UMLクラス図からのワークロードの見積もりとスケジュール作成

竹村 司

Using UML Class Diagrams to Estimate Workload and to Prepare Development Schedule

Tsukasa Takemura

近年, オブジェクト指向開発ではUML®（Unified Modeling Language）を用いた分析・設計が盛んに行われるようになってきた. しかし, 大規模なシステムを設計した場合にはクラス間の依存関係が複雑になり, クラスの実装やテストの実行順序の決定が困難になる. 本論文では, この問題を解決するため開発した, UMLのクラス図から実装タスクのスケジュールを生成する手法およびツールとその評価について述べる.

This paper describes a method using UML® class dependencies to prepare a schedule for development project. For the development of a large-scale system, a huge number of classes are defined in its model represented in UML. This makes it difficult to analyze the dependencies among the classes and to determine the order of implementation tasks. A tool has been developed to generate automatically development schedules from UML class diagrams and its effectiveness has been evaluated.

Key Words & Phrases：統一モデリング言語, クラス図, プロジェクト管理, ワークロード見積もり, 多変量解析
Unified Modeling Language (UML), Class Diagram, Project Management, Workload Estimation, Multivariate Analysis

1. はじめに

近年のオブジェクト指向技術の普及に伴い, 膨大な数のクラスから構成されるシステムが開発されることが多くなってきた. このようなシステムを開発するプロジェクトでは, クラスの実装（コーディング）やテストの順序を正しく計画しておくことが重要である. クラスの実装順序やテスト順序を誤って計画してしまうと, あるクラスやパッケージの実装時やテスト時に他のクラスやパッケージの実装やテストの不必要な完了待ちが発生し, プロジェクトを計画通りに遂行することができなくなるからである. しかし, 大規模なシステムではクラス間の依存関係が複雑になり, 分析・設計時に後工程であるクラスの実装やテストの順序を決定することが困難になってしまう.

筆者の参加していたプロジェクトでは, 会計システム構築のためのビジネス・コンポーネント群を独自アーキテクチャのフレームワークからEnterprise JavaBeans（EJB）へのポーティングした. このプロジェクトでは33個のコンポーネントに分類された529個のクラスの再設計と実装, テストを行った. このプロジェクトにおけるコンポーネント（UMLではパッケージに該当する）間の依存関係を**図1**に示す. この図からも判るように, UMLパッケージ間には複雑な依存関係が存在する. このため, この程度の規模のプロジェクトでも手作業でクラスの実装順序を決定することは困難となっている.

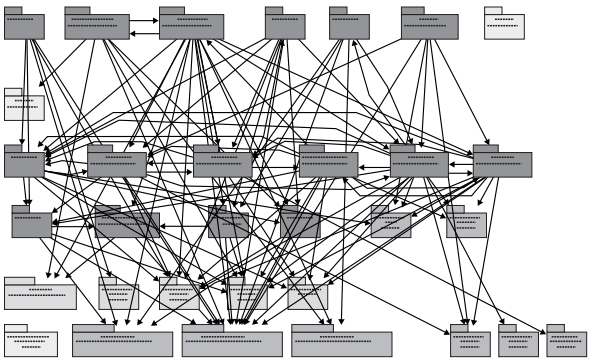

図1. コンポーネント間の依存関係

この問題を解決できない場合, 実装・テスト段階で発生する不必要な待ち時間を計画段階で最少化す

提出日：2004年8月17日

ることができないばかりか, 予想することすら不可能になる. このため, 今後ますます増加すると予想されるUML®を用いたプロジェクトでは, この問題を解決することがプロジェクトの成否に大きく影響する.

このような問題の解決するために, 筆者はRational Rose®を用いてUnified Modeling Language（UML）で記述したクラス図から, Microsoft® Project用の入力データを自動生成するツールを作成した. 本論文では, この手法・ツールおよびその評価について述べる.

