マス目実装ガイド

改版履歴

版数	日付	担当者
	改版内容	
0.1	2009/12/31	kenjift
	初版作成。	

マス目実装ガイド 目次

1. 概要

1.1. 本書について

本文書は、マス目を利用するための実装ガイドである。マス目はマスターメンテナンスシステムである。本書では、マス目の特長や利用局面、利用方法、応用方法などについて説明する。

1.2. マスターメンテナンス

コンピュータシステムには一般的にマスターデータが存在し、コンピュータシステム上の任意のアプリケーションはそのマスターデータを利用して動作するようになっている。

近年のアプリケーションではデータの永続化を RDB で行っており、マスターデータも RDB 上に格納される場合がほとんどである。

マスターデータは、受発注システムで言うところの商品データであったり、工場データであったり、取引先データであったりする。これらはそのシステムの本質的なビジネスプロセス（受発注システムでは受注処理など）では変更はありえない。しかし別のトリガ（新規商品開発、工場新設など）により変更が発生する可能性は大いにありえる。

マスターデータの変更頻度が少ない場合には、DB を直接操作し、メンテナンスを行うことで対応できる。しかしマスターデータの変更頻度が多い場合には、システムに別途マスターメンテナンスの機能を用意することが多い。

また近年では、直接 DB を操作するのが好ましくないといった風潮になっている。そのため権限のあるものが権限の範囲内の操作を行い、操作時の証跡（ログなど）を記録しておくことが求められている。こういった観点からマスターメンテナンスの機能を用意することもある。

1.3. マスターメンテナンス機能

マスターメンテナンス機能は、一般的に以下のような機能を保持している。

- 画面からの入力機能および入力補助機能
- 他システムからのデータの一括登録機能
- 入力時のバリデーション機能
- 関連するデータとの整合性を保つ機能
- 履歴管理機能
- 有効日や反映日などを設けた事前登録機能

1.4. マスターメンテナンス機能の問題点

上記であげたマスターメンテナンスの機能のうち特に有効日や反映日などを設けた事前登録機能は、システムの本質的な仕様を複雑にする要因になっている。

たとえば、受発注システムで工場の責任者を出力する場合を考えてみよう。

工場やその責任者はたまに変更されるかもしれないが、受発注システム上ではマスターデータと位置づけられる。しかし工場にひもづく責任者は、7 月までは A さんだったが、8 月からは B さんになるなどがあるため、データ構造が 1 対多になっている。つまり工場と日付が決まらないと、責任者は一意にならない。

責任者を出力する機能では、このデータ構造を意識して「出力時の現在日付や月末などの特定の日付を利用し責任者を一意にする」といったことを考えなければならない。

本来であればシステムにかかわる本質的な仕様のみを把握しておけばいいのに、マスターデータのデータ構造を意識して複雑な仕様を盛り込み、複雑な実装をしなければならなくなってしまう。このような仕様はマスターデータを扱う際に、システムのあらゆるところで発生する。

そこでマス目では、システムのマスターデータに関する横断的関心事を分離し、システム特有の関心事のみを意識できるようにサポートすることを目的として用意されたライブラリです。

1.5. マス目の特徴

マス目は、マスターメンテナンス機能を提供する。これ自体をシステムとして利用しても良いし、マス目のライブラリを利用し、システムに組み込んでも良い。マス目が提供するマスターメンテナンス機能の特徴は以下のとおりである。

- 既存のデータモデルに変更を加えない形でのメンテナンス
- データの事前登録を可能
- 事前登録時を含むデータ整合性を保障する
- Web からの修正を可能とする UI を用意

1.5.1. データの事前登録機能

マス目でのデータの事前登録は、現在の情報を利用する場合（または過去の情報を利用する場合）には有効である。しかし予約システムなど、未来に変わる情報を元に登録する場合には利用できない。未来の情報を利用するためには、データの主キーに日付を入れなければならず、「既存のデータモデル」を利用する思想と反しているため、今バージョンでは対応しない。

