JUnit による単体テストの工数削減と ROI を向上させる秘策！

～効率的な JUnit の実装方法とは？～

目次

■ はじめに

今日のデジタル経済では、企業が消費者の要望に答えるためにソフトウェアを迅速に市場に届ける必要があります。さらに、高品質でセキュアなアプリケーションでこれを達成するためには、ソフトウェアの変更を漏れなく、迅速にテストしなければなりません。しかし、テストの実施はボトルネックにもなり、納期を遅らせ、ビジネスに直接的な影響を及ぼす可能性があります。

ソフトウェア開発の遅延の大きな要因の 1 つは、開発プロセスの後半で発見されるバグです。これは、短納期に対応しようと、テストフェーズにかける時間やコストを最小限にするため、テストと製品の品質にしわ寄せがいきます。この問題に対処する 1 つの方法として、JUnit を使った単体テストがあります。これによって開発の早い段階でバグを潰し、手戻りを減らします。しかし、この方法にも時間とコストの面で欠点があります。

これらの欠点を補うために、Parasoft Jtest（以下、Jtest）を活用し、Java 開発のテスト効率を上げることができます。新しいコードの開発でも、レガシーアプリケーションの保守開発でも、Jtest は単体テストに関連する時間とコストの大幅な改善を促進します。「副次的な」メリットとして、メンテナンスが容易で意味がある単体テストスイートが得られます。では、Jtest を使うことでどのくらいコストが削減できるのでしょうか？それをどのように実現するのでしょうか？

■ 単体テスト作成時の ROI の向上

Parasoft 社の ROI 調査

ここでの例として、大手金融機関のクラウド移行戦略の一環で、新しい Java アプリケーションを開発中とします。プロジェクトの概要は以下のとおりです。

- 開発チームは **20 人**。1 人あたり年間 **1,000 万円**となり、年間の開発コストは **2 億円。**
- アプリケーションのリリースは**四半期**ごとで、1 回あたりのリリースコストは **5,000 万円**。

Parasoft 社の調査では、平均的な開発プロジェクトは、単体テストの作成におよそ **30%**の時間を費やしているそうです。

(年間の開発コスト) ÷ (四半期ごとのリリース) × (単体テストに費やす時間) = (単体テスト作成コスト)

2 億円 ÷ 4 × 30% = 1,500 万円

さらに Parasoft 社の調査によると、Jtest を使うことで単体テスト作成工数を半分にする事ができます。Jtest を使うと多くの手動で書かなければならないコードを自動で作成することができるため、工数の削減に繋がります。つまり、単体テスト作成の工数が開発の 30%ではなく、15%へ削減することができます。

(単体テスト作成コスト) × (Jtest による効率化) = (Jtest を使った単体テスト作成コスト)

1,500 万円 × 50% = 750 万円

これを見ると、1 回のリリースあたり**750 万円**のコストが削減され、1 年間で**3,000 万円**のコスト削減につながります。また、コストだけではなく、時間も節約されます。2 週間 10 営業日のスプリントを採用して開発を行っている場合、30%つまり 3 日が単体テストに割り当てられます。
1 回のリリース（3 ヶ月）につき合計 6 つのスプリントがあります（例：開発用に 5 スプリント、最終テスト用に 1 スプリント)。この数値から計算すると、1 リリースにつき 15 日間が単体テストに使用されます。
Parasoft の調査ではこの期間を半分にするので、1 リリースあたり 7.5 日間単体テストが行われます。

(開発スプリント数) ✕ (スプリントの日数) ✕ (単体テストに費やす時間) ✕ (Jtest による効率化)＝ (Jtest を使った単体テスト作成コスト)

5 スプリント ✕ 10 日 x 30% ✕ 50% ＝ 7.5 日

1 リリースあたり **7.5 日**の削減につながり、年間で **22.5 日**の削減になります。この節約は市場へのリリースで競合他社に勝つために時間がひっ迫している状況では非常に貴重です。

弊社の ROI 調査

ここまで、Parasoft 社の調査データを見てきましたが、弊社では、新卒の Jtest 未経験者による社内調査を行いました。 同一のソースコードに対して、ゴールをカバレッジ 100%に設定した時にどの程度工数が削減されるか、という調査です。この調査では、ファイルによって波はあるものの、平均で **33%**の工数削減に繋がりました。

