千葉工業大学　社会システム科学部

プロジェクトマネジメント学科

平成25年度　卒業論文

テストを基準にした

ソフトウェア開発プロセスの調査

|  |  |
| --- | --- |
| 指導教員印 | 学科受付印 |
|  |  |

ソフトウェア開発管理コース

矢吹研究室

1042060　清水 竜吾／Ryugo SHIMIZU

**目次**

[**第1章** **序論** 2](#_Toc378632468)

[**1.1.** **研究背景** 2](#_Toc378632469)

[**1.2.** **研究目的** 3](#_Toc378632470)

[**1.3.** **プロジェクトマネジメントとの関連性** 3](#_Toc378632471)

[**第2章** **ソフトウェア開発について** 4](#_Toc378632472)

[**2.1.** **ソフトウェア開発とは** 4](#_Toc378632473)

[**2.2.** **ソフトウェア開発プロセスについて** 4](#_Toc378632474)

[**2.2.1.** **要求（要件）定義工程とは** 4](#_Toc378632475)

[**2.2.2.** **設計工程とは** 4](#_Toc378632476)

[**2.2.3.** **実装（コーティング）工程とは** 4](#_Toc378632477)

[**2.2.4.** **テスト工程とは** 5](#_Toc378632478)

[**2.2.4.1.** **主なテスト技法** 5](#_Toc378632479)

[**2.2.5.** **導入（運用）工程とは** 6](#_Toc378632480)

[**2.2.6.** **上流・下流工程とは** 6](#_Toc378632481)

[**2.3.** **開発プロセスの種類** 7](#_Toc378632482)

[**2.4.** **開発プロセスの比較** 9](#_Toc378632483)

[**第3章** **Gitについて** 10](#_Toc378632484)

[**3.1.** **バージョン管理システム** 10](#_Toc378632485)

[**3.2.** **用語説明** 12](#_Toc378632486)

[**3.3.** **Gitの簡単な使い方** 16](#_Toc378632487)

[**第4章** **GitHubについて** 19](#_Toc378632488)

[**4.1.** **バージョン管理システム** 20](#_Toc378632489)

[**4.2.** **GitHub APIについて** 21](#_Toc378632490)

[**4.2.1.** **APIとは** 21](#_Toc378632491)

[**4.2.2.** **GitHub APIとは** 22](#_Toc378632492)

[**4.2.2.1.** **GitHub APIリスト** 23](#_Toc378632493)

[**4.3.** **GitHub APIを実際に使ってみる** 31](#_Toc378632494)

[**4.3.1.** **cURLとは** 31](#_Toc378632495)

[**第5章** **本論** 34](#_Toc378632496)

[**5.1.** **研究方法** 34](#_Toc378632497)

[**5.2.** **研究手順** 34](#_Toc378632498)

[**5.3.** **研究対象** 35](#_Toc378632499)

[**5.4.** **研究ツール** 35](#_Toc378632500)

[**5.4.1.** **データフローチャート** 35](#_Toc378632501)

[**5.4.2.** **プログラム概要図** 36](#_Toc378632502)

[**5.4.3.** **実際のコード** 37](#_Toc378632503)

[**5.4.3.1.** lineCounter.shのコード説明 37](#_Toc378632504)

[5.4.3.2. lineCountScriptCreator.pyのコード説明 44](#_Toc378632505)

[**第6章** **考察** 47](#_Toc378632506)

[**6.1.** **研究結果** 47](#_Toc378632507)

[**6.1.1.** **メインコードよりテストコードの方が多い傾向** 47](#_Toc378632508)

[**6.1.2.** **メインコードとテストコードがともに成長する傾向** 49](#_Toc378632509)

[**6.1.3.** **テストコードが殆ど書かれていない傾向** 51](#_Toc378632510)

[**6.1.4.** **テストコードが初期状態から殆ど変化がない傾向** 53](#_Toc378632511)

[**6.1.5.** **その他** 54](#_Toc378632512)

[**6.2.** **考察** 56](#_Toc378632513)

[**6.3.** **今後の課題** 56](#_Toc378632514)

[**参考文献** 57](#_Toc378632515)

**図目次**

[図 1‑1.GitHub登録者・リポジトリ数の推移 1](#_Toc378517876)

[図 2‑1.ウォーターフォール・モデル図 6](#_Toc378517877)

[図 2‑2.アジャイルソフトウェア開発図 7](#_Toc378517878)

[図 2‑3.開発プロセスの比較図 8](#_Toc378517879)

[**図 3‑1.集中型バージョン管理イメージ図** 9](#_Toc378517880)

[図 3‑2.分散型バージョン管理図イメージ図 10](#_Toc378517881)

[図 4‑1.2012年1月1日時点での言語別コミット数 19](#_Toc378517882)

[図 4‑2.APIイメージ図 20](#_Toc378517883)

[図 4‑3.GitHub API使用例 21](#_Toc378517884)

[図 5‑1.データフローチャート図 35](#_Toc378517885)

[図 5‑2.本研究のプログラム概要図 36](#_Toc378517886)

**表目次**

[表 4‑1. GitHubで開発が進められているソフトウェア 18](#_Toc378517887)

[表 4‑2. GitHub APIリスト 22](#_Toc378517888)

[表 5‑1. 本研究対象リスト 34](#_Toc378517889)

# **序論**

本章では，本研究の背景・目的・プロジェクトマネジメントとの関連性を記す．

* 1. **研究背景**

これまでのソフトウェア開発の現場では，主にウォーターフォール型開発手法が採用されていた．それに代わってアジャイル型開発手法が普及してきている．ウォーターフォール型開発手法は仕様を最初にすべて決めてしまうため，スケジュール管理は容易だが，開発途中の仕様変更は難しい．その一方で，アジャイル型開発手法は簡単なウォーターフォール型開発手法をサイクルして開発を行うため，開発途中の仕様変更が容易である．

アジャイル型開発手法では，テスト駆動開発手法（Test Driven Development）がよく採用される．これは，先にプログラムに必要な機能について初めにテストを考案し，そのテストが動作する必要最低限な実装を行い，コードを洗練させ，短い工程で計画・開発を繰り返し行う手法である．

実際の製品として販売・配布されている多くのソフトウェアはテストを含めて，開発プロセスに関する情報は一般には知ることができなかったが，近年では開発プロセスに関する情報を一般でも見ることができるオープンソースソフトウェア（OSS）の開発が盛んである．OSS開発には，OSSホスティングサービスを利用しての開発されることが多い．ホスティングサービスとはソースコードやドキュメントのバージョン管理するシステムやバグ・テスト情報を記録できるシステムなどの複合システムである．

最もよく利用されているホスティングサービスの一つがGitHubである．実際，GitHubの**図 1 1.GitHub登録者・リポジトリ数の推移**をみるとリポジトリ数は，サービスを開始した2008年には38,423件だったのに対して，2012年には4,614,306件と大幅に増加した．ユーザ数も同期間に41,157人から2,763,437人へと大幅に増加した．

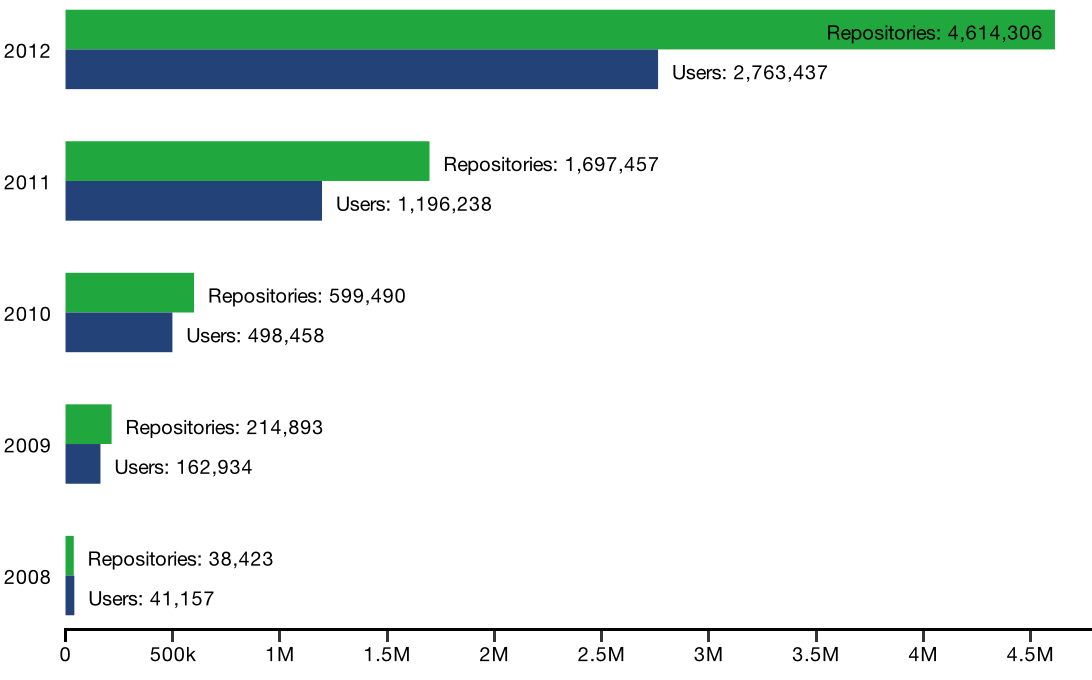


図 1‑1.GitHub登録者・リポジトリ数の推移

このように多くのプロジェクトをホストするGitHubのプロジェクトを調査・解析することによって，近年のソフトウェア開発傾向を調べることができると思われる．

* 1. **研究目的**

本研究では，現実のソフトウェア開発プロセスであるテスト工程に着目して調査し，ソフトウェア開発プロセスを明らかにすることを目的とする．

* 1. **プロジェクトマネジメントとの関連性**

本研究は，現実のソフトウェア開発がどのようなプロセスで行われているかを調べる研究である．ソフトウェア開発におけるプロジェクトマネジャーは，開発プロセスについて深く理解しておくことが望ましく，本研究の成果はそれに貢献することが期待される．

# **ソフトウェア開発について**

本章では，本研究の目的である『テスト工程』に着目する前にソフトウェア開発の基本知識や種類などについて記す．

* 1. **ソフトウェア開発とは**

ソフトウェア開発とは一般的にコンピュータ・ソフトウェアの設計や製造の一連の工程のことである．ソフトウェア開発において一般的に行われる一連の工程は『要求（要件）定義工程・設計工程・実装（コーティング）工程・テスト工程・導入（運用）工程』である．

* 1. **ソフトウェア開発プロセスについて**

ソフトウェア開発プロセスの際に取られる各工程について記す．

* + 1. **要求（要件）定義工程とは**

要求定義工程では，利用者が業務を進める際に，そのシステムなどを使ってどのようなことがしたいのか，どのようなことができなければ困るのかといった内容を，開発者による聞き取りなどを通じて明らかにし，要求定義書として文書化する工程である．また，開発者は利用者がそのソフトウェアにてどのようなことを行いたいのかを元に，その行いたいことを実現するのに実装しなければならない機能や，満たさなければならない性能などを開発者が検討して明確にし，要件を定義する工程である．

* + 1. **設計工程とは**

設計工程には以下の二つの段階がある．以下はそれを説明する．

* **外部設計**

外部設計では，利用者が必要とする要件に基づいて，利用者から見てシステムがどのように動作するかを決めていく．主に操作画面やユーザ・インターフェースの構成や，書式や，データベースの構造などを決める段階である．

* **内部設計**

内部設計では，ハードウェアの構成や調達するソフトウェアパッケージやシステムの処理方式など要件定義書に記載された機能の実現に必要なシステムの構成を決定し，開発するシステムを大まかな機能ごとにサブシステムに分割し，それらのサブシステム間を繋ぐインターフェース使用などを設計する．その後サブシステムをさらに小さいモジュールごとに分割し，各モジュール間のインターフェースなどを定義する段階である．

* + 1. **実装（コーティング）工程とは**

実装（コーティング）工程では，設計工程にて，定義された仕様を元に機能を組み込んでいく工程である．

* + 1. **テスト工程とは**

1. **モジュールテスト（ユニットテスト）**

テスト工程では，モジュールごとにテストを行い，バグ（論理上の誤り）を検出して修正を行う．

1. **ソフトウェア結合テスト**

モジュールごとにテストを行った後にモジュールを組み合わせてモジュール間のインターフェースに不具合がないかを検証する．また，検証するには様々な技法がある．それらの技法は後述の節に記す．

1. **システム結合テスト（システムテスト）**

モジュール間のインターフェースに不具合がなかったらシステム全体を稼働させて行うテストを行う．そのテストでは，必要な機能・性能が満たされているかを検証する．テスト内容としては，必要な機能が正常に動くか・要求された性能か・エラー処理が正常に動作するか・負荷をかけても動作するか・ユーザ・インターフェースが使いやすいかを検証するテストを行う．

* + - 1. **主なテスト技法**

主なテスト技法は以下のものである．

* **ボトムアップテスト**

複数のモジュールを結合させたサブシステムを，下位モジュールから順に結合しながら検証していくテスト方法．複数のモジュールを使用したプログラムを開発する場合は，バグの発生個所を特定しやすくするために，全てのモジュールを組み合わせて一気にテストするのではなく少しずつ組み合わせながら順次テストしていく方式が一般的である．

* **トップダウンテスト**

ボトムアップテストとは逆に，上位モジュールから順に結合しながらテストする方法．

* **ビッグバンテスト**

複数のモジュールを結合させたサブシステムを，全てのモジュールを組み合わせてから一気に動作検証するテスト方法．それぞれのモジュールは単体テストを完了している必要がある．大規模なシステムでビッグバンテストを行うと問題点の特定が難しいため，小規模なシステムや構造の単純なプログラムのテストに用いられる場合がほとんどである．

* **ホワイトボックス法とブラックボックス法**

チェックを行うために必要な値を含めたデータのことを，テストデータと言う．テストデータの作成技法として，プログラムを作成する開発者の視点で作るホワイトボックス法，ユーザ側の視点で作るブラックボックス法がある．

ホワイトボックス法では，プログラムの処理手順や処理内容などの内部構造を意識して，バグが発見できるようにテストデータを作り込む．プログラムの作成者であるエンジニアが，モジュールテスト用のテストデータを作成するときに用いる技法である．

それに対して，ブラックボックス法は，プログラムの内部構造を意識せず，「入力データAを処理すると，Bが結果の値として返ってこなければならない」という要件をクリアするかを検証するテストデータを作るときに用いる技法である．ソフトウェア結合テスト以降のテストで，要件定義書に定められた機能を満たすことを検証するためのテストデータ作成などに使われる技法である．

* + 1. **導入（運用）工程とは**

ソフトウェア開発はテスト環境にて開発し，実際の稼働環境へシステムを移す．ある時点で旧システムから新システムへ一気に切り替える一斉移行や，システムの機能や支店・部門などの単位ごとに新システムへ切り替えて行く順次移行などの方法がある．

一度完成したシステムも，ハードウェアの故障への対応や，業務の変化に伴う修正などの保守が，随時行われる．障害の発生を事前に防ぐために定期的に行われる予防保守と，障害発生後に行われる事後保守がある．

* + 1. **上流・下流工程とは**

ソフトウェア開発において行われる一連の工程は，『要求定義（要件定義）工程・設計工程』を上流工程といい，『実装（コーティング）工程・テスト工程・導入（運用）工程』を下流工程と言う．

上流工程とは，コンピュータ・ソフトウェアの設計や製造の初期工程のことである．上流工程にて，システム全体の規模や，開発にかかる期間やコストなども上流工程にて決定される．

下流工程とは，実装や導入など具体的なソフトウェアの構築・配備に関する工程のことである．

* 1. **開発プロセスの種類**
     1. **ウォーターフォール・モデル**

ウォーターフォール・モデルはソフトウェア開発において，古典的なプロセスモデルである．先進的な情報システム開発の世界や小規模開発においては必ずしも最適なモデルとは言えませんが，今も大きな規模のソフトウェア・システムを開発する場合の基本となるモデルです．

ウォーターフォール・モデルは，図 2-1のように『要件定義工程・設計工程・実装（コーティング）工程・テスト工程・導入（運用）工程』などの各段階に分けあたかも水が落ちるかのように工程を進め，前の工程へ戻らないような進め方をする．

要求（要件）

定義工程

設計工程

実装（コーティング）工程

テスト工程

導入（運用）工程

図 2‑1.ウォーターフォール・モデル図

ウォーターフォール・モデルでは，工程を明確に区別しているため，開発の進捗管理がしやすく多人数での共同作業なる大規模システムの開発においても，信頼性の高いシステムを構築できるというメリットがある．

一方では．様々な欠点がある．コーティングを始めるまでにすべての要求と設計を確定しなくてはならない．しかし，要求事項や設計の詳細をすべて明確にするのは簡単ではない．ビジネスにスピードを求められる時代ではシステムへの要求事項も時々刻々と変化しているのが普通である．現実の世界はウォーターフォール・モデルが求めるようには，厳密には工程後に区切られて動かないのである．また，ウォーターフォール・モデルでは大量のドキュメントの作成に必要とし，開発時間がかかるため，工程の流れが最後まで終了しない限りユーザはたとえ部分的にでもシステムを入手して使い始めることができない．

