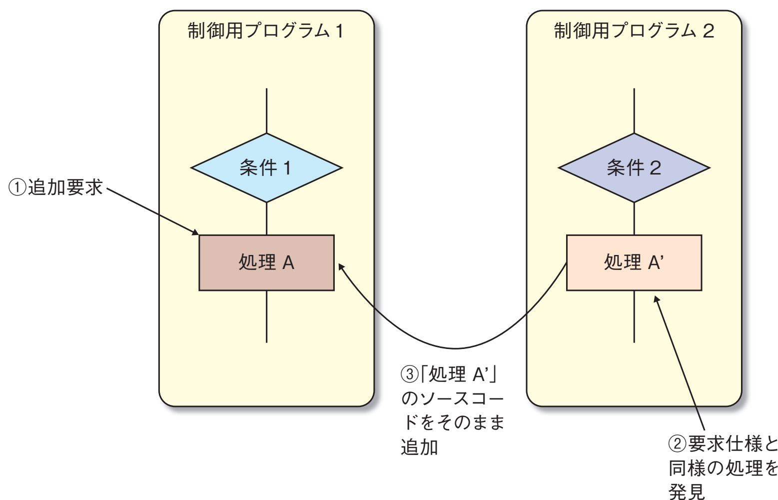
ストを行うべきである。改造元の仕様に対してどのような考えに基づいて改造したかを文書化しておくと、この回帰テストが効率的に実施出来る。

例 既存の制御装置に対して、客先より新たな要求仕様が追加された場合

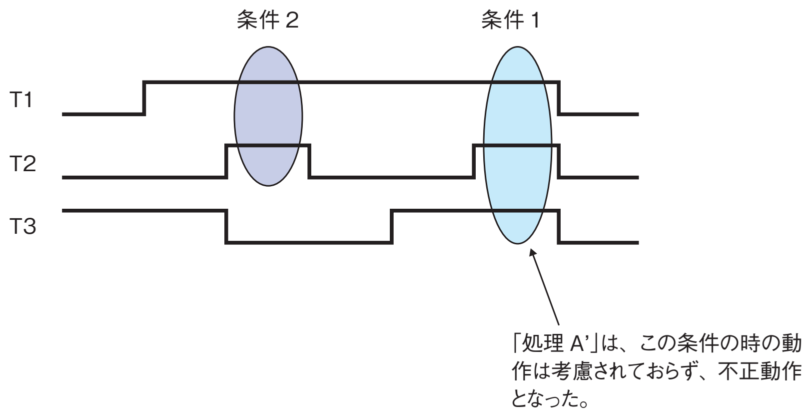
客先要求は、「制御用プログラム 1」に「条件 1」の時に動作する「処理 A」を追加するものであったため、「制御用プログラム 1」を改造して客先要求を実現することにした。改造元のソースコードを確認すると、別の「制御用プログラム 2」に「処理 A」と同様の動作をしている「処理 A'」を発見したため、「制御用プログラム 1」に「処理 A'」のソースコードをそのまま追加した。

ところが、制御装置が異常を検出し、動作不能となってしまった。

原因を調査したところ、「処理 A'」は「条件 1」の時の動作は考慮されていなかったため、「処理 A'」を追加した「制御用プログラム 1」では、「条件 1」での動作が不正となっていた。そこで、「処理 A'」に対して制御仕様に基づき「条件 1」でも正しく動作するように修正した「処理 A''」を作成し、これを「制御用プログラム 1」に追加したところ、正常に動作するようになった。



図C-13：既存の制御用プログラムの改造



図C-14：制御タイミング

関連する品質特性

- **●信頼性（成熟性）**
- **●保守性（解析性、変更性、試験性）**
- **●移植性（環境適応性）**

C-13 事前に構造解析を行うことにより効率的にソフトウェアを再設計する

作法概要

- 機能追加と改造の繰り返しによって複雑化したプログラムは、適当な時期に再設計(リファクタリング)を行う。
- 再設計は、構造解析や複雑度解析などを実施して、問題となりそうなモジュールを特定した上で実施する。

メリット

あらかじめ見直しが必要な箇所を特定することによって、低コストで効率の良い再設計が可能となる。

留意点

①設計見直しの要否の判断を含め、あらかじめ再設計計画を決めておくことが重要である。
②再設計計画には、目的、再設計規模、期間、工数、手順(設計、実装、試験、品質評価など)、設計見直し後の継続的改善方法などを含める。
③再設計後の機能や性能、タイミングなどについて、再設計前と比較して問題がないかどうか確認することが必要であり、その確認手段も準備しておく。

●解説

機能追加や流用、改造の繰り返しによって複雑化したソフトウェアを、更に機能追加や保守などを行いながら継続的に使用するためには、適切な時期に再設計を実施することが望ましい。

再設計を効率よく低コストで実施し、高品質で、生産性と保守性の高いソフトウェアに再生するためには、再設計技術・手法の適用が有効である。次にその一例を示す。

①静的解析ツールまたは実行パス解析ツールによる要再生箇所の抽出。
②モジュールの相互依存や階層を飛び超した依存関係などの分析と、問題箇所の特定。
③コードクローン検出ツール(CCFinder(http://www.ccfinder.net/)など)によるソースコードの類似性の分析と問題箇所の特定。

例 ソフトウェア再設計の手順

10 年以上前に C と C++ を用いて開発したソフトウェアを、機能拡張しながら使用を続けてきた。今後も使い続ける可能性が高いが、ソフトウェアの複雑化に伴い、保守が困難になってきた。

そこで、ソフトウェアの構造を見直すことによって、保守性、移植性及び信頼性を向上させることを計画し、再設計を次の手順で実施した。

①分析

各種ツールを使用してソフトウェア構造を見える化し、構造を分析し、危険箇所を特定した。

②設計と実装

構造分析の結果を利用し、複雑な構造の単純化、関数の共通化の再設計及び静的解析で検出した

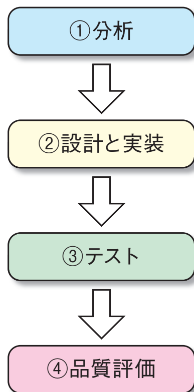
危険箇所の修正を実施した。

③テスト

改善部分についてのテスト、及び従来のテストケースを用いた回帰テストを実施した。

④品質評価

最終成果物の品質を測定し、総合評価と妥当性評価を実施した。



図C-15：ソフトウェア再設計の流れ

関連する品質特性

●信頼性(成熟性)
●保守性(解析性、変更性、試験性)
●移植性(環境適応性)

関連する作法

A-26　派生開発でソースコードの再利用性が低ければ設計の見直しを行う(P80 参照)

C-14 機能の変更・追加のときにはソフトウェアの静的構造を崩さないように注意する

作法概要

既存のソフトウェアを流用または改造して機能変更または追加を行う場合、ソフトウェアの静的構造を出来るだけ崩さないように注意する。

メリット

既存のソフトウェアを流用または改造して開発を繰り返す場合でも、当初の品質を維持することが期待出来る。

留意点

①ソフトウェアの静的構造が明確になっていることが重要であるため、新規開発などで新たにアーキテクチャ設計を行う場合は、ソフトウェアの静的構造を文書化しておくことはもちろん、そのような構造にした根拠と理由を記述しておくことは、その後の流用や改造、保守にとって重要である。

②改造にあたって、ソフトウェアの静的構造を変更しなければならない場合には、構造の複雑化を最小限にするための変更ルール(例えば、変更の条件や参照可能なクラスの種類など)を決めておくのが良い。

