ソフトウェア構造を見れば品質がわかる！
構造メトリクスとバグの関係

最新研究事例紹介と追試結果：
構造メトリクスとソフトウェア品質との相関関係

テクマトリックス株式会社
システムエンジニアリング事業部
綾井環 ・ 小向順 ・ 福永一寛

2014年10月15日

- Phase 1

ソフトウェア定量的測定のススメ

ソフトウェアの定量的測定とは？

- ソースコードの規模、複雑さ、構造、保守性などの特性について数値や尺度等に指標化すること。
- 尺度や測定方式をまとめてメトリクスと呼ぶ。(JIS X 0141)

なぜ必要か？

- 全体把握することが不可能なほど巨大・複雑化したソフトウェアを、ある特性を用いて把握し、ソフトウェアに関わるプロセスや品質をコントロールするため。

たとえば：

- 定量的管理の有無でプロジェクトの成否は左右される。

定量的管理	成功率
あり	45.6%
なし	24.3%

参考：矢口ら、日経コンピュータ2008年12月1日号「第2回プロジェクト実態調査」

定量化と開発プロジェクト成否の関連について

ソフトウェア開発に寄与するSW・構造・開発者・進捗等のメトリクスは完全には特定できていない

- メトリクスを統計的に解析することによって、品質に関わる要素を特定している組織もあるが、一般化はできていない。

不具合予測に関するメトリクスについての研究は活発に続いている [畑ら2012]

- ソフトウェア工学における重要なテーマの一つ。
- 不具合の潜在が疑われるモジュールを、早期に予測することが求められている。

- Phase 2

最新研究事例紹介：
システムデザインと構造複雑度のコストについて

アーキテクチャとコストについての研究と研究論文著者紹介

Sturtevant, Daniel Joseph,
“System design and the cost of architectural complexity”,

Thesis (Ph. D.) Massachusetts Institute of Technology, Engineering Systems Division,2013

- **MacCormack らの研究グループ：**
  - **ソフトウェア製品アーキテクチャ※に関する研究を行う**
    ※ アーキテクチャ：設計要素間の依存性(結合状態)をシステムの特徴として扱おうというアイディア [Simon1962]
  - **アーキテクチャの測定、アーキテクチャ変化の定量化について有益な発表を行っている数少ない研究グループ(MacCormack, Rusnack, Baldwin ら)** **[立木2010]**
  - **Sturtevantも所属**
    - **2008 - 2013　MIT, ESD: PhD in Engineering Systems**
    - **2013年当時 The MathWorks(MATLAB開発元) システムアーキテクト**
  - **Alan D. MacCormack:ソフトウェア科学、企業研究等Awards & Honors 多数**

構造複雑度が開発会社に課すコストは？

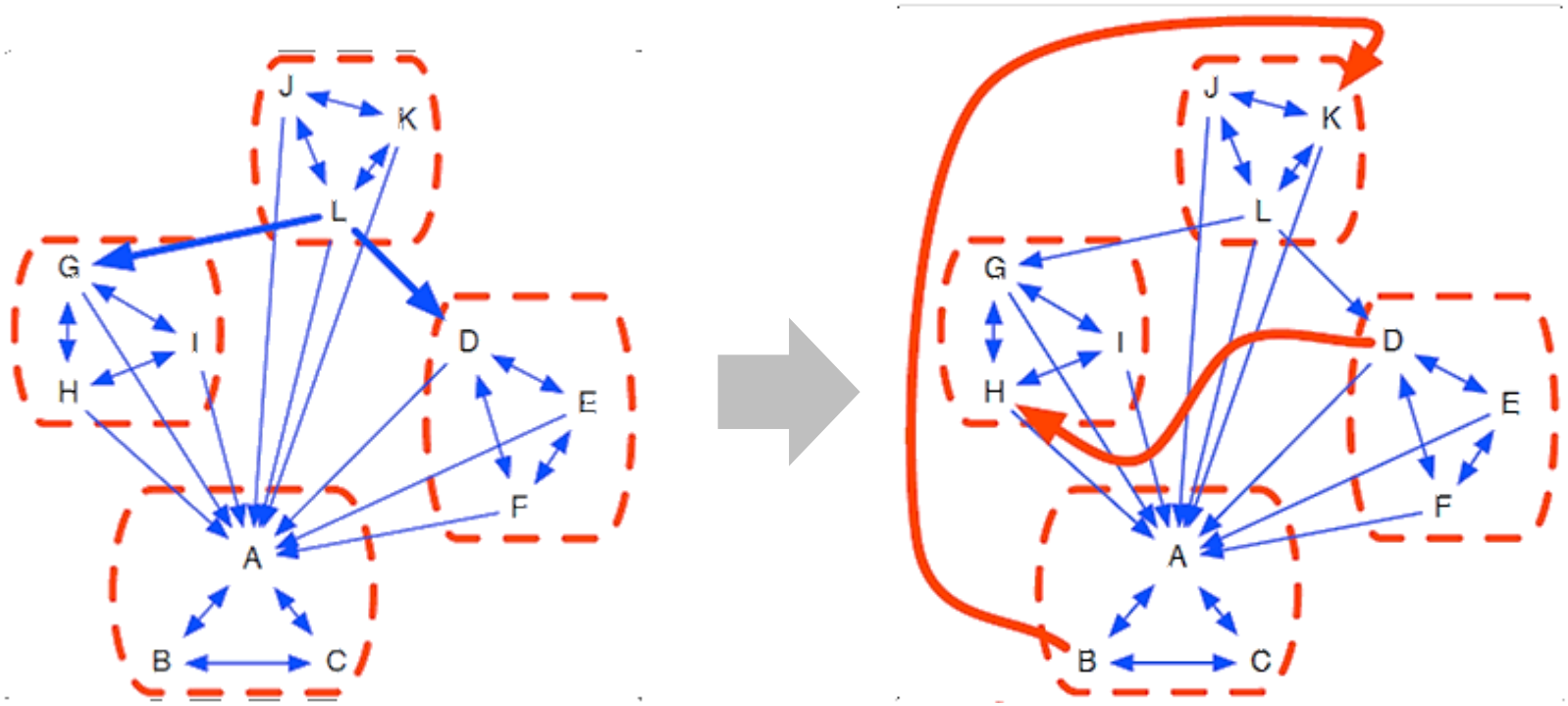
コストを発生させるものとして、以下三つを検討する

1. 複雑度はバグ密度を増加させるか？
2. 複雑度は生産性を減損させるか？
3. 複雑度は離職率を増加させるか？

Sturtevant 2013 要約

ソフトウェアの構造的な複雑度が爆発的に増大する例

設計上のルールが間違って破られたとき、
全ての要素は依存関係を持つ



**たった2か所、誤った依存が混入しただけで、
すべての要素に依存関係が発生している**

引用：Sturtevant 2013 Webinar “Slide 10”