対象クラス（Java コード）	テスト手法	メソッド数	LOC	JUnit のみ (h:mm:ss)	Jtest 利用 (h:mm:ss)	削減率
ClassA.java	MockMvc	2	93	0:29:42	0:12:39	**57.41**%
ClassB.java	MockMvc	3	97	0:34:10	0:25:51	**24.34%**
ClassC.java	MockMvc	1	99	0:32:56	0:20:23	**38.11%**
ClassD.java	MockMvc	5	82	0:36:33	0:37:29	-2.55%
ClassE.java	MockMvc	1	61	0:15:05	0:16:09	-7.07%
ClassF.java	MockMvc	3	92	0:38:03	0:26:44	**29.74%**
ClassG.java	標準	4	46	0:25:04	0:15:47	**37.03%**
ClassH.java	標準	2	35	0:29:19	0:07:17	**75.16%**
ClassI.java	標準	7	121	1:21:25	0:46:42	**42.64%**
ClassJ.java	標準	5	136	1:22:39	1:01:16	**25.87%**
計		33	862	**6:44:56**	**4:30:17**	**33.25%**

33%の工数削減ということは、つまり、**67%**の工数で単体テストを実行する事ができます。
これを先程同様に計算すると次の様になります。

(単体テスト作成コスト) ✕ (Jtest による効率化) = (Jtest を使った単体テスト作成コスト)

1,500 万円 ✕ 67% ＝ 1,005 万円

この場合でも、1 回のリリースあたり 1500 万円 － 1,005 万円で **495 万円**のコストが削減され、1 年間で **1,980 万円**のコスト削減につながります。また、時間に関しては以下のとおりです。

(開発日数) ✕ (単体テストに費やす時間) ✕ (Jtest による効率化) = (Jtest を使った単体テスト作成コスト)

50 日 ✕ 30% ✕ 67% ＝ 10.5 日

1 リリースあたり 15 日 - 10.5 日で **4.5 日**の削減、1 年間で **18 日**の削減につながります。

削減されたコスト、日数をまとめると以下の様になります。
お客様の開発プロジェクトにあわせて削減コストを算出してみてはいかがでしょうか？

削減率	削減されたコスト（年間）	削減された日数（年間）
50%	3,000 万円	22.5 日
33%	1,980 万円	18 日

Jtest を利用することによってなぜ、これほどまでに工数を削減することができるのかを具体的に説明します。

■ なぜ Jtest を利用すると工数が削減されるのか？

わずか 30 秒で単体テストを実装

JUnit を使った単体テストは、テストコードを書き、入力値を設定し、期待した動作をしているか検証を行います。そのため、テスト対象と同じくらい、またはそれ以上のコード量を書く必要があります。
通常、一からテストコードを書いていくか、IDE の機能を使ってテストコードの空のテンプレートを作成後、テストコードを実装する必要がありますが、Jtest を使うと、簡単なテストであればわずか 30 秒で実行可能なテストコードの作成が可能です。
次の 3 ステップを実行するだけでテンプレートに沿った単体テストコードが実装できます。

