デジタルスチルカメラへの ZIPC 適用

富士フイルムソフトウエア株式会社
ソフトウエア事業部　機器ソフト開発部　制御ソフトグループ
伊藤　洋一

1．はじめに

弊社では、デジタルカメラ、民生用／業務用プリンタの組込み制御ソフトウエアを開発しております。特にデジタルカメラにおいては競合他社との激しいシェア争い／市場ニーズの変化／マーケティング戦略等、色々な要因があり、商品のライフサイクルが短くなっており、さらに近年は低価格化／付加価値としての新規機能増加／多機種戦略で複数商品の同時開発をしなければならない状況になっております。

2．ZIPC 導入の背景

今回 ZIPC の導入効果を評価することになった背景は、(1)近年の開発ではプロジェクト終盤での不具合が増加しており、コスト的に見逃すことが出来ないレベルまで達した、また、(2)現状の開発現場の状況を何とか改善しなければ、という機運が生まれたためであります。

これら問題点を明らかにするために、まず現状の問題点を洗い出すことから始めました。弊社にはプロジェクト毎にソフトウエア出荷後の不具合を管理するデータベースがあります。これらを使い、終了したプロジェクト全ての不具合項目を分類する作業を行いました。

図 1 より、不具合の多くは設計段階で発生しており、ソフトウエアの品質向上には設計段階での不具合低減が有効である事が分かります。また、仕様変更／仕様不備による不具合も 15%近くあり、見逃す訳にはいきません。

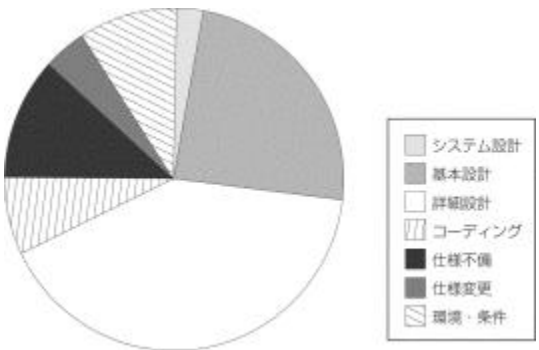

図１　不具合発生の原因となった工程

設計段階で不具合が発生する原因を考察すると、以下の原因が考えられます。

(1) 要求分析／基本設計の曖昧さ

分析工程で必要なシステム／要求仕様が曖昧であるため（提出される要求仕様書は我々をスペシャリストと見込んでいる)､その結果を示す設計書も曖昧であるといった状況下にあります。さらにツール／フォーマット等が決まっておらず、各設計者がそれぞれの手法で設計書を残しています。この結果、内部でのデザインレビューに一定の効果が期待できないというのが現状です。

(2) 詳細設計での仕様書は後回し

分析工程で大体の仕様を把握し、設計者が頭の中で設計してしまい（設計者は状態遷移表をかくのと同じ頭で直接コーディングしている！）実装工程に入ってしまう。プログラミング言語ですべき設計は無理に行っていません。具体例としては、

- データ構造は構造体定義で直接設計
- 状態遷移は ROM テーブルで直接設計
- インターフェースは関数プロトタイプ宣言で設計
- ヘッダファイルが一番有用な仕様書

つまり、仕様書が書かれたとしても、それらはソースコードから作成することになり、何のための仕様書か分からなくなってしまいます。但し、この考え方も全て間違っている訳ではなく、ソースコードと仕様書が一致していれば問題はないとも考えられます（この場合、設計書という言い方の方が正しいと思われます。）どちらにしても仕様書は後回しであることは変わりありません。

(3) 仕様変更は設計書に残らない

本来なら、上流工程に戻り、影響範囲をしっかりと見極め、再設計を行わなければならないのですが、上流工程での設計書が曖昧なために、同様に設計者が頭の中で設計してソースコードを作成しています。全ての影響範囲を洗い出すことが出来た場合は良いのですが、それが出来なかった場合モグラ叩き状態に陥ることになります。
勿論、上記の様な設計であるため、

(4) 単体試験／結合試験とデバッグとの区別の曖昧さ

といった問題も生じ、結果的に納期までにデバッグ／試験を繰り返すことにより品質を高めていくという開発になります。

現状分析の結果、以下の問題点が考えられます。

(1) 分析工程での仕様の曖昧さ。
(2) 設計工程での設計書が可視化されていない。
(3) 製造工程でのソースコードと設計工程での設計書の不一致。
(4) テスト工程とデバッグとの区別の曖