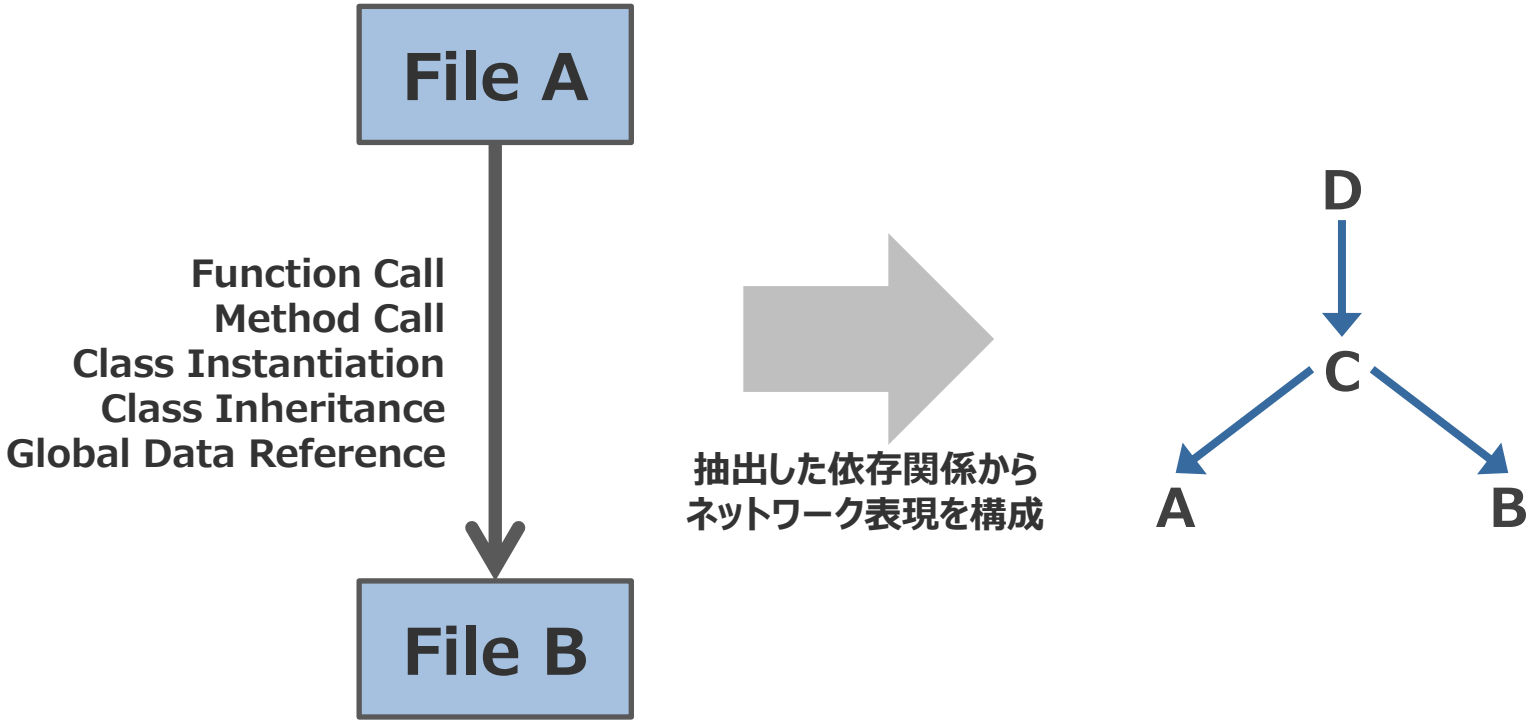
分析対象

分析アプローチ

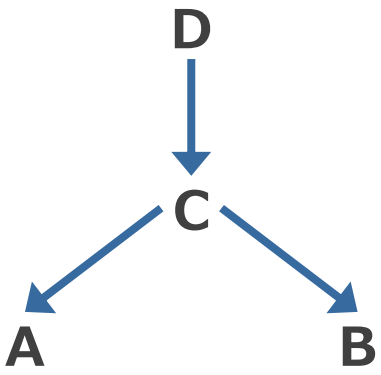
- 1. 米国にて成功している開発会社: “Iron Bridge 社” (仮名)
  - 商用大規模ソフトウェア開発
- 2. **大規模商用ソフトウェア**1つ、8リリースを追う
  - ファイル間の依存関係を抽出する
  - 複雑度を計測する
  - 各開発期間ごとにバグ発生数、発生箇所、開発アクティビティを計測する
  - コストと消耗に関する重要な情報を抽出する
- 3. **バグと複雑度に関して回帰分析**を用いて検証
- 4.**シミュレーション**を行い、分析で得られた要素のImpactサイズを確定

Sturtevant 2013 要約

解析手法 - Step1. ファイル間の依存を抽出



解析手法 - Step2. 直接依存関係から間接依存関係の導出



直接依存
Direct Dependencies

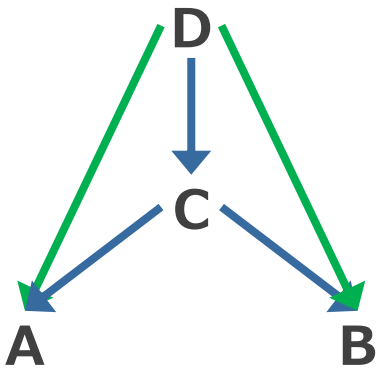
	A	B	C	D
A				
B				
C	●	●		
D			●	

Traditional Network View

ネットワーク表現

Design Structure Matrix

DSM表現



間接依存
Indirect Dependencies

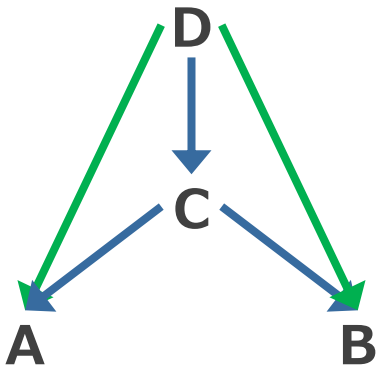
	A	B	C	D
A				
B				
C	●	●		
D	●	●	●	

引用：Sturtevant 2013 Webinar “Slide 23”