① テストケーステンプレートを作成

クリックひとつでテストケースのテンプレートを作成します。テストケースのスケルトンだけではなく、値の設定が必要なパラメータの定義や空のアサートも作成してくれます。

② テスト対象メソッドに渡すパラメータを設定

テスト対象メソッドに渡すパラメータを手動でテストコード内に記述します。

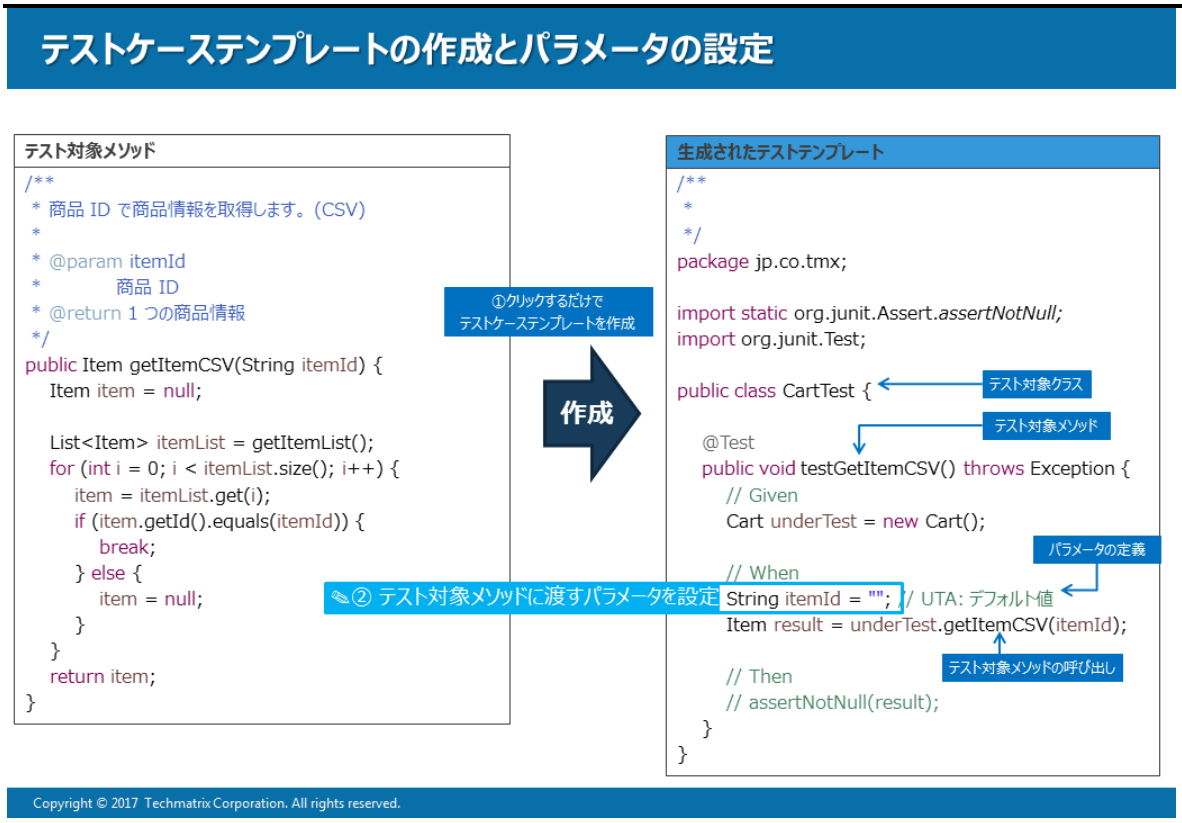


図 1 テストケーステンプレートの作成とパラメータの設定例

③ アサートのテンプレート作成

テスト結果（期待値）を現在の動作に合わせる場合は、テストを実行し、期待値が正しいことを確認後、クリックひとつでアサートを作成します。

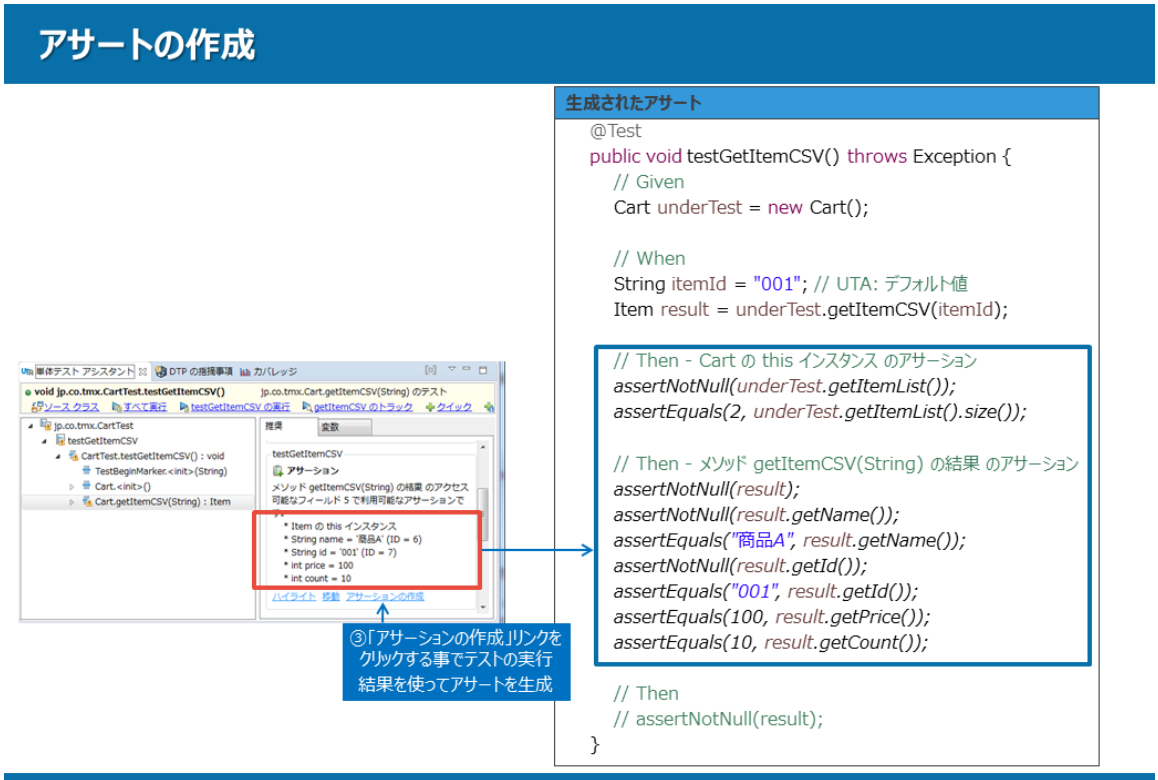


図 2 アサートの作成例

クリック操作と簡単なテストケースの修正でテストを実装することができました。テストを実施する上で必要となる作業が大幅に軽減されたことがお分かりいただけたのではないでしょうか。時間にすると、わずか 30 秒でテストケースの実装が可能です。テストケースの入力値、期待値はあらかじめユーザーが用意する必要があるものの、Jtest ではこのようにテストケースの作成を大きく効率化してくれます。

なぜテストが失敗したのかをひと目で確認

もうひとつ単体テストで意外と面倒なのが、テストのデバッグ作業です。テストが失敗したり、期待通りに動作しない場合、通常だとステップ実行によって変数の値の変化を見たり、デバッグログなどを挟んでプログラムがどのように動いたかを確認します。

この作業も、Jtest を使えば、実行したテストが期待通りの動作にならなかったときに、テスト対象のどのコードが実行され、どのように処理がされたのかを瞬時に確認することができます。