2. クラス図からのガントチャートの導出

2.1 クラス間の依存関係とタスク間の依存関係

オブジェクト指向の手法を用いて設計されたクラスを実装する場合, 開発プロジェクトを効率的に進めるためには, クラス間の依存関係を考慮してクラスの実装順序を決定する必要がある. 2つのクラスの間に依存関係が存在する場合, 被依存クラスを依存クラスより先に実装しておく必要があり, プロジェクトを計画・実施するにあたってはクラスの依存関係に応じて実装やテストのタスクをスケジュールする必要がある. 例えば, クラスAを継承してクラスBを実装する場合, クラスAを先に実装してからクラスBを実装するようなスケジュールを作成しておかなければ, 実装時にコーディングが完了したにもかかわらずコンパイルを行うことができないという問題や, 単体テストを行うための前提となるクラスが存在しないという問題が発生する可能性がある. このような問題が発生すると, プロジェクトを計画通りに実施することが不可能になり, 計画自体の見直しを余儀なくされる.

2.2 UMLのクラス間の依存関係

UMLのクラス図に含まれるクラス間の依存関係には**表1**に示すものがある. この中には, 操作の引数として他のクラスが使用される場合のような, クラス図上は明示されないクラス間の依存関係が含まれる.

2.3 タスク間の依存関係の生成

筆者の開発したツールでは, 次の方法によりUMLのクラス図からガントチャートを導出している.

- クラス図からパッケージおよびクラスを抽出し, 各パッケージおよび各クラスに対応するタスクをガントチャート上に生成する.
- クラスがパッケージに含まれる場合（**図2**）, クラスに対応するタスクをパッケージに対応するタスクのサブタスクとする.

  例えば, あるパッケージPackageにクラスJ, K, Lが含まれる場合には**図3**のようなガントチャートが生成される.
- 表1に示したUMLのクラス間の依存関係を抽出し, その依存関係をもとにタスク間の依存関係を生成する. 被依存クラスに対応するタスクを, 依存クラスに対応するタスクの先行タスクとする.

例えば, 図2のようにクラスLがクラスKを所有し, クラスKがクラスJを所有している場合には, **図4**のようなガントチャートが生成される. また, **図5**のようにクラスBとクラスCがクラスAを継承している場合には, **図6**のようなガントチャートが生成される.

ここに示した手法を用いて, すべてのパッケージとすべてのクラス間の依存関係から, すべてのタスク間の依存を導出することができる.

表1. クラス間の依存関係

クラス間の関係	依存クラス	被依存クラス
継承	子クラス	親クラス
実現	実装クラス	インターフェイスクラス
集約・コンポジション	所有クラス	被所有クラス
関連	相互依存	
依存	依存クラス	被依存クラス
属性にクラスを持つ	属性を持つクラス	属性となるクラス
操作の引数にクラスを使用する	操作を持つクラス	引数となるクラス

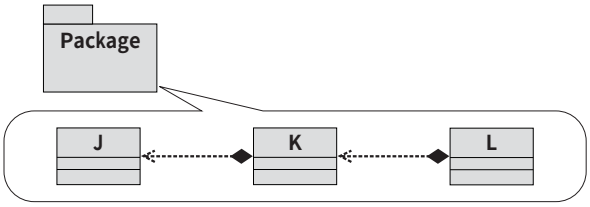

図2. 3つのクラスを包含するパッケージ

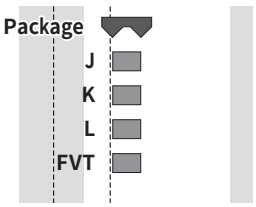

図3. 図2に対応するガントチャート

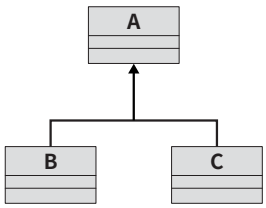

図5. 継承関係のUML表記

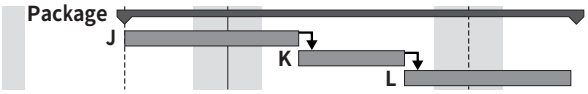

図4. 所有関係のガントチャート

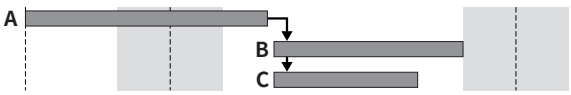

図6. 継承関係のガントチャート

3. タスク間の循環依存

3.1 クラスの循環参照とタスクの循環依存

オブジェクト指向設計において, 2つのクラス間の相互参照や複数クラス間の循環参照が発生することがある. 循環参照自体も望ましいものとはいえないが, 設計上回避できない場合もあり, 実装・テストを行うことが必要となる. しかし, 前述の手法を用いただけではガントチャート上のタスク間にも循環依存関係が発生しスケジュールを作成することが不可能となる. ここではタスク間の循環依存を解決する手法を提案する.