解析手法 - Step3. Visibility Score を間接依存グラフから取得

Visibility Fan-In/Out (VFI/VFO)

間接的な依存要素数 も含めた Fan-In/Out



	A	B	C	D
A				
B				
C	●	●		
D	●	●	●	

	Visibility Fan-In	Visibility Fan-Out
File A	3	1
File B	3	1
File C	2	3
File D	1	4

引用：Sturtevant 2013 Webinar “Slide 25”

解析手法 - Step4. Visibility Score を基に構造複雑度を分類

Architectural Complexity (構造複雑度) の分類

Peripheral・・・複雑度 低

Peripheralに属するファイルは、周辺に影響を及ぼさず、また、残りの多くのファイルから影響を受けない。

Utility・・・複雑度 やや低

システムの大半はこれにUtilityに属するファイルに依存する。しかし、このUtilityに属するファイルは他のファイルに対して依存がすくない。依存は内包され、安定している。

Control・・・複雑度 やや高

Controlに属するファイルは機能性を操作するため、他のファイルへの依存が高い。

Core・・・複雑度 高

Coreに属するファイルは、他のファイルに対して直接あるいは間接的に共同依存する。また、大きな循環を含むシステムに不可欠な形成する。これらの領域は小さなコンポーネントに分解するのが難しく、また、大きくなりすぎる場合は、管理不能になる可能性がある。

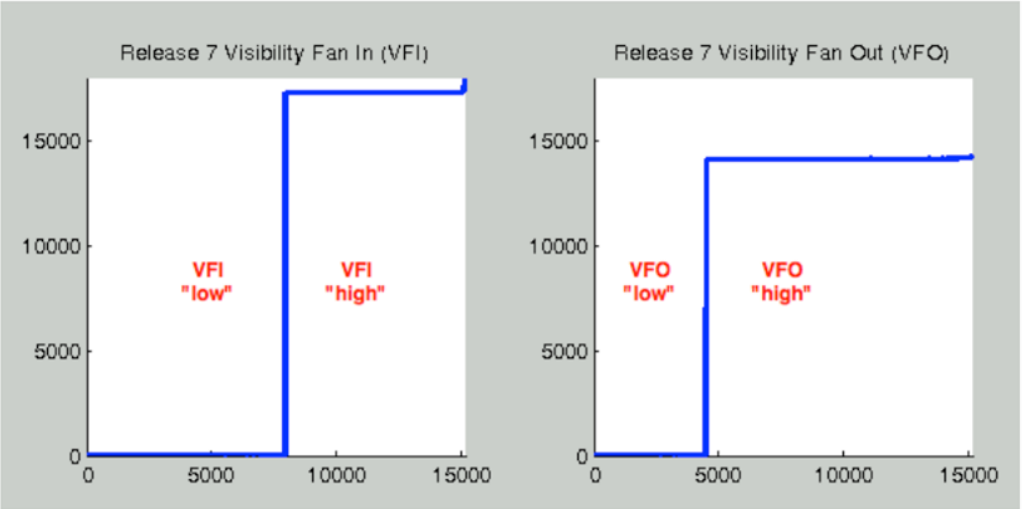
Sturtevant 2013 要約

解析手法 - Step4. Visibility Score を基に構造複雑度を分類

VFI･VFO High/Low の判定：

全ファイル中で最大VFI･VFOの1/2を閾値とする

個々のファイルの最大VFI･VFOそれぞれ閾値以上だったらHigh、下回ったらLowに分類

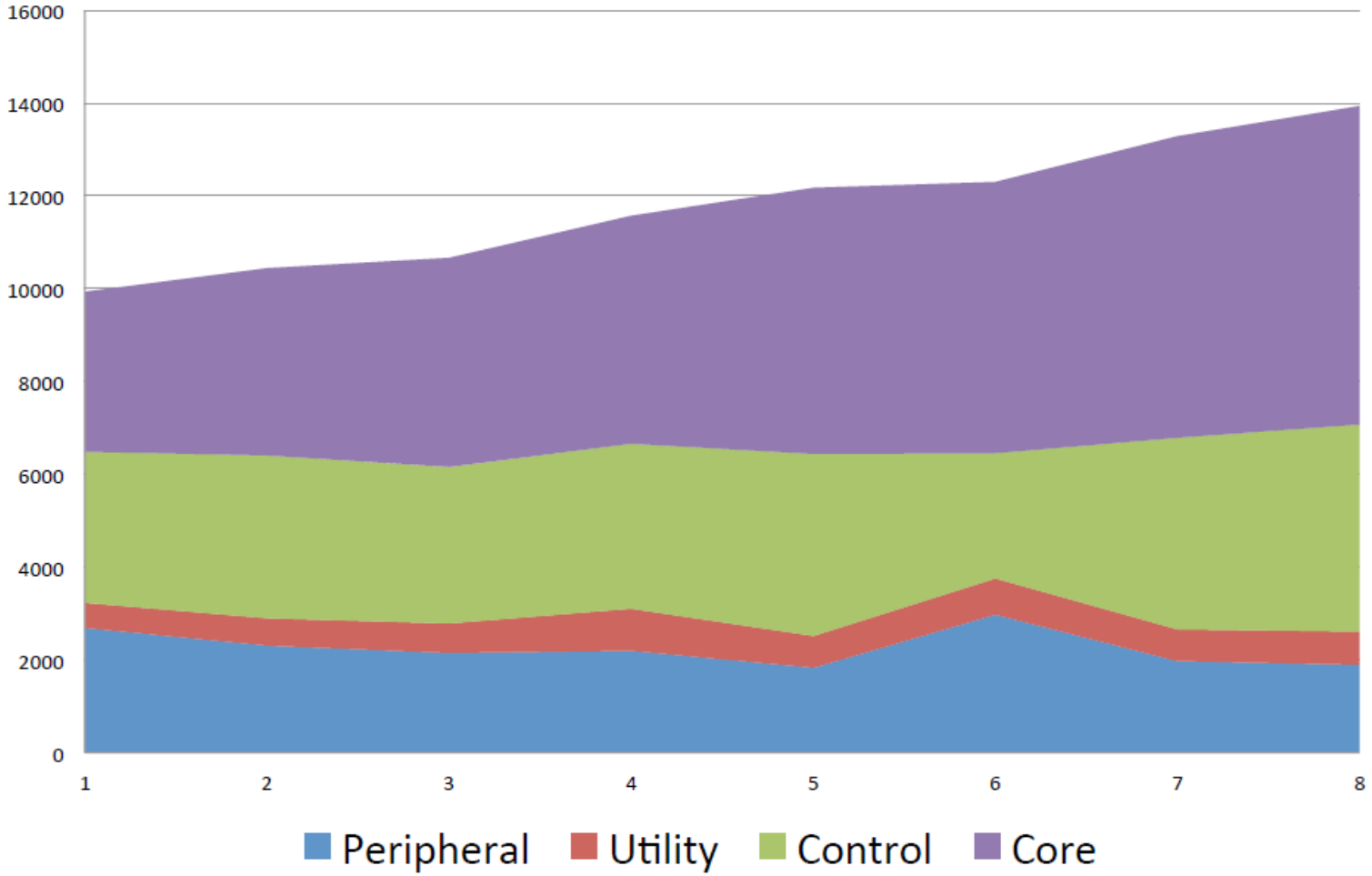


VFO	Low	Low	High	High
VFI	Low	High	Low	High
Architectural Complexity	Peripheral	Utility	Control	Core