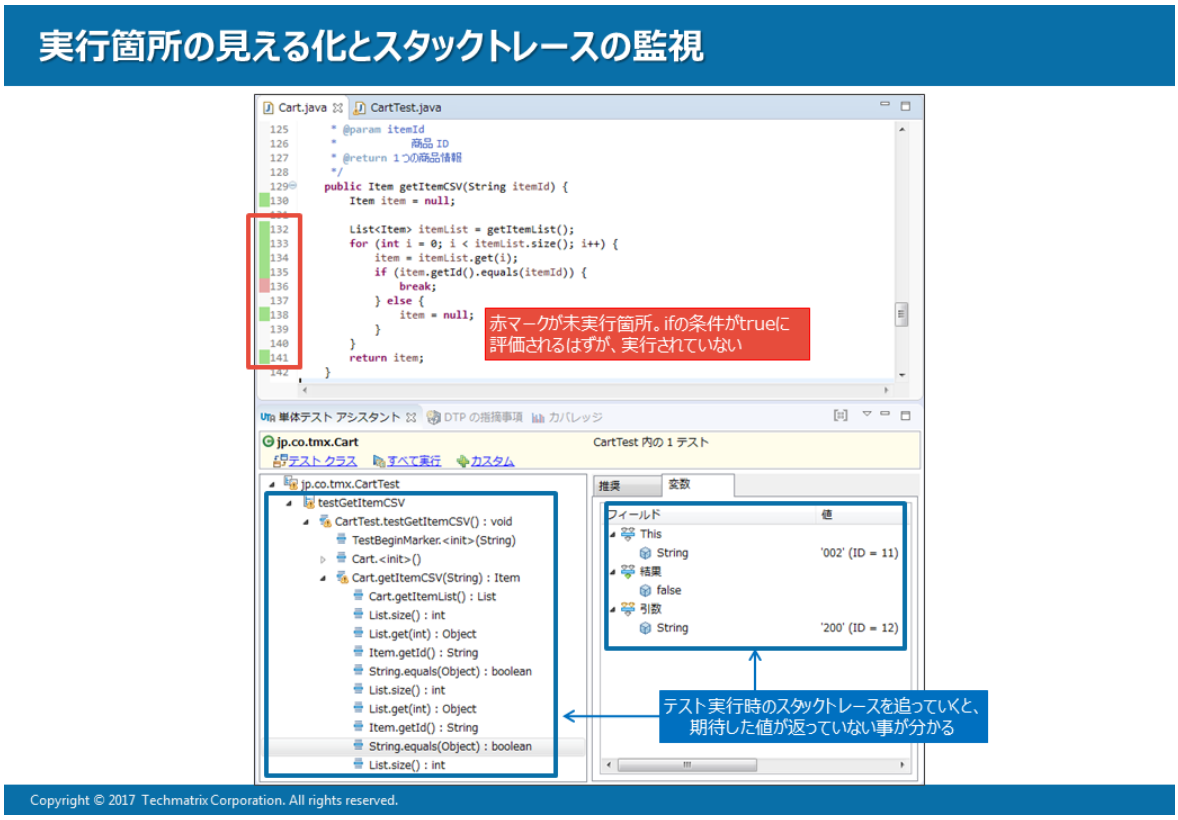


図 3 実行箇所の見える化とスタックトレースの監視

カバレッジや実行時の処理フローを追うことで、実行されていないコードを見つけ出すことができます。さらに、Jtest では、実行された処理の経路（スタックトレース）を監視しているため、メソッドの戻り値や引数の値を確認してなぜ実行されなかったのか、テスト実行時の状態を見える化できます。

モックの監視やパラメータライズのテンプレートまで作成

さらに、Jtest では Mockito と PowerMock をサポートしています。一般的にモックを利用するためには、呼び出し手順の習得や慣れが必要になってきますが、Jtest がモックのテンプレートを作成したり、モック化可能な箇所をレポートしたりしてくれるため、経験の少ない開発者でも簡単に使うことができるようになります。

さらに、JUnitParams による CSV を使ったパラメータライズドテストケースのテンプレート作成などさまざまなテンプレートを作成できます。これによって、単体テストを手順化することができるため、経験の少ない開発者でも迷うことなくテストを実装することができ、単体テストの実装レベルを平準化することができます。

Spring フレームワークを使った単体テストを効率化

エンタープライズ分野でのフレームワークとしては、スタンダードになっている Spring フレームワークは独自の単体テスト用のフレームワークやモックも提供されているため、フレームワークの仕組みに沿ったテストが効率良く実施できます。しかし、テストケースを適切にセットアップするには、通常の JUnit のテストケースと比べても、さらに多くの手作業によるコーディングが必要です。Jtest を使えば、Spring フレームワーク独自の面倒なコードを自動で作成できます。

テストケースのテンプレート作成

通常の JUnit テストケースと同様にクリックひとつでテストケースのテンプレートを作成します。

PeopleController.java　　テスト対象クラス・メソッド

```
@Controller
@RequestMapping("/people")
public class PeopleController {
    @Autowired
    protected PersonService personService;

    @GetMapping("/search/{name}")
    public ResponseEntity<Person> search(@PathVariable("name") String name, HttpSession session, Locale locale) {
        Person person = personService.findPerson(name);
        if (person == null) {
            return new ResponseEntity<Person>(HttpStatus.NOT_FOUND);
        }
        return new ResponseEntity<Person>(person, HttpStatus.OK);
    }
}
```

図 4 テスト対象クラス

たとえば、このソースコードであれば以下のような手順を手動でコーディングする必要があります。

- テスト対象の Controller とそれが依存する PersonService を使用して Spring コンテナーを設定する。
- findPerson ハンドラーメソッドに有効なリクエストを送信する。
- 戻り値のレスポンスを検証する。

Jtest を使えばこれらの面倒な手続きを自動で作成できるため、手作業のコーディングを大幅に減らせます。

PeopleControllerTest.java（抜粋） テストクラス・メソッド

```
@Autowired
PeopleController controller;
@Autowired
PersonService personService;
@Autowired
MockHttpSession session;
MockMvc mockMvc;

@Before
public void setup() {
    mockMvc = MockMvcBuilders.standaloneSetup(controller).build();
}
@Configuration
static class Config {
    @Bean
    public PeopleController getPeopleController() {
        return new PeopleController();
    }
    @Bean
    public PersonService getPersonService() {
        return mock(PersonService.class);
    }
}
@Test
public void testSearch() throws Throwable {
String name = "name";
    // session.setAttribute("", "");
    ResultActions actions = mockMvc.perform(get("/people/search/" + name).session(session));

    // Then
    // actions.andExpect(status().isOk());
    // actions.andExpect(header().string("", ""));
    // assertNotNull(session.getAttribute(""));
    // assertEquals("", session.getAttribute(""));
    // String response = ""; // UTA: 期待されるレスポンス値の設定
    // actions.andExpect(content().string(response));
}
```