3.2 UMLクラス図と有向グラフ

図7に示すように, UMLのクラス図を, クラスを頂点(vertex)とし依存関係を辺(edge)とする有向グラフととらえることができる. 同様に, タスク間の依存関係も有向グラフととらえることができる.

3.3 循環依存と有向グラフ

タスク間の依存関係を有向グラフとしてとらえた場合, タスク間の循環依存の検出は, 有向グラフにおいて強連結部分グラフを発見することと等価となる. 有向グラフにおいて強連結部分グラフを発見するアルゴリズムはすでに存在するので, タスク間の循環依存を検出するアルゴリズムも実装可能である.

タスク間の循環依存を解決する手段として, クラス間の依存関係に強度という概念を導入する. 被依存クラスには, 実装またはテスト時に必須のクラスと必須とは限らないクラスがある. 例えば, インターフェイスクラスと実装クラスの関係では, 後者をコンパイルするためには前者は必須である. 集約やコンポジションでは被所有クラスが存在しなくても所有クラスを実装することが可能な場合もある. 本論文では, このような依存関係の度合いを依存関係の強度と定義する.

クラス間の依存関係の強度の定義から, 強度の弱いクラス間の依存関係では, 対応するタスク間の依存関係も弱いことは自明である. このため, 循環依存の中で強度の最も弱い依存関係でタスク間の循環依存を切れば, タスクの実行時への影響が少なくなる. これは, 以下のアルゴリズムにより実現できる.

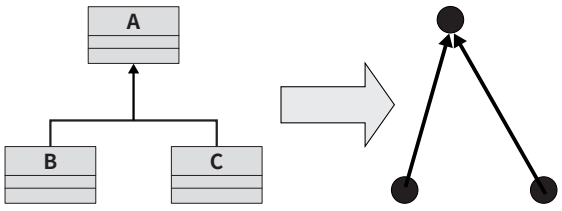

図7. クラス図と有向グラフ

3.3.1 アルゴリズム

(1)すべての強度のタスク間依存関係を有向グラフで表現し, 強連結成分に分解する.
(2)(頂点数2以上の)強連結部分グラフが存在する間, 各強連結部分グラフに対して以下を繰り返す.
(2.1)強連結部分グラフから, 最も強度の弱い依存関係を除去する.
(2.2)残された依存関係だけを用いて, 強連結成分に分解する.
(3)終了

図8はこのアルゴリズムを模式的に表したものである. 図中, 実線の矢印は強度の強い依存関係を表し, 破線の矢印は強度の弱い依存関係を表す. この図では, 破線の矢印までを含めた有向グラフには, 楕円で示された強連結部分グラフ(循環依存関係)が存在するが, 実線の矢印だけの有向グラフには, 強連結部分グラフが存在しない.

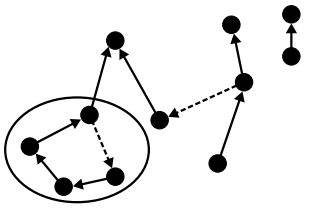

図8. 循環参照と強連結部分グラフ

ここで示した手法を用いることにより, 全タスクの依存関係から循環依存を取り除き, すべてのタスクの実行順序を決定することができる. つまり, 全クラスの開発のためのタスクの実施順序を示すガントチャートを得ることができる. ただし, この段階では各タスクの実施に必要な期間は未定である.

3.4 クラスの複雑度とワークロード

各クラスの複雑度とそのクラスに対応するタスクのワークロードには相関関係があると考えられる. つまり, クラスが複雑であるほどそのクラスの実装にかかるワークロードも増加すると考えられる. UMLのクラス図から取得可能な, クラスの複雑度に関する指標には, 以下のものがある.

- クラス内の属性の数
- クラス内の操作の数
- クラス内の操作の引数の総数
- クラスの持つ関連の数
- クラスの持つ集約・コンポジション関係の数
- クラスの持つ依存関係の数
- クラスの持つ継承関係の数
- クラスの持つ実現関係の数

これらの値と実際のワークロードの関連については「考察」の節で詳述する.