1.6. 構成

マスターメンテナンスシステムは以下のような構成になっている。

- Core
- Struts
- Struts-web

必須ライブラリは以下の通り。

- Hibernate
- Spring

2. スタートガイド

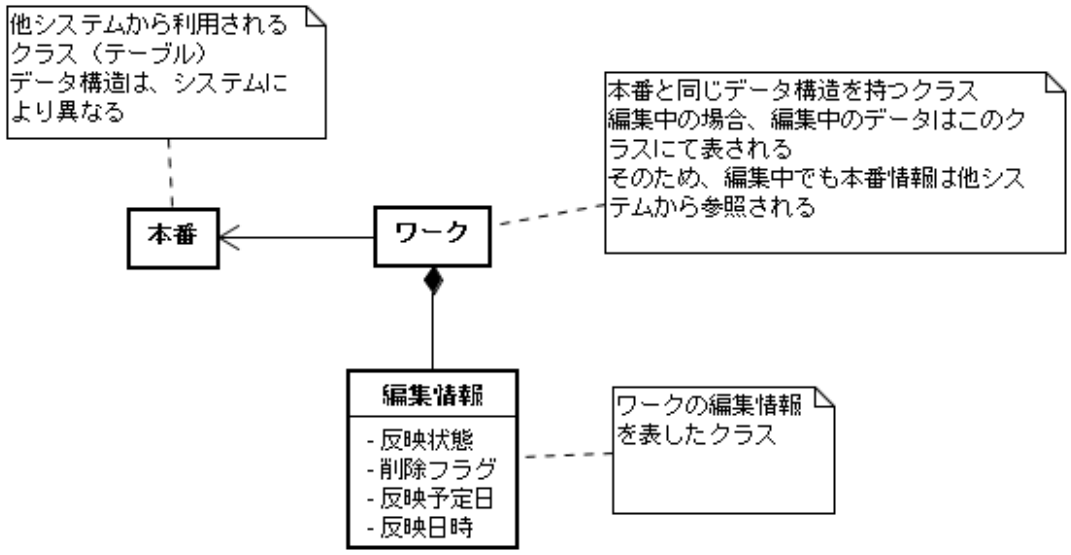
簡単な利用方法を紹介予定

後日記載する

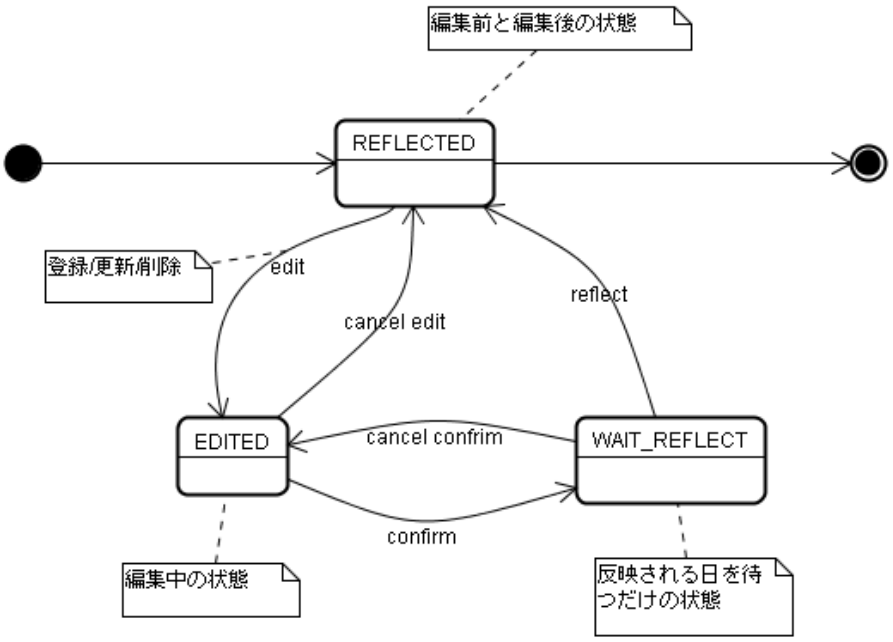
3. メンテナンス機能

3.1. メンテナンス方法

データ構造は以下の通りとなる。



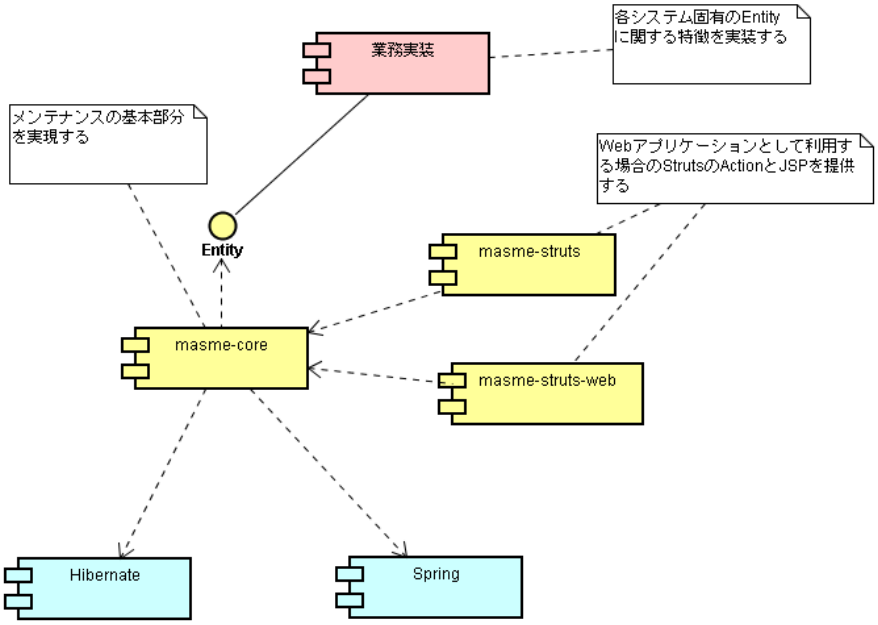
ワークの状態は、以下のように変わる。



- edit・・・修正処理。正確には、登録/変更/削除の 3 種類がありえる。ワークに対してのみ修正を行い、本番への変更は行わない。
- confirm・・・確認処理。定義した整合性が満たされることを確認し、反映待状態にする。
- reflect・・・反映予定日になった段階で、ワークの情報を本番に反映する。登録/変更/削除それぞれと同様の処理を行う。

edit と confirm には、取り消し処理がある。reflect は取り消しできない。

3.2. 全体構成



3.3. 機能説明

マス目の基本理念は、システムからマスターメンテナンスという関心事を分離することだ。マス目に個々のシステム独自の概念を教えることで、それ以外の一般的なマスターメンテナンス処理をすべて引き受けることが目的となっている。

マス目ではメンテナンスするうえでのシステム独自の概念を、以下のように定義する。これらをマス目に提供する必要がある。

- エンティティのデータ構造
- エンティティの整合性

また、システム独自の横断的関心事を以下のように定義する。これは必要に応じて、AOP を利用し実装することになる。

- ログ管理方針

- 権限方針