●解説

ソフトウェア開発では、一度開発した既存のソフトウェアを流用したり、改造するなどして、新しいソフトウェアを開発することが多い。このようなソフトウェア開発は、流用または改造の繰り返しによって当初のソフトウェアの静的構造(構成要素の責務と境界、構成要素間の依存関係)を崩し、その構造を複雑化させ、保守や機能追加などを困難とし、品質低下の原因となる。

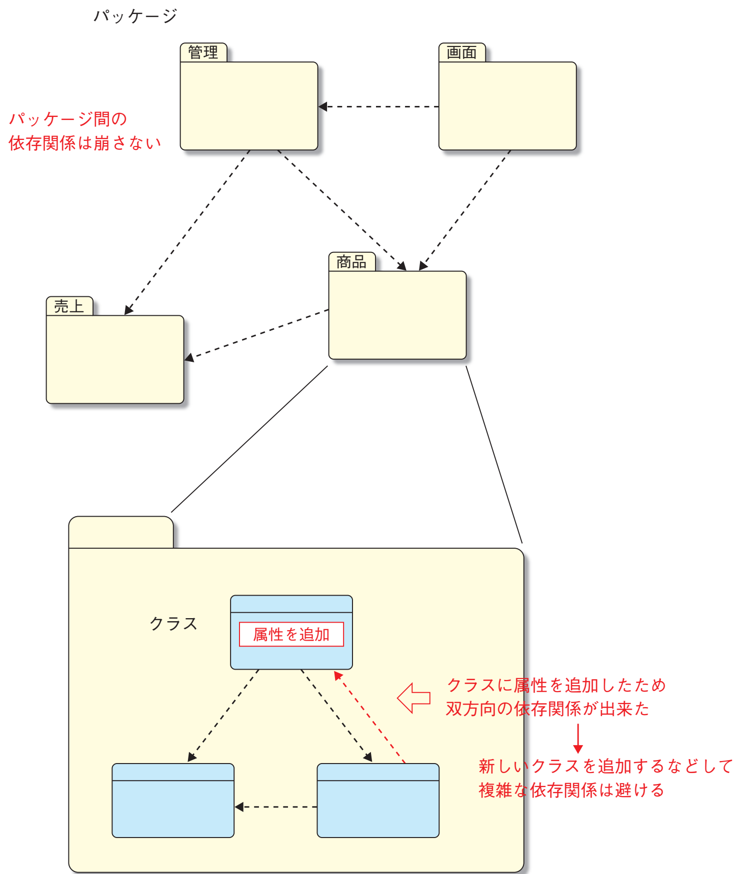
そのため既存ソフトウェアの仕様変更や機能追加を高品質で手戻りなく行うためには、ソフトウェアの静的構造をなるべく崩さないようにすることが重要である。

例1 パッケージやクラス間の依存関係

ソフトウェアの静的構造として、パッケージまたはクラス間の依存関係、及び責務の記述を例にとる。

機能を追加・変更する場合に、図 C-16 の上部に示したパッケージ間の依存関係に変更が生じないようにする。

また、図 C-16 の下部のように、あるクラスに属性を追加した結果、新たな依存関係が生じるケースもある。こういった場合は属性を追加するのではなく新たにクラスを設けるなどして、既存の依存関係に変更が生じないように設計を行う。



図C-16：パッケージやクラス間の依存関係

例2 構成要素の責務の記述

表 C-1 の黒字箇所のように構成要素へ責務を割り当てていたが、赤字の責務の追加を行った。検討してみると、元の構成要素と関連がないことが分かった。こういった場合はより適切な構成要素に追加する、あるいは構成要素を新設するなどの方法をとる。

構成要素	責務の記述
画面管理	商品コードの入力処理、商品データベースの呼出し処理を行う。 入力されたコードから、商品名、価格を表示する。
商品管理	商品コードから、該当する商品名、価格、属性を検索する。 (追加)登録された会員情報を保持する ⇦ 関連のない責務を割り当てない
販売管理	日々の売上情報から、商品別の売上を集計する。 日ごと、月ごとの売上を集計する。
・・・	・・・

表C-1：構成要素の責務の記述例

関連する品質特性

- ●信頼性（成熟性）
- ●保守性（解析性、変更性、試験性）
- ●移植性（環境適応性）

C-15 効率を意識してテーブル設計を行う

作法概要

テーブルを設計する際は、データアクセスとデータ管理が効率的に行えるよう考慮する。

メリット

- データの総容量を削減することが出来る。
- データの管理と変更を局所化することが出来る。
- データの複製を行うときに、最低限のコピーで済むため、処理速度も向上する。

留意点

①すべてのデータをまとめて管理せずに、責務ごとに管理出来るようにデータの責務を明確にして、責務外のデータを混在させないように監視する必要がある。
②データの保存または読込みを行うときに、分割された関連データをもれなく処理する必要がある。

●解説

データの構造を設計するときに、アクセス時の効率が考慮されておらず適当なデータの集まり程度の設計となっていると、データ構造が肥大になり不要な密結合が発生し、障害と性能低下の温床となってしまう。

そこで、テーブル設計を行うときに、データアクセスとデータ管理をより効率的に行えるように、次の観点に基づいて最適化を行う。

①データの重複を避け、適切な単位で分割する。
②分割した各テーブルについて、更に分割出来ないか検討する。

例 表 C-2 のテーブルを使った最適化

最初のステップは、データの重複を避け、適切な単位で分割することである。

テーブル行の内容を意味でグループ化して、そのグループを一意的に表せるキー（主キー）を抽出する。表 C-2 では、名前 ID から藩主までをユーザ情報として 1 つのグループとして分割する。その中で一意に表すことの出来る主キーは、名前 ID となる。

更に同様の手順で、イベント日時からイベント名に名前 ID を付加して、イベント情報としてグループ化する。その結果、表 C-2 を表 C-3 と表 C-4 のテーブルに分割する。

名前 ID	氏名	藩	藩主	イベント日時	イベント状態	イベント名
100	Ryoma	Ronin	NotExist	1865/5/21	unacceptable	Satsuma-Choshu Alliance
200	Kogoro	Chosyu	Takachika	1865/5/21	unacceptable	Satsuma-Choshu Alliance
100	Ryoma	Ronin	NotExist	1866/3/7	acceptable	Satsuma-Choshu Alliance
300	Takamori	Satsuma	Hisamitsu	1866/3/7	acceptable	Satsuma-Choshu Alliance
200	Kogoro	Chosyu	Takachika	1866/3/7	acceptable	Satsuma-Choshu Alliance
301	Toshimiti	Satsuma	Hisamitsu	1866/3/7	acceptable	Satsuma-Choshu Alliance

表C-2：変更前のテーブル

名前 ID	氏名	藩	藩主
100	Ryoma	Ronin	NotExist
200	Kogoro	Chosyu	Takachika
300	Takamori	Satsuma	Hisamitsu
301	Toshimiti	Satsuma	Hisamitsu

表C-3：ユーザ情報

名前 ID	イベント日時	イベント状態	イベント名
100	1865/5/21	unacceptable	Satsuma-Choshu Alliance
200	1865/5/21	unacceptable	Satsuma-Choshu Alliance
100	1866/3/7	acceptable	Satsuma-Choshu Alliance
300	1866/3/7	acceptable	Satsuma-Choshu Alliance
200	1866/3/7	acceptable	Satsuma-Choshu Alliance
301	1866/3/7	acceptable	Satsuma-Choshu Alliance