引用：Sturtevant 2013 Webinar “Slide 22”

Step4. 各リリースごとの構造複雑度の分類結果

構造複雑度がバージョンごとに増加している



引用：Sturtevant 2013 Webinar “Slide 30”

Step5. 複雑度の品質の分析(回帰分析)

- 命題「複雑度がバグ密度を増大させる」が支持された
- 構造複雑度が高い順でバグ密度は高くなる (有意水準0.1%)
  係数の大きさ Core > Control > Utility > Peripheral

Predicting LOC changed in a file to fix bugs. (Negative binomial model)

Parameter	Model 1: controls	Model 2: cyclomatic complexity	Model 3: architectural complexity	Model 4: combined
LOC in file	0.00156486***	0.0011712***	0.00143183***	0.00104115***
Non-bug lines change	0.00372536***	0.00353601***	0.00355368***	0.00335322***
File age	-0.10050305***	-0.11730352***	-0.1026859***	-0.11853279***
Cyclomatic: mid		0.774729***		0.70392074***
Cyclomatic: high		0.93363115***		0.95513134***
Cyclomatic: very high		0.91923347***		0.96444595***
Architectural: utility			0.2018549*	0.35797922***
Architectural: control			0.94111466***	0.84721344***
Architectural: core			1.14823521***	1.14683088***
Residual Deviance	30370	30418	30428	30475
Degrees of Freedom	94353	94350	94350	94347
AIC	227861	227512	227403	227079
Theta	0.030212	0.030692	0.030836	0.031295
Std-err	0.000285	0.00029	0.000291	0.000295
2 x log-lik	-227837.302	-227482.025	-227373.406	-227042.861

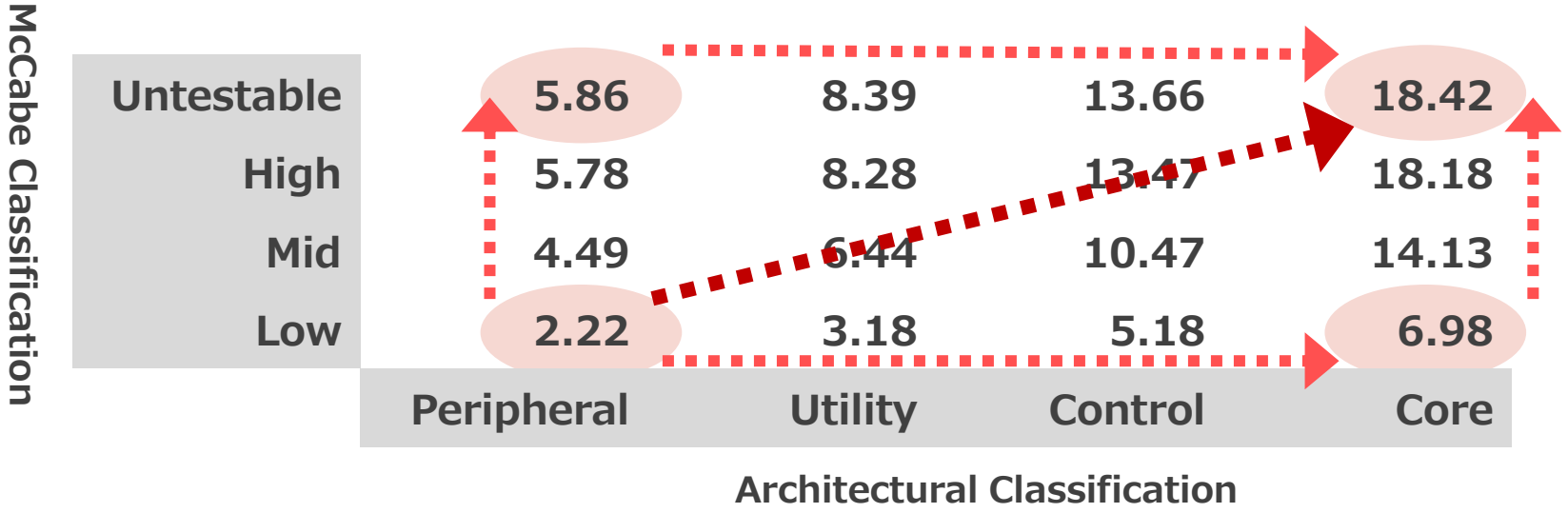
N = 94364 files observations (from 8 releases)
Dummy variables for each of 8 releases omitted.
*Significance codes: .<0.1, *<0.05, **<0.01, ***<0.001*

引用：Sturtevant 2013 Webinar “Slide 37”

分析結果を元にしたバグ予測シミュレーション結果

バグ修正に要する修正行数 予測シミュレーション結果

McCabe : x2.6
Architectural : x3.1
Combined : x8.3



対象のソフトウェア典型的なソースコードファイルを仮定
サイズ：550LOC, ファイル年齢：4.198

引用：Sturtevant 2013 Webinar “Slide 39”

事例検証：オープンソースソフトウェアを用いた追試検証

- 検証その１：複雑度-バグ密度の関連
- 検証その２：複雑度-潜在的コードエラーの関連

対象ソフトウェア：

Apache Ant™ (ビルドツールソフトウェア)

Apache Software Foundation, Open source project

開発言語：Java　対象バージョン：1.7, 1.8･･･2008年から5年間

検証使用ツール	ツール名称	
変更履歴	Bugzilla	Mozilla Foundation
リポジトリ	Subversion	Apache Software Foundation
構造的複雑度	Lattix	Lattix, Inc.
手続き的複雑度	Understand	SciTools, Inc.
潜在的コードエラー	Jtest	Parasoft