- 履歴管理方針

3.4. エンティティ

3.4.1. インタフェース

メンテナンス対象のエンティティは、以下のインタフェースを実装させる。

- RealEntity・・・本番エンティティを表す
- WorkEntity・・・ワークエンティティを表す

3.4.2. Configuration

3.4.2.1. アノテーションによる設定

以下のアノテーションを使って、エンティティに対する情報を設定する。

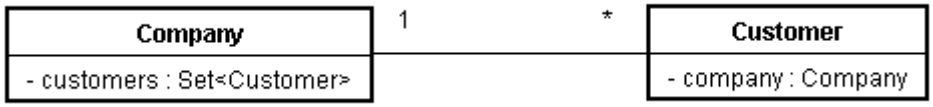
- MaintenanceTargetClass・・・メンテナンス対象クラスに対するアノテーション
- MaintenanceTargetProperty・・・メンテナンス対象プロパティに対するアノテーション
- ReferenceTargetProperty・・・メンテナンス対象プロパティが、単純な型でなく別のメンテナンス対象エンティティを参照している場合の設定

3.4.2.2. 設定ファイルによる設定

現在、未サポート。

3.4.3. 参照プロパティ

ReferenceTargetProperty で示されたプロパティを「参照プロパティ」と呼ぶ。以下の例の場合、Customer クラスの company プロパティが参照プロパティとなる。また、参照プロパティの逆側となった場合には、Company クラスの customers がそれに当たる。



3.5. バリデーション

マスターデータは整合性が取れていなければならない。整合性が取れていない場合には、他システムに対しての動作が保障できなくなってしまう。

マス目では、整合性検証のためのバリデーションを 2 種類定義する。

3.5.1.　１エンティティ内でのバリデーション

ひとつのエンティティに閉じたバリデーション。たとえば「名称が全角 5 文字であること」や「銀行の支店が指定されていること」などである。これは以下のように実装する。