筆者が開発したツールでは, UMLのクラス図から上記の指標をクラスごとに抽出し, その値に基づいてワークロードを計算し, ガントチャートのタスクに割り当てている.

これにより, タスクの実行順序だけではなく, 各タスクの予想実施期間を反映したガントチャートを得ることができる.

3.5 人的リソースの割り当て

以上の方法で, 本ツールを用いてUMLのクラス図からガントチャートを生成することができる. このガントチャートはタスク間の依存関係と各タスクのワークロードを含んでおり, 実行可能なスケジュールとなる. しかし, タスクの依存関係だけからは同時に並行して実施可能と考えられるタスクでも, 実プロジェクトでは人的リソースが有限であるために並行して実施できない場合がある. このため, プロジェクト管理ツールで各タスクに人的リソースを割り当てて平準化することにより実プロジェクトに即したスケジュールを得ることになる.

4. 考察

4.1 タスクのスケジュール

本論文で示した手法の効果を示すためには, 本手法を用いたプロジェクトと用いていないプロジェクトを比較する必要がある. しかし, 本論文は実プロジェクトを対象としているため, このようなプロジェクト間の比較を行うことはできない. このため, 本論文ではここで述べた手法の効果を傍証する.

前述のように, クラスの実装順序やテスト順序を誤って計画してしまうと, 実装時やテスト時に被依存クラスの完了待ちが発生しプロジェクトを計画通りに遂行することができなくなる. 本論文で述べたプロジェクトでは, このようなクラス間の依存関係に起因する待ち時間は発生しなかった. スケジュール作成時には予期しなかった待ち時間がプロジェクト実施時に発生した場合, 開発期間が予定を上回ってしまう. これに対して, 本プロジェクトでは, ほとんどのコンポーネント開発期間の実績値が予測値を下回っている. このことは, 本プロジェクトではクラス間の依存関係に起因する待ち時間は発生しなかったことを裏付けている.

4.2 クラスの複雑度とワークロード

ここでは, 筆者の参加するプロジェクトの前期が終了した時点でのデータ(18個のコンポーネントに含まれる282個のクラス)に基づいて, クラスの複雑度の指標からワークロードを求める手法の有用性を検証する.

4.2.1 相関関係

本ツールでは, 前述の「クラスの複雑度とワークロードには相関関係がある」という仮説のもとにワークロードを算出している. この仮説を検証するために, 前述のデータとコーディング期間についての相関関係を求めた(**表2**を参照). なお, 本論文の対象として使用したクラス図では, 単純な関連を使用していないので, 表2では3.4節で述べた複雑度の指標のうち「クラスの持つ関連の数」を表示していない. この結果,「クラスの持つ実現関係の数」を除いて,「コーディング期間」との単相関係数の値が0.6以上で比較的強い相関関係があることを確認することができた. これにより, 本仮説は正しいと考えることができる.

4.2.2 重回帰分析

複雑度とワークロードの関係を検証するために,「コーディング期間」を目的変数とし, 複雑度の各データを説明変数として, 多変量分析の一種である重回帰分析を行った. 重回帰分析を行うにあたっては,「コーディング期間」との相関関係が比較的弱い「実現関係の数」を説明変数から除いた. また, 説明変数間に強い相関関係がある場合には多重共線性(multi-collinearity)が発生し重回帰分析の精度が低下するため[1], 説明変数から「操作の引数の総数」,「集約・コンポジション関係の数」,「依存関係の数」を除いた.

その結果, 有意0.00153で, 重相関係数0.777, 補正後の重決定係数0.547という値を持つ分析精度の比較的よい次の重回帰式を得た.

$$D_C = 0.899 \times N_A + 0.638 \times N_I + 11.4 \qquad \text{式1}$$

ここで, D_Cはコーディング期間(日), N_Aは属性の数, N_Iは継承関係の数である.

ここで得た重回帰式自体は, プロジェクトの性質, 対象システムの性質, プロジェクトメンバーの経験・スキルに依存するものと考えられるが, 特定の組織やプロジェクトについての分析精度の高い重回帰式を決定することは可能であり, 本手法によって得られる重回帰式をプロジェクトの計画に利用することは有意義だと考えられる.