検証1：複雑度-バグ密度の関連

Ant Ver.1.7,Ver.1.8のいずれのバージョンも、複雑度の増加により、バグ密度が高くなっている傾向

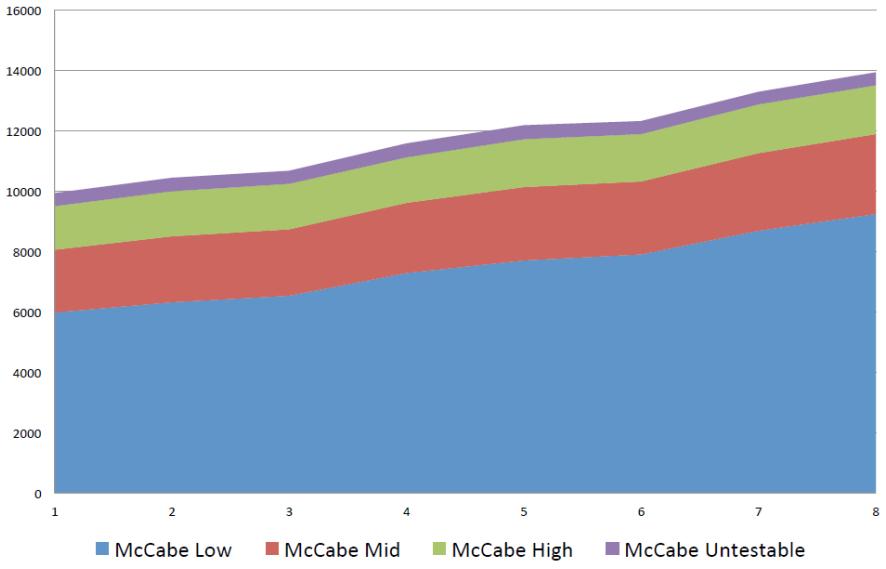
Ver.1.7

	Peripheral	Utility	Control	Core
Untestable	–	–	–	–
High	–	–	36%	67%
Mid	13%	–	17%	38%
Low	4%	8%	4%	8%

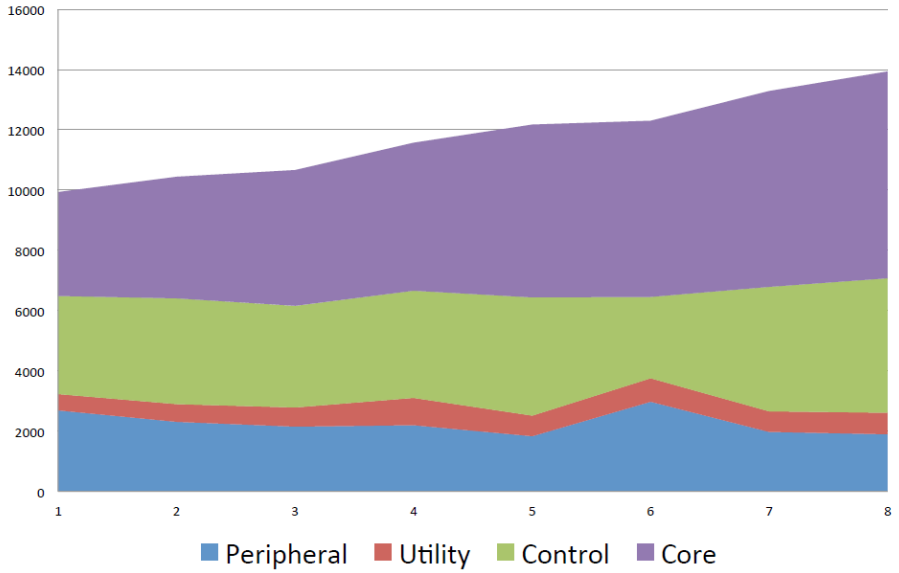
Ver.1.8

	Peripheral	Utility	Control	Core
Untestable	–	–	–	–
High	–	–	26%	20%
Mid	13%	–	22%	13%
Low	10%	13%	12%	11%

検証2：複雑度-潜在的コードエラーの関連



McCabe複雑度分類



構造複雑度分類

- バージョンアップにより、McCabe複雑度よりも、構造複雑度の増加の方が顕著
- 関数/メソッド内の複雑さ(＝McCabe複雑度)よりも、コンポーネント間の複雑さ(＝構造複雑度)の方が、開発者の理解を難しくしており、バグの発生に繋がりやすくなっていると予想される

検証2：複雑度-潜在的コードエラーの関連

Ant Ver.1.7, Ver.1.8 いずれのバージョンも、複雑度の増加により、潜在的コードエラーの密度が高くなっている傾向

Ver.1.7

	Peripheral	Utility	Control	Core
Untestable	–	–	–	–
High	–	–	5.64	7.00
Mid	1.25	–	2.16	5.76
Low	0.50	0.12	0.64	0.90

Ver.1.8

	Peripheral	Utility	Control	Core
Untestable	–	–	–	–
High	–	–	8.35	12.20
Mid	1.25	–	3.00	5.65
Low	0.72	0.38	0.64	1.20

事例検証：オープンソースソフトウェアを用いた検証結果

- **検証その１：複雑度-バグ密度の関連**

  検証対象Antについても、

  「構造複雑度が増すとバグ密度が増加する」

  という Sturtevant の結論と類似の傾向が確認できた

- **検証その２：複雑度-潜在的コードエラーの関連**

  「構造複雑度が増すと、潜在的コードエラーも増加している」ことが確認できた

ソフトウェア定量化による予測についての最新研究事例まとめ

構造複雑度が高い箇所は低い箇所にくらべて バグ密度が 3.1 倍もある

複雑度の高い箇所を担当して開発者は、低い箇所を担当している開発者に比べて生産性が 50% も落ちる。

複雑度の高い箇所を担当している開発者の離職率は、低い箇所を担当している開発者に比べて 10倍 も高くなる。

Acknowledge :

- 生産性や離職率については、検証事例がさらに必要。
- バグ密度に対するネットワークベースメトリクスの影響を扱った研究は過去にもあるが、商用に開発された成熟したソフトウェアシステムのバグに対するネットワークベースの複雑度メトリクスの影響を、複数のリリースにわたって調査したものは本研究が初めて。
- 成功した一企業内に対してだけ計測を実施したため、本研究の成果が使用可能な場は、複雑性が許容範囲内にコントロールされた状況における複雑性のコストについてだけである。
- 複雑性のコストにフォーカスしたので、複雑性は生まれながらに悪ではない。複雑性は価値を追加する。マネージャの合理的意思決定(利益-Cost)についても検討が必要。