テスト対象の準備やMockMvcの宣言

MockMvcの初期化

テスト実行のためのインスタン生成やMockMvcの生成

デフォルトのパラメータ値を設定

MockMvcを使ってハンドラーへリクエスト送信

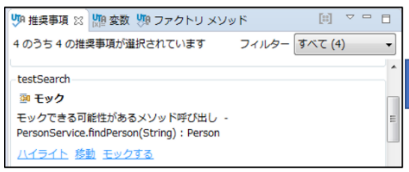
結果検証のテンプレートも生成

図 5 Jtest が自動で作成した Spring テストケースのテンプレート

また、通常の JUnit テストと同様にパラメータの設定やアサートの作成を行うことができます。

MockMvc の監視が可能

Jtest では Spring が提供する MockMvc もサポートするため、テスト実行時にモック化可能な箇所を検知し、レポートします。そこからクリックするだけでモックテンプレートの作成が可能です。



作成

PeopleControllerTest.java（抜粋） テストクラス・メソッド

```
public void testSearch() throws Throwable {
    Person findPersonResult = null;

    when(personService.findPerson(anyString())).thenReturn(findPersonResult);
    // When
    String name = "name";
    ResultActions actions = mockMvc.perform(get("/people/search/" +
name).session(session));
```

MockMvcが呼び出すメソッドのモックテンプレートを生成

図 6 MockMvc の監視とモックテンプレートの作成

これによって、Spring の単体テストやモックに慣れない開発者の学習コストを削減し、テスト実施の敷居を下げます。もちろん、Spring の単体テストでも Jtest の機能であるパラメータライズのテンプレート作成や、単体テストの追跡や監視といった機能も利用できます。

■ 最後に

JUnit による単体テストは迅速に品質を高めるための重要な要素です。単体テストによって後の工程でのバグの発生や手戻り工数は削減され、ビジネスに貢献する新機能の開発に集中できます。

また、JUnit を使うと迅速なフィードバックを得ることができるため、コードの変更がアプリケーションの機能を壊していないかをすばやく確認し、対処できます。実装担当者が変わった場合でもテストを資産として利用することができ、メンテナンス性も飛躍的に向上します。JUnit を使った単体テストは繰り返しテストができるので、思い切ったリファクタリングといった作業も気軽に行うことができます。

さらに Jtest をお使いになれば、JUnit の単体テストでネックとされていたテストの実装工数を削減し、より効果的に単体テストを実施することができます。もちろん、有償ツールならではの充実したサポートや、単体テスト運用のご提案から導入時のご支援なども有償・無償のサービスにて対応可能です。ぜひご検討下さい。

Parasoft について

Parasoft は、欠陥のないソフトウェアの効率的なデリバリーを支援するソフトウェア ソリューションの研究開発に取り組んでいます。SDLC を加速する一方で、ソフトウェアの欠陥に関するリスクに対処するため、Parasoft は Development Testing Platform および継続的テスト プラットフォームを提供しています。Parasoft のエンタープライズ向けおよび組み込み向けの開発ソリューションは、業界で最も包括的なものです。その範囲は静的解析、単体テスト、要件のトレーサビリティ、カバレッジ解析、機能テストと負荷テスト、開発 / テスト環境などに渡ります。Fortune 500 の大半の企業が、アジャイル、リーン、DevOps、コンプライアンス、セーフティ クリティカルな開発のための取り組みを進める中、最高品質のソフトウェアを一貫して効率的に生産するために、Parasoft のソリューションを頼りとしています。

技術資料および体験版

Jtest 体験版

- ご利用になれる期間は 14 日間です。
- 本書でご紹介した単体テストだけでなく、コードカバレッジ計測や、その他の機能である静的フロー解析を含む、すべての機能を無償でご利用いただけます。
- ご評価を円滑に進めるための技術的なお問い合わせも受け付けております。

Jtest 技術資料

- 以下の技術資料をご提供しております。
  - ソフトウェアの品質を向上させるためのアイデアや事例などの技術情報
  - 弊社が過去に開催したセミナーのプレゼンテーション資料

Jtest の単体テスト動画（YouTube）

単体テストを効率化する - Jtest の単体テストアシスタント

お問い合わせ先

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　ソフトウェアエンジニアリング営業部

https://www.techmatrix.co.jp/product/jtest
parasoft-info@techmatrix.co.jp