味さ

問題(1)に関しては開発プロセス全体として改善が必要であるが、現場の設計者としては(1)、(3)、及び(4)の問題点は深刻です。

これら問題点により、以下の問題が発生しています。

(1) ソースコード解析ミス
(2) コミュニケーションミス

ソフトウエアを流用する場合、設計者は、まずソースコード解析から始めます。この結果をもとに前任者から不明点のヒアリングを行い、開発を行うことになります。この時、ソースコード解析が不十分だとヒアリングで全ての内容が明らかにされません。また、要求仕様書が曖昧／設計内容が可視化されていないため、前任者が忘れていた仕様は開発後期になり発覚し、大きな問題となってしまうことがあります。

最も怖いのは、ソフトウエアを流用して問題点が発生した場合、既に終了したプロジェクトに対しても同じ問題が存在している可能性が高くなります。

3. ZIPC 導入のねらい

現状の開発では、分析工程から設計工程への接続性も悪く（分析は自然言語／ドローツールで、設計はプログラミングとなり全く接続性がない)、これらを改善するためのツールとして ZIPC の導入を検討しました。また、忙しいあまり良いソフトウエアを開発するための技術に対する興味がなくなっている開発現場の意欲を高めるためのきっかけにならないだろうかという狙いもありました。

オブジェクト指向分析／設計（OOAD）をしようと考察しましたが、現在の我々にはそれだけの時間がなく、段階を踏んで OOAD での開発をするためにもまずは ZIPC で現状のソフトウエアを可視化するのがベストであると判断しました。また、ZIPC は Rose オプションも用意されて要る／Konesa-RealTime なるツールもあり、設計資産（モノ／ヒト共に）を流用出来る拡張性も備えています。

ZIPC 導入は、以下の 3 つの効果が得られると考えています。

(1) エディタ機能による設計書の共通化（タスク関連図／シーケンス図／タイミング図／状態遷移図／状態遷移表）
(2) 分析／設計／実装工程のシームレスな接続
(3) ソースコードとドキュメントの一致

4. ZIPC 評価方法

ZIPC の評価は、すぐに実際の製品に適用するのは無理と判断し、既に終了したプロジェクトのハードウエア／ソフトウエア資産を使い、テストプロジェクトを立ち上げて評価を行いました。我々の

ソフトウエアシステムはリアルタイムOS を導入しており、機能ごとにタスク分割しています。今回はハードウエアに近いタスクはそのまま流用し、それらのコントロールを司る撮影制御タスクとズーム制御タスクに適用しました。

ZIPC の機能であるエディタ／チェッカ／ジェネレータを使い、コードジェネレーションまで行うことにより、ソースコードとドキュメントを一致させ、ソフトウエアを可視化させるのが狙いです。

今回のZIPC 評価ポイントは、

(1) デザインフェーズ：EDT／CHK
(2) シミュレーションフェース：SIM
(3) ジェネレーションフェーズ：GEN
(4) 時間の都合上、ATV／VIP の評価は行わない。

図2 タスク構成図

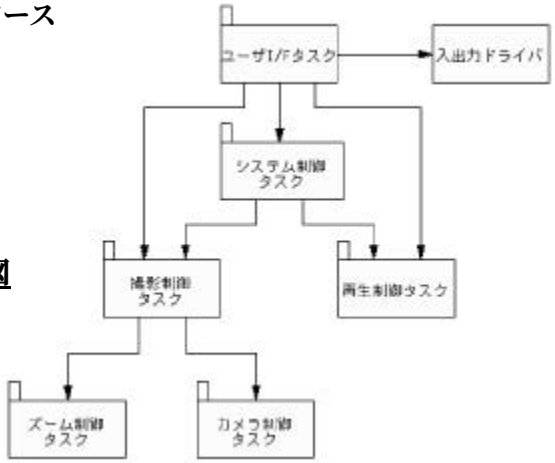

表1 タスク構成

タスク名	内容
撮影制御タスク	ユーザ I/F タスク、システム制御タスクからメッセージを受信し、ズーム／カメラ制御のコントロールを行うタスク。
ズーム制御タスク	光学ズームレンズの制御を行うタスク。撮影制御タスクからのメッセージを受信して光学ズームレンズを動かす。
再生制御タスク	ユーザ I/F タスク、システム制御タスクからメッセージを受信し、スマートメディアの静止画／動画ファイルの再生コントロールを行うタスク。