4.3 予測値と実績値

前節で示したように, プロジェクト前半の実績に基づいてクラスの複雑度とコーディング期間の関係を表す重回帰式を得た. この重回帰式をプロジェクト後半(コンポーネント数15個, クラス数に241個)に対して適用して得た予測値と実績値を**表3**に示す. 本プロジェクトでは, 各コンポーネントに一人の担当者が割り当てられた. 表中, 斜体は後期から開発に参加したメンバーが担当したコンポーネントを示す. また, 一人の担当者が同一期間に並行して複数のコンポーネントを開発したケースが発生したが, このようなケース

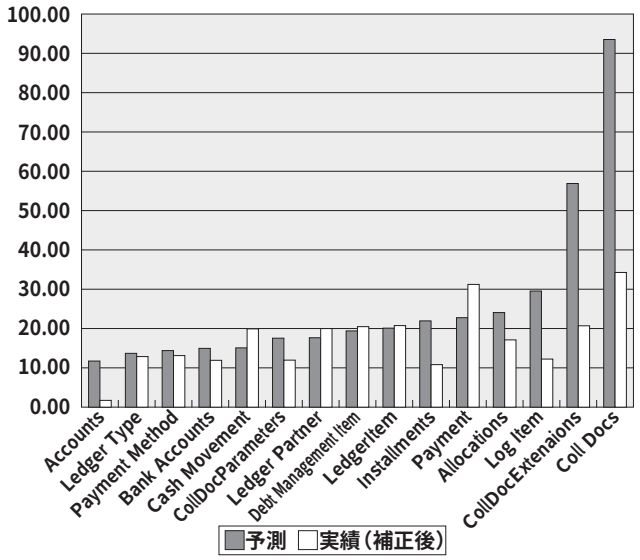

図9. コーディング期間の予測値と実績値（補正後）

では単純な開発期間ではワークロードを正しく評価できない. このため, 開発が重複した期間は複数のコンポーネントに対して同等のワークロードをかけたものとみなしてコーディング期間の補正を行った.

図9に予測値と補正後の実績値の関係をグラフとして示す. グラフから, 一部の例外を除いて予測値と実績値がかなり近い値であることが判る. 予測値が大きく外れたコンポーネントについて調査したところ, それぞれに特徴があることが判明した. これらの特徴は以下の通りである.

- Account

このコンポーネントは抽象クラスのみを含み実装クラスを含まない. このためコーディング作業がほとんど発生しなかった.

- LogItem

このコンポーネントはLedgerItemと内容が似ており, しかも同一のメンバーが担当した. このため, コーディング作業が予測より早まった.

- CollDocExtensions

クラス数が多いにもかかわらず同じパターンの構造しか現れない. このため, コーディング作業が予測より早まった.

- CollDocs

多数のstatic finalで宣言された属性が単一のクラスに含まれていた（全76属性のうち40属性がstatic final）. これらはクラス図では属性として扱われるが, コーディング時には変数ではなく定数扱いとなる. このため, 予測値が大きくなりすぎた.

以上の特徴からこれらのコンポーネントでは実績値が予測値を大きく下回ったものと推測できる.

4.4 予測値の有用性

前述の例外的なコンポーネントを除いて, 予測値と実績値を検討する. **表4**は予測値と実測値の平均を比較したものである. この表からも判るように, 誤差が4.7%であり, コーディングにかかるワークロードの見積もりとしてはかなり精度の高いものと考えることができる.

5. 関連研究

5.1 複雑度とメトリクス

オブジェクト指向設計に対する複雑度のメトリクスとしてはChidamber他[2][3]のメトリクスが有名である. 彼らは, オブジェクト指向設計に対する6種類のメトリクスを提案しそれらを分析的に評価した後に, 2つの実プロジェクトから収集したデータを用いてこれらのメトリクスの実用性を示している.

UMLクラス図の複雑度から種々のメトリクスの値を予測する研究には以下のものがある. Genero他[4]は, UMLクラス図の構造的な複雑度からクラス図の保守性を予測するモデルを構築しており, 神谷他[5]は, 複雑度メトリクスを用いてエラー修正時間を予測できることを示している. また, Marchesi[8]は, オブジェクト指向分析時のUMLのユースケース図およびクラス図の対するメトリクスを提案し, それらメトリクスを用いて実装に必要なコストと時間をプロジェクトの初期の段階で見積もることを提唱している.