AbstractCustomer.java

```
   public void validate () {
      if (this.getName().length() >100) {
         throw new MasmenValidateException("整合性失敗"+ this.getName
());
      }
   }
```

3.5.2.　参照プロパティに関するバリデーション

参照プロパティに関するバリデーションを指定しなければならない場合には、通常の方法では指定できない。理由は、それぞれが反映される日付を持っているため、参照プロパティのエンティティとの反映順序が影響するからである。

たとえば「customer には、ひとつの company が指定されていること」といった場合、ワークに存在する company が本番にまだ反映されていないかもしれない。バリデーションの条件に、先に反映されることという条件が必要になる。

この種類のバリデーションは以下のように実装する。メソッド名は、参照プロパティを指定する際に同時に指定する。引数は、無か参照プロパティのクラスのみである。参照プロパティを直接使ってはいけない。

AbstactCustomer.java

```
   public void validateForCompany(AbstractCompany company) {
      if (company==null) {
         throw new MasmenValidateException("整合性失敗"+company);
      }
      if (this.getName().equals(company.getName())) {
         throw new MasmenValidateException("整合性失敗"+company);
      }
   }
```

参照プロパティに対するバリデーションを定義しておけば、反映順番の保証はマスターメンテナンスシステムが行う。

3.5.3.　メソッドによる実装

現在は、エンティティを表すクラスのメソッドでのバリデーションを実装のみをサポートしている。メソッドは、ワークおよび本番の両方に対して有効になるように実装する。たとえばスーパークラスを設けることや、インタフェースを用意すること、本番を継承してワークを作ることなどが考えられる。

3.6. サービスクラス

マス目が提供する基本のサービスクラスは以下の通り。

- LoadTargetService・・・メンテナンス対象オブジェクトを取得するためのサービスクラス。全件取得、ID を指定した取得、修正中の一覧取得などのサービスを提供する。
- EditTargetService・・・メンテナンス対象オブジェクトに対する修正を行うサービスクラス。登録/更新/削除や修正キャンセルなどのサービスを提供する。
- ConfirmTargetService・・・メンテナンス対象オブジェクトに対する確認処理を行うサービスクラス。確認、確認キャンセル、反映日修正のサービスを提供する。
- ReflectTargetService・・・メンテナンス対象オブジェクトに対する反映処理を行うサービスクラス。反映サービスを提供する。

また、引数にメンテナンス対象オブジェクトではなく、直接 Entity を取る以下のサービスも提供する。

- EditEntityService
- ConfirmEntityService

3.7. Web 層

Struts を利用した単純な Web 層を提供する。

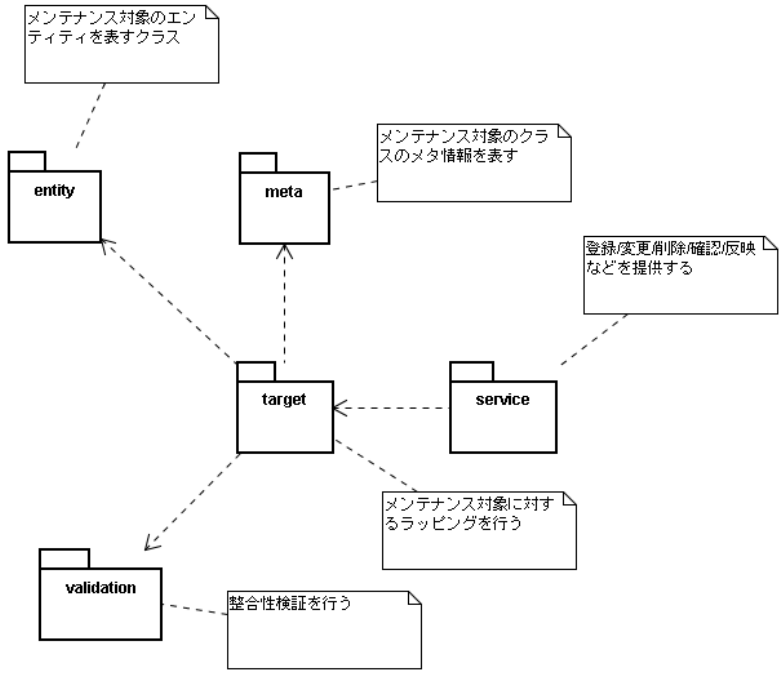
- masme-struts・・・マス目のサービスを利用する Action クラスとサポートクラスを提供する
- masme-struts-web・・・masme-struts を利用する JSP（WAR プロジェクト）