表2 作成したSTM のサイズ（ZIPC 2000 GEN マトリクステーブル：標準型生成）

タスク名	STM 枚数	イベント数 (注1)	状態数 (注2)	関数数 (注3)	ステップ数
撮影制御タスク	35	84	165	386	7772
ズーム制御タスク	11	22	60	109	2517
再生制御タスク	7	59	37	130	2787

(注 1) GED設計書(*.ged)より算出。
(注 2) タスク名_0zip.h(自動生成コード)より算出。
(注 3) タスク名_0stm.h(自動生成コード)より算出。FNC設計書は含まない。
(注 4) 有限会社マウントシステム社製CCooDoo 使用。

5. ZIPC 適用方法

今回の開発では、同時に「操作仕様とプログラムの分離」も同時に行いました。外部設計を見直し、撮影制御タスクはなるべく撮影に関する部分のみを行える様にタスク結合度を弱め、操作仕様の仕様変更（仕様変更の大部分は操作仕様の変更の割合が多い）に耐えうる構成にしました。

各タスクは以下の順番で適用を行っています。（表3）

表3　各フェーズでのZIPC 適用方法

タスク名	フェーズ1 （3ヶ月）	フェーズ2 （2ヶ月）
撮影制御タスク ズーム制御タスク	従来ソースコードを全て捨て新規設計。階層 STM 化。ZIPC でコードジェネレーションまで行う。	特になし。
再生制御タスク	EDT で STM 設計まで（階層化しない1つの STM。）コードジェネレーションはしない。 従来ソースコードと ZIPC で作成した STM を一致させる。	フェーズ1と同等の機能を維持し階層 STM 化＆コードジェネレーションまで行う。

以下の流れで作業を行いました。

(1) 前プロジェクトのソースコード解析（ドキュメントが無いため、従来手法と変わらず）
(2) 外部インターフェースの決定（基本設計）
(3) 階層STM構成の決定､STM作成(詳細設計）
(4) イベント名称シミュレーションでSTMの動作を確認、MSCトレースで外部インターフェースの動作確認（シミュレーションデバッグ）
(5) ソースコード自動生成
(6) アクションセル、関数設計書の中身の実装
(7) ターゲットでのエミュレーションデバッグ

実際は、(3)～(7) はスパイラル型の開発となり、(4) または (7) で問題が起こった場合、(3) の工程に戻り作業を行いました。リバース機能は使用しないことにし、ドキュメントとソースコードを完全に一致させる様にしました。

6. ZIPC 活用方法

以下、私が設計して感じた ZIPC 活用方法です。

(1) ドキュメント検索機能は有用

状態遷移表ファイルはバイナリデータなので、そのままでは grep 等で検索出来ませんが、ドキュメントツリーにファイルを登録しておけば、全てのファイル内の文字列を検索することが出来ます。この機能により、元々全体を広く見渡しやすい状態遷移表から、文字列を検索することが出来ますので、開発後期になっての仕様変更等に対応しやすくなります。

(2) 状態遷移と処理の分離を心がける

遷移先の記述は､状態遷移表のセル内､もしくは ZIPC システムコールを使えば関数設計書内に記述出来ます。関数内に記述した場合、遷移先が分かりにくくなるという欠点があります。遷移先を全て状態遷移表のセル内で閉じる様に設計することにより、設計者以外が見てもプログラムの動作が分かりやすくなります。

(3) 置換ファイルは有用

セル内には日本語 2 バイトコードを記述することも出来ます。分析／設計の初期段階では、日本語により記述し、後に置換ファイルを使用してその部分をソースコードに置き換えることが出来ます。これにより、分析／設計工程をシームレスに接続することが可能になります。

(4) リバース機能はなるべく使わない

ターゲット上のデバッグで問題点が発生した場合、その場でコードを修正してリバース機能を使いドキュメントとソースコードの一貫性を保つことは可能です。しかし、リバース機能を使うためには、ソースコード上にリバース情報を埋め込む必要があります。この情報を埋め込むと、ソースコード上のインストラクション 1 行毎に置換情報を埋め込みますので、コードが非常に見にくくなります。