* + 1. **アジャイルソフトウェア開発**

アジャイルソフトウェア開発は，ソフトウェア開発において，ソフトウェア工学において迅速かつ適応的にソフトウェア開発を行う軽量な開発手法群の総称である．

近年，アジャイルソフトウェア開発手法が数多く考案されている．ソフトウェア開発で実際に採用される事例も少しずつではあるが増えつつある． アジャイルソフトウェア開発手法の例としては，エクストリーム・プログラミング (XP) などがある．

アジャイルソフトウェア開発は，図 2-1のように『要件定義工程・設計工程・実装（コーティング）工程・テスト工程・導入（運用）工程』などの各段階に分けあたかも水が落ちるかのように工程を進め，前の工程へ戻らないような進め方をする．

**1期間中に**

**複数回繰り返し**

レビュー

テスト工程

実装（コーティング）工程

設計工程

仕様変更

要求（要件）

定義工程

設計工程

実装（コーティング）工程

テスト工程

レビュー

図 2‑2.アジャイルソフトウェア開発図

アジャイルソフトウェア開発では，工程を明確に区別していなく，開発の進捗管理がしにくい．一方では，様々な利点がある．ウォーターフォール・モデルとは違い，すべての要求と設計をしなくても，ある程度の要求と設計を立ててしまえばプロセスを進めることができる．そのため，IT市場の進化への適応することができる．また，ウォーターフォール・モデルでは，ドキュメントを詳しく作らないことが多いかわりに，ユーザは部分的なシステムを使うことができる．

* + 1. **アジャイルソフトウェア開発**

テスト駆動開発 (Test Driven Development; TDD) とは，プログラム開発手法の一種で，プログラムに必要な各機能について，最初にテストを書き，そのテストが動作する必要最低限な実装をとりあえず行った後，コードを洗練させるという短い工程を繰り返すスタイルである．また，テストを最初に書くことをテストファストという．

* 1. **開発プロセスの比較**

開発手法にはこれらのような開発プロセスがある．開発手法にはそれぞれメリットもあればデメリットの面があるため，ソフトウェア開発規模によって適宜変更する必要がある．

一般的には以下の図のように，IT市場の進化などに対して適応的開発を行う必要がある場合は，アジャイルソフトウェア開発を行う．また，銀行システムなど大きくニーズの変化が起きにくいようなプロジェクトでは計画重視のウォーターフォール・モデルを採用する．

適応的開発　　　　　　　　　　　計画重視

アジャイルソフトウェア

開発

ウォーターフォール

モデル

反復型開発

図 2‑3.開発プロセスの比較図

# **Gitについて**

Gitとは、プログラムのソースコードなどの変更履歴を記録・追跡するための分散型バージョン管理システムである．もとはLinuxカーネルのソースコード管理に用いるためにリーナス・トーバルズによってLinuxカーネルのような巨大プロジェクトにも対応できるように開発された．Gitでは、各ユーザのローカル環境に、全履歴を含んだリポジトリが複製される．そのため、たとえインターネットに繋げなく、コアリポジトリにアクセスできない環境でも、履歴の調査や変更の記録といったほとんどの作業を行うことができる．これが「分散型」と呼ばれる理由である．

* 1. **バージョン管理システム**

バージョン管理システムとはソフトウェア開発ではソースコードを作成しながらソフトウェアを作り上げていきますが，バグの修正や機能の追加ごとにソースコードの状態を記録し，それぞれのバージョンを管理する必要がある．そのようなソースコードの管理をするソフトウェアが「バージョン管理システム」である．ソフトウェア開発を複数人で開発を行う場合に必要不可欠なソフトウェアとなっている．

バージョン管理システムには集中型バージョン管理システム（SVN（Apache Subversion））や，分散型バージョン管理システム（Git）などがある．

本稿ではGitバージョン管理システムを取り扱う．Gitバージョン管理システムは分散型バージョン管理システムに分類される．Gitバージョン管理システムはLinux開発のために2005年ごろにGitの原型となるプログラムが開発された．当初，この管理システムはLINUXのバージョン管理システムを置き換える目的で作成された．LINUXカーネルは高度な変更が加えられるソフトウェアのため，既存のバージョン管理システム（SVNなど）十分な機能や性能が確保できないため，新しく開発された．

* **集中型バージョン管理システム**

集中型バージョン管理システムは図3-1のようにリポジトリをサーバに集中させて配置するため，1つのリポジトリしか存在しない．データが中央のサーバに集中されるので管理がシンプルであるが，サーバに接続できない状況だと最新のソースコードが取得できないため開発がほとんど行えない．また，サーバが故障してデータが消えてしまった場合は，最新のソースコードをロストしてしまいます．

リポジトリ

**開発者Ａ**

**開発者Ｂ**

**Commit**

**Checkout**

**Commit**

**Checkout**

**図 3‑1.集中型バージョン管理イメージ図**

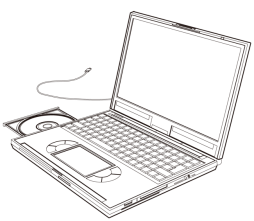
* **分散型バージョン管理システム**

分散型バージョン管理システムは，図3-2のようにサーバにある特定のリポジトリを自分のアカウント以下のリポジトリに複製し，複製したリポジトリは自分の好きなように編集でき，サーバにあるリポジトリとは異なるリポジトリとなる．

分散型は複雑であるが，図3-2のように複数リポジトリがあるのが特徴である．手元の開発環境にもリポジトリがあるため，リモートのリポジトリに接続できなくても開発することができる．

**開発者Ａ**

**開発者B**



Git



Git

**GitHub**

Git

Git

**Fork**

**Pull Request**

**Pull**

**Push**

**Pull**

**Push**

**ローカル環境**

**ローカル環境**

図 3‑2.分散型バージョン管理図イメージ図

* + 1. **バージョン管理システム比較**

集中型バージョン管理システムと分散型バージョン管理システムのどちらが良いかは双方にメリット・デメリットがあるので，ケースバイケースである．

しかし，今日ではGitとGitHubの普及により，今後は分散型が圧倒的に普及するのは明らかである．ルールによっては，分散型バージョン管理システムで集中型バージョン管理システムのような管理もできる．そのため，今後，集中型バージョン管理システムを使うケースが少なくなると考えられている．同様に，今後集中型バージョン管理システムから分散型バージョン管理システムへ移行していくと思われる．

* 1. **用語説明**

以下に本章にて出てくる専門用語やGitHubの提供する機能の説明をする．

* + 1. **リポジトリ**

ファイルやディレクトリの状態を記録する場所である．保存された状態は，内容の変更履歴として格納され，変更履歴を管理したいディレクトリをリポジトリの管理下に置くことで，そのディレクトリ内のファイルやディレクトリの変更履歴を記録することができるのである．

* + 1. **コミット**

ファイルやディレクトリの追加・変更を，リポジトリに記録するために行う操作のことである．

* + 1. **圧縮コミット**

複数のコミットを一つのコミットにまとめることである．

* + 1. **クローン**

サーバにあるリモートリポジトリをローカルにコピーすることである．

* + 1. **プッシュ**

リモートリポジトリに自分の変更履歴がアップロードされて，リモートリポジトリ内の変更履歴がローカルリポジトリの変更履歴と同じ状態になるようにすることである．

* + 1. **ブランチ**

履歴の流れを分岐して記録していくためのものである．分岐したブランチは他のブランチの影響を受けないため，同じリポジトリ中で複数の変更を同時に進めていくことができるのである．

* + 1. **プル**

リモートリポジトリから最新の変更履歴をダウンロードしてきて，自分のローカルリポジトリにその内容を取り込むことである．

* + 1. **プルリクエスト**

相手に対して自分のブランチをpullしてもらうように要求する機能のことである．

* + 1. **Origin**

clone元のリモートリポジトリのことである．

* + 1. **Upstream**

当該リポジトリよりも上流に位置するリモートリポジトリのことである．

* + 1. **ステージング**

修正作業によって更新したもののうち，コミット対象の更新を選り分けるための作業のことである．

* + 1. **ステージングエリア**

ステージングを行う領域のことである．

* + 1. **Revert**

ステージングエリアに追加した変更を戻すことである．

* + 1. **タグ**

コミットを参照しやすくするために，わかりやすい名前を付けるもののことである．

* + 1. **チェックアウト**

ブランチ内から作業ディレクトリに展開されたファイル・ディレクトリ群のことである．

* + 1. **HEAD**

現在使用しているブランチの先頭を表す名前のことである．

* + 1. **トラック**

Git上で変更内容を追跡できるようにすることである．特に，ローカルブランチとリモートブランチとを紐付けすることを指す．また，ファイルやディレクトリに対して言う場合は，Gitオブジェクトに変換してリポジトリに格納することを指すのである．

* + 1. **作業ツリー**

ブランチ内から作業ディレクトリに展開されたファイル・ディレクトリ郡のことである．

* + 1. **マージ**

当該ブランチに対して別のブランチの差分を取り込むことである．

* + 1. **リベース**

マージの一種で，当該ブランチの更新差分を無かったことにして，別ブランチの最新を取り込んだ後，当該ブランチの更新差分を適用する処理することである．

* + 1. **チェリーピック**

マージの一種で，全ての差分を取り込むのではなく，特定のコミットを指定して取り込むことである．

* + 1. **競合**

マージ（リベースを含む）の際に，同一のファイル・ディレクトリに対しての更新が発生していることである．

* + 1. **フェッチ**

リモートリポジトリの最新を取り込むこと

* + 1. **二等分**

二分探索の要領で，バグが入り込んだコミットを突き止める作業

* + 1. **フォーク**

GitHubのサービスで，相手のリポジトリを自分のリポジトリとしてコピー・保持できる機能のことである．

* + 1. **Issue**

一つのタスクを一つのIssueに割り当てて，トラッキングや管理を行えるようにするための機能である．

* + 1. **Wiki**

Wiki機能は，いつでもだれでも文書を書き換えて保存できるため，共同で文章を作成できる機能である．

* + 1. **Gist**

コードの断片など，リポジトリに入れるほどでもないコードを管理，公開するのに重宝する機能である．また，ブログなどを書いている人はGistにサンプルコードを登録すれば，ブログの記事に埋め込むこともできる．

* + 1. **News Feed**

フォローしているユーザやWatchしているプロジェクトの活動情報が流れてくる機能である．最新の動向を確認するときに使われる．

* + 1. **GitHub API**

GitHubでは開発者向けにAPIを公開している．また，GitHub APIについては後述に記す．

* + 1. **アドオン**

ソフトウェアに追加される拡張機能のことである．

* + 1. **作業ツリー**

ローカルリポジトリ上にある現在の作業ファイルである．作業ツリーの変更点はaddによるインデックスに追加されることである．

* + 1. **ディレクトリ**

　フォルダのことである．ファイルを分類・整理するための保管場所である．

* + 1. **フォロー**

手助けすることである．

* + 1. **フォロワー**

ユーザがフォローしている相手から視点のことである．

* + 1. **スター**

GitHub内のお気に入り機能である．

* + 1. **リビジョン**

ある期間内までの過去のプロジェクトデータやある程度まとまったプロジェクトデータを記録したものである．

* + 1. **ウィキ（Wiki）**

　Webサーバ上のハイパーテキスト文書を書き換えるシステムの一種である．それらシステムに使われるソフトウェア自体や，システムを利用して作成された文書群全体を指してウィキと呼ぶこともある

* + 1. **Organizations**

　Organizationsとは普段，個人的にGitHubを使用している人が仕事などでGitHubを仕事用に使用したい場合にアカウントをもう1つ作成するのではなく，同じアカウントで会社用に使用するアカウントとして使用するのがOrganizationsである．

* 1. **Gitの簡単な使い方**

Gitを使うにはGitをインストールしなければならない．本稿では、LinuxのUbuntuを使って説明をする．大体の捜査は端末（ターミナル）にてコマンドを入力して行う．

1. 端末にて以下のコマンドを打ち込む

|  |
| --- |
| sudo apt-get install git-core |

1. Gitの設定をする

適宜変更して端末にて以下のコマンドを打ち込む

|  |
| --- |
| git config --global user.name "Ryugo Shimizu"  git config --global user.email "dxxxxxxx@it-chiba.ac.jp" |

1. リポジトリの作成

適宜変更して端末にて以下のコマンドを打ち込む

|  |
| --- |
| mkdir test  cd test  vi test.html |

Vimエディッターで

|  |
| --- |
| <HTML>  <HEAD>  <TITLE>hello world</TITLE>  </HEAD>  <BODY>  <font size=50>test page</font>  </BODY>  </HTML> |

の内容のファイルを作成後

|  |
| --- |
| git init  git add test.html |

と入力する．

また、リポジトリに入れたいファイルが多数ある場合は

|  |
| --- |
| git add . |

と入力してもよい．

以上でリポジトリの作成は終了である．

1. コミットする

|  |
| --- |
| git commit –m “test commit” |

でコミットすることができる．

1. 変更履歴の確認

|  |
| --- |
| vi test.html |

Vimエディッターで

|  |
| --- |
| <HTML>  <HEAD>  <TITLE>hello world</TITLE>  </HEAD>  <BODY>  <font size=50>test page</font><br>  <font>welcome to you</font>  </BODY>  </HTML> |

に変更して保存する．

変更後ステータスを確認すると、

|  |
| --- |
| git status |

と入力すると

|  |
| --- |
| # On branch master  # Changes to be committed:  # (use "git reset HEAD <file>..." to unstage)  #  # modified: test.html  # |

と表示される．これは前回のコミットから現在までで変更されたファイル名を表している．

|  |
| --- |
| git log |

と入力すると

|  |
| --- |
| commit eee760579db4956d1f430a6107eb8d9223a330e3　　 ＃変更ナンバー  Author: xxx xxxxx dxxxxxxx@it-chiba.ac.jp　　　　　　　＃だれが  Date: Sat Jan 25 13:43:41 2014 +0900　　　　　　　　 ＃いつ  index create　　　　　　　　　　　　　　　　　　 ＃なにをした |

と表示される．

また、

|  |
| --- |
| git log |

と入力すると上記のログとともにDiffデータ（変更点）の取得することができる．

* リポジトリ内で名前変更or移動/削除する場合

普段は「mv・rm」で移動できるのだが、Gitにてバージョン管理を行っている場合は以下のコマンドを使用する．前者は名前変更or移動であり、後者は削除コマンドである

|  |
| --- |
| git mv test.html index.html |
| git rm -f test.html |

* 削除したファイルを元に戻すコマンド

削除してからコミットしてない場合のみに有効．削除してしまってからの場合はリバートを行うことで復元することができる

|  |
| --- |
| git checkout test.html |

以上が簡単な使い方である．

# **GitHubについて**

GitHubとはアメリカのサンフランシスコを拠点とする，GitHub社によって運営されているソフトウェア開発プロジェクトのための共有ウェブサービスであり，Gitバージョン管理システムである．

GitHubはGistと呼ばれるpastebinスタイルのサイトも運営している．それらのコンテンツは具体的には個々のリポジトリのためのウィキおよびウェブページでありGitリポジトリを通して編集される．

尚現在では，GitHubを就職活動の選考書類として，GitHubアカウントを使用する会社も出てきている．

また，GitHubとは別に和製GitHubと言われる，株式会社ビズリーチが運営するIT・ウェブエンジニアの協業を支援するサイト「コードブレイク」を正式オープンし，プログラムのソースコードを共有できるリポジトリを提供するGitホスティングサービス「gitBREAK」（ギットブレイク）が開始されたが，本稿では近年のソフトウェア開発傾向を調べるために，世界中で使われ，ユーザ数が多いGitHubを用いることとする．

GitHubで開発が進められている有名なソフトウェアの一部は以下の表のとおりである．

表 4‑1. GitHubで開発が進められているソフトウェア

|  |  |
| --- | --- |
| Ruby on Rail | Rubyにて使われる代表的なWEBフレームワーク |
| node | JavaScriptにて人気のあるプラットフォーム |
| jQuery | JavaScriptライブラリ |
| Symfony2 | PHPにてつくられたフルスタックWEBフレームワーク |
| Bootstrap | ついったーのようなインターフェースをつくれるコンポーネント集 |

また，GitHubで公開されているプロジェクトには様々なプログラミング言語があり，また，背景で述べたように，GitHubは最もよく利用されているホスティングサービスの一つのため，GitHubで使われている言語は世界的に使われている言語ともとらえられるため，以下の図から，近頃はJavaScriptやJAVAが人気である．

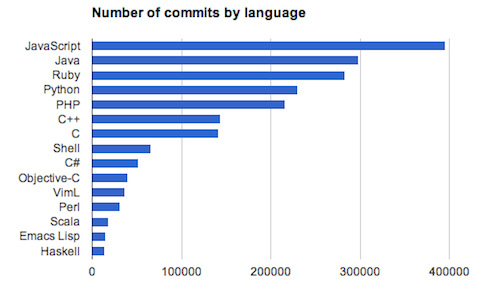


図 4‑1.2012年1月1日時点での言語別コミット数

* 1. **バージョン管理システム**

本章で出てくるGitとGitHubは別物である．

GitとはGitリポジトリというデータの貯蔵庫にソースコードなどを入れて利用する．このGitリポジトリを置く場所をインターネット上に提供しているのが　GitHubである．つまり，GitHubで公開されているソフトウェアのコードはすべてGitでされている．Gitについて理解しておくことはGitHubを使う上で重要である．