これらの研究に対して, 本論文ではUMLクラス図のみから得られる複雑度をもとに重回帰分析を行うことで各クラスの開発に必要なワークロードを高い精度で予測することの妥当性を示した.

前述の研究以外では, Siau[6]は, UMLで用いる図と他のオブジェクト指向開発方法論で用いている図を種々の複雑度のメトリクスを用いて比較し, UMLの図は複雑度が高いことを示している.

5.2 UMLと有向グラフ

本論文と同様にCiupke[7]はUMLクラス図内のクラス間の関係を有向グラフととらえた研究を行っている. この研究では, 有向グラフに対するクエリを分析のために用いることができることを示している.

Jeron他[12]は, UMLのクラス間, 操作間, およびクラス－操作間の関係からテスト依存グラフ（Test Dependency Graph）と呼ばれるモデルを生成し, このモデルから統合テストの実行順序を決定し, テスト・スタブの数を最小化する方法を提案している.

これに対し, 本論文では, 実際のプロジェクトの経験からUMLのパッケージを開発担当者に割り当てるという前提に基づき, クラスの開発順序のみならず, パッケージをまたがる依存関係を用いてパッケージ

表2. 相関関係

	属性	操作	引数の総和	集約・コンポジション	依存	継承	実現	コーディング期間
属性	1							
操作	0.598547	1						
引数の総和	0.667044	0.896715	1					
集約・コンポジション	0.601247	0.837959	0.912753	1				
依存	0.637511	0.769872	0.856769	0.851723	1			
継承	0.705451	0.879702	0.944688	0.927198	0.933477	1		
実現	0.472481	0.490363	0.597292	0.603843	0.719077	0.679918	1	
コーディング期間	0.720572	0.659555	0.626434	0.619255	0.713298	0.714545	0.443597	1

表3. コーディング期間の予測値と実績値（単位：日）

コンポーネント	予測	実績	実績（補正後）
Accounts	12.77	2	2.00
LedgerType	14.44	14	14.00
PaymentMethod	15.12	14	14.00
BankAccounts	15.43	13	13.00
CashMovement	15.80	20	20.00
CollDocParameters	18.08	13	13.00
LedgerPartner	18.38	20	20.00
DebtMgmtItem	18.38	38	21.00
LedgerItem	20.28	40	21.50
Installments	22.70	24	12.00
Payment	23.55	32	32.00
Allocations	24.54	30	18.00
LogItem	28.38	32	13.50
CollDocExtensions	57.16	21	21.00
CollDocs	84.00	35	35.00

の開発スケジュールを生成する手法を示している.

5.3 UMLとプロジェクト管理

UMLとプロジェクト管理の関連に関しては以下のような研究がある.

Lin他[9]は, ソフトウェア開発プロジェクトのプロセスの構造をオブジェクト指向の継承関係と全体―部分関係を用いて表現することによって, プロジェクト管理を容易にすることを提唱している. Noack[10]もまたソフトウェア開発プロセスの構造をUMLで表現することによって, ソフトウェア開発プロセスのガイドラインを示すことを提唱している. 本論文で述べた研究とは異なり, これらの研究はソフトウェア開発プロセス自体をUMLで表現しようというものである.

これら以外の, ソフトウェア開発プロセスとスケジューリングに関する研究としては, Padgerg[11]の研究がある. この研究ではソフトウェア開発プロセスのスケジュール法を評価するための離散シミュレーション・モデルを提唱している.

表4. 合計と平均の比較

	予測値	実績値	誤差
平均	18.89	18.05	4.7%

6. 結論

6.1 手法およびツールの有効性

「考察」で述べたように, 本手法およびツールはUMLを用いて分析・設計したシステムを実装する際には有効である. 今後UMLを使用したプロジェクト開発はますます増加すると予想されるので, 本手法およびツールの必要性もさらに高くなってくると考えられる. 特に, クラス間の依存関係が複雑な大規模システムを複数の担当者で開発するプロジェクトにおいて, 不必要な待ち時間を発生させないようなスケジュールを作成することでき, プロジェクト期間の予想外の延長を防ぐことができる. このため, 大規模システムのプロジェクト管理にとっての有用性が高いと考えられる.