3.8. 履歴管理

現在未サポート

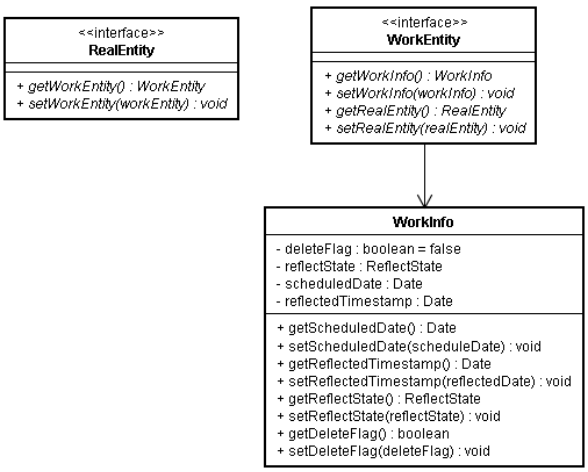
4.　内部構成および拡張ポイント

4.1.　パッケージ構成

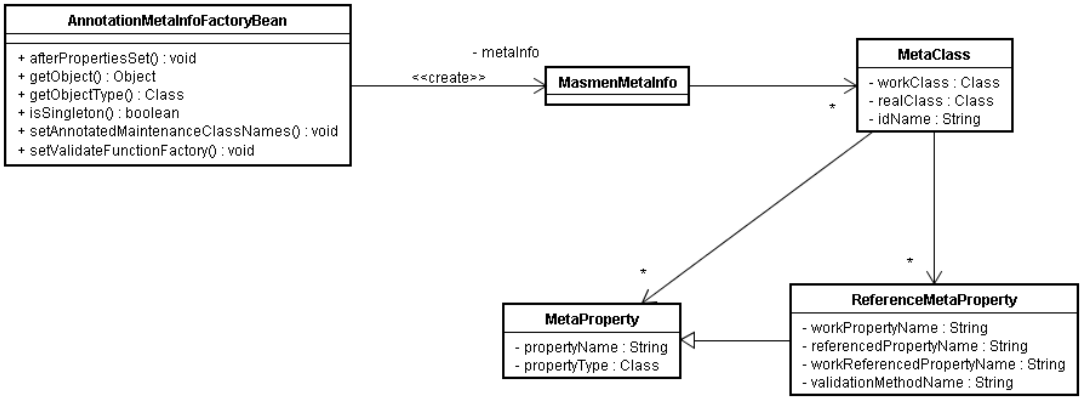


4.2.　各種パッケージ

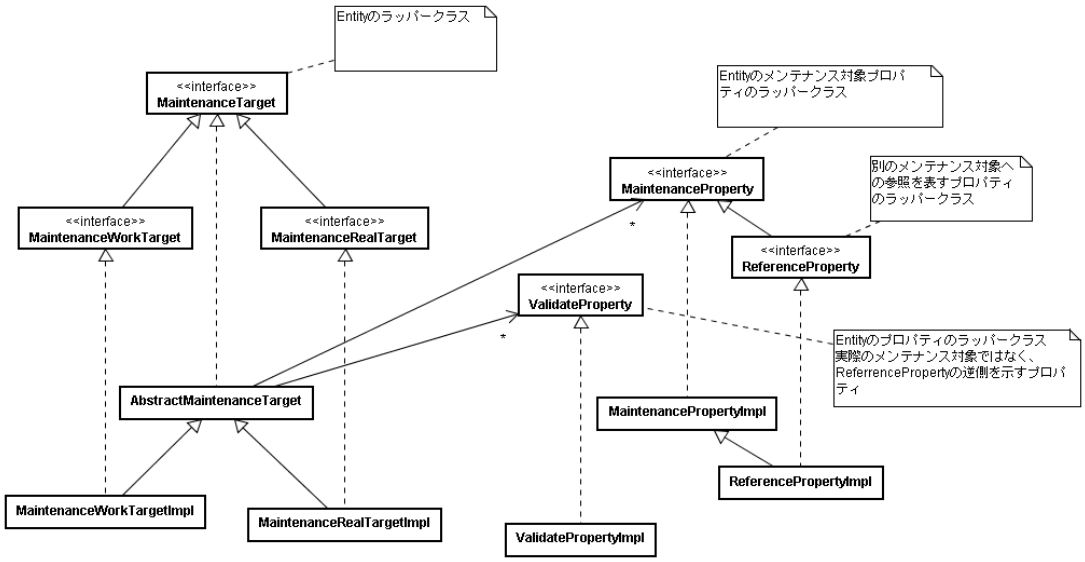
4.2.1.　entityパッケージ



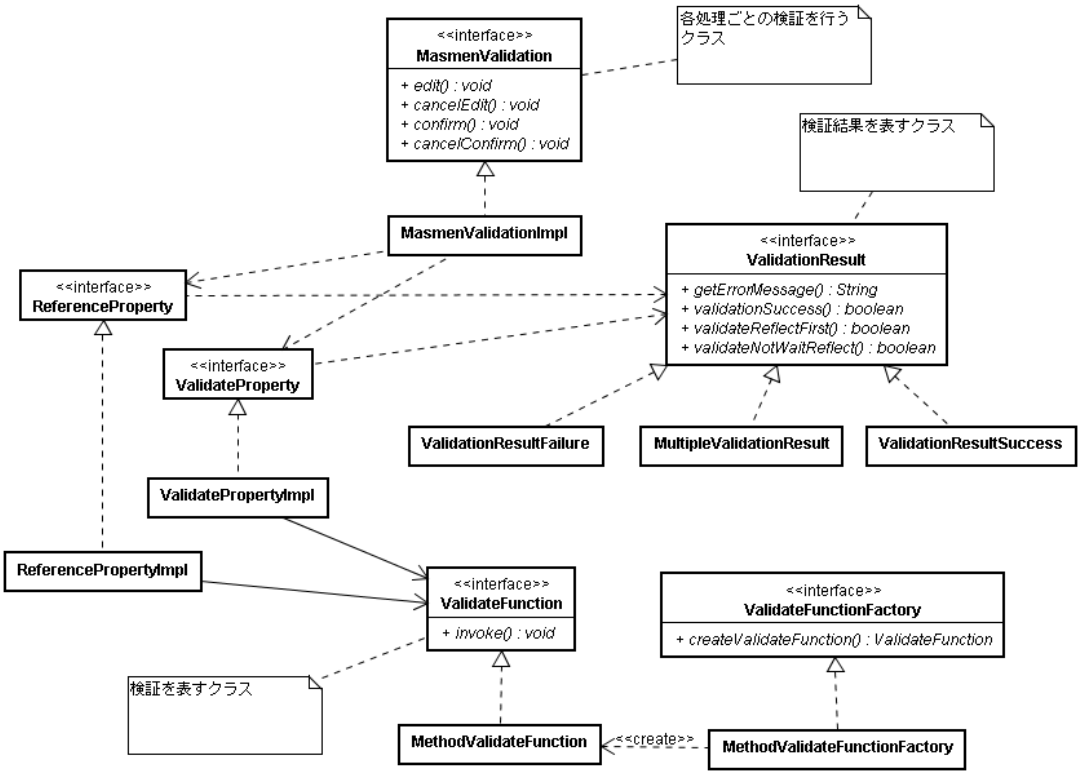
4.2.2. metaパッケージ



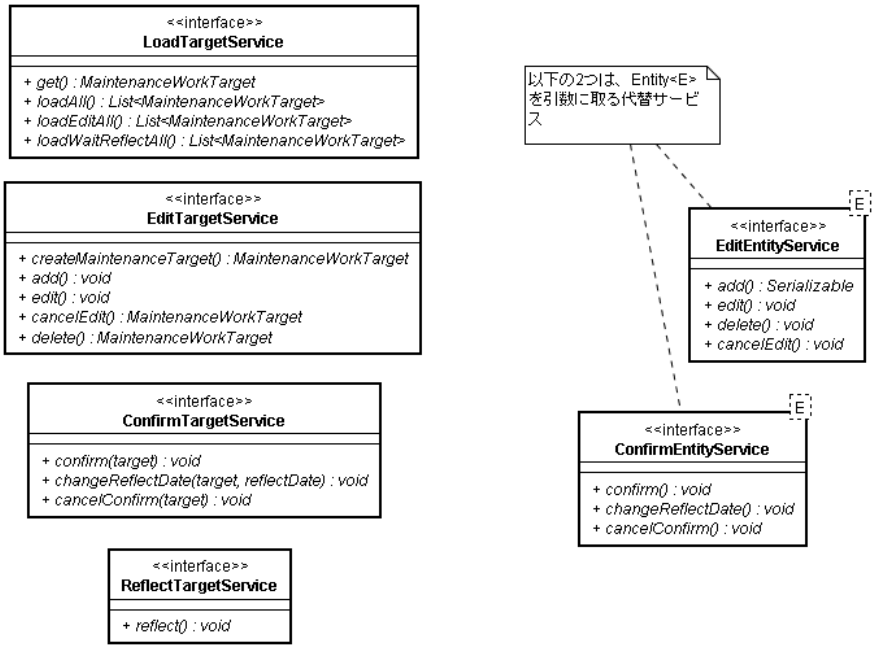
4.2.3. targetパッケージ



4.2.4. validationパッケージ



4.2.5. serviceパッケージ



4.3. 処理ごとのバリデーション内容と意義

マス目の一番の特徴となる処理が、バリデーションである。反映の順序を保証しなければならないため、処理ごとに異なるバリデーションを実施している。

4.3.1. ワークが整合性を保っているかどうかの検証

この検証は、変更処理に対してと変更の取り消し処理に対して行わなければならない。検証内容は以下の通りである。

4.3.1.1. 参照先のエンティティの整合性検証