7. ZIPC 2000？ それとも ZIPC 2001？

ZIPC2001 にはジェネレータモードとして、ZIPC2000（旧バージョン）GEN と ZIPC2001 GEN の 2 種類のジェネレータを搭載しています。前者は ZIPC 2000 からのユーザのために用意されていると思われますが、自動生成コードが異なります。

ZIPC 2000 ジェネレータは従来と同じくマトリクステーブルは、「標準型」「処理抜粋型」「オフセット型」「ノンテーブル型」の 4 種類、生成ファイルはイベント解析関数毎、イベントアクティビティ毎に生成されるので、STM 数が増えてもファイル構成は変わりません。

ZIPC2001 ジェネレータは STM の記述により、「Basic 型生成」「Full 型生成」の 2 種類となります。生成ファイルは、ソース／ヘッダファイル共に STM 毎に生成されるため、STM が増える度に生成ファイルが増加していきます。

（makefile のメンテナンスが面倒であるという欠点を除き、)アクティビティのインライン生成強化、ZIPC システムコールのサポート拡大など性能向上の面も伺えます。

実際に（表２）で作成した撮影制御タスクの STM の自動生成されたコードをジェネレーションした結果を（図３）に示します。なお、撮影制御タスク／ズーム制御タスク共に全ての STM は以下の条件を満たした「単純な STM」としました。(ZIPC 2000 GEN ではスケジュール駆動生成されない／ZIPC 2001 GEN では Full 型生成されない様な条件)

- E 型標準
- ステート開始／終了／モードアクティビティなし
- 並列状態なし
- 状態階層 STM なし
- ディスパッチアクティビティなし

また、C コード生成設定オプション［STM 設定］は以下の通り：

- マトリクステーブル：標準型
- イベントアクティビティ：インライン生成
- 同一処理共通化：処理関数／遷移関数／処理関数生成チェック全てのチェック ON
- その他は初期設定値を使用

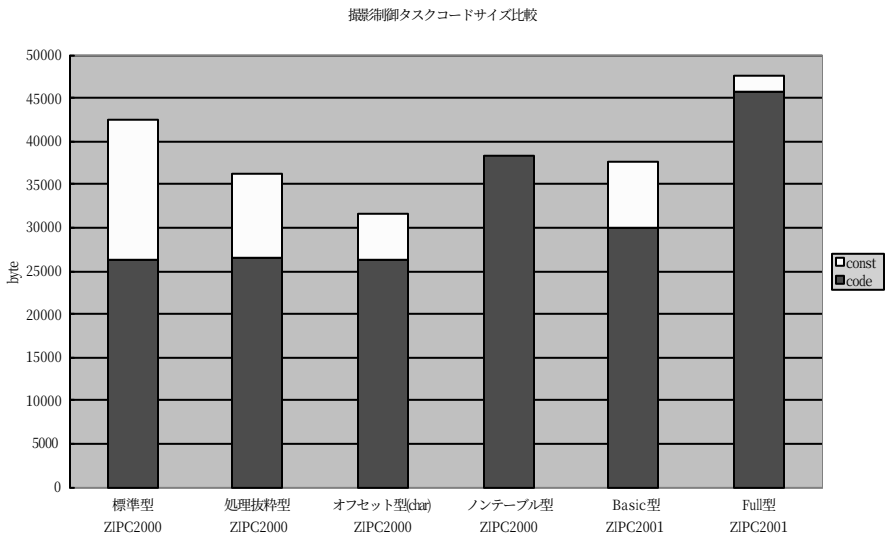

図３　ジェネレータモードによるコードサイズ比較（撮影制御タスク）

ZIPC 2000 GEN オフセット型（char）は一番 ROM 使用効率が良く、処理時間も標準型と変わらず、テーブルサイズも標準型より小さくなります。登録可能な状態数は最大 254 までですが、通常ではあまり問題にならないと考えられます。

コードと STM の一体化を求めるのであれば、ZIPC 2000 標準型がベストですが一番 ROM 使用効率が悪くなります。コードの読みやすさはそのままデバッグ容易性に直結します。デバッグのためにはやはり ZIPC 独自の実装アーキテクチャを理解する必要があります。