- Phase 3

ソフトウェア開発プロセスにおける構造複雑度の活用提案

研究事例ふりかえり

Sturtevantが明らかにしたこと

- 構造的な複雑さが、手続き的な複雑さと同程度にソフトウェアの欠陥に影響する。
- 既知ソフトウェア工学的手法によってコントロールされた品質の高いソフトウェア※に対しても、構造複雑度を監視することで、ソフトウェア欠陥を更に予防することが可能と考えられる。

※Iron Bridge Software社 [Sturtevant2013]の大規模商用ソフトは、構造的な複雑さによる残留バグがあった。

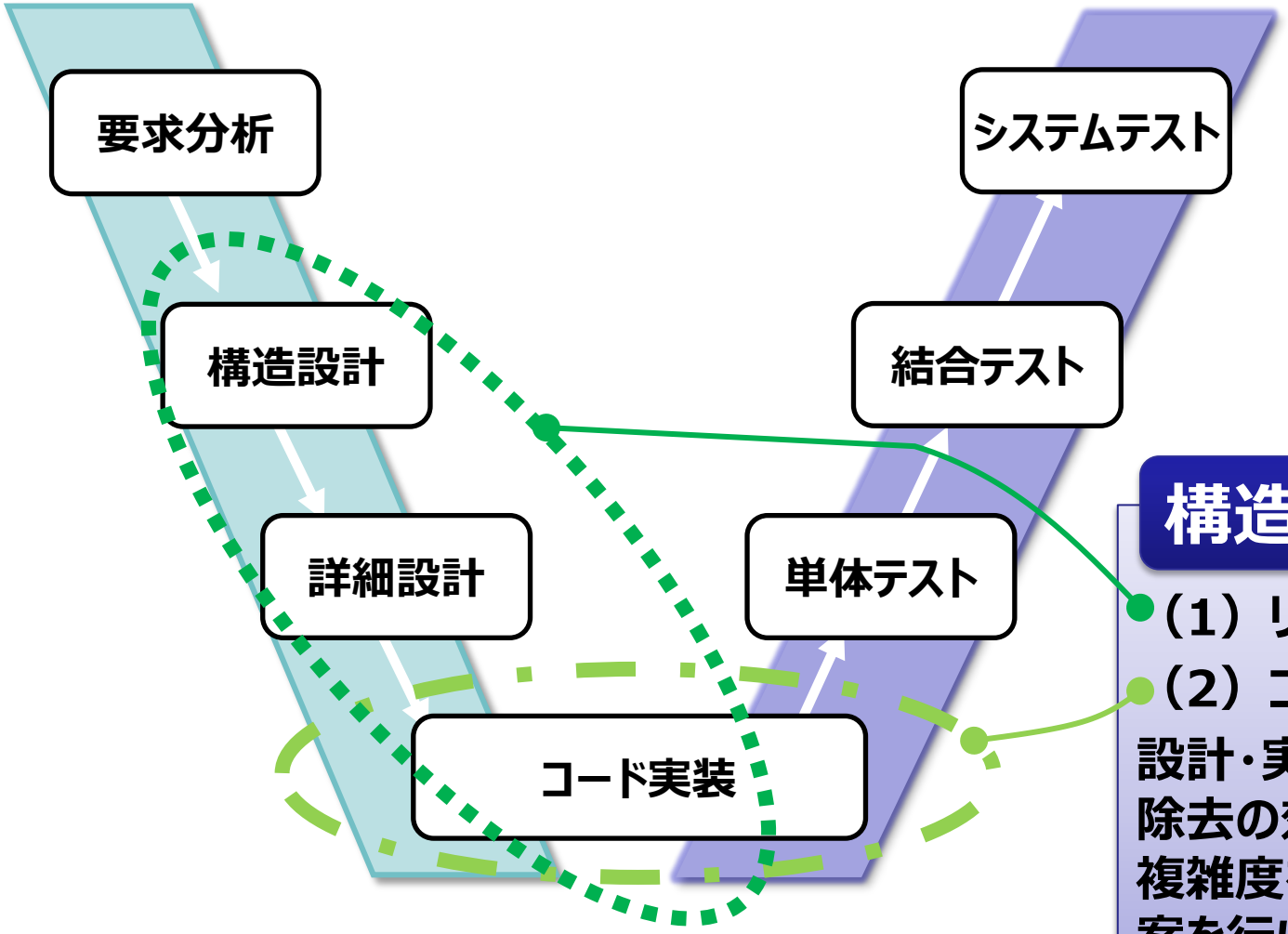
追試結果から考えられること

- あるソフトウェアプロジェクトに対する構造複雑度によるバグ予測の適用可否については、追試検証で示したようにSturtevantのバグ予測と同様の傾向になることが当該プロジェクトで確認されたならば、構造複雑度によるバグ予測が適用可能で、欠陥予防に役立つ。

さらに検証が必要なこと

- Sturtevantが示した構造複雑度によるバグ予測は品質が高く管理されたソフトウェアについては適用可能と考えられるが、品質が低く管理されていないソフトウェアにも適用可能かどうかは不明。ソフトウェアプロジェクトごとに検証する必要あり。