表C-4：変更後のイベントテーブル

次のステップは、更に分割可能かを検討することである。

表C-2を分割して得られた表C-3のユーザ情報において、藩主は藩によって決まるため、表C-5に示すようにユーザ情報テーブルから分離させ、新たに表C-6の藩情報テーブルを独立させる。

名前 ID	氏名	藩
100	Ryoma	Ronin
200	Kogoro	Chosyu
300	Takamori	Satsuma
301	Toshimiti	Satsuma

表C-5：変更後のユーザ情報

藩	藩主
Ronin	NotExist
Chosyu	Takachika
Satsuma	Hisamitsu

表C-6　藩情報

以上のような作業を進め、関係の薄いデータを分離することによって、テーブルのデータ構造の重複を避け、関係の浅いデータが一体化するのを防ぎ、データ変更時の更新漏れと不整合を起こりにくくする。また、複製時に不必要なデータのコピーが行われず、更に、排他制御の単位が小さくなるため、処理速度が向上する。

関連する品質特性

●保守性
●効率性(資源効率性)

C-16 データアクセスを局所化し データ更新処理による不整合を防ぐ

作法概要

データに対する更新または参照処理が複雑な手続きを要する場合、アクセスが必要な処理に対して、データアクセスのためのAPI(Application Interface)を提供する。

メリット

データ更新または参照が複数のモジュールから実行されても、データの不整合が発生しない。

留意点

データアクセスのための API を提供した場合、該当データを隠蔽して、直接アクセス出来ないようにする。

●解説

画面構造が複雑になってくると、多くの画面から、同一データに同じ参照手順でアクセスする必要が出てくる。このような場合に、データアクセスの手続きを各画面で実装していると、クローンコード(類似コード)が多数発生し、仕様変更時などにすべて変更しなければならなくなり、保守性を低下させてしまう。

そのために、データに対する参照手順をある共通ルーチン内に隠蔽し、この共通ルーチンを呼び出す API を用いてアクセスするようにする。こうしておけば、仕様変更時にも共通ルーチン内部を修正するだけで済み、影響範囲を制限することが出来て保守性が向上する。

例 複数画面を有する、大型の事務機器組込みソフトウェア

データの更新方法が画面によって異なり、不整合が発生していた。データを更新する複雑な手続きが各画面で実装されており、データの仕様が変更になったときに、すべての画面での処理を仕様通りに修正出来ていなかった。

そこで、データを隠蔽し、複数の手続きを含むアクセス用の API を公開することにした。どの画面もこの API を通してのみアクセス可能となったため、不整合は発生せず、仕様変更があっても画面処理側に影響は起きなくなった。

関連する品質特性

- ●保守性
- ●信頼性

C-17 モジュールの参照方向を上位から下位へ統一する

作法概要

モジュールやパッケージなどの参照方向は、コントロールする側からコントロールされる側への一方向を原則とする。

メリット

- 複雑度が低下して、修正時の影響が少なくなる。
- モジュールやパッケージなどの再利用性が向上する。

留意点

原則を破って上位参照が必要な場合は、アーキテクトなど必ずアーキテクチャに詳しい専門のレビューを受ける。

●解説

モジュール分割を行う場合、モジュールが他のモジュールを呼び出す方向、いわゆるモジュールの参照方向は、上位から下位への一方向に統一するように設計する。このように設計することによって、下位モジュールとの関係を疎結合にし、下位モジュールの再利用性を上げることが出来る。また、関係が疎であるので、上位モジュールを修正しても下位モジュールに影響が及ばず、修正による不具合を最小限に押さえることが出来る。

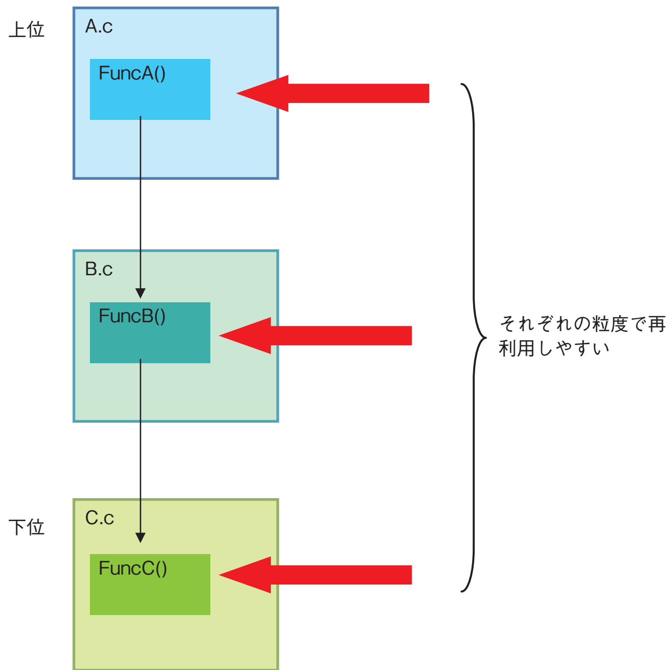
しかし、初期設計でモジュールの参照方向を上位から下位への一方向に統一するようにアーキテクチャ設計を行っても、その後の改変時に、アーキテクチャ違反となる修正が入ってしまう。主な理由としては、修正者がアーキテクチャ設計を理解していない、もしくは、理解しているが、納期などの時間的制約のもとで動作させるために違反してしまう、という2点が挙げられる。

前者に関しては、修正者のスキル不足などによって、当初の設計意図と逆方向の参照が行われてしまい、その結果、構造が複雑化し、品質の低下が発生する。これを防ぐためには、修正時のアーキテクトなどによるレビュー、もしくは参照方向違反を検出するツールの活用がある。

後者に関しては、納期を理由に違反修正していることをバックログに記録しておき、のちのバージョンアップ時に確実にリファクタリングをかけて修正することが重要である。

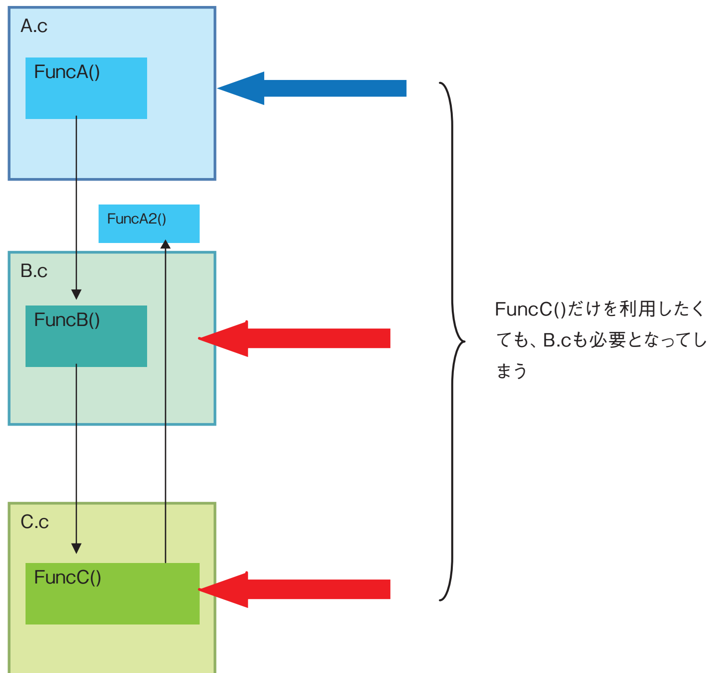
例 参照方向の違いによる影響

上位から下位への一方向参照の場合、依存関係が疎になるので、どの階層においても再利用が容易になる。



図C-17：一方向参照の原則に沿ったブロック図

一方、一方向参照の原則が崩れ、下位モジュールが上位モジュールを参照すると、下位モジュールを再利用するとき、それが参照している上位モジュールだけでなく、その上位モジュールが参照するすべてのモジュールが必要になってしまう。