ZIPC 2001 GEN Full 型生成は、const テーブルサイズは ZIPC 2000 標準型に比べ約 1/8 に減りますが、コードサイズが約 1.5 倍となってしまい現実的ではありません。ZIPC 2001 GEN Basic 型生成も ZIPC 2000 GEN 標準型と処理抜粋型の中間という結果になりました。ZIPC 2001 よりサポートされた ZIPC システムコールを使うのであれば、Basic 型生成されるよう STM の記述方法を考慮しなければなりません（この点は ZIPC 2000 GEN でもスケジュール駆動になればコードサイズが大きくなると考えられますので同じ条件でしょう。）

また、これを ZIPC 導入前とコードサイズを比較してみます。ZIPC 導入前の撮影制御タスクは、ZIPC 導入後の「撮影制御タスクの機能」＋「ズーム制御タスクの機能」＋「OSD 制御機能」が含まれています。マトリクステーブルは一番 ROM 使用効率の高いオフセット型と比較してみます。（**図 4**）

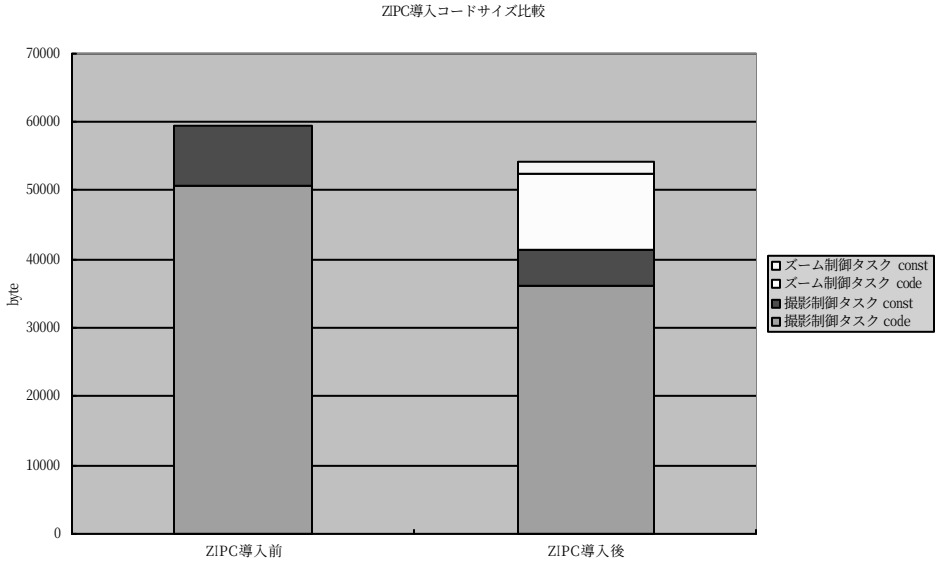

図 4　ZIPC 導入前／導入後コードサイズ比較（撮影制御タスク）

ZIPC 適用後「OSD 制御機能」が削除されているため、そのまま比較することは出来ません。ZIPC 適用前の「OSD 制御機能」モジュールのトータルステップ数 1700Steps、平均バイト数 3～4byte/step とすると、約 7Kbyte を使用するとしますと、コードサイズは ZIPC 適用前とそれほど変わらないと思われます。残念ながら私達の評価では ZIPC を導入すればコードサイズが減るとは言い切れません。

再生制御タスクは、フェーズ 1 とフェーズ 2 で ZIPC 適用方法を変えています。フェーズ 1 ではエディタで STM を記述し、実際のソースコードは STM を元に従来コードを流用して作成しました。このときの STM は階層化されていない 1 枚の STM でした。

フェーズ 2 ではフェーズ 1 の機能を実現し、STM を階層化して ZIPC でのコードジェネレーションまで行いました。フェーズ 1 と 2 の差は、STM の階層化のみです。これらを踏まえコードサイズを比較してみます。ジェネレーションコードはマトリクステーブル：オフセット型と比較します。（図 5）

この結果を見ると ZIPC 適用によりむしろコードサイズが増加することが分かります。階層化によりかえってコードサイズを増加させる要因にもなります。しかし階層化の恩恵は大きく、パソコンの画面では表示しきれない STM をメンテナンスするよりはるかに効率が良いはずです。