ソフトウェア開発プロセスにおける構造複雑度の応用



V字型ソフトウェア開発プロセス例

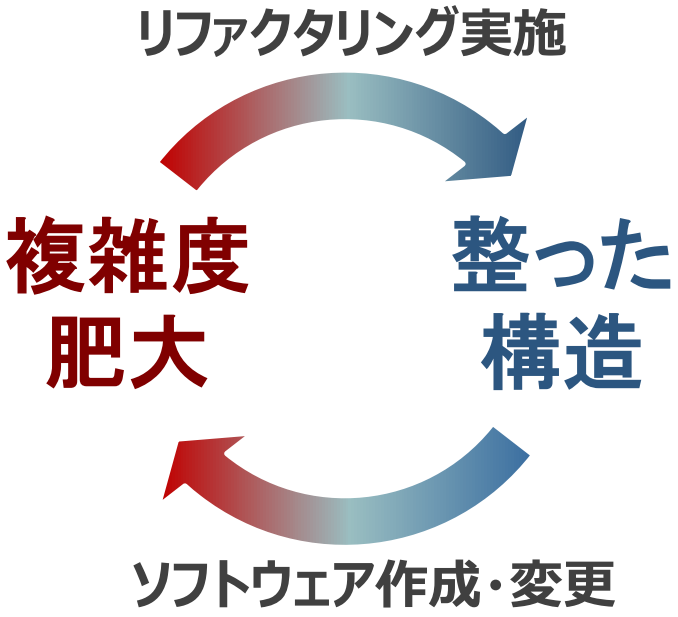
構造複雑度の応用

（1）リファクタリング

（2）コードレビュー効率化

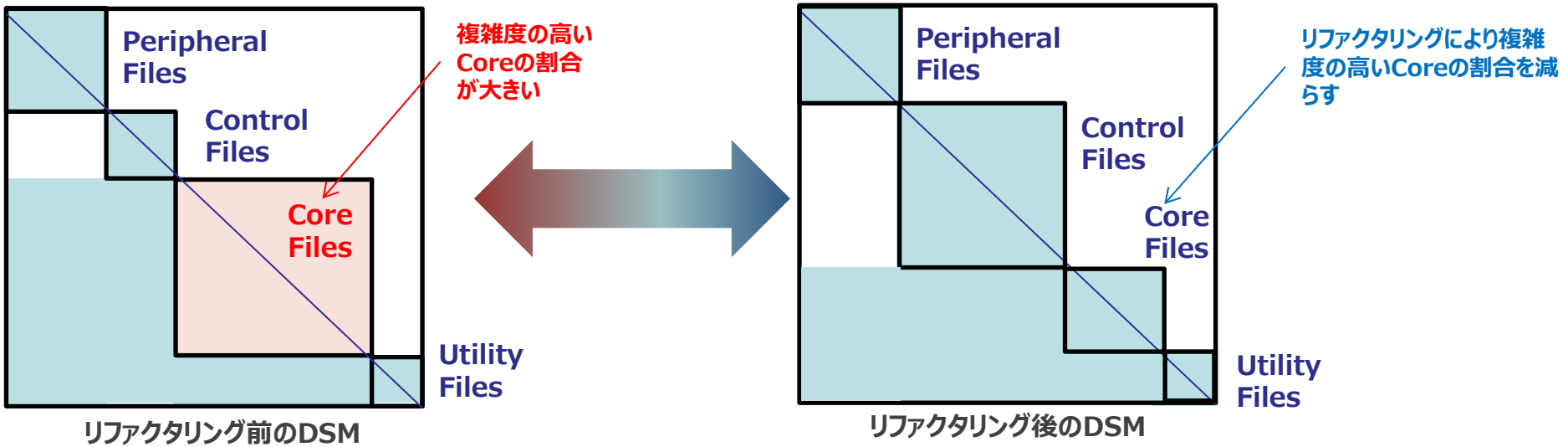
設計・実装プロセスにおける欠陥除去の効率化を目的として、構造複雑度を用いた上記2施策の提案を行います。

構造複雑度を用いたソフトウェア開発プロセス改善の施策提案　その１



提案その１：リファクタリング

一般に用いられているソフトウェアリファクタリングのルールに、構造複雑度による監視ルールを追加します。ソフトウェア開発に進むにつれ、構造的な複雑さを起因とする欠陥が発生しやすい状況になるのを回避します。

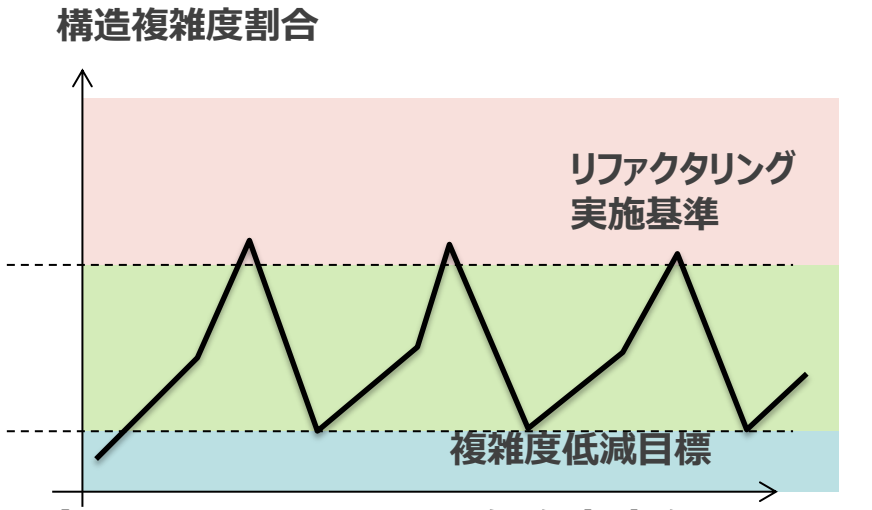


構造複雑度を用いたソフトウェア開発プロセス改善の施策提案　その１

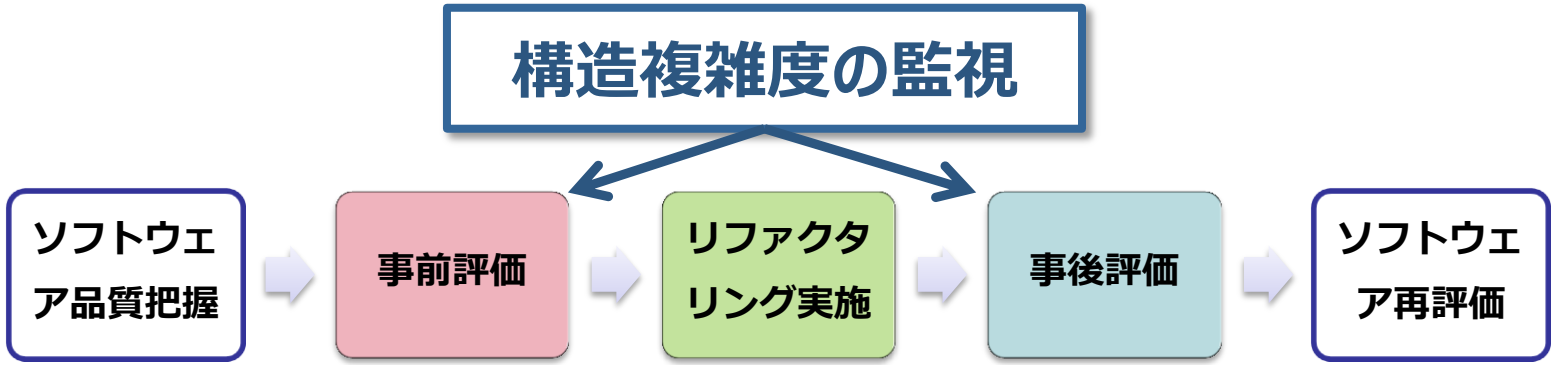
【構造複雑度に注目したリファクタリング手法】

直接的なコンポーネント間依存関係だけでなく、間接的な依存関係にも注目し、In/Out両方の依存を多数持つコンポーネントを減らしていくため、以下を実施していきます。