* 1. **GitHub APIについて**

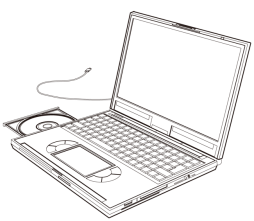
本節では本研究においてツールを作成する際に，GitHub APIを使用する．そのため本節ではAPI・GitHub APIについて調査する．

* + 1. **APIとは**

アプリケーションプログラミングインタフェースのことである．ソフトウェアコンポーネントが互いにやりとりするのに使用するインターフェースの仕様である．APIには，サブルーチン，データ構造，オブジェクトクラス，変数などの仕様が含まれる．APIには様々な形態があり，POSIXのような国際規格，マイクロソフトの Windows API のようなベンダーによる文書，プログラミング言語のライブラリがある．

APIは，アプリケーションから利用できる，オペレーティングシステムやプログラミング言語で用意されたライブラリなどの機能の入り口となるものである．主に，ファイル制御，ウインドウ制御，画像処理，文字制御などのための関数として提供されることが多い．

つまり，簡単にいえば，アプリケーションをプログラムするにあたって，プログラムの手間を省くため，もっと簡潔にプログラムできるように設定されたインターフェースの事である．



**API提供サーバ**

**API利用サーバ**

**一般ユーザ**

1. **WEBサイトにアクセス**

**②APIと通じてデータ取得**

**③JSON形式等でデータを返送**

**④データを表示**

図 4‑2.APIイメージ図

* + 1. **GitHub APIとは**

GitHub APIとはGitHubにて使用するために用意されたAPIである．

GitHub APIを使用した例は図 3-5のようになる．



**拡大**

図 4‑3.GitHub API使用例

図3-5のように，GitHub内のデータをJSON形式で呼び出すことができる．

このようにJSON形式でデータを取得することで，プログラムに読み込んだりできる．また，API種類『DELETE・POST・PUT』などを使えば，GitHub外からのGitHubの操作することができる．

今回ソフトウェア開発プロジェクトの情報を読み込むときに使用する．

* + - 1. **GitHub APIリスト**

本項では，GitHubにあるAPIをリスト形式で説明をする．

表 4‑2. GitHub APIリスト

|  |  |  |
| --- | --- | --- |
| **API種類** | **APIコード** | **意味** |
| **Events** | | |
| GET | /users/{userName}/events | ユーザが実行したイベントの一覧を取得する． |
| GET | /networks/{userName}/{repoName}/events | ネットワーク内でユーザが実行したリポジトリの共有イベントの一覧を取得する． |
| GET | /orgs/{org}/events | 組織の共有イベントの一覧を取得する． |
| GET | /repos/{userName}/{repoName}/events | ユーザが実行したリポジトリのイベントを取得する． |
| GET | /events | 共有イベント名を取得する． |
| GET | /users/{userName}/events/orgs/{org} | 組織でユーザが実行したイベントの一覧を取得する． |
| GET | /repos/{userName}/{repoName}/issues/events | リポジトリでユーザが実行したイベントの一覧を取得する． |
| GET | /users/{userName}/received\_events | ユーザが受信したイベントの一覧を取得する． |
| GET | /users/{userName}/events/public | ユーザの共有イベント名の一覧を取得する． |
| GET | /users/{userName}/received\_events/public | ユーザが受信した共有イベントの一覧を取得する． |
| **Issue Comments** | | |
| DELETE | **/repos/{userName}/{repoName}/issues/comments/{id}** | リポジトリのIssueのコメントを削除する． |
| GET | **/repos/{userName}/{repoName}/issues/comments/{id}** | リポジトリのIssueのコメントを取得する． |
| GET | **/repos/{userName}/{repoName}/issues/{id}/comments** | リポジトリのIssueの返信コメントを取得する． |
| POST | **/repos/{userName}/{repoName}/issues/{id}/comments** | リポジトリのIssueにコメントを作成する． |
| POST | **/repos/{userName}/{repoName}/issues/comments/{id}** | リポジトリのIssueにコメントを編集する． |
| **Git Date** | | |
| GET | /repos/{userName}/{repoName}/git/commits/{sha} | 使用ユーザはGitリポジトリからコミットを取得する． |
| GET | /repos/{userName}/{repoName}/git/trees/{sha} | 使用ユーザはGitリポジトリからツリー形式に取得する． |
| GET | /repos/{userName}/{repoName}/git/tags/{sha} | 使用ユーザはGitリポジトリのタグを取得する． |
| GET | /repos/{userName}/{repoName}/git/refs/tags | 使用ユーザのGitリポジトリの全ての証明書タグを取得する． |
| GET | /repos/{userName}/{repoName}/git/refs | 使用ユーザのGitリポジトリのすべて証明書を取得する． |
| GET | **/repos/{userName}/{repoName}/git/refs/{ref}** | 使用ユーザのGitリポジトリの証明書を取得する． |
| GET | **/repos/{userName}/{repoName}/git/blobs/{sha}** | 使用ユーザのGitリポジトリのブロブを取得する． |
| GET | **/repos/{userName}/{repoName}/git/trees/{sha}?recursive=1** | 使用ユーザのGitリポジトリの１つ前のツリーを入手する． |
| POST | **/repos/{userName}/{repoName}/git/blobs** | 使用ユーザのGitリポジトリにブロブを作成する． |
| POST | **/repos/{userName}/{repoName}/git/trees** | 使用ユーザのGitリポジトリにツリーを作成する． |
| POST | **/repos/{userName}/{repoName}/git/refs/{ref}** | 使用ユーザのGitリポジトリの証明書を更新する． |
| POST | **/repos/{userName}/{repoName}/git/tags** | 使用ユーザのGitリポジトリ内にタグオブジェクトを作成する． |
| POST | **/repos/{userName}/{repoName}/git/commits** | 使用ユーザのGitリポジトリ内にコミットを作成する． |
| POST | **/repos/{userName}/{repoName}/git/refs** | 使用ユーザのGitリポジトリの証明書を作成する． |
| **Issue Events** | | |
| GET | **/repos/{userName}/{repoName}/issues/{id}** | リポジトリの指定したIssueの変化を取得する． |
| GET | **/repos/{userName}/{repoName}/issues** | リポジトリのIssueの変化を取得する． |
| GET | **/repos/{userName}/{repoName}/issues/{issueId}/events** | リポジトリの指定したIssueの変化を取得する． |
| **Issue** | | |
| GET | **/repos/{userName}/{repoName}/issues/{id}** | 選択されたIssueを取得する |
| GET | **/repos/{userName}/{repoName}/issues** | リポジトリのIssueの一覧を取得する． |
| GET | **/issues** | ユーザのIssueの一覧を取得する． |
| POST | **/repos/{userName}/{repoName}/issues/{id}** | 指定されたリポジトリのIssueを編集する． |
| POST | **/repos/{userName}/{repoName}/issues** | 指定されたリポジトリにIssueを作成する． |
| **Mile Stones** | | |
| DELETE | **/repos/{userName}/{repoName}/milestones/{id}** | マイルストーンを削除する． |
| GET | **/repos/{userName}/{repoName}/milestones** | マイルストーンの一覧を表示する． |
| GET | **/repos/{userName}/{repoName}/milestones/{id}** | 指定したマイルストーンの表示する． |
| POST | **/repos/{userName}/{repoName}/milestones** | マイルストーンを作成する． |
| POST | **/repos/{userName}/{repoName}/milestones/{id}** | マイルストーンを更新する． |
| **Labels** | | |
| DELETE | **/repos/{userName}/{repoName}/issues/{id}/labels** | 指定したリポジトリのIssueから全てのラベルを削除する． |
| DELETE | **/repos/{userName}/{repoName}/labels/{id}** | 指定したリポジトリから指定ラベルを削除する． |
| DELETE | **/repos/{userName}/{repoName}/issues/{id}/labels/{id}** | 指定したリポジトリのIssueから指定したラベルを削除する． |
| GET | **/repos/{userName}/{repoName}/milestones/{id}/labels** | すべてのマイルストーンを取得する． |
| GET | **/repos/{userName}/{repoName}/labels** | リポジトリ内のラベルがついたもの一覧を取得する． |
| GET | **/repos/{userName}/{repoName}/labels/{id}** | リポジトリ内の選択されたラベルを取得する． |
| GET | **/repos/{userName}/{repoName}/issues/{id}/labels** | リポジトリ内のIssueにつけられラベルを取得する． |
| POST | **/repos/{userName}/{repoName}/issues/{id}/labels** | リポジトリ内のIssueにラベルを追加する． |
| POST | **/repos/{userName}/{repoName}/labels/{id}** | リポジトリ内のラベルを更新する． |
| POST | **/repos/{userName}/{repoName}/labels** | リポジトリ内にラベルを作成する． |
| PUT | **/repos/{userName}/{repoName}/issues/{id}/labels** | Issueのすべてのラベルを交換する． |
| **Organization Members** | | |
| DELETE | **/orgs/{org}/public\_members/{userName}** | ユーザのメンバシップを非公開にする． |
| DELETE | **/orgs/{org}/members/{userName}** | メンバを組織から削除する． |
| GET | **/orgs/{org}/members/{userName}** | メンバを取得する． |
| GET | **/orgs/{org}/members** | 組織のメンバのユーザ一覧を取得する． |
| GET | **/orgs/{org}/public\_members/{userName}** | パブリックメンバを取得する． |
| GET | **/orgs/{org}/public\_members** | パブリックメンバの一覧を取得する． |
| PUT | **/teams/{id}/members/{user}** | メンバを追加する． |
| PUT | **/orgs/{org}/public\_members/{userName}** | ユーザの役割を変更する． |
| **Organization** | | |
| GET | **/user/orgs** | 組織に認証されたユーザの一覧を取得する． |
| GET | **/orgs/{org}** | 組織を取得する． |
| GET | **/users/{userName}/orgs** | 組織のユーザの一覧を取得する． |
| POST | **/orgs/{org}** | 組織を編集する． |
| **Organization Teams** | | |
| DELETE | **/teams/{id}/members/{userName}** | チームメンバを削除する． |
| DELETE | **/teams/{id}/repos/{userName}/{repoName}** | チームからリポジトリを削除する． |
| DELETE | **/teams/{id}** | チームを削除する． |
| GET | **/teams/{id}/members/{userName}** | チームメンバを取得する． |
| GET | **/teams/{id}/members** | チームのメンバの一覧を取得する． |
| GET | **/teams/{id}** | チーム詳細を取得する． |
| GET | **/orgs/{org}/teams** | 組織内のチームの一覧を取得する． |
| GET | **/teams/{id}/repos/{userName}/{repoName}** | チーム内のリポジトリを取得する． |
| GET | **/teams/{id}/repos** | チームのリポジトリの一覧を取得する． |
| POST | **/orgs/{org}/teams** | チームを作成する． |
| POST | **/teams/{id}** | チームを編集する． |
| PUT | **/teams/{id}/members/{userName}** | チームメンバを追加する． |
| PUT | **/teams/{id}/repos/{userName}/{repoName}** | チームリポジトリを追加する． |
| **Pull Requests** | | |
| GET | **/repos/{userName}/{repoName}/pulls** | リポジトリ内のプルリクエストの一覧を取得する． |
| GET | **/repos/{userName}/{repoName}/pulls/{id}/commits** | リポジトリ内のコミットへのプルリクエストの一覧を取得する． |
| GET | **/repos/{userName}/{repoName}/pulls/{id}/files** | リポジトリからファイルへのプルリクエストの一覧を取得する． |
| GET | **/repos/{userName}/{repoName}/pulls/{id}** | リポジトリから指定したプルリクエストを表示する． |
| GET | **/repos/{userName}/{repoName}/pulls/{id}/merge** | リポジトリからマージへのプルリクエストを取得する． |
| POST | **/repos/{userName}/{repoName}/pulls/{id}** | プルリクエストを更新する． |
| POST | **/repos/{userName}/{repoName}/pulls** | プルリクエストを作成する． |
| PUT | **/repos/{userName}/{repoName}/pulls/{id}/merge** | マージをプルリクエストする． |
| **Pull Requests Review Comments** | | |
| DELETE | **/repos/{userName}/{repoName}/pulls/comments/{id}** | プルリクエストのコメントを削除する． |
| GET | **/repos/{userName}/{repoName}/pulls/{id}/comments** | プルリクエストへのコメントの一覧をする． |
| GET | **/repos/{userName}/{repoName}/pulls/{id}** | プルリクエストへの指定されたコメントを取得する． |
| POST | **/repos/{userName}/{repoName}/pulls/comments/{id}** | コメントを編集する． |
| POST | **/repos/{userName}/{repoName}/pulls/{id}/comments** | コメントを作成する． |
| **Repo Collaborators** | | |
| DELETE | **/repos/{userName}/{repoName}/collaborators/{user}** | 共同リポジトリを削除する． |
| GET | **/repos/{userName}/{repoName}/collaborators** | 共同リポジトリの一覧を取得する． |
| GET | **/repos/{userName}/{repoName}/collaborators/{user}** | 共同リポジトリの協力者を取得する． |
| PUT | **/repos/{userName}/{repoName}/collaborators/{user}** | 共同リポジトリのユーザを追加する． |
| **Repo Commits** | | |
| DELETE | **/repos/{userName}/{repoName}/comments/{id}** | コミットコメントを削除する． |
| GET | **/repos/{userName}/{repoName}/commits** | コミットの一覧を取得する． |
| GET | **/repos/{userName}/{repoName}/commits/{sha}** | 指定したコミットを取得する． |
| GET | **/repos/{userName}/{repoName}/comments** | コミットへのコメントの一覧を取得する． |
| GET | **/repos/{userName}/{repoName}/commits/{sha}/comments** | 指定したコミットへのコメントを取得する． |
| GET | **/repos/{userName}/{repoName}/comments/{id}** | 指定したコメントを取得する |
| GET | **/repos/{userName}/{repoName}/compare/{base}...{head}** | 2つのコミットを比較する． |
| POST | **/repos/{userName}/{repoName}/comments/{id}** | コミットのコメントを更新する． |
| POST | **/repos/{userName}/{repoName}/commits/{sha}/comments** | コミットのコメントを作成する． |
| **Repo Deploy Keys** | | |
| DELETE | **/repos/{userName}/{repoName}/keys/{id}** | リポジトリの暗号鍵を削除する． |
| GET | **/repos/{userName}/{repoName}/keys/{id}** | リポジトリの暗号鍵を取得する． |
| GET | **/repos/{userName}/{repoName}/keys** | リポジトリの暗号鍵の一覧を取得する． |
| POST | **/repos/{userName}/{repoName}/keys** | リポジトリの暗号鍵を作成する． |
| POST | **/repos/{userName}/{repoName}/keys/{id}** | リポジトリの暗号鍵を編集する． |
| **Repo Downloads** | | |
| DELETE | **/repos/{userName}/{repoName}/downloads/{id}** | リポジトリのダウンロードデータを削除する． |
| GET | **/repos/{userName}/{repoName}/downloads/{id}** | リポジトリのダウンロードデータを取得する． |
| GET | **/repos/{userName}/{repoName}/downloads** | リポジトリのダウンロードデータの一覧を取得する． |
| POST | **/repos/{userName}/{repoName}/downloads** | リポジトリから新しくダウンロードする． |
| **Repo Forks** | | |
| GET | **/repos/{userName}/{repoName}/forks** | フォークデータの一覧を取得する． |
| POST | **/repos/{userName}/{repoName}/forks** | フォークを作成する． |
| **Repo Hooks** | | |
| DELETE | **/repos/{userName}/{repoName}/hooks/{id}** | フックを削除する． |
| GET | **/repos/{userName}/{repoName}/hooks** | フックの一覧を取得する． |
| GET | **/repos/{userName}/{repoName}/hooks/{id}** | 指定したフックを取得する． |
| POST | **/repos/{userName}/{repoName}/hooks** | フックを作成する． |
| POST | **/repos/{userName}/{repoName}/hooks/{id}** | フックを編集する． |
| POST | **/repos/{userName}/{repoName}/hooks/{id}/test** | フックのテスト（検証）を行う． |
| **Repos** | | |
| GET | **/repos/{userName}/{repoName}/branches** | ブランチの一覧を取得する． |
| GET | **/user/repos** | 認証済みユーザのリポジトリの一覧を取得する． |
| GET | **/users/{userName}/repos** | 指定したユーザの共有リポジトリの一覧を取得する． |
| GET | **/repos/{userName}/{repoName}** | ユーザはリポジトリを取得する． |
| GET | **/orgs/{org}/repos** | 指定した組織の共有リポジトリの一覧を取得する． |
| GET | **/repos/{userName}/{repoName}/contributors** | 貢献者の一覧を取得する． |
| GET | **/repos/{userName}/{repoName}/languages** | 言語の一覧を取得する． |
| GET | **/repos/{userName}/{repoName}/teams** | チームの一覧を取得する． |
| GET | **/repos/{userName}/{repoName}/tags** | タグの一覧を取得する． |
| POST | **/user/repos** | 認証されたユーザのためにリポジトリを作成する． |
| POST | **/repos/{userName}/{repoName}** | ユーザのリポジトリを編集する． |
| POST | **/orgs/{org}/repos** | 組織のリポジトリを作成する． |
| **Repo Watching** | | |
| DELETE | **/user/watched/{userName}/{repoName}** | リポジトリのウォッチをやめる． |
| GET | **/user/watched** | ユーザによってウォッチされているリポジトリの一覧を取得する． |
| GET | **/repos/{userName}/{repoName}/watchers** | ユーザがウォッチしているリポジトリの一覧を取得する． |
| GET | **/users/{userName}/watched** | ユーザがウォッチしたリポジトリの一覧を取得する． |
| GET | **/user/watched/{userName}/{repoName}** | 自分のリポジトリウォッチしているユーザの一覧を取得する． |
| PUT | **/user/watched/{userName}/{repoName}** | リポジトリをウォッチする |
| **User Followers** | | |
| DELETE | **/user/following/{userName}** | フォローを削除する． |
| GET | **/user/following** | 認証しているユーザのフォローしている一覧を取得する． |
| GET | **/users/{userName}/followers** | フォロワーの一覧を取得する． |
| GET | **/user/followers** | 認証されたユーザのフォロワーの一覧を取得する． |
| GET | **/users/{userName}/following** | フォローしているユーザの一覧を取得する． |
| GET | **/user/following/{userName}** | フォローしているかどうかを取得する． |
| PUT | **/user/following/{userName}** | ユーザをフォローする． |
| **User Public Keys** | | |
| DELETE | **/user/keys/{id}** | 公開鍵を削除する． |
| GET | **/user/keys** | 公開鍵の一覧を取得する． |
| GET | **/user/keys/{id}** | 指定した公開鍵を取得する． |
| POST | **/user/keys/{id}** | 公開鍵を更新する． |
| POST | **/user/keys** | 公開鍵を作成する． |

