Introduction and Goals

Requirements Overview

Tool Rental Agreement Service (TRAS) is a service for creating a rental agreement for large tools (e.g. a jackhammer) based on inputs provided by a point-of sale application. This application will be run by employees of a home improvement store, such as Home Depot. There are specific business requirements, detailed in the appendix, which outline how the service should handle discounts, holidays, and so on.

Quality Goals

Functional Suitability: TRAS should adhere in all respects to the requirements listed in the appendix, and there should be a suite of automated tests which cover positive and negative cases for those requirements.

Maintainability: TRAS should be maintainable, with sufficient separation among service components to enable easy scaling to external services, easy replacement of data repositories, easy incorporation of API calls, etc.

Architecture Constraints

The service will be implemented in Java, but there are otherwise no explicit constraints placed on the architecture. It does not have to integrate with any existing systems. That said, the small size and specificity of the service's responsibilities strongly suggest that it be implemented as a single microservice with sufficient isolation of internal responsibilities to allow incremental migration to separate services as usage and/or performance requirements change.

System Scope and Context

Business Context

TRAS is initially meant to accept inputs from a point-of-sale system and respond with rental agreement details in text format. However, it would also be useful in other contexts, including self-service tool rental forms on a retail website, or a warehouse tool that needs to reconcile inventory, so being mindful of those other use cases within reason will help

prevent rework in future.

External Interface

CheckoutService will be instantiated with a ToolRepository instance (likely to be supplied via a dependency injection library), with the `checkout()` method being the entrypoint to the service.

Input: `checkout()` takes in these parameters:

- toolCode: string
- rentalDays: integer
- discountPercent: integer
- checkoutDate: datetime

Output: the output is an instance of the RentalAgreement class, which will make all the outputs listed in the specification document available either directly or through domain objects. When the `toString()` method is called on RentalAgreement, it will generate a form representation of these outputs, formatted as following:

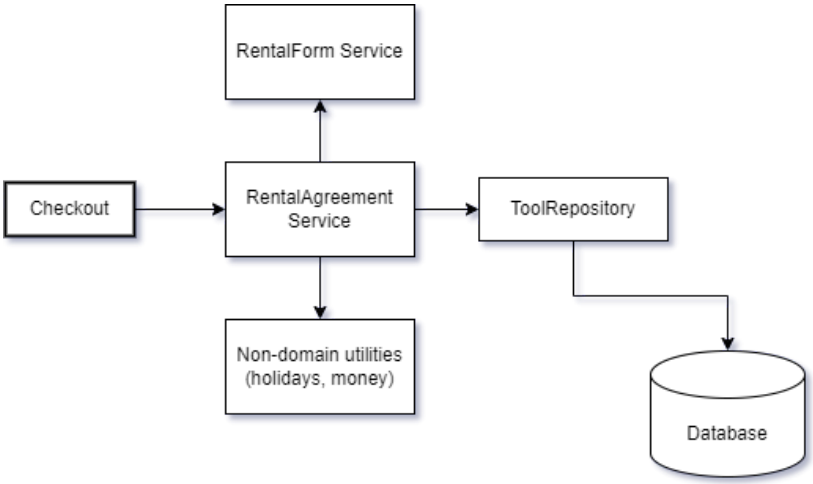
```
Tool code: [string, currently 4 uppercase alpha characters, e.g. CHNS]
Tool type: [string]
Tool brand: [string]
Rental days: [int]
Check out date: [date, mm/dd/yy]
Due date: [date, mm/dd/yy]
Daily rental charge: [currency (USD), e.g. $1.49]
Charge days: [int]
Pre-discount charge: [currency (USD)]
Discount percent: [percent, e.g. 10%]
Discount amount: [currency (USD)]
Final charge: [currency (USD)]
```

Solution Strategy

Our general approach will be to build this as a microservice, which will internally use a simple n-tier layering strategy. This allows us to create and maintain a clear external interface, while retaining an organizational structure that would allow for breaking out the pieces of the service in the event that scaling requires it. Further, the separation of concerns and isolation afforded by the n-tier approach will provide testability and efficient tracing, improving overall software quality.