図C-18：一方向参照の原則が崩れたブロック図

関連する品質特性

- ●信頼性
- ●保守性

C-18 機能変更を行う前にパフォーマンスに与える影響を十分検討する

作法概要

- 部分的なソフトウェアの変更でも他の部分に影響し得ることを認識する。
- 機能追加または変更に伴うパフォーマンスへの影響について、事前に十分検討する。

メリット

機能追加または変更に伴う影響を緩和することが出来る。

留意点

変更が他の部分へ影響を与える場合、影響があったとしても変更を実施するのか、または実施しないのかは十分に検討する。

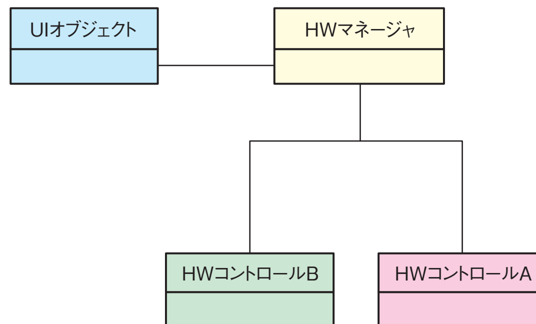
●解説

機能追加や、操作性向上のための変更などを行うと、他の部分の動作に何らかの影響が発生する。変更に伴う影響は必ず発生することを認識し、変更の実施前に十分検討しておかなければならない。

また、処理の単純な追加またはデータ構造の変更によってパフォーマンスへの影響が発生することはよく見られるため、とくに注意する。

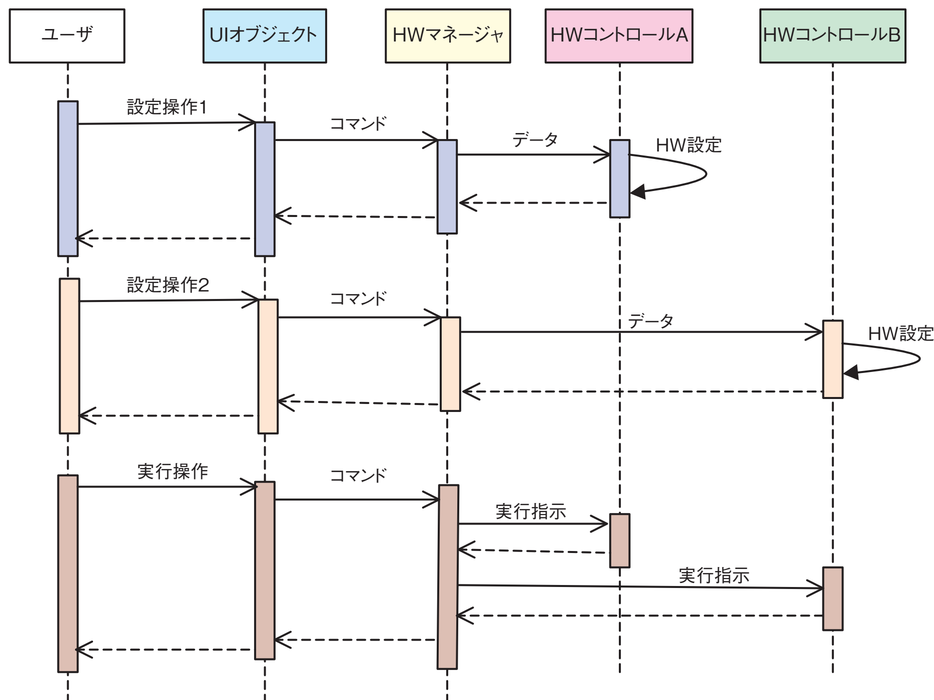
例 操作性を向上させたために、パフォーマンスに影響を及ぼした場合

図 C-19 のオブジェクト図に示すシステムがある。ユーザからの操作は UI オブジェクトで受ける。UI オブジェクトは、ユーザからの操作をコマンドに変換し、HW マネージャへ送信する。そのコマンドは、HW マネージャによって各 HW を設定するためのデータに変換され、対応する HW コントロールへ送られる。各 HW コントロールは、それぞれが対応する HW へその値を設定したり、HW を動作させたりする。



図C-19：オブジェクト図

全体の動作としては、まず各 HW に対して動作させるために設定値を与え、その後 HW を動作させる。図 C-20 にそのシーケンス図を示す。



図C-20：変更前のシーケンス図

図 C-20 に示すように、ユーザによる設定操作を UI オブジェクトが受ける。UI オブジェクトは HW マネージャにコマンドを送信する。HW マネージャはコマンドから該当する HW を特定し、設定するデータを決定する。決定されたデータは該当の HW コントロールに送られ、HW コントロールが HW へ設定する。