* 1. **GitHub APIを実際に使ってみる**

GitHub APIを使う方法はさまざまな方法があるが、本稿では、LinuxのUbuntuでcURLを使って説明をする．大体の捜査は端末（ターミナル）にてコマンドを入力して行う．

* + 1. **cURLとは**

cURLとは、様々なプロトコルを用いてデータを転送するライブラリとコマンドラインツールである．また、cURLを使うにはインストールする必要性がある．

1. cURLをインストールする．

|  |
| --- |
| sudo apt-get install curl |

アプリをインストールなどするためには、OSの管理者権限が必要なため、「sudo」を使い、スーパーユーザ（Root）権限を取得する．「apt-get install」とはUbuntuのアプリパッケージ（アプリリポジトリ）からインストールしますという意味である．

1. cURLの動作をテストする

前者のコマンドを打つと「千葉工業大学」のホームページのソースコードが表示される．また、後者を入力すると「千葉工業大学」のホームページのソースコードが指定したファイル名で保存される．このように、表示・保存されればcURLの動作している．

|  |
| --- |
| curl www.it-chiba.ac.jp  curl -o index.html www.it-chiba.ac.jp |

* + 1. **APIを実際に使ってみる**

前節で、cURLのインストール・動作テスト完了後以下の手順で実行する．

尚本節での説明にはjQuery（https://github.com/jquery/jquery）を使用する．

以下にはAPIで実際にデータを取得する方法を記載する．

|  |
| --- |
| curl "https://api.github.com/repos/jquery/jquery" |

以下のJSON形式のデータが帰ってくる．

|  |
| --- |
| {  "id": xxxxxxx,  "name": "Hello-World",  "full\_name": "xxxxx/Hello-World",  "owner": {  "login": "xxxxx",  ・  ・  ・  "site\_admin": xxxxx  },  "private": xxxxx,  ・  ・  ・  "subscribers\_count": xxxx  } |

尚、上のデータはjQueryのリポジトリデータである．

また、ログインをしてからAPIを取得することも可能である．方法は以下に記載する．

ログインしてからAPIを取得することによって、自分が所属しているプライベートリポジトリの情報を取得することもできる．

|  |
| --- |
| curl -s -u ユーザ名:パスワード "https://api.github.com/repos/jquery/jquery" |

しかし、これでは取得するたびにいちいちログイン情報を記載しなくてはならないので、手間を省くために、以下のコマンドを入力してGitHubのアカウント情報を保存する．

|  |
| --- |
| echo 'ユーザ名:パスワード' > github.passwd  chmod 600 github.passwd |

作業フォルダにGitHubのアカウント情報を記載した「github.passwd」というファイルを作成する．これを作成することによって、「-s -u ユーザ名:パスワード」の代わりに、「-s -u $(cat github.passwd)」と入力するだけでログインした状態でデータを取得するようになる．

また、「chmod」と言うコマンドはディレクトリやファイルのアクセス権限（パーミション）をコントロールするコマンドを使用して600番（ファイルは所有者以外一切の操作できない）を使用して、権限の変更を行う．

上記のアカウント情報を記載したファイルの作成とファイルへのアクセス権限を設定したら以下のコマンドを入力する

|  |
| --- |
| curl -s -u $(cat github.passwd) "https://api.github.com/repos/jquery/jquery" |

以上でログイン状態にてAPIを通して、JSON形式のデータ取得することができる．

同様に

|  |
| --- |
| curl -s -u $(cat github.passwd) "https://api.github.com/repos/{ユーザ名}/{リポジトリ名}/commits" |

にすると、指定したリポジトリのIssues情報が取得することができる．

また、以下のようにすると取得した結果から該当文字を検索して表示することもできる

|  |
| --- |
| curl -s -u $(cat github.passwd) "https://api.github.com/repos/{ユーザ名}/{リポジトリ名}/issues" | grep title |

しかしながら、標準では30件しか表示されないので、ループ処理を行わなければならない．それを自動で行うのが後述のプログラム（api.py）である．また、Pythonで製作されているので、Python・Pythonモジュール（easy\_install）をインストールしなければならない．

これら二つをインストールするには

|  |
| --- |
| sudo apt-get install python python-setuptools  sudo easy\_install requests |

以上の二つのコマンドを端末（ターミナル）に入力する．

**api.py**

|  |
| --- |
| #!/usr/bin/python  import sys, json, requests  tmp = open('github.passwd').readline().rstrip('\n').split(':');  username = tmp[0]  password = tmp[1]  url = sys.argv[1]  count = 0  while (url is not None):  r = requests.get(url, auth=(username, password))  print >> sys.stderr, r.headers['status'],  for item in r.json():  count = count + 1  print json.dumps(item)  if (r.links.has\_key('next')):  url = r.links['next']['url']  else:  url = None  print >> sys.stderr, count |

を作業用ディレクトリに保存し、

|  |
| --- |
| python api.py "https://api.github.com/repos/jquery/jquery/commits?per\_page=100" > commits.txt |

と入力することによって，JSON形式でコミットの情報をすべて取得することができる．

* + 1. **データの整理**

取得したデータはJSON形式なので、jq（http://stedolan.github.io/jq/）とawk言語を使用し、整理する．

|  |
| --- |
| ./jq '.commit.committer.date,.sha,.commit.message' commits.txt | awk '{ printf("%s", $0); if (NR % 3 == 0) printf("\n"); else printf(","); }' > commits.csv |

jqで指定した属性のデータの値を抜き、抜いたデータをawk言語がCSV形式に変換する仕組みである．

上記のコマンドでは「DateTime（投稿時間）・SHA（ハッシュ）・Message（コミット内容）」の3列をCSV形式でコミット履歴を整形しています．

# **本論**

本章では，これまでの調査を含めた研究方法と，現状の進捗における研究状況を報告する．

* 1. **研究方法**

本研究はGitHubにホスティングされているソフトウェアのテストコードの構成を調べ，それを解析するためのツールを作成する．そのツールを用いて，GitHubでホスティングされているプロジェクトを網羅的に調査し，現在のソフトウェア開発の実情を明らかにする．現在のソフトウェア開発の実情を調べる方法の例として，ソースコードを製品とテストに分け，それぞれの成長の様子を調べるということが考えられる．

* 1. **研究手順**

本研究の流れを以下に記す．

ツールを作成するための調査

解析するためのツールを作成

作成したツールをGitHubでホスティングされているオープンソースソフトウェアに使用

結果の考察

1. **ツールを作成するための調査**

ツールを作成するには様々な調査が必要である．GitHubでホスティングされているソフトウェアのソースコードを製品とテストに分けそれぞれの成長の様子を調べる方法の仮説を立て検証する．一方で成長の様子を取得するためのGitHub APIを使用するためにGitHub APIを調査する．

1. **解析するためのツールを作成**

本研究では解析するためのツールを作る際にGitHubのデータを用いるためにGitHub APIを採用する．GitHub APIにて必要なデータをGitHubでホスティングされているソフトウェアから取得させるプログラムを作成する．

1. **作成したツールを使用**

GitHubでホスティングされているソフトウェアを網羅的に無作為抽出（ランダムサンプリング）を行い，標本を抜く．無作為抽出で取得した標本に対して②で作成したツールを使用し解析する．

1. **結果の考察**

上記の手順で研究を行い，調査内容・解析が齎すデータの考察を行うことによって，近年のソフトウェア開発プロセスを明らかにし，テスト駆動開発の実情が明らかになることが期待される．

* 1. **研究対象**

　本研究の対象は以下の表の計37個を調査対象とした．

表 5‑1.本研究対象リスト

|  |  |
| --- | --- |
| https://github.com/jquery/jquery | jquery/jquery |
| https://github.com/joyent/node | joyent/node |
| https://github.com/mbostock/d3 | mbostock/d3 |
| https://github.com/angular/angular.js | angular/angular.js |
| https://github.com/jashkenas/backbone | jashkenas/backbone |
| https://github.com/blueimp/jQuery-File-Upload | blueimp/jQuery-File-Upload |
| https://github.com/mrdoob/three.js | mrdoob/three.js |
| https://github.com/adobe/brackets | adobe/brackets |
| https://github.com/jekyll/jekyll | jekyll/jekyll |
| https://github.com/hakimel/reveal.js | hakimel/reveal.js |
| https://github.com/moment/moment | moment/moment |
| https://github.com/visionmedia/express | visionmedia/express |
| https://github.com/Modernizr/Modernizr | Modernizr/Modernizr |
| https://github.com/LearnBoost/socket.io | LearnBoost/socket.io |
| https://github.com/jashkenas/underscore | jashkenas/underscore |
| https://github.com/plataformatec/devise | plataformatec/devise |
| https://github.com/less/less.js | less/less.js |
| https://github.com/ariya/phantomjs | ariya/phantomjs |
| https://github.com/jashkenas/coffee-script | jashkenas/coffee-script |
| https://github.com/discourse/discourse | discourse/discourse |
| https://github.com/defunkt/jquery-pjax | defunkt/jquery-pjax |
| https://github.com/bower/bower | bower/bower |
| https://github.com/mozilla/pdf.js | mozilla/pdf.js |
| https://github.com/gruntjs/grunt | gruntjs/grunt |
| https://github.com/caolan/async | caolan/async |
| https://github.com/Prinzhorn/skrollr | Prinzhorn/skrollr |
| https://github.com/madrobby/zepto | madrobby/zepto |
| https://github.com/scottjehl/Respond | scottjehl/Respond |
| https://github.com/janl/mustache.js | janl/mustache.js |
| https://github.com/maker/ratchet | maker/ratchet |
| https://github.com/visionmedia/jade | visionmedia/jade |
| https://github.com/rstacruz/nprogress | rstacruz/nprogress |
| https://github.com/twitter/typeahead.js | twitter/typeahead.js |
| https://github.com/thoughtbot/paperclip | thoughtbot/paperclip |
| https://github.com/xing/wysihtml5 | xing/wysihtml5 |
| https://github.com/Shopify/dashing | Shopify/dashing |
| https://github.com/resque/resque | resque/resque |

* 1. **研究ツール**

本節では，研究ツールの仕組みを記述する．

* + 1. **データフローチャート**

以下は本研究のデータフローチャートである．

****

図 5‑1.データフローチャート図

* + 1. **プログラム概要図**

以下は本研究で実際に作ったプログラムの詳細である．

**GitHub**



**プロジェクト**

**データ**



**プロジェクト**

**データ**



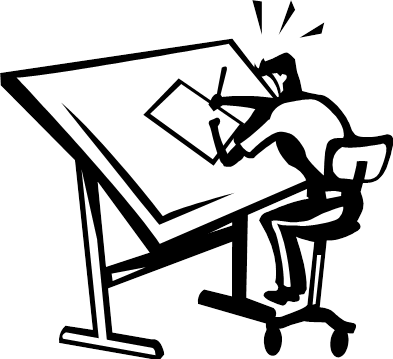
**コミット情報CSV**

**（HASH・時刻）**



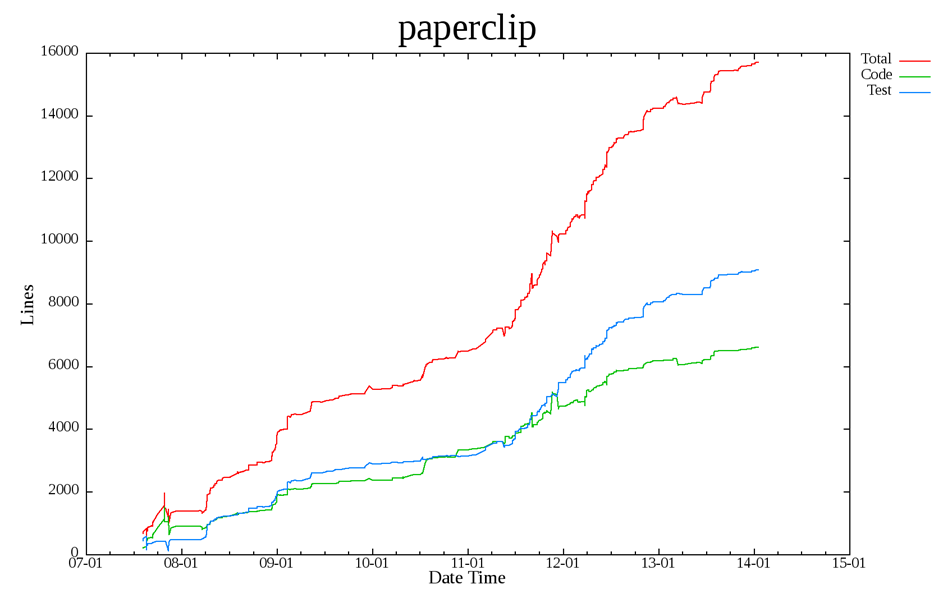
**コミット情報CSVχ**

**（HASH・時刻・メイン/テストコード）**



**グラフ作成ツール**

**結果**



**①**

**②**

**③**

**④**

**⑤**

**⑥**

**コードカウントツール**

**ローカル環境**

図 5‑2.本研究のプログラム概要図

1. 「git clone」を使用してプロジェクトデータのリクエストをGitHubに送る．
2. GitHubからプロジェクトデータをローカルに取得
3. 取得したプロジェクトデータからコミット情報（HASH・時刻）を取得し、CSV形式で出力する．
4. ③で取得したCSV形式のデータにコードカウントツールを使用して、コミットされた状態にロールバックさせ、メインコード（全体の行数）とテストコード（testディレクトリ内の行数）のバイナリデータを除き、カウントをする．
5. ③で取得したCSV形式のデータに④で取得した、メインコードとテストコードのデータを追加する．
6. ⑤で更新されたCSV形式のデータをグラフ作成ツールにポストし、グラフを作成する．
   * 1. **実際のコード**

本プログラムのコードは以下の二つから成り立つ．

* + - 1. lineCounter.shのコード説明

|  |
| --- |
| git clone https://github.com/$1/$2  cd $2  git log --pretty=format:"%H,%cd" --date=iso --first-parent --no-merges > $2-commits.csv  cd ..  cat $2-commits.csv | python **lineCountScriptCreator.py** $2 > $2-count.sh  bash $2-count.sh > $2-count-result.csv 2> $2-error.log  rm -f count-result.csv  cp $2-count-result.csv count-result\_tmp.csv  awk -F ',' 'BEGIN{OFS=","} {print $0,$3-$4;}' count-result\_tmp.csv > count-result.csv  rm -f count-result\_tmp.csv  gnuplot <<EOF  set datafile separator ","  set xdata time  set timefmt "%Y-%m-%d %H:%M:%S"  set format x "%y-%m"  set xl "Date Time" font 'Times New Roman,25'  set yl "Lines" font 'Times New Roman,25'  set terminal png size 1680,1050  set lmargin 14  set rmargin 15  set tmargin 5  set border 15 lw 2  set tics font "Times New Roman,20"  set title "$2"  set title font 'Times New Roman,50'  set key outside font 'Times New Roman,20' spacing 1.5  set out "lines.png"  plot 'count-result.csv' using 2:3 with lines lw 2 title 'Total', 'count-result.csv' using 2:5 with lines lw 2 title 'Code','count-result.csv' using 2:4 with lines lw 2 title 'Test'  EOF  rm -f $2-lines.png  mv lines.png $2-lines.png |

|  |
| --- |
| git clone **<URL>** |

「git clone」とは、リポジトリを複製（ローカル環境へコピー）するためのコマンドである．オプション指定をすることによって、以下のようなことができる．

* ディレクトリ名の指定

git clone <URL> <ディレクリ名>

* 最新リビジョンのみの取得

git clone –depth 1 <URL>

※git cloneはリポジトリの履歴情報までも含めてリポジトリを複製するため、大規模なプロジェクトの場合は多くのファイルを取得・コピーしなければならないことがある．最新のソースファイルを取得する目的のみで使用する場合は、「--depth」オプションで取得するとよい．

|  |
| --- |
| cd **<PASS>**  ・  ・  cd .. |

「cd」とは、指定したディレクトリに移動することができる．<PASS>には相対パス（現在位置から、目的のファイルやフォルダまでの道筋）・絶対パス（最上位階層から目的のファイルやフォルダまでのすべての道筋）のどちらでも使用できる．以下の記号を使うことによって、ディレクトリ操作をすることもできる．

* ルート・ディレクトリ(システム上最上層のディレクトリ)

cd /

* 親ディレクトリ(一階層上のディレクトリ)

cd ..