Building Block View

Overall System



Checkout

The checkout service will be responsible for:

1. taking in the point-of-sale parameters
2. creating the rentalAgreement instance
3. printing the rentalAgreement string

RentalAgreement

The RentalAgreement instance will be responsible for:

1. gathering the tool and rental terms information from the tool repository
2. implementing domain considerations, e.g. how to count the number of rental days
3. calling the rental form service to get the string output

RentalForm

The RentalForm service will take in all of the tool and rental terms information and format them into the output string.

ToolRepository

The ToolRepository will wrap calls to the data layer and return denormalized domain objects. This is preferable to echoing back the database structure to the service layer because it will help protect the rest of the system from changes in that structure.

Misc. non-domain utilities

More a point of guidance than a strictly-defined structure, this serves as a reminder that non-domain utilities should not be within domain structures. Examples of these within this project are holidays, money, and currency. The rationale for this is that requirements around those often change and expand, as when entering new markets, and do so for many projects within the organization at once. Therefore, having them isolated enough to provide for expanding them into their own libraries or services is a good idea.

Data Model

Some useful conversation with the product team around the data model will be necessary before finalizing this. Answers to the following questions would influence the model design:

Do rental terms adhere to the tool type, the tool code, or neither?

This will affect whether normalization of the tool rental terms away from the tool definition is useful, and if so what columns will be used to define them.

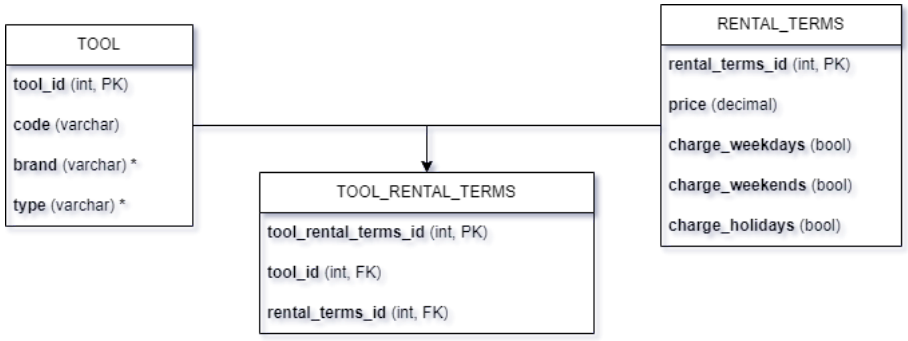
Will historical rentals have to maintain reference to their original terms, in the case of changing terms for the same tool type or tool code?

If so, we would likely persist rentalAgreements back to the database, or in a separate database.

Will reporting/business intelligence need rental term change information, or rental information?

This will influence the decision to represent the rental terms separately from tool information, and if so, to use a history table type. Using a history table will make later retrieval of the rental terms in effect at the time of a given rental possible.

All this said, with the information we have, this is a satisfactory representation of the data model as understood presently. It is subject to change based on conversations with Product (already a good reason for isolating the data layer using a repository class):



Cross-cutting Concepts

Money

For the sake of simplicity, we will be representing money by wrapping the BigDecimal class. Once Java Money makes it into a standard library, that will be the preferable choice.

Currency

Currency will be represented using USD. This is a simplifying assumption which should be stated in documentation and isolated. Preference is to hard-code use of the US locale rather than implementing USD display by hand, as the former is easier to find using search and easier to replace using context- or configuration-based locale later.

Holidays

Holidays are also simplified here, as there are only two to represent. This also should be isolated as requirements around this are likely to change.

Non-functional requirements

1. All public methods should be unit tested, with coverage requirements set by the organization
2. There should be javadoc comments on the entrypoint class at minimum
3. General code quality standards and guidelines set by organization should be followed

Appendix A: requirements document

See the technical assessment document for full business requirements for the service.