すべての設定が完了後、ユーザが実行操作を行うと各 HW へ実行指示が出され、あらかじめ設定された値で HW が動作する。

このソフトウェアに対して、操作性向上のために次の機能追加を行う。

① HW 設定前の設定操作の取り消し

② HW 動作順の制御

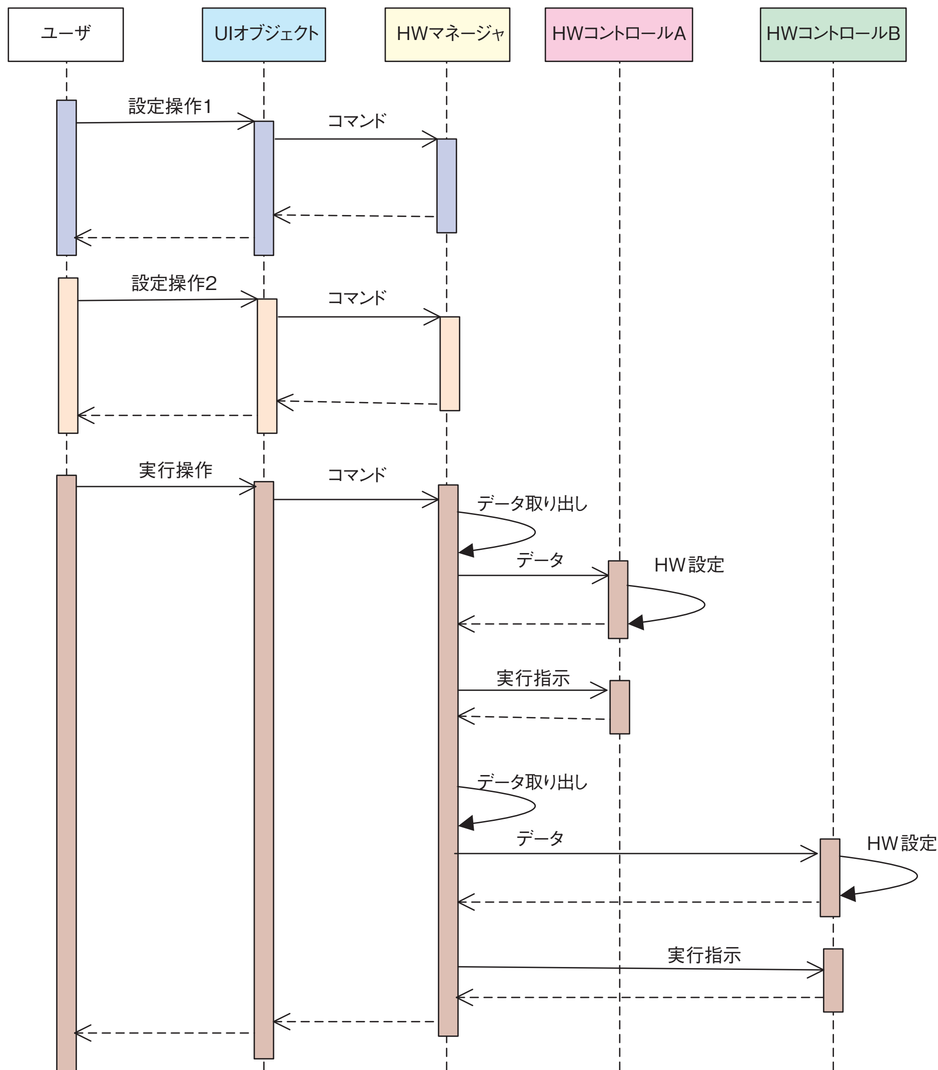
これらの機能追加を実現するために、HW マネージャに次の変更が必要となった。

③設定操作では HW コントロールへのデータ送信は行わない。

④設定操作で作成されたデータは実行操作まで保存しておく。

⑤実行指示のときに設定データを取り出し、HW 設定後に HW を動作させる。

この変更を実施したあとのシーケンス図を図 C-21 に示す。



図C-21：変更後のシーケンス図

図 C-21 に示すように、実行操作内でデータの取出し、HW への設定、実行指示が行われるため実行操作が従来の処理よりも長くなり、操作性を向上させた結果、パフォーマンスが悪化してしまった。

関連する品質特性

●保守性（変更性、安定性）
●効率性（時間効率性）

Part D

システムで扱う
周辺デバイス操作に関する工夫

D-1 多重割込みはどうしても必要な場合のみ使用する

作法概要

多重割込みを使用する前に、要求されるリアルタイム性などを検討し、その使用をどうしても必要な場合に限定する。

メリット

多重割込みをどうしても必要な場合だけに限定することにより、テストしやすくなる。

留意点

どうしても多重割込みが必要な場合は、リアルタイム OS(RTOS)を使用する。

●解説

多重割込みが可能なシステムについて、ハードウェアは同一という条件で、多重割込みを使用しないシステムと比較する。多重割込みが可能なシステムでは、外部からの入力に対して出力を返すまでのレイテンシ(遅延時間)が小さく、その分だけ、リアルタイム性が良い。しかし、多重割込みのタイミングのパターンは複雑になり、それを網羅したテストを行うのは非常に難しく、不具合が出た場合の解析も困難になる。

多重割込みを使用する前には、多重割込みを使わない場合のレイテンシなどのリアルタイム性を計測し、多重割込みを使用する必要があるかどうかを判断して、どうしても必要な場合だけに限定する。

例 LAN 間接続用のルーター

レイテンシを低くするため、パケット受信に多重割込みを使用することにした。テストは正常に通過したが、実際に運用してみると、輻輳時にパケットが壊れるケースがあることが分かった。莫大なデータを追跡した結果、タイミングによっては予期せぬ資源の競合が発生し、パケットのデータが破壊されることが分かった。

こういったケースは回避が困難と判断し、多重割込みを使用しない解決法を調査した。その結果、デバイスを定期的にポーリングすることにより十分なレイテンシが得られることが分かったため、設計を変更し実装を行った。

関連する品質特性

●機能性(合目的性、正確性)
●保守性(解析性、変更性、安定性、試験性)

D-2 デバイス初期化時の副作用を回避する

作法概要

マイコン周辺デバイスの初期化は、リセット後の状態を勘案して実施する。

メリット

リセット後のデバイス初期化時に、機器の誤動作を防止することが出来る。

留意点

①デバイスのリセット後のポート状態が、入力モードか出力モードかを確認する。
②デバイスに接続された外部回路の論理を確認する。
③デバイスの初期化は、外部回路をアクティブにしない手順で行う。

●解説

マイコン周辺デバイスの初期化は、デバイスのリセット後のポート状態を理解し、外部接続された回路の論理を把握して行わないと、予期しない動作が発生する原因となる。

リセット後に入力モードとなる、双方向のパラレル I/O デバイスでは、リセット直後のポートの論理状態は、デバイス内部または外部のプルアップ/ダウン抵抗で決まる。また、リセット後のポート出力用のデータレジスタの値は、不定になる場合または強制的に Low にされる場合がある。

このため、データレジスタの初期値を適切に設定せず、ポートモードレジスタを出力モードへ切り替えると、プルアップ/ダウン抵抗で設定した論理と異なる出力をする結果となり、外部回路がアクティブになる可能性がある。これを防ぐには、次の事項に留意する。

①リセット直後のデバイスの内部状態に関して、ポートのモードの種別(入力か、出力か)や、Hi/Low/ハイインピーダンスの論理状態などをデバイスのマニュアルで確認する。
②ポートに接続された外部回路の論理が、Hi アクティブか Low アクティブなのかを把握し、かつそのため、ポートがプルダウンまたはプルアップされているかを回路図などから確認する。
③リセット直後、モードレジスタで方向を設定する前から出力データレジスタの設定は有効であるため、出力として使用するビットには、外部回路がアクティブにならないようにあらかじめ初期値を設定する。

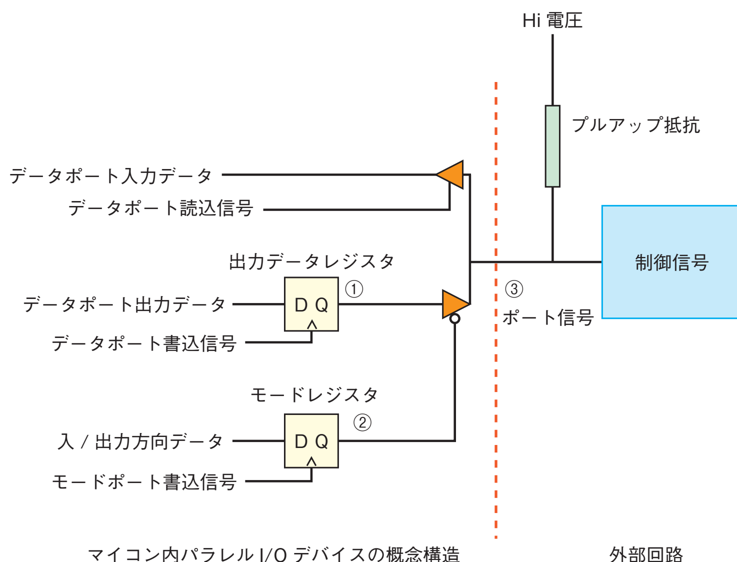
例 パラレル I/O デバイスによって、外部回路のヒータ制御を行う

図 D-1 の回路例において、ポートは、ハードリセット後にヒータが OFF となるように外部抵抗でプルアップされており、ポートが Low になった時だけヒータが ON となるように負論理で設計されていた。