- ✓ 多数のIn/Outの依存関係を持つコンポーネントがある場合には、適切な責任分担を検討し、複数のコンポーネントに分割します。
- ✓ 構造的な循環依存が発生しているコンポーネントを見つけ、設計で意図した階層構造に従うようコンポーネント間の依存関係を修正します。



例）ソフトウェアの構造複雑度変化とリファクタリングタイミング

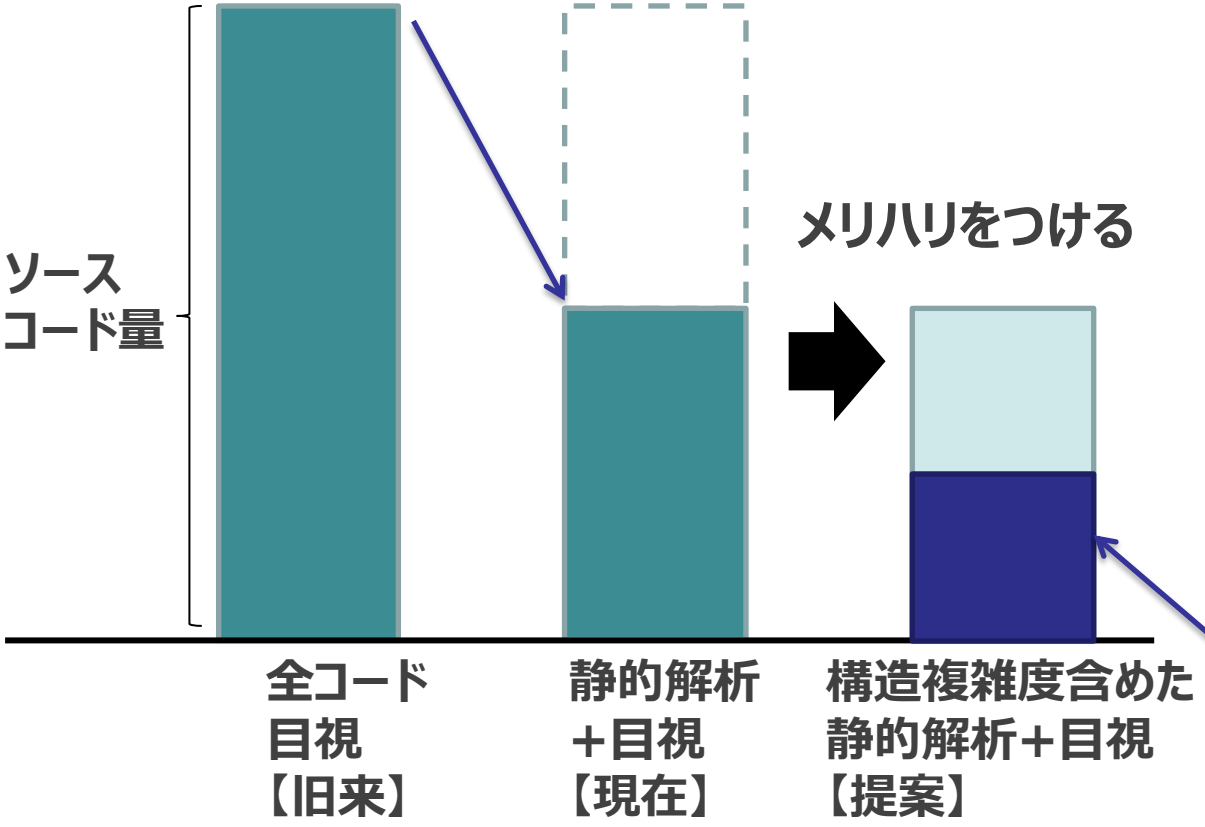


リファクタリングライフサイクル

構造複雑度を用いたソフトウェア開発プロセス改善の施策提案　その２

提案その２：レビュー効率化

目視によるコードチェック量を減らし、開発時に不足しがちなレビューアーの負担を軽減する。



例)人手によるソースコードレビュー量の変化

- ソースコードレビュー前の静的解析時に構造複雑度メトリクスも監視し、構造的に複雑な箇所をリスクの高いコードと特定する。
- リスクの低いコードは開発者自身＋静的解析でチェックし、リスクの高いコードは静的解析に加えて人手(開発者以外の目視)によるチェックを行う。

危険な箇所の特定。危険な箇所を対象に重点的にレビュー実施し、残留バグを取り除く。

References

- Sturtevant, Daniel Joseph, “System design and the cost of architectural complexity”, Thesis (Ph. D.) Massachusetts Institute of Technology, Engineering Systems Division,2013
- Sturtevant, Daniel Joseph, “Technical Debt in Large Systems: Understanding the Cost of Software Complexity”, MIT SDM Systems Thinking Webinar Series, May 6,2013 URL: http://sdm.mit.edu/news/news_articles/webinar_050613/sturtevant-webinar-technical-debt.html
- MacCormack, Alan. et al. "The Architecture of Complex Systems: Do Core-Periphery Structures Dominate? ." Academy of Management Best Paper Proceedings, 2010
- Baldwin, Carliss Y., “Hidden Structure: Using Network Methods to Map Product Architecture”, Harvard Business School Finance Working Paper No. 13-093, 2013
- Herbert A. Simon “The Architecture of Complexity”, Proceedings of the American Philosophical Society, Vol.106, No.6, Dec. 1962
- 畑英明ら, “不具合予測に関するメトリクスについての研究論文の系統的レビュー”, コンピュータソフトウェア. 29 (1), pp.106-117, Feb.2012
- 矢口竜太郎ら,“成功率は31.1% 第2回 プロジェクト実態調査（対象800社）”, 日経コンピュータ2008年12月1日号, pp.36-53
- JIS X 0141:2009 システム及びソフトウェア技術－測定プロセス，財団法人 日本規格協会, 2009
- 立本博文, “複雑システムの設計進化：ソフトウェアのアーキテクチャ変化の測定”, 東京大学ものづくり経営研究センターディスカッションペーパー No.333, 2010

ご清聴ありがとうございました

テクマトリックス株式会社
システムエンジニアリング事業部

TEL: 03-5792-8606
FAX: 03-5792-8706
Mail: lattix-info@techmatrix.co.jp
URL: www.techmatrix.co.jp/quality/lattix/