* ホームディレクトリ（ユーザ上最上層ディレクトリ）

cd ~/

|  |
| --- |
| git log **<オプション>** |

「git log」とはリポジトリの履歴情報を見るコマンドである．デフォルトでは直近のコミットしか見られないため、オプションを使うことによってさまざまな履歴情報を見ることができる．

また、最後に「 > “ファイル名”」と入れることによって、情報を書き出すことができる．

* 各コミットのパッチを表示する

-p -<表示する数>

* 変更点を単語単位で表示する

--word-diff

* 各コミットで変更されたファイルの統計情報を表示する

--stat

* ブランチ指定

--first-parent

* SHA-1 チェックサムの全体 (40文字) ではなく最初の数文字のみを表示する

--abbrev-commit

* コミット情報の後に変更されたファイルの一覧を表示する

--name-only

* 完全な日付フォーマットではなく、相対フォーマット (“2 weeks ago” など) で日付を表示する

--relative-date

* マージコミットの除外

--no-merges

* 検索

--grep <ワード>

* コミットを別のフォーマットで表示する．

--pretty=format:"<オプション記号>"

|  |  |
| --- | --- |
| %H | コミットのハッシュ |
| %h | コミットの短縮ハッシュ |
| %T | ツリーのハッシュ |
| %t | ツリーの短縮ハッシュ |
| %P | 親のハッシュ |
| %p | 親の短縮ハッシュ |
| %an | Authorの名前 |
| %ae | Authorのメールアドレス |

|  |  |
| --- | --- |
| %ad | Authorの日付 |
| %ar | Authorの相対日付 |
| %cn | Committerの名前 |
| %ce | Committerのアドレス |
| %cd | Committerの日付 |
| %cr | Committerの相対日付 |
| %s | 件名 |

|  |
| --- |
| cat **<ファイル名.csv>** | python lineCountScriptCreator.py **<リポジトリ名>** > **<ファイル名.sh>** |

「cat」とは、ファイルを開くコマンドである．また、「|（パイプ）」はCATで開いた内容をPython言語で作成された「lineCountScriptCreator.py」にスルーするという意味である．lineCountScriptCreator.pyはGitの履歴情報から一つ一つロールバックさせて行き、ロールバックするたびにソースコードの行数をカウントするスクリプトを作成するツールである．lineCountScriptCreator.pyについては後程説明をする．

|  |
| --- |
| bash **<lineCountScriptCreator.pyにて作成されたスクリプト名.sh>** > **<リポジトリ名.csv>** 2 > **<ファイル名-error.log>** |

「bash」とは、ユーザによってキーボードから入力された文字や、マウスのクリックなど与えられた指示をOSの中核部分に伝え、対応した機能を実行するようにOSに指示を伝える．今回ではlineCountScriptCreator.pyにて作成されたスクリプトを実行し、結果をCSV形式に書き出し、エラーがあればエラーログに書き出すという意味である、」

|  |
| --- |
| cp **<ファイル名.csv>**  **<ファイル名\_tmp.csv>** |

「cp」とは、指定したファイルをコピーするコマンドである．今回では元ファイルを残すために使用している．

|  |
| --- |
| awk -F ',' 'BEGIN{OFS=","} {print $0,$3-$4;}' **<ファイル名\_tmp.csv>** > **<ファイル名.csv>** |

「awk」とは、CSV形式の文字ファイルの処理を行う際に用いられる言語であり、awkは渡されたファイルの一行ずつ処理を行う．上記の例だと、「-F」はCSV形式の区切りの文字列を表している．「BEGIN」は、読み込み開始とともに実行するという意味である．「OFS」は出力時の区切りを設定している．printは出力する内容を表している．print内で「$0」というのは元のデータをそのまま出力という意味である．また、「$3-$4」というのは「三列目－四列目（コード合計－テストコード）」を表している．前者のファイル名には整理したファイル名、後者には整理後の保存したいファイル名を入力する．

|  |
| --- |
| rm -f **<count-result\_tmp.csv>** |

「rm」とは、ファイルやディレクトリを削除するコマンドである．-ｆというのは警告メッセージを表示しないオプションである．

|  |
| --- |
| gnuplot <<EOF  ・  ・  EOF |

「EOF」とは、<<EOFからEOFまでGnuplot（グラフ作成ツール）で実行する．という意味である．

|  |
| --- |
| set datafile separator ","  set xdata time  set timefmt "%Y-%m-%d %H:%M:%S"  set format x "%y-%m"  set xl "Date Time" font 'Times New Roman,25'  set yl "Lines" font 'Times New Roman,25'  set terminal png size 1680,1050  set lmargin 14  set rmargin 15  set tmargin 5  set border 15 lw 2  set tics font "Times New Roman,20"  set title "$2"  set title font 'Times New Roman,50'  set key outside font 'Times New Roman,20' spacing 1.5  set out "lines.png"  plot 'count-result.csv' using 2:3 with lines lw 2 title 'Total', 'count-result.csv' using 2:5 with lines lw 2 title 'Code','count-result.csv' using 2:4 with lines lw 2 title 'Test' |

上記の部分はGnuplotのコンフィグである．

* datafile

入力データの区切りの文字列を指定．

* xdata

Xデータの種類を指定．

* timefmt

入力データの時間表記書式を指定

* format

出力データの時間表記書式を指定

* xl・yl

X・Y軸のラベル名を指定

* terminal/size

ターミナルで出力する画像形式を指定して、サイズで出力サイズを指定

* <オプション記号>margin

オプション記号（詳細は以下に表形式で記載する）で設定した位置のグラフ外のスペースを指定する．

|  |  |
| --- | --- |
| **記号** | **効果** |
| l | 左側余白の設定をする． |
| r | 右側余白の設定をする． |
| t | 上側余白の設定をする． |
| b | 下側余白の設定をする． |

* border/lw

グラフの外枠の太さを指定している．

また、枠には12ビットの整数が与えられている．特定箇所の枠を設定したい場合は，その枠の数字を加算して表記する．本プログラムでは15と表記されているので「**1+2+4+8＝15**（下+左+上+右＝上下左右）」となる．

|  |  |  |
| --- | --- | --- |
| **ビット** | **plot効果** | **splot効果** |
| 1 | 下 | 底面の左手前 |
| 2 | 左 | 底面の左後ろ |
| 4 | 上 | 底面の右手前 |
| 8 | 右 | 底面の右後ろ |
| 16 | - | 垂直面の左 |
| 32 | - | 垂直面の後ろ |
| 64 | - | 垂直面の右 |
| 128 | - | 垂直面の手前 |
| 256 | - | 天井面の左後ろ |
| 512 | - | 天井面の右後ろ |
| 1024 | - | 天井面の左手前 |
| 2048 | - | 天井面の右手前 |

* tics font

見出し目盛のフォント・サイズを「，」で区切り指定

* title

指定されたタイトルを挿入する．本プログラムでは$2にはリポジトリ名が変数として代入されているため、調査対象リポジトリ名が代入されるようになっている．

* title font

タイトル目盛のフォント・サイズを「，」で区切り指定

* key <オプション> font spacing

凡例の表示設定を行っている．オプションについては以下の表のとおりである．

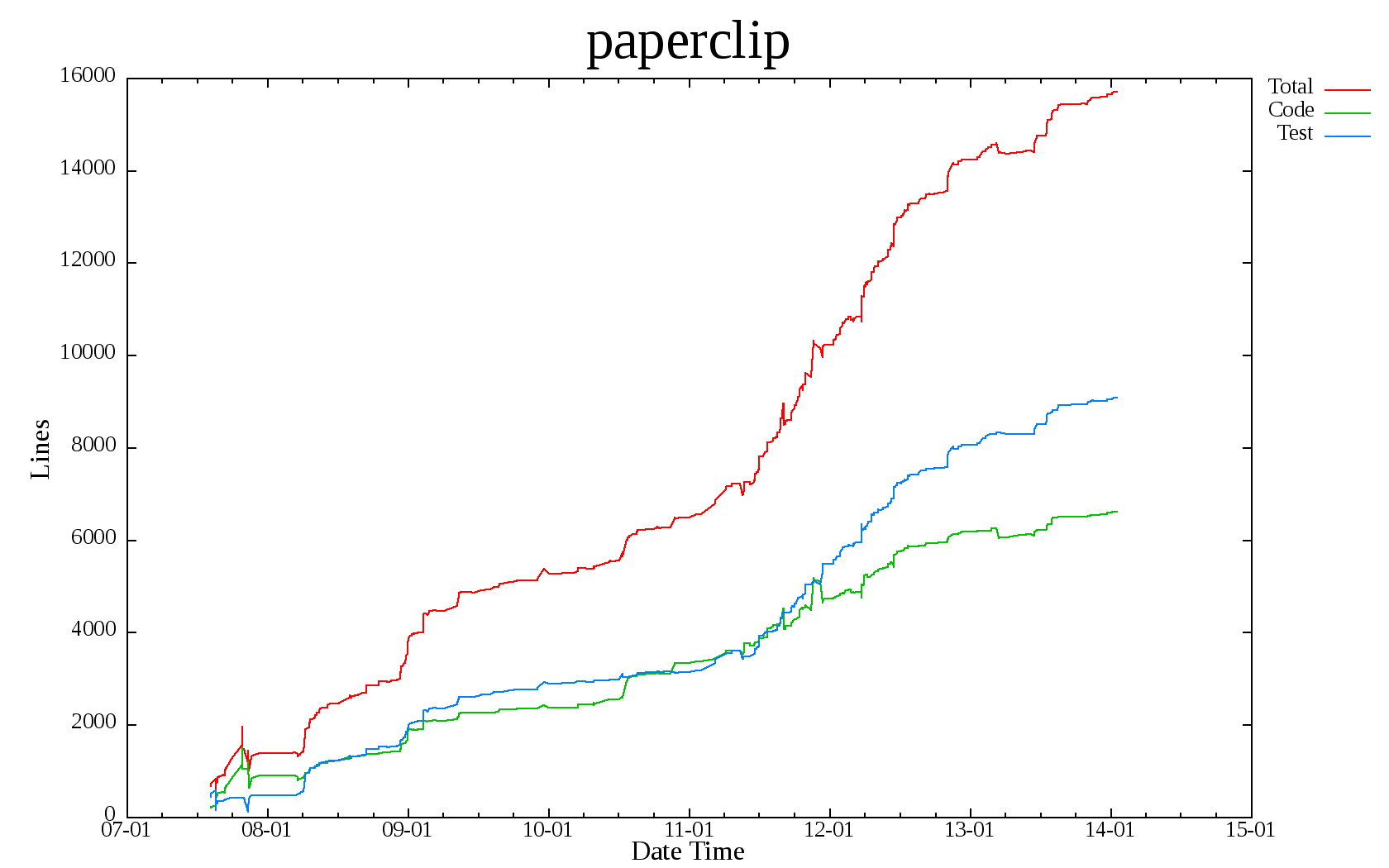
|  |  |
| --- | --- |
| **値** | **効果** |
| left | 左側に表示 |
| right | 右側に表示 |
| top | 上側に表示 |
| bottom | 下側に表示 |
| outside | 図の右側の外に表示 |
| below | 図の下側の外に表示 |

* plot

二次元のグラフを描画するコマンド．

1. 使用するファイル名をシングルクォーテーションで囲み、指定する．（例：'count-result.csv'）
2. using X:YのようにX軸とY軸の値を設定する（例：using 2:3）
3. with lines lw \*を使って線の太さを設定する（例：with lines lw 2）
4. title '×××'のように値にラベルを付ける．（例：title 'Test'）
5. 他の値を入力する場合「,」で区切り1-4同様の式を作成する．

本プログラムでは以下のようなグラフを作成することができる．



* + - 1. lineCountScriptCreator.pyのコード説明

|  |
| --- |
| from datetime import datetime  from dateutil.parser import parse  from dateutil import tz  import sys  myProject = sys.argv[1].replace("'", "\\'")  print("rm %s-error.log") % (myProject)  for line in sys.stdin:  x = line.split(',')  myHash = x[0]  #git logの結果はタイムゾーンがばらばら．すべてUTCに統一する．  myDate = datetime.strftime(parse(x[1]).astimezone(tz.gettz('UTC')), '%Y-%m-%d %H:%M:%S')  print("echo %s >&2") % (myHash)  print("cd %s") % (myProject)  print("git checkout -f %s 2>> ../%s-checkout-error.log") % (myHash, myProject)  print("cd ..")  print("if [ -e %s/test ]; then") % (myProject)  print(" echo %s,%s,$(grep -rI '' '%s' | grep -v '^%s/\.git/' | wc -l),$(grep -rI '' '%s/test' | wc -l)") % (myHash, myDate, myProject, myProject, myProject)  print("else")  print(" echo %s,%s,$(grep -rI '' '%s' | grep -v '^%s/\.git/' | wc -l),0") % (myHash, myDate, myProject, myProject)  print("fi") |

|  |
| --- |
| from datetime import datetime  from dateutil.parser import parse  from dateutil import tz  import sys |

指定されたモジュールの中から指定した巻子や変数を読み込みます．うえから、

「datetime」モジュールから「datetime」を取り込む．

「dateutil.parser」モジュールから「parse」を取り込む．

「dateutil」モジュールから「tz」を取り込む．

「sys」モジュールを使えるようにする．

|  |
| --- |
| myProject = sys.argv[1].replace("'", "\\'") |

渡された引数（今回はリポジトリ名）を「 ' 」から「\\'」に置換してmyProject変数に保存する．

|  |
| --- |
| print("rm %s-error.log") % (myProject) |

“リポジトリ名”-error.logの削除を行う

|  |
| --- |
| for line in sys.stdin: |

　for ~ fi　までの間の処理を、読み込んだファイルの一行一行処理を繰り返す．

|  |
| --- |
| x = line.split(',')  myHash = x[0] |

　CSV形式には「Hash,Date」と別れているので「 , 」xに配列を使って変数として入れる．そして、MyHash変数にx[0]すなわち、Hashを代入している処理．

|  |
| --- |
| myDate = datetime.strftime(parse(x[1]).astimezone(tz.gettz('UTC')), '%Y-%m-%d %H:%M:%S') |

git logで取得したデータの時間は投稿した人の国のタイムゾーンが記載されているので全てをUTC（協定世界時）に変換してmyDate変数に代入。

例）2014-01-17 00:38:56 -0500 → 2014-01-17 05:38:56

|  |
| --- |
| print("echo %s >&2") % (myHash)  print("cd %s") % (myProject)  print("git checkout -f %s 2>> ../%s-checkout-error.log") % (myHash, myProject)  print("cd ..") |

エラーログを出力している。

|  |
| --- |
| print("if [ -e %s/test ]; then") % (myProject)  print(" echo %s,%s,$(grep -rI '' '%s' | grep -v '^%s/\.git/' | wc -l),$(grep -rI '' '%s/test' | wc -l)") % (myHash, myDate, myProject, myProject, myProject)  print("else")  print(" echo %s,%s,$(grep -rI '' '%s' | grep -v '^%s/\.git/' | wc -l),0") % (myHash, myDate, myProject, myProject)  print("fi") |

IF[条件式]…Aの処理…else…Bの処理という構文

条件式はリポジトリにtestディレクトリの中にフォルダがあるかないかを判別している．testディレクトの中にファイルがある場合はAの処理を行う．testディレクトの中にファイルがない場合はBの処理を行う．

処理をまとめるとAでは、「Hash，Date，.gitフォルダ（履歴情報）を除いた行数，testディレクトリの行数」を出力という処理を行っている．

Aの処理だけでは、testディレクトリの中にファイルがない場合にエラーが出てしまい、正常にカウントできないので、testディレクトリの中にファイルがない場合は「testディレクトリの行数」を0と出力するようになっている．