6.2 今後の課題

6.2.1 ツールの改良

現時点では, クラスの複雑度からワークロードを求める際に重回帰式を手動で与えている. ツールをより使いやすくするためには, プロジェクトの実績に基づいて重回帰式を修正しスケジュールにフィードバックする手法を自動化する必要がある.

6.2.2 手法の強化

本手法では, 以下のような事項が考慮されていない.

- クラスが抽象クラスであるのか具象クラスであるのか
- 継承元のクラスの複雑度
- 親クラスから継承した操作を実装する必要があるか否か

これらの事項を考慮することによって本手法の精度がさらに高まると考えられる. さらに, クラス図だけではなくシーケンス図やコラボレーション図からもクラス間の依存関係を抽出することや, クラスの複雑度

の抽出にステートチャート図を用いることによって本手法を強化することが考えられる.

本手法はクラスの設計情報を用いて, クラス設計以降の工程のスケジュールを決定している. しかし, より上流の工程において, プロジェクト全体のスケジュールやワークロードを求める必要がある場合がある. このため, ユースケース図や分析段階のクラス図から大まかなスケジュールやワークロードを求める手法が必要である.

謝辞

本論文は, 情報処理学会のオブジェクト指向シンポジウム2003で優秀論文賞を受賞した論文[13]をベースに加筆・修正したものである. 本論文を本誌に転載することを許可していただいた情報処理学会に感謝します.

参考文献

[1] 管 民郎：Excelで学ぶ多変量解析入門, オーム社, 2001
[2] Chidamber, S.R. and Kemerer, C.F. : A metrics suite for object-oriented design, IEEE Transaction on Software Engineering, Vol.20, No.6, pp.476-493, 1994
[3] Chidamber, S.R., Darcy, D.P., Kemerer, C.F. : Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis, IEEE Transaction on Software Engineering, Vol. 24, No.8, pp.629-639, 1998
[4] Genero, M., Piattini, M., Jimenez, L. : Empirical validation of class diagram complexity metrics, Proceedings of XXI Internatinal Conference of the Chilean Computer Science Society, pp.95-104, 2001
[5] 神谷 年洋, 楠本 真二, 井上 克郎, 毛利 幸雄：複雑度メトリクスを用いたエラー予測の一手法―アプリケーションフレームワークを用いた開発への適用, 情報処理学会論文誌, Vol.42, No.6, pp.1601-1609, 2002
[6] Siau, K. and Cao, Q. : Unified Modeling Language (UML) a complexity analysis, J. Database Manage. (USA) Vol.12, No.1, pp.26-34, 2001
[7] Ciupke, O. : Analysis of object oriented program using graphs, ECOOP '97 Workshops Proceedings of Objecct Oriented Technology, pp.270-271, 1998
[8] Marchesi, M. : OOA metrics for the Unified Modeling Language, Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering (Cat. No.98EX143), pp.67-73, 1998
[9] Lin, J. and Yeh, C. : An object-oriented formal model for software project management, Proceedings of Sixth Asia Pacific Software Engineering Conference, pp.552-559, 1999
[10] Noack, J. : Extending the software development process with a toolkit of UML-centred techniques, Proceedings of International Conference on Software Methods and Tools, pp.87-96, 2000
[11] Padberg, F. : Using process simulation to compare scheduling strategies for software projects, Proceedings of Ninth Asia-Pacific Software Engineering Conference, pp.581-590, 2002
[12] Jeron, T. et al. : Efficient strategies for integration and regression testing of OO systems, Proceedings of 10th International Symposium on Software Reliability Engineering, pp.260-269, 1999
[13] 竹村 司：UMLクラス図からのワークロードの見積もりとスケジュール作成, オブジェクト指向最前線2003, 情報処理学会OO2003シンポジウム 予稿集, pp.153-160, 2003

日本アイ・ビー・エム株式会社
ソフトウェア開発研究所
副主管開発技術担当部員

竹村 司 Tsukasa Takemura

[プロフィール]
1986年京都大学大学院工学研究科修士課程修了(情報工学専攻).同年日本アイ・ビー・エム入社.以来,種々のインダストリー向けアプリケーションソフトウェアやソフトウェアコンポーネントの開発に従事.現在,ソフトウェア開発研究所に所属.オブジェクト指向シンポジウム2003優秀賞受賞.共訳「MDA モデル駆動アーキテクチャ」.ソフトウェアのモデル化と開発過程に興味を持つ.