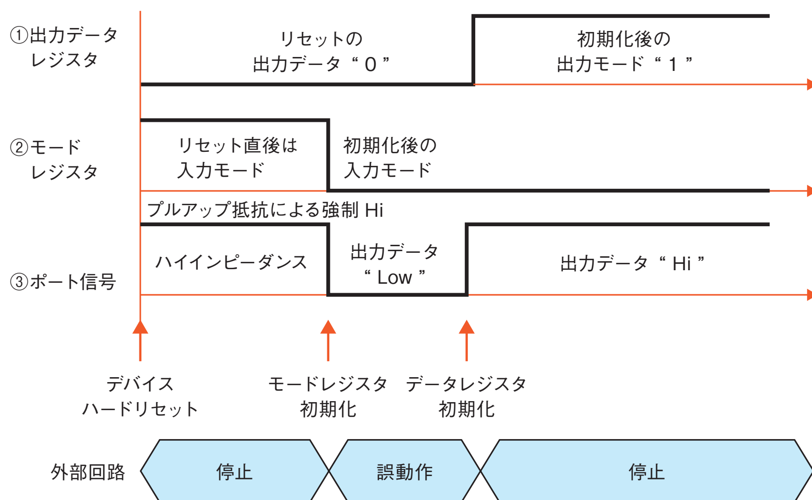
デバイスの初期化は、図 D-2 のタイミングチャート例に示す手順で行った。デバイスのハードリセット後にデータレジスタ値を変更しないまま、モードレジスタを入力モードから出力モードへ切替えを

行ったため(モードレジスタ初期化)、ハードリセット時のデータレジスタ値“0”がそのまま出力され、ポート出力はLowへ遷移した。その後、ヒータを非アクティブにするために、データレジスタへ“1”を書き込んだ(データレジスタ初期化)が、その間にヒータが異常加熱してしまった。

この対策として、データレジスタ初期化後にモードレジスタ初期化を行うように修正し、正しい動作になるようにした。



図D-1：回路図



図D-2：タイミングチャート

関連する品質特性

●機能性(正確性)
●信頼性(成熟性)

D-3 周辺デバイスのI/Oアクセスは専用関数・マクロを用いる

作法概要

周辺LSIのレジスタまたはメモリをアクセスする場合、アクセス専用の関数またはマクロを用いて行う。

メリット

- クロス環境でのデバッグが容易になる。
- I/O アクセスログの取得が容易になる。

留意点

①性能要求が厳しい場合は、関数呼出しによる実行時間のオーバヘッドが懸念されるため、マクロ化やインライン関数、インライン展開による最適化などで対応することを考える。そのときには実行可能コードのサイズが増大するため、それに伴うキャッシュ効率の低下などの副作用に注意する。

②マクロ化で対応した場合には、マクロが展開されたあとにはソースコード上で区別がつかないため、機械的な検索などのツールによる支援が限定される。

●解説

周辺 LSI のレジスタまたはメモリをアクセスする場合、C または C++ 言語を利用していれば、アクセス専用の関数、メソッド、またはマクロを定義し、これらを用いて I/O アクセスを行う。これによって I/O アクセスしている箇所を機械的に抽出出来るようになり、次のようなメリットが生まれる。

①クロス環境でデバッグを行うときに、周辺 LSI の動作を模擬するスタブへの切替えをコンパイルスイッチで行うことが出来る。

② I/O アクセスログの取得を統一して行うことが出来る。

③上位の処理ルーチンに対して、I/O アクセス方法の詳細を隠蔽することが出来る。

関連する品質特性

- 保守性（解析性、変更性、安定性、試験性）
- 信頼性（成熟性）
- 移植性（環境適応性）
- 生産性

D-4 ハードウェアを保護するための機能レイヤを設ける

作法概要

- アクチュエータへの指令値は、論理的に求めたあとにハードウェア特性を考慮して補正する。
- 指令値を論理的に求める部分とハードウェア特性を考慮する部分とは、ソフトウェア構造上、分離しておく。

メリット

- アクチュエータへの指令値を直前に漏れなく補正することによって、ハードウェアへの誤指令、ひいてはハードウェアの破壊を防止することが出来る。
- 論理的に指令値を求める部分と補正を行う部分とを分離することにより、制御ソフトウェアの変更が容易になる。

留意点

指令値の補正を行う部分が指令値の時間的変化量を扱う場合、論理的に指令値を求める部分との間でその時間的変化量を受け渡すことによって、変化量の再計算を抑える。

●解説

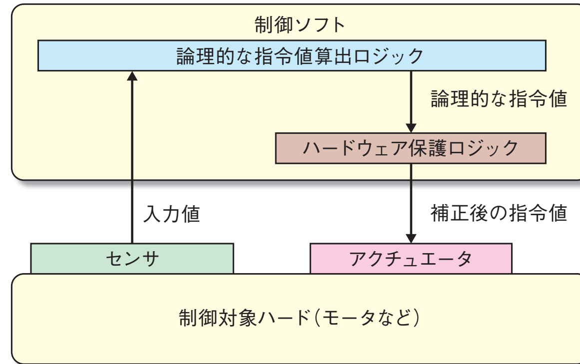
センサからの入力値に基づいてアクチュエータへの指令値を決定し、これに基づいてハードウェア(モータやインバータなど)を制御する制御システムの開発においては、指令値の算出方法は制御設計工程の結果として決まる。一方で、ハードウェアの特性として、指令値に制約がある場合がある。例えば、指令値の上限が設定されている場合や指令値の時間的変化量に制限がある場合などである。

このように、制御設計の結果に基づいて論理的に求められる指令値がハードウェア特性による制約を満たすとは限らないため、ハードウェア特性を考慮した保護ロジックを組み込む必要がある。このとき、指令値を論理的に求める部分と、ハードウェア特性を考慮した保護ロジックとは分離しておくのがよい。これによって確実にハードウェアを保護することが出来ると共に、次のような、制御方式かハードウェア特性のいずれか一方が変わった場合にも、変更を容易に実施出来る。

①制御方式の変更、チューニング、またはキャリブレーション

②制御対象ハードウェアの変更

例 **指令値を論理的に求める部分とハードウェア特性を考慮する部分とを分離した、レイヤ構造の一例を図 D-3 に示す。**



図D-3：レイヤ構造

関連する品質特性

- ●機能性（合目的性、正確性）
- ●信頼性（成熟性、障害許容性）
- ●保守性（変更性）
- ●移植性（設置性）
- ●安全性

D-5 デバイスの寿命に関する特性を考慮して設計する

作法概要

不揮発性メモリーなどの寿命があるデバイスについては、設計にあたって次の項目を考慮する。
①バッファによる利用回数の低減や、書き込みを分散する平準化など実効的な寿命を長くする工夫
②デバイスの一部の故障がシステム全体に影響しないような冗長性の担保
③デバイスの寿命が尽きた場合に対するエラー処理

メリット

そのデバイスが機能しなくなることによって、システム全体の信頼性や可用性に影響が及ぶのを軽減することが出来る。

留意点

①デバイスの寿命が尽きて機能しなくなった場合に、デバイスの交換を通知するなどのエラー処理をあらかじめ設計しておかないと、システムは機能不全から回復できなくなる。従って、デバイスの寿命が尽きた場合に対応するテストケースを十分に検査しておく必要がある。
②フラッシュメモリを用いた記憶機器や二次電池などは、デバイスメーカやOSベンダなどによって管理モジュールが提供されており、多くの場合、それらは特許化されている。従って、デバイスによっては、その寿命特性に対する処理として、それらメーカーやベンダなどの提供する管理モジュールを採用する方がライセンスにも抵触せず、また信頼性も保証され、有利になる。
③平準化処理を含むソフトウェアまたはファームウェアのバージョンアップがある場合には、平準化処理によるマッピングに影響が生じる可能性があるので注意すること。影響する場合、記憶デバイスに保存されている内容が正しく復元できなくなる場合がある。
④あくまでデバイス寿命に対するシステム健全性を維持することを目的としており、悪意のあるユーザ操作や外部からの攻撃に対する安全性は保証出来ないことを念頭におく。