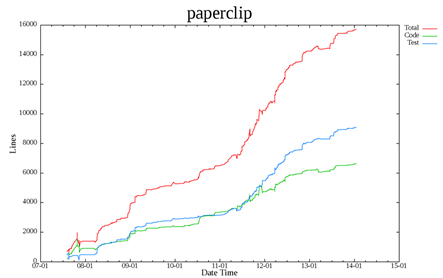
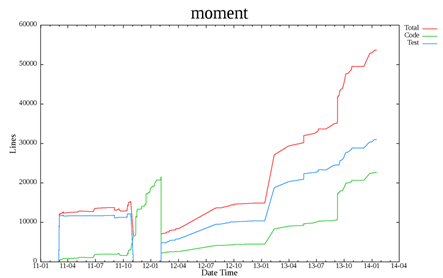
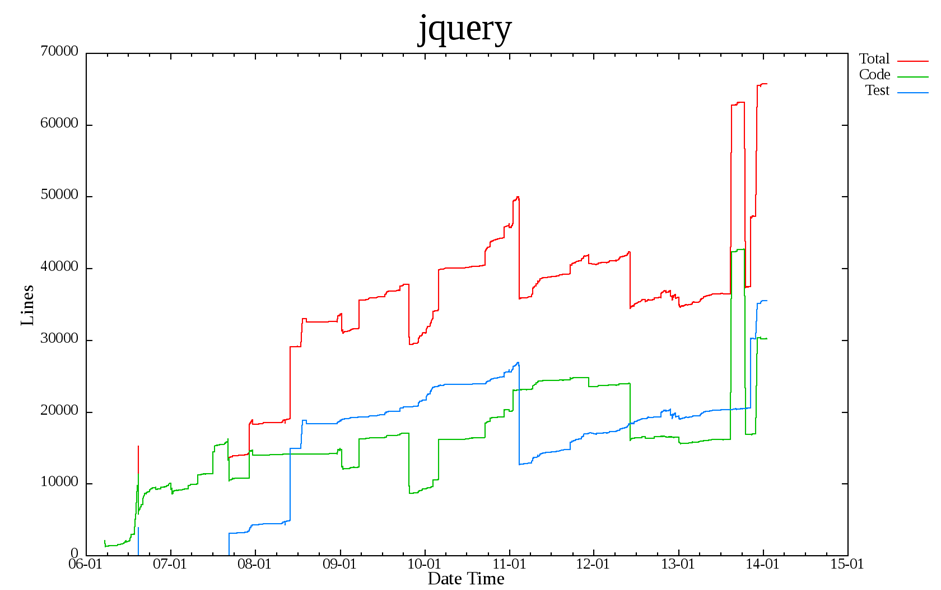
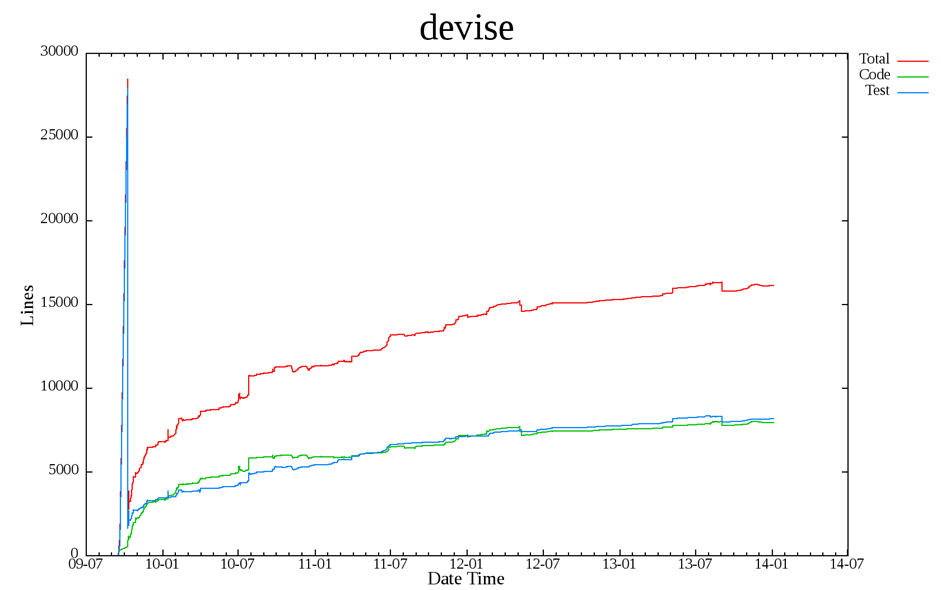
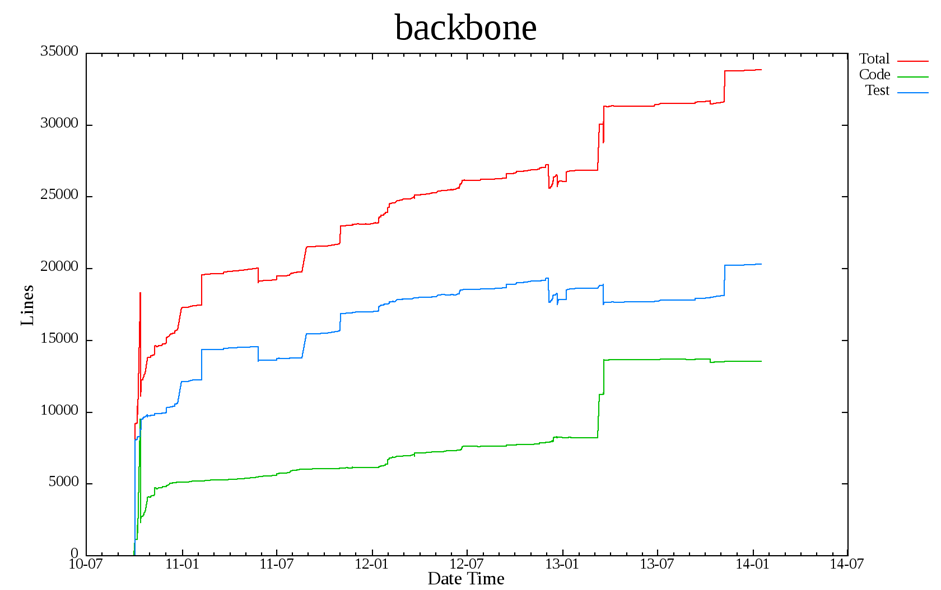
# **考察**

本章では，これまでの調査方法を用いて，GitHub内のソフトウェア開発プロジェクトかたコミットの変更履歴を調査した結果・考察を記述する．

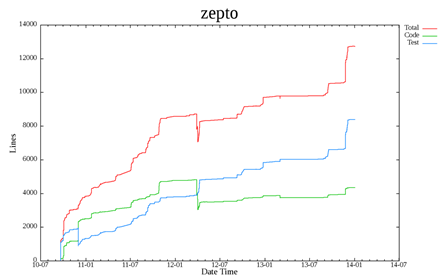
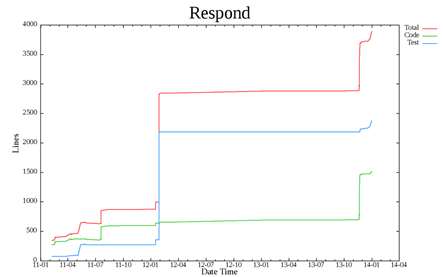
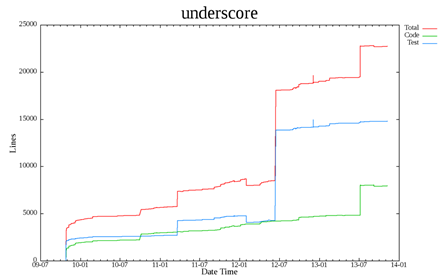
* 1. **研究結果**

研究の結果，得られたグラフを主に以下の四種類に分類し，突発的に変動を起こしている部分のコミットを変更履歴から調査し，得られた結果を記載する。

1. メインコードよりテストコードのほうが多い
2. メインコードとテストコードが共に成長する
3. テストコードが殆ど書かれていない
4. テストコードが初期状態から殆ど変化がない
   * 1. **メインコードよりテストコードの方が多い傾向**



メインコードよりテストコードの方が多い傾向続き

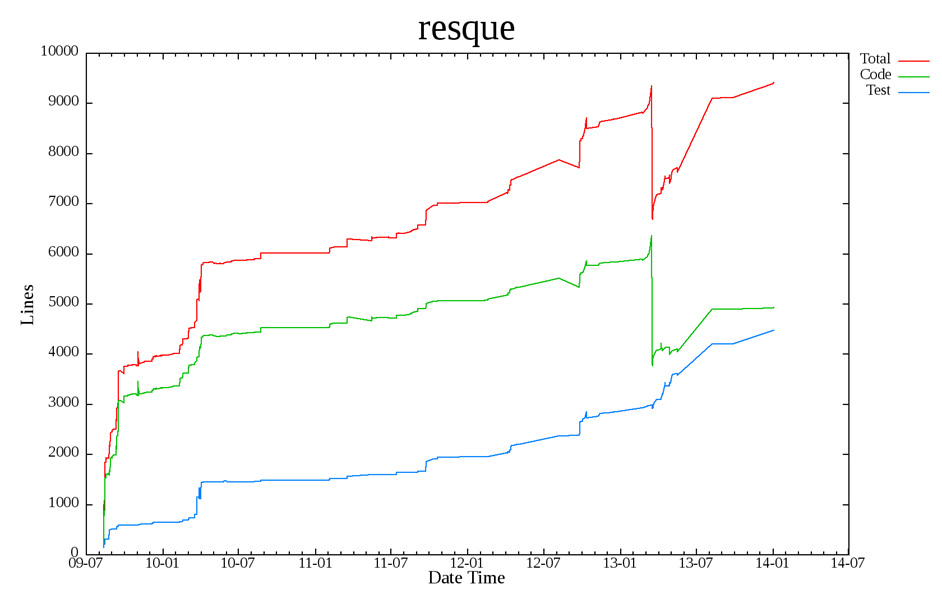
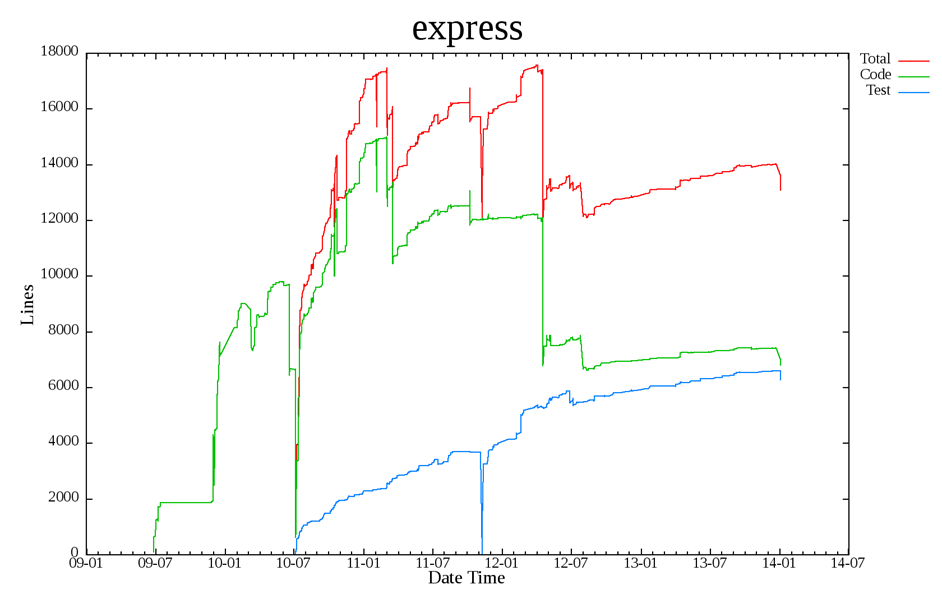
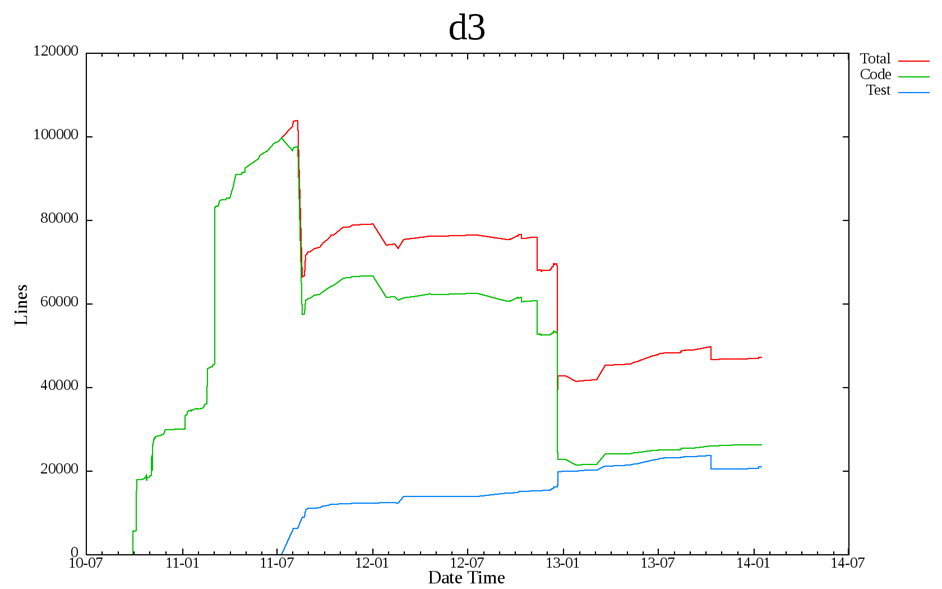
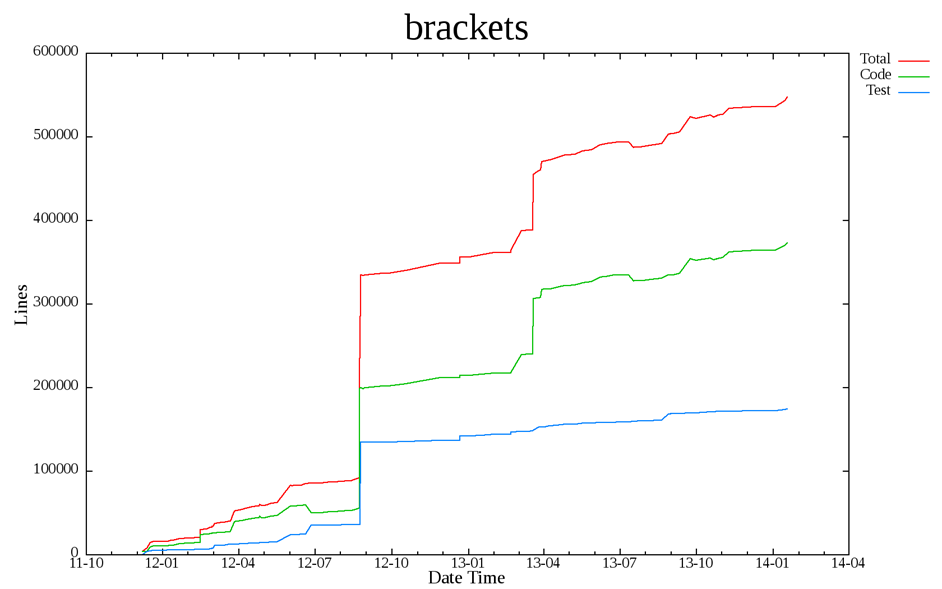
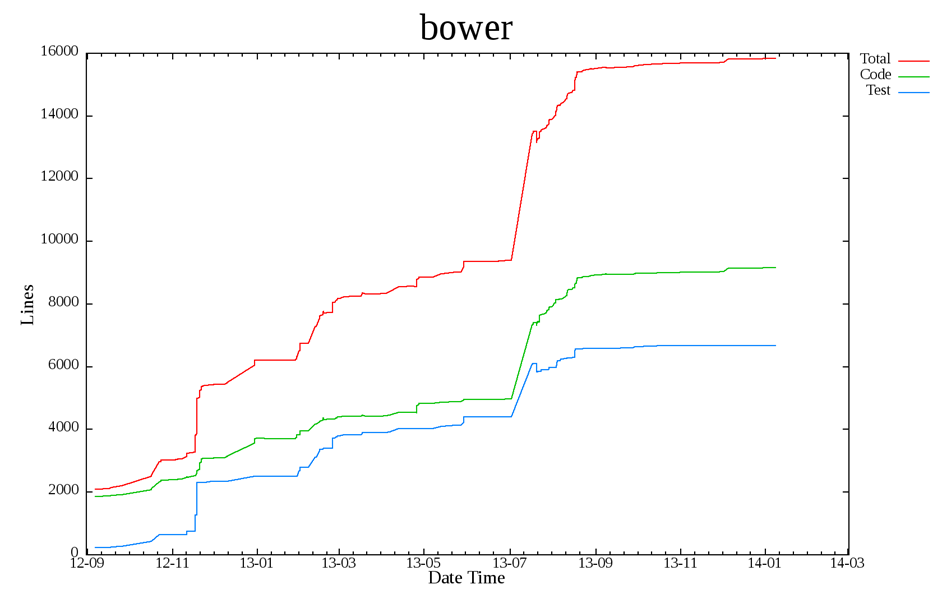
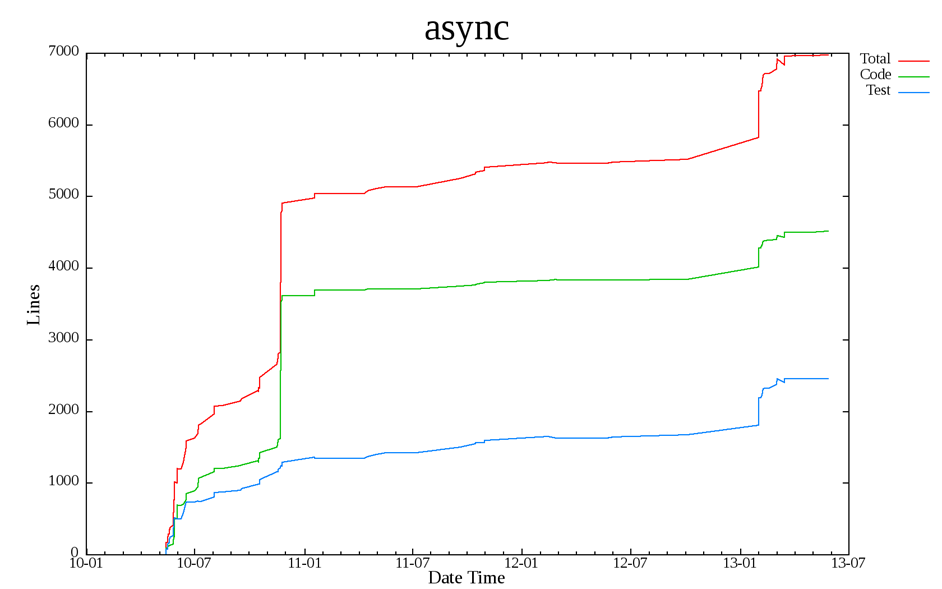