（変更後の）ワークエンティティが参照するエンティティに対して 整合性を保っている状態かどうかを検証する。

これは従業員の情報を変更し、特定の店舗を割り当てる際の以下の検証になる。

- 割り当てる店舗が存在するか
- 割り当てる店舗の営業開始日 < 従業員の就業開始日

この検証はワークエンティティが削除対象でない場合に （削除フラグを ON 時、登録取消時）にのみ行う。

4.3.1.2. 被参照先のエンティティの整合性検証

（変更後の）ワークエンティティが参照されているエンティティに対して 整合性を保っている状態かどうかを検証する。

これは店舗の情報を変更した際に、その店舗を参照している全ての従業員を対象に 以下の検証を行うことになる。

- 店舗が削除される場合には、参照しているすべての従業員も削除対象になっているか
- 変更後の店舗の営業開始日 < 全ての従業員の就業開始日

4.3.2. エンティティの反映状態に関して整合性を保っているかどうかの検証

この検証は、確認処理に対してと確認の取り消し処理に対して行わなければならない。検証内容は以下の通りである。

4.3.2.1. 参照先のエンティティの整合性検証

（確認対象である）エンティティ A が別のエンティティ B を参照している場合には、 エンティティ A の反映時にはエンティティ B の本番情報が存在し 特定の整合性条件を満たしている必要がある。そのような状態を実現するためには、 エンティティ A を確認時に、エンティティ B が 以下のいずれかの状態である必要がある。

- エンティティ B のワークが確認済状態にあり、 エンティティ B の反映日 < エンティティ A の反映日（確認処理においてエンティティ B のワークが条件を満たしていないことはありえない）
- エンティティ B の本番が、エンティティ A に必要な条件を満たしている

この検証はエンティティ A が削除対象でない場合のみ行う。

4.3.2.2. 被参照先のエンティティの整合性検証

（確認対象である）エンティティ A が別のエンティティ B を参照されている場合には、 エンティティ B の反映時にはエンティティ A の本番情報が存在し 特定の整合性条件を満たしている必要がある。そのような上状態を実現するためには、 エンティティ A が削除されるまたは条件を満たさなくなる修正の場合の確認時に、 以下のいずれかの状態である必要がある。

- エンティティ A を参照するエンティティ B の本番が存在しないこと（確認処理においてエンティティ B のワークがエンティティ A を参照することはありえない）
- エンティティ A を参照するエンティティ B の本番がある場合には、（修正または削除されたエンティティ B のワークが） 確認済状態でエンティティ B の反映日 < エンティティ A の反映日

4.4. 拡張ポイント

後日記載する。