●解説

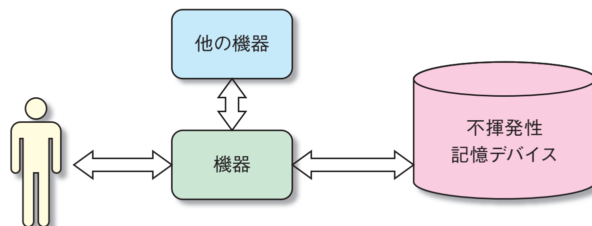
フラッシュメモリやNVRAMなどの不揮発性記憶デバイスは、書き込み回数の上限や内蔵電池容量などに起因する寿命を持つ。このため、これらのデバイスを構成要素に持つ組込みシステムは、その寿命の特性を考慮した設計が必要になる。

ある種の設定情報を寿命がある記憶デバイスに保存する場合、書込み部位を平準化して冗長な書込みを行い、読出し時のエラー補正をするように設計する。書込み回数の上限や電池切れなど、寿命が

尽きた場合に対するエラー処理も、あらかじめ設計しておく必要がある。

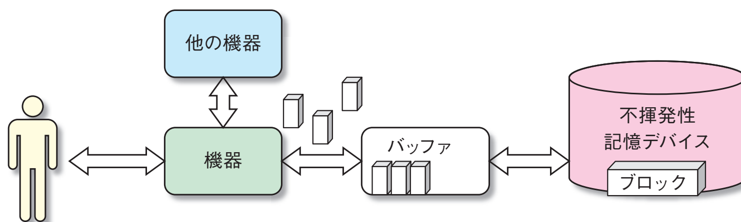
例 通信機能を持つ機器

この機器は、ユーザーの操作と他の機器からの通信による入出力とを機能として持つ。入力の一部によって機器の設定が変更され、この変更された設定は不揮発性記憶デバイスに記録される。また、機器の動作に伴って変化した機器の状態も、必要に応じて同様に記録される。



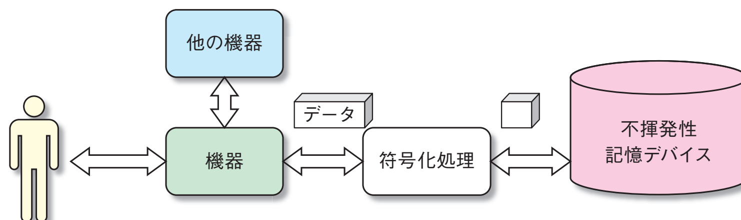
図D-4：機器の機能の仕組み

これらの記録に対する書込みは、不揮発性記憶デバイスとして書込み回数が有限なものを使用しているため、設計上の工夫を必要とする。1つは、書込みをブロック単位にバッファにまとめることによって、実際の書込み回数を削減する工夫である。ただし、設定変更または状態変化のタイミングと書込みタイミングが同期しないことになるため、機器の稼動停止時に書出しを必ず行うように設計し、十分にその動作を検証する必要がある。また、意図しない電源断などの障害発生時に、未書込み分が失われるリスクがあることも、承知した上で設計する必要がある。



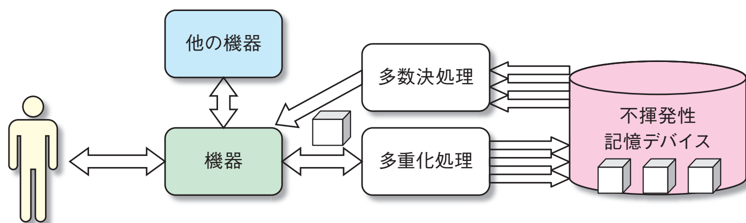
図D-5：設計上の工夫①

その他の工夫としては、書き込む情報量を最小化するために、可逆圧縮などの符号化処理を行うというものがある。これは実行時間とのトレードオフになる。



図D-6：設計上の工夫②

寿命がある不揮発性記憶デバイスの場合、特定のブロックのみが寿命となって、読出しデータが異常になることがある。このため、機器の設定など、データ健全性に対する要求が高いものについては冗長化しておく。具体的には、複数のブロックに同じ設定データを書き込んでおき、読み出し時には常に多数決処理を行うという設計にしておく。



図D-7：寿命がある不揮発性記憶デバイスの場合

関連する品質特性

信頼性（障害許容性、回復性）

関連する作法

A-22　故障を回避する（P70 参照）

D-6 ハードウェアの機能と物理的特性を十分に分析してから設計する

作法概要

ハードウェアが実現している論理的機能とユーザに提供すべき機能の間に、ハードウェアの物理的特性による乖離がある場合は、十分にハードウェアの物理的特性の分析を行う。

メリット

ハードウェアの物理的特性に制限されずに、ユーザが期待する機能を提供することが出来る。

留意点

ハードウェアに関する情報を入手するだけではなく、必要に応じてハードウェアを動作させてその特性を調べなければならない。

●解説

ソフトウェアの制御対象となる個々のハードウェアは、物理的特性のためにユーザに提供すべき機能をそのまま実現出来ているわけではない。ハードウェアとして備えるべき機能・性能と、ユーザに提供すべき機能との間に往々にして乖離が発生する。

このように、ユーザに提供する機能とハードウェアが実現している機能との間に乖離がある場合、ソフトウェアがその差を埋める必要がある。

そのために、とくに次のような場合には、ハードウェアの機能を十分に分析してソフトウェアの設計を行う。

①時間制約など機能仕様がシビアである。

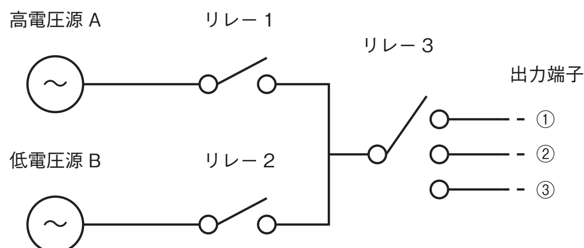
②ハードウェア制御の手順が多い。

③過去の操作に依存して、ハードウェアの制御を行う。

④他のハードウェアの状態に依存して、対象ハードウェアの制御を行う。

また、実際の作業では、ハードウェア開発者(開発元)からの情報以外に、ハードウェアを動作させ、その特性を確認しながら、ソフトウェアの作成を行う場合もあり得る。

例 制御対象ハードウェア



図D-8：制御対象ハードウェア

図 D-8 はソフトウェアで制御するハードウェアの例を単純化して示したものである。このハードウェアの機能は電圧を出力することであり、ユーザから見た機能は次のようになる。

①指定した電圧を出力する。

②指定した出力端子から出力する。

一方、ハードウェアの仕様は次のようになる。

① 10V 以上の電圧を得たい場合は高電圧源 A の出力を、それ以下なら低電圧源 B の出力をそれぞれ利用する。

②電圧源 A と B には、電圧値を設定してから電圧出力が安定するまでの待ち時間があり、その値は異なる。

③電圧源 A と B を直接接続してはいけない。

④リレー 1 と 2 には、接続が安定するまでのセトリング時間がある。

例えば、「5V の電圧を出力端子①から出力」していたときに、以前の状態とは異なる「15V の電圧を出力端子③から出力」する動作をさせるとすれば、次の手順を取らなければならない。