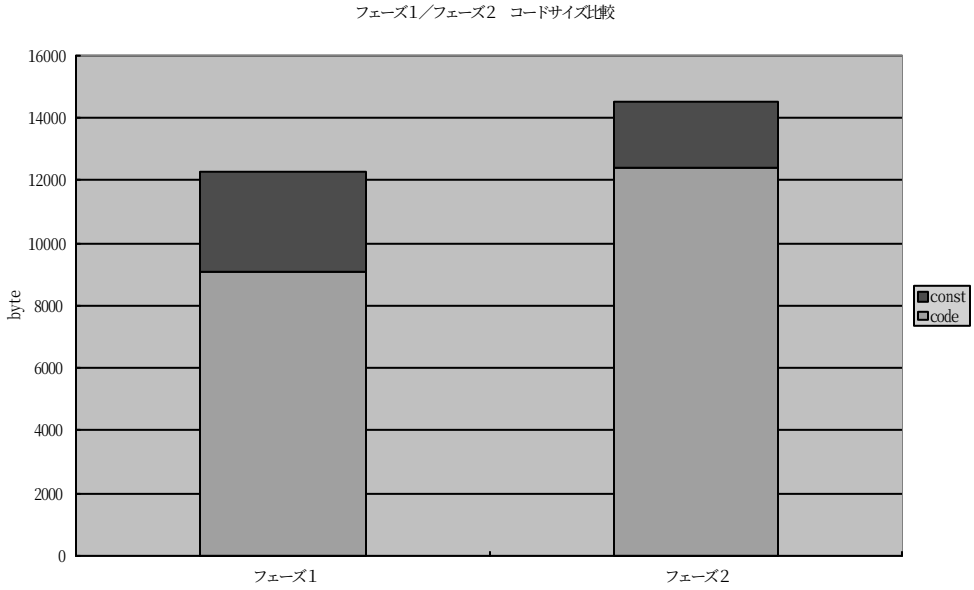

図 5　フェーズ 1／フェーズ 2 コードサイズ比較（再生制御タスク）

8．コードジェネレーションへの長い道程

ZIPC WATCHERS Vol.5 にあるように、ZIPC 2001 は前バージョンに対して多くの要望を取り入れ機能向上したのが伺えます。状態遷移表設計は ZIPC 2001 でサポートされた機能を前提に設計を開始しました。

しかし、自動生成したソースコードが、ターゲットデバイスコンパイラでのコンパイルが通らないという、思いもよらぬ（？）問題に直面しました。この時点で ZIPC2001 ジェネレータに何点かの不具合が発見され、対策を取っている間は ZIPC2000 ジェネレータを使用していました。その後、キャッツのサポートと改修作業で何とか ZIPC2001 ジェネレータでのコードジェネレーションを完了しましたが、この時間的ロスがなければ、より生産効率を上げられたかもしれません。

更なるツールバージョンアップと品質向上に期待したい所です。

9．まとめ

今回の導入評価目標であった項目についてまとめます。

(1) エディタ機能による設計書の共通化

ドキュメントを書くための統一したツールとして使用出来ることにより、各設計者の設計書が共通化出来ます。我々が良く使うシーケンス図は、ATV で使用出来るため、外部設計完成度を高め、設計工数の削減につながると考えられます。

(2) 分析／設計／実装工程のシームレスな接続

分析工程で階層化状態遷移表のイメージを作成し、成果物はそのまま設計工程へとつながり、実装工程は設計した状態遷移表よりコードジェネレーションを行い、ターゲットデバッグへと、全ての工程が前工程の成果物を基につながることが出来ます。

(3) ソースコードとドキュメントの一致

状態遷移表といくつかの ZIPC 設計書がソースコードと一致しているため、ターゲットデバッグの方法も変わりました。何か起こったら、現在の状態／イベント番号より状態遷移表のどこで止まったかが分かります。全体を広く見渡しやすい状態遷移表からほとんど原因をつかむことが出来て、デバッグ効率もあがりました。

ZIPC に全く不満がない訳ではありません。Windows アプリケーションとしてのチューニング不足なのか、細かい部分で不満を感じることはありました。このあたりはもう少しがんばってもらいたいところです。

また今回は見送った ATV／VIP 機能ですが、今後適用範囲が広がれば必ず必要になってくると思われます。これら機能を使えばターゲットがない状態でもコード品質を高めることが可能になると思います。（実際にはハードウエアの完全

なシミュレーションは不可能なので時間に関連する部分は無理ですが。）

最後に今回この様な機会を与えて頂いたキャッツ株式会社様に感謝いたします。