メインコードよりテストコードの方が多い傾向は37プロジェクト中，9プロジェクトが該当する．

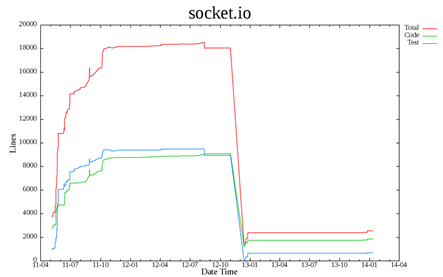
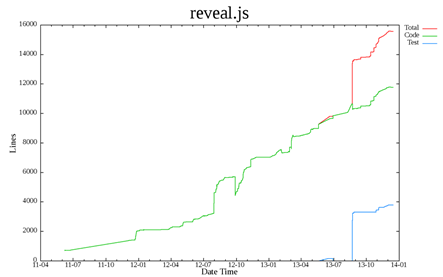
momentではプロジェクトでは丸印の位置にて，「testをメインコード内に移動」させたのでtestコードが検知されなくなってしまったが，その後「構造の再編成し、ビルドとテストプロセスの分割」を行ったため、メインコードが減り、テストコードが増えたとコミットに記載されていた情報から分かった．

Respondでは，一つ目の丸で「Qunitをローカル環境へ移した」ことが分かった．Qunitとは，JavaScriptで単体テストを自動化にて行うためのフレームワークである．また、二つ目の丸では機能の追加を行ったのとクロスドメインへ対応になったとコミットに記載されていた情報から分かった．

* + 1. **メインコードとテストコードがともに成長する傾向**



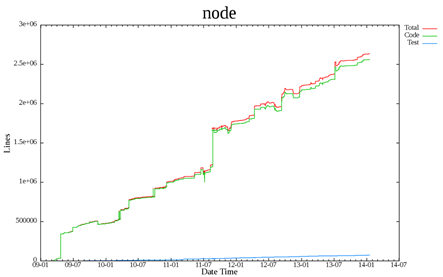
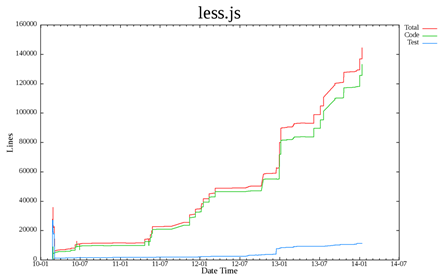
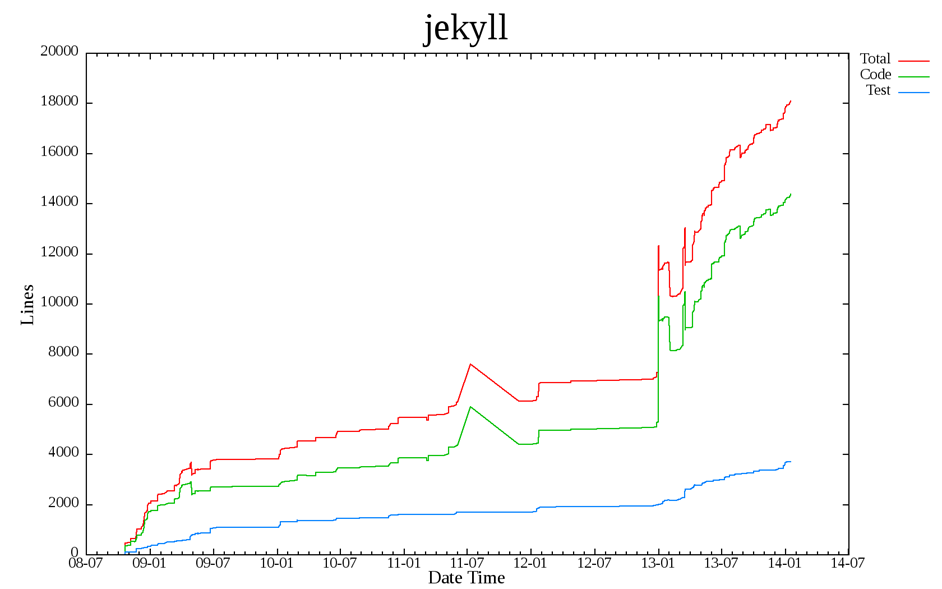
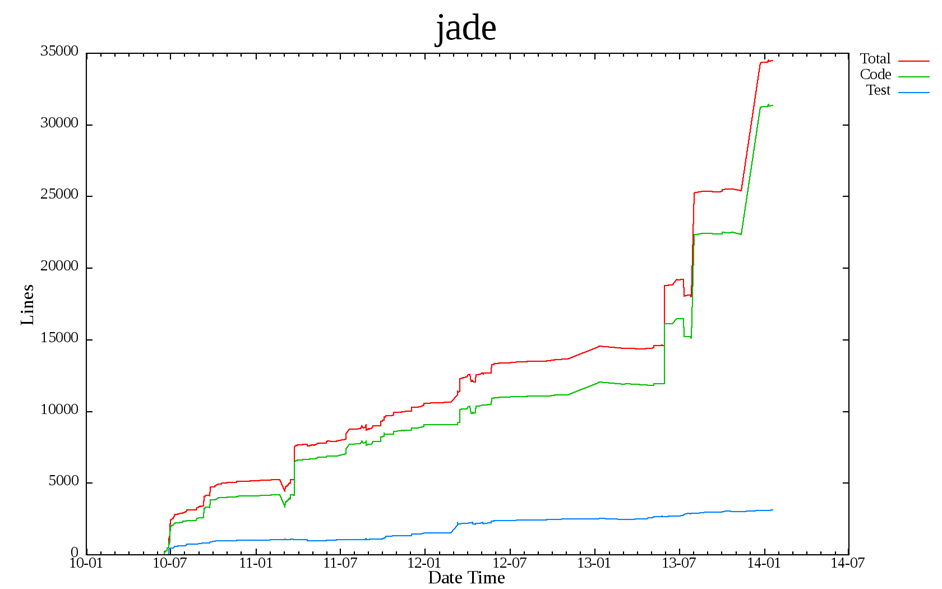
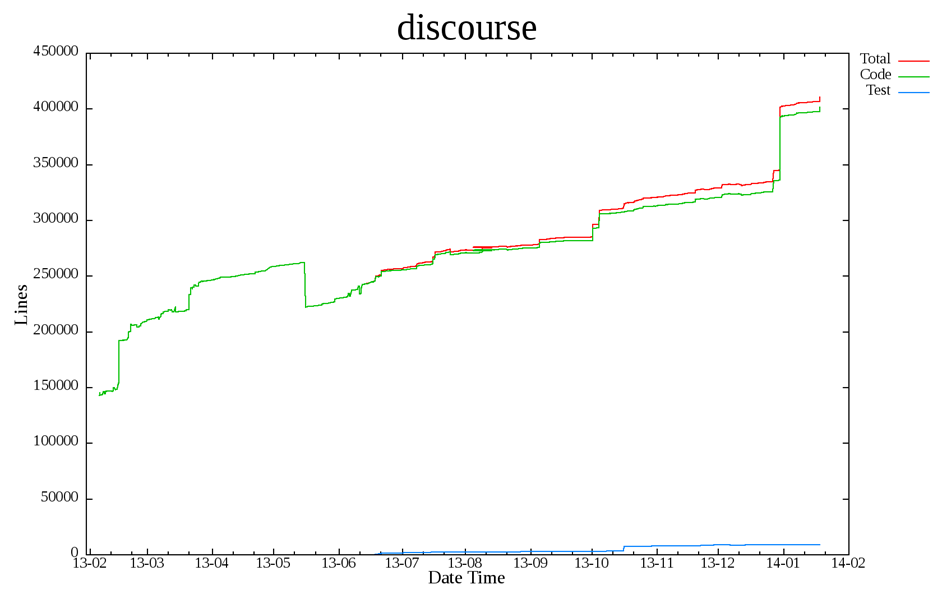
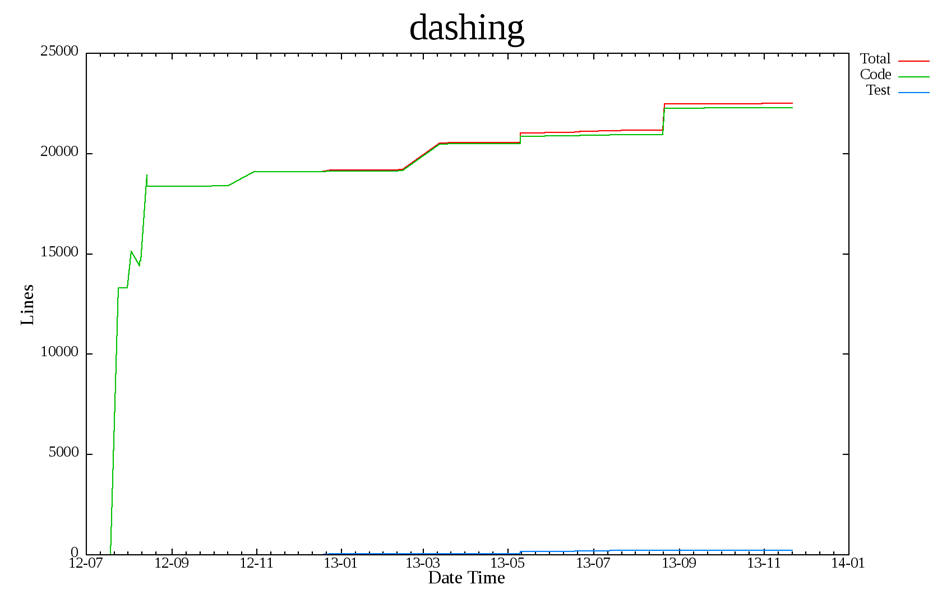
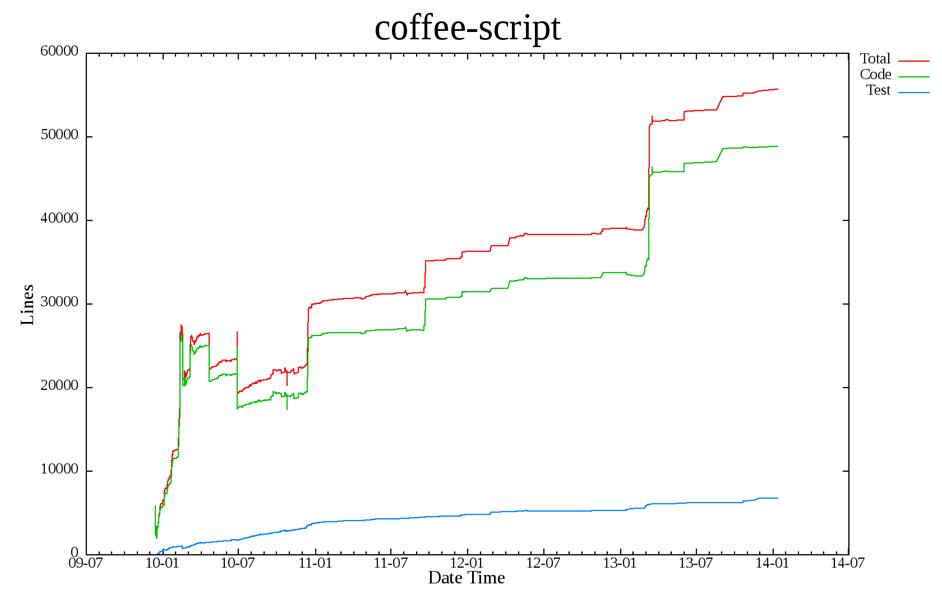
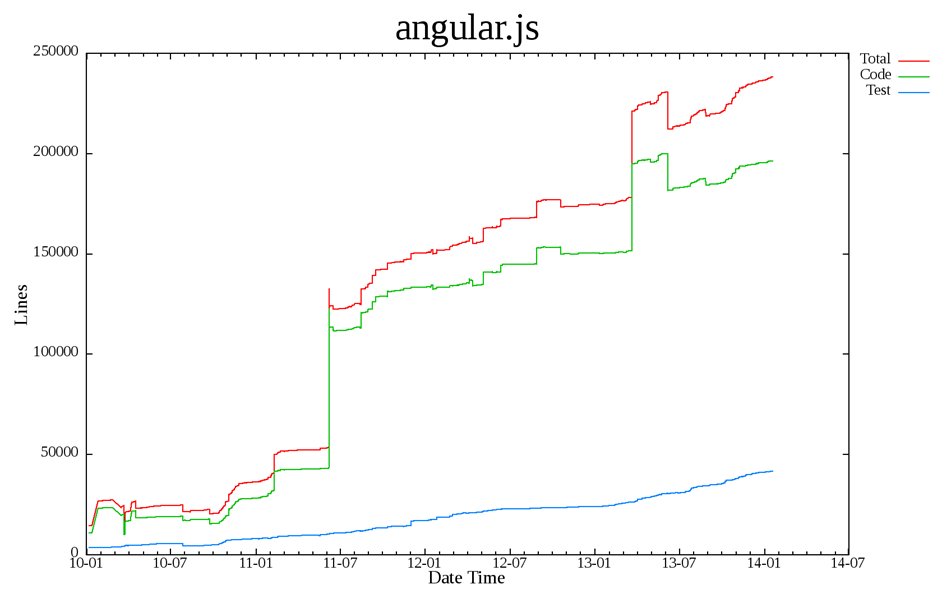
メインコードとテストコードがともに成長する傾向の続き



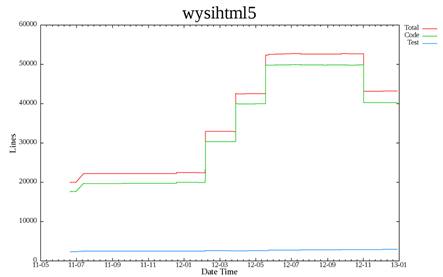
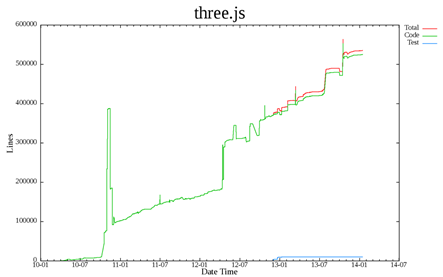
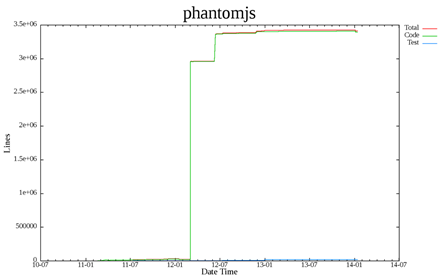
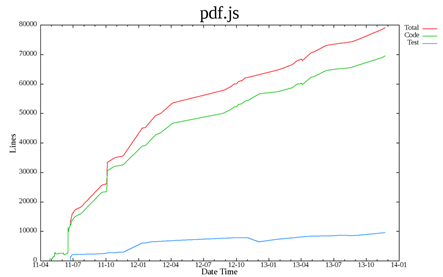
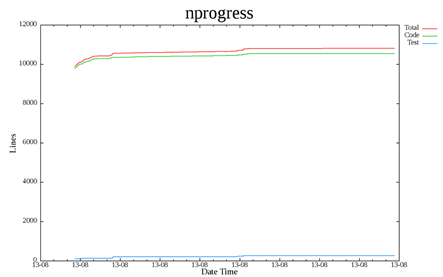
メインコードとテストコードがともに成長する傾向は37プロジェクト中，8プロジェクトが該当する．

d3では，一つ目の丸で「env-jsをライブラリに追加」していたことが分かった．env-jsとは，JavaScriptで書かれたブラウザ環境のシミュレーションである．また、二つ目の丸では更新していく過程で不要になったファイルを削除したものだとコミットに記載されていた情報から分かった．