①予期せぬ出力が発生しないように、リレー 3 をすべての出力端子から OFF。

②以前出力していた側のリレー（リレー 2）を OFF。

③高電圧源 A を 15V に設定。

④高電圧源 A の待ち時間を待って、リレー 1 を ON。

⑤リレー 1 のセトリング時間を待ってリレー 3 を出力端子③に接続。

このように、ソフトウェアはリレーの接続順番、以前の接続の状態、待ち時間など、複数のことを考慮しながら、ユーザの操作に対応してハードウェアを制御する必要がある。

関連する品質特性

機能性（合目的性）

D-7 ハードウェアの操作を出来るだけ遅らせてパフォーマンスを向上させる

作法概要

●ハードウェアに対するアクセスを1箇所に集める。
●ユーザ操作に伴うハードウェアの操作を、必要になるまで実施しない。

メリット

ハードウェアを動作させる回数を減らすことで、パフォーマンスを改善出来る。

留意点

①ユーザ操作が、実際のハードウェアの設定に反映されているか管理する必要がある。ハードウェアの状態によってハードウェアの動作が変化する場合、ユーザ操作による状態の変化を見るのではなく、実際のハードウェアの状態を把握して、ハードウェアを動作させる必要がある。
②ハードウェアの応答時間が十分短い場合、これらの工夫は逆にパフォーマンスを低下させる原因ともなる。

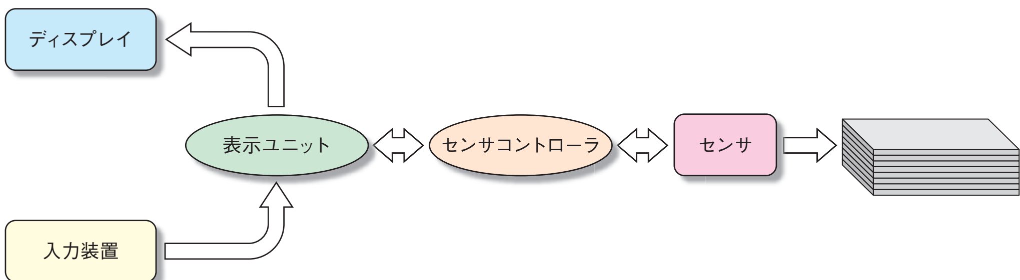
●解説

ソフトウェア処理と比較すると、ハードウェアの設定や動作などには、一般的にその完了までに時間がかかる。このハードウェアに対する操作を極力減らすことが出来れば、パフォーマンスが向上する。

具体的には、ハードウェアをアクセスする箇所を1箇所にまとめ、管理出来るようにしておく。更に、ユーザ操作に伴うハードウェアへの操作を可能な限り遅らせることによって、重複する操作を回避でき、必要のない操作を省略することが出来る。

例 紙の残量をセンサで検知し、ディスプレイに表示するコピー機

その概略図を図D-9に示す。

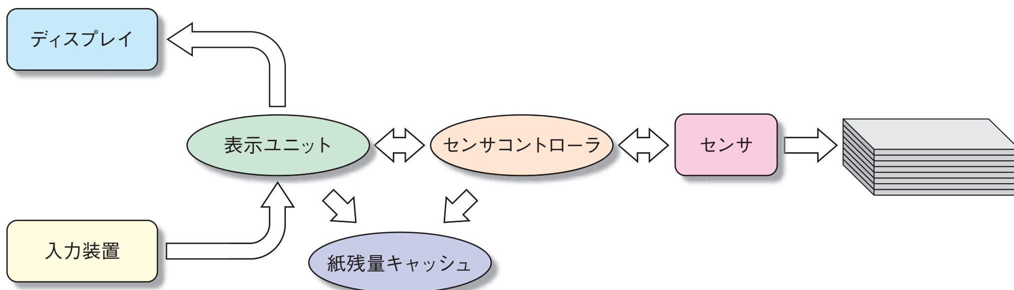


図D-9：毎回センサを動作させる構成

ユーザからのコピー機への操作に対して、結果をディスプレイに表示する。このとき同時に、センサの走査によって紙の残量を取得し、これを表示に反映する。

この方法ではユーザからの操作のたびにセンサを動作させる時間が必要となり、ユーザインターフェースが使いづらいという印象を与えていた。

図 D-10 に改善した構成例を示す。



図D-10：キャッシュを持った構成

センサコントローラは、一度取得した紙残量を紙残量キャッシュに格納しておく。表示ユニットは、紙残量がディスプレイの表示に必要な場合は、通常、紙残量キャッシュから取得する。こうすることで、センサを動作させることなくディスプレイへの表示が行われ、パフォーマンスが改善する。

ただし、ユーザが紙残量を変化させる操作を行った場合は、キャッシュを無効化する。これによって次の表示要求時にはセンサが動作し、紙残量キャッシュのデータを更新する。

関連する品質特性

効率性（時間効率性）

参考文献

[1] "A Pattern Language: Towns, Buildings, Construction", Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, Shlomo Angel, Oxford University Press 1977

[2] "Software Design Methods for Concurrent and Real-Time Systems", Hassan Gomaa, Addison-Wesley Publishing Company 1993

[3] "Design Patterns: Elements of Reusable Object-Oriented Software", Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley Publishing Company 1994

[4] "Pattern-Oriented Software Architecture Volume 2 Patterns for Concurrent and Networked Objects", Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann, Addison-Wesley Publishing Company 2000

[5] "Desigining Concurrent, Distributed, and Real-Time Applications with UML", Hassan Gomaa, Addison-Wesley Publishing Company 2000

[6] 「組込みシステム概論」、戸田望 編著、高田広章・枝廣正人・清水徹・中島達夫・平山雅之 共著、CQ 出版社 2008

[7] "A Practical Guide to SysML", Sanford Friedenthal, Alan Moore Rick Steiner, Morgan Kaufmann OMG Press 2009

編著者（敬称略・五十音順）

岩橋　正実　　三菱電機メカトロニクスソフトウエア株式会社
遠藤　雅光　　キヤノンソフトウェア株式会社
小笠原公一　　三菱電機株式会社
小野　康一　　日本アイ・ビー・エム株式会社
菊地　一成　　キヤノン株式会社
島袋　　潤　　株式会社日立製作所
舘　　伸幸　　ルネサス マイクロシステム株式会社
谷本　晃仁　　セイコーエプソン株式会社
中村　　洋　　株式会社レンタコーチ
楢木野公彦　　横河医療ソリューションズ株式会社
山本　浩数　　パナソニック株式会社

石井　正悟　　IPA/SEC（東芝ソリューション株式会社）
十山　圭介　　IPA/SEC（株式会社日立製作所）
浜田　直樹　　IPA/SEC（株式会社アックス）
三原　幸博　　IPA/SEC（株式会社アルパイン）

監修

安全ソフトウェア構築技術部会

編集・著作

独立行政法人 情報処理推進機構
技術本部 ソフトウェア・エンジニアリング・センター

SEC BOOKS

組込みソフトウェア向け設計ガイドESDR〔事例編〕

2012年11月12日　1版1刷発行

著　　者　独立行政法人 情報処理推進機構（IPA）
技術本部 ソフトウェア・エンジニアリング・センター（SEC）

発 行 人　松本　隆明

発 行 所　独立行政法人 情報処理推進機構（IPA）
〒113-6591
東京都文京区本駒込二丁目28番8号
文京グリーンコート センターオフィス
URL　http://sec.ipa.go.jp/

© 独立行政法人 情報処理推進機構　技術本部 ソフトウェア・エンジニアリング・センター *2012*

ISBN978-4-905318-14-9　Printed in Japan