* + 1. **テストコードが殆ど書かれていない傾向**



テストコードが殆ど書かれていない傾向続き

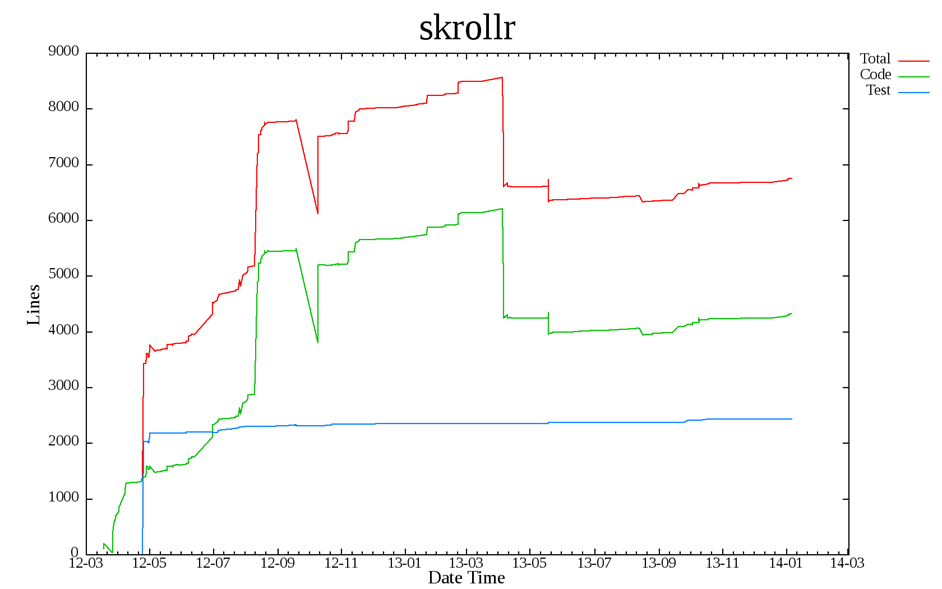
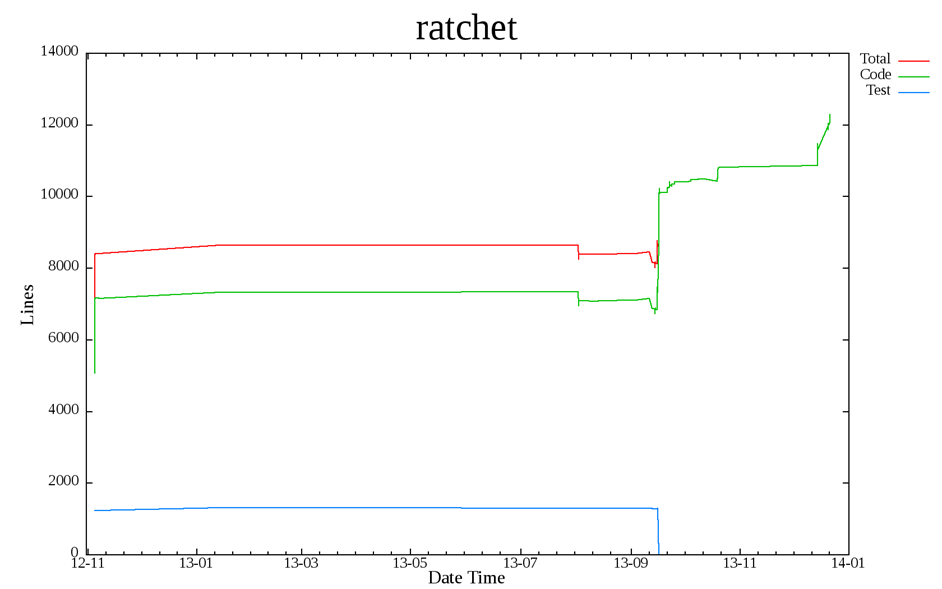
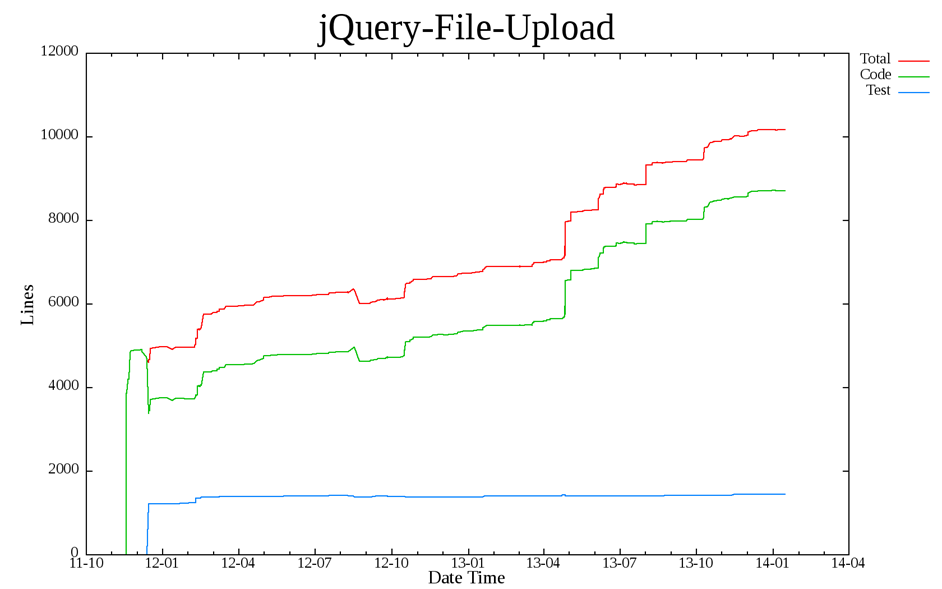


テストコードが殆ど書かれていない傾向は37プロジェクト中，13プロジェクトが該当する．

angular.jsでは，一つ目の丸で「チュートリアルの追加」をしていたことが分かった．また、二つ目の丸では「datetimesymbolsext.jsを取り込んだ」とコミットに記載されていた情報から分かった．

phantomjsでは，「Qtを追加」していたことがコミットに記載されていた情報から分かった．またQtとは，C++で開発されたUIフレームワークである。

* + 1. **テストコードが初期状態から殆ど変化がない傾向**

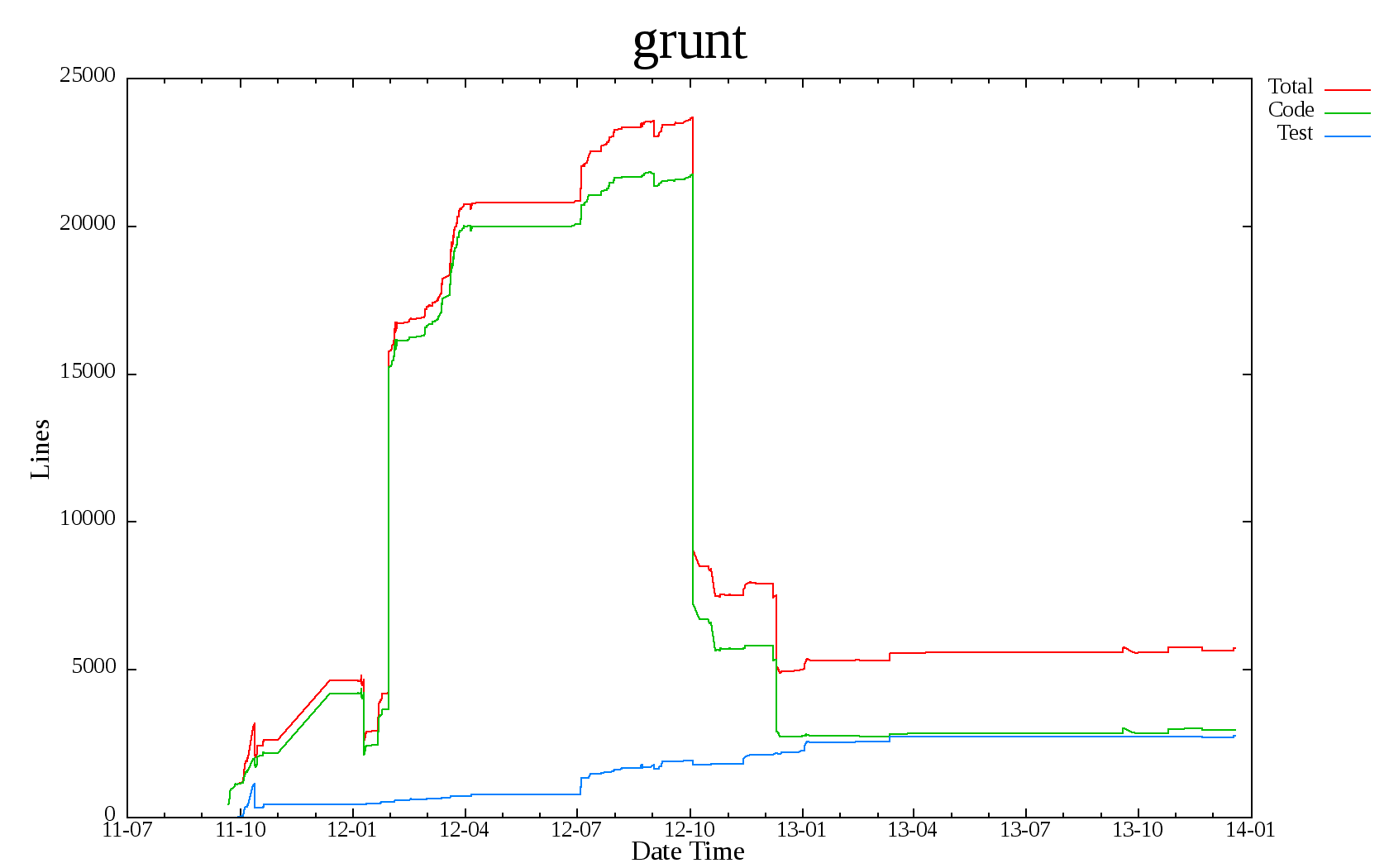


テストコードが初期状態から殆ど変化がない傾向は37プロジェクト中，8プロジェクトが該当する．

* + 1. **その他**

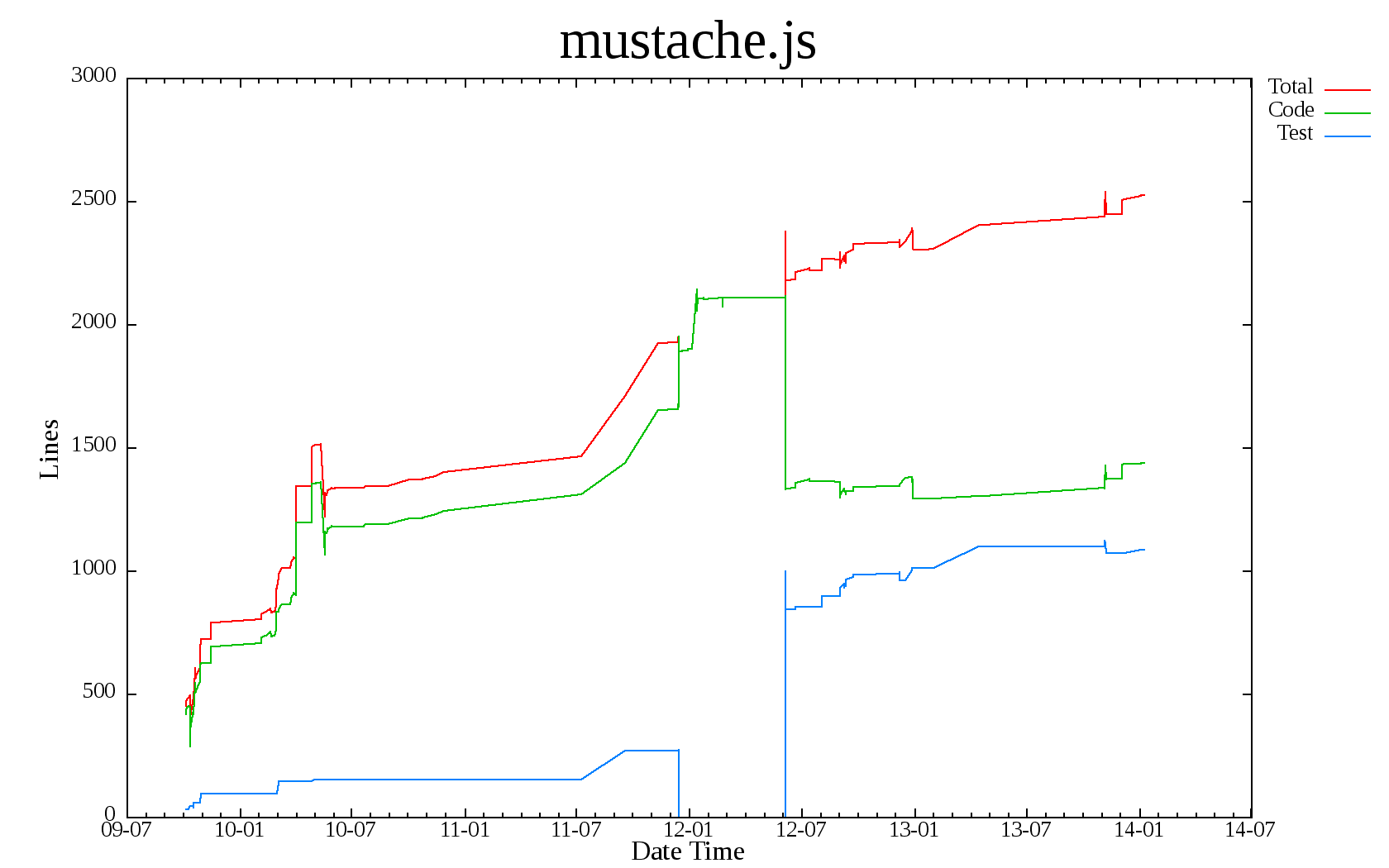
上記の傾向当てはまらなかったのは以下の37プロジェクト中，3プロジェクトである．

* + - 1. **gruntプロジェクトの場合**



gruntでは，「プロジェクトの分割化」をしていたことが分かった．もともとgruntの中に入っていた機能を別にして一つのプロジェクトgrunt-initに分割されていた．

* + - 1. **mustache.jsプロジェクトの場合**

****

mustache.jsでは，一つ目の丸で「testをメインコード内に移動」させたのでtestコードが検知されなくなってしまった。また二つ目の丸では、「コード間の負荷の分散・構造編成や関数の整理など」をしていたことがコミットに記載されていた情報から分かった．

* 1. **考察**

　実験結果より，ソフトウェア開発プロジェクトの成長の様子を調査することができた．

また，一般的なソフトウェア開発ではウォーターフォール型開発手法を採用しているプロジェクトは少ない．何故ならば、ウォーターフォール型開発手法を採用していれば，今回の結果ではコードの変動は大きく、変動後は細かい訂正を行うため、多少ギザギザ変動するが、そのような動きをしているプロジェクトは少なかった。

また，最近ではテスト駆動開発手法（Test Driven Development）が採用されていると言われているが、そもそもテストコードがないプログラムや、テストコードが伸びていないのに、メインコードが急激に伸びているソフトウェア開発プロジェクトもあった。

テスト駆動開発手法（Test Driven Development）がよく採用される．これは，先にプログラムに必要な機能について初めにテストを考案し，そのテストが動作する必要最低限な実装を行い，コードを洗練させ，短い工程で計画・開発を繰り返し行う手法である．

# **参考文献**

[1] Mint（経営情報研究会). 図解でわかる ソフトウェア開発のすべて. 日本実業出版社, 2000, 327p.

[2] 佐藤聖規, 和田貴久, 河村雅人, 米沢弘樹, 山岸啓, 川口耕介. Jenkins実践入門. 技術評論社, 2011, 336p.

[3] The GitHub Blog. The Octoverse in 2012. 2012-12-19. https://github.com/blog/1359-the-octoverse-in-2012

[4] パスサポ | めざせ！ 情報処理技術者試験 パスサポ … 技術評論社. グルーピングで楽に覚える情報処理試験用語. 2009-10-08. http://gihyo.jp/lifestyle/serial/01/ipa-terminology/

[5] Internet.Watch. 和製GitHub「ギットブレイク」公開、プライベートリポジトリが無料・無制限. 2013-05-29. http://internet.watch.impress.co.jp/docs/news/20130529\_601351.html

[6] apis.io. APIS.io :: The open-source API directory. 2013-05-29. http://apis.io/github

[7] ITエキスパートのための問題解決メディア - ＠IT. ガチで5分で分かる分散型バージョン管理システムGit. 2013-07-05. http://www.atmarkit.co.jp/ait/articles/1307/05/news028.html

[8] 飯山教史, 町田欣史, 高橋和也, 小堀一雄. 現場で使えるソフトウェアテスト. 翔泳社, 2008, 344p.

[9] ジェームズ・ウィテカー, ジェーソン・アーボン, ジェフ・キャローロ, 長尾高弘. テストから見えてくるグーグルのソフトウェア開発. 日経BP社, 2013, 432p.