Tool Time - High Level Design Document

Jay Houghton
jay@computer.org
7-July-2024

---

1 Introduction and Objectives

This document describes the high level design for Tool Time - an application to compute rental agreements for tool rentals.

1.1 Functionality

Tool Time is a command line tool rental application that provides a store the capability to rent tools to customers. The application takes a set of inputs about the rental and produces a Rental Agreement that can be utilized at point-of-sale as a contract covering the terms of the rental.

1.2 Objective, Goals, & Constraints

- *Primary Objective* - Design and implementation for Tool Time.
- *Goal* - Improve efficiency and customer experience
- *Goal* - Deterministic and correct computation of a Rental Agreement and its components.
- *Constraint* - To facilitate an iterative approach to building such tools, the first phase will consist of a command line application, without a UI and without a database.

1.3 Stakeholders

The following stakeholders are identified as having interest in the application:

- Store - The store will gain the ability to further automate tool rental products, increasing revenue and efficiency.
- Store Clerk - The store clerk will gain efficient and reduction of human error by utilizing the application to perform rental calculations.
- Customer - The customer will have a better experience with less errors, and predicable, clearly communicated rental terms.

2 Solution Strategy & Scope

2.1 Scope

The application will be utilized by store clerks performing tool rental for customers. The application has no external interfaces and is capable of complete standalone operation. The application is required to be deterministic - producing the same outputs given the same inputs.

2.1.1 Business & Technical Interfaces

The store clerk will be interacting with the application using a command line on a computer system associated with tool rental. There are no other business or technical interfaces. Of note, the clerk may perform printing of the application output (Rental Agreement) for purposes of book and record keeping, however this action is out of scope for this design.

2.2 Solution Overview

The application will be structured using inspiration from a typical MVC desktop application, with distinct differences being that there is no UI view and no persistent data store. The logical View component consists solely of a command line interaction and output printed to standard output streams. The logical Controller subsystem will comprise the business logic and agreement computation. Finally, a Model will be proposed that matches the domain objects that participate in tool rental.

Concepts

To augment the details from requirements, the following concepts are highlighted:

- Chargeable Day - a day determined to have a charge for a particular tool rental. By contrast, non-chargeable days will have zero rental cost.
- Holiday - either the Fourth of July (fourth day in July) or Labor Day (first Monday in September).
- Pre-Discount Charge - the subtotal of the rental cost before application of the discount.

Rental Period (Duration)

The days which are chargeable start the day after the check-out date and extend the number of days indicated by the *Rental days* parameter.

2.2.1 View

The View subsystem is simplistic. A command line application will be invoked and its output sent to standard output streams for viewing by the Store Clerk.

2.2.2 Model

The Model is based on the domain objects involved in tool rental. From our requirements the following objects and associated properties will be designed for the application.

Tool

- code - textual globally unique identifier for tool.
- type - classification of tool using commonly known tool object types (ladder, chainsaw, etc).
- brand - manufacturer name of tool.
- schedule of charges - details of how the tool rental cost will be computed.
    - daily rate - rate per chargeable day
    - weekday flag - indication to charge on a day that is a week day (Monday-Friday)
    - weekend flag - indication to charge on a weekend day (Saturday or Sunday)
    - holiday flag - indication to charge on a holiday

Day Type - classification of a day

- weekday - straightforward indication of the day being a week day
- weekend - straightforward indication of the day being a weekend day
- holiday - indication of a holiday, including specialized logic for when the effective weekday-celebration when the Fourth of July falls on a weekend day.

Rental - aggregate object, composed of:

- tool
- rental day count - number of days for rental (duration)
- discount - integer representing a scaled (0-100) percent discount to apply to subtotal
- check out date (start)

Agreement (Rental Agreement)

- Rental
    - Tool
    - Duration
    - Checkout out date
    - Discount (percent)
- Due date
- Daily rental charge
- Charge days - number of days with charge fees
- Pre-discount charge
- Discount amount
- Final charge (grand total)

2.2.3 Controller

The logical Controller subsystem will consist of two parts:

1. **Business logic layer** - specific logic for domain-based input and output data validation, embodiment of rules that might be specific to the Store.
2. **Service logic layer** - discrete computation of the Rental Agreement, mostly without regard to Store-specific business rules. Some flexibility might be desired to accommodate variations of Rental Agreement computation - but this is not in scope for this design.

2.3 Design Philosophy & Patterns

A primary consideration is growth. How will the application be utilized and how can improvements be accommodated in the future. To address this, the focus will be on a modular design philosophy, where the system can be decomposed into containing subsystems.
Additionally, the following guidance is provided for design:

- Each subsystem should be of a single purpose, to ensure cohesion, and allowing for *things-that-change-together-stay-together*.
- Business logic should be independent of any computation.
- Consider the following design patterns:
    - Service Locator for obtaining references to service objects.
    - State Machine pattern may be useful for inspecting each day in duration.
    - MVC or MVC-like for separation of concerns.
- Consider a POSIX-compatible command line program arrangement for command input.

2.4 Technology

The system is expected to be implemented on the Java Platform. Use of the latest LTS release is suggested. Use of the Oracle OpenJDK is forbidden due to lack of security updates. Consider Coretto or Adopt instead.

3 Structural Design

The primary subsystems comprising the Tool time software system are:

- **Main** - Program entry point.
- **Controller** - Business logic and flow coordination.
- **Service Layer** - Utility services for configuration, data, and rental.

3.1 Subsystems

3.1.1 Main

This is the entry point of the command line application. This component should gather input, interact with the Controller by delegating computation of the Rental Agreement, and finally, output the resulting agreement according to requirements.

3.1.2 Controller

The controller will organize business logic, primarily consisting of validation, and manage the delegation of computation of the Rental Agreement to the Service Layer. This includes the following responsibilities:

1. Accepting input as a Rental model object
2. Validating and providing feedback on the content of the Rental model object
3. Calling into the service layer to delegate computation of the Rental Agreement
4. Handling error conditions on all the above in a normalized fashion, providing the caller with a straightforward way to relay exceptional conditions to the user.

3.1.3 Service Layer

The Service Layer will consist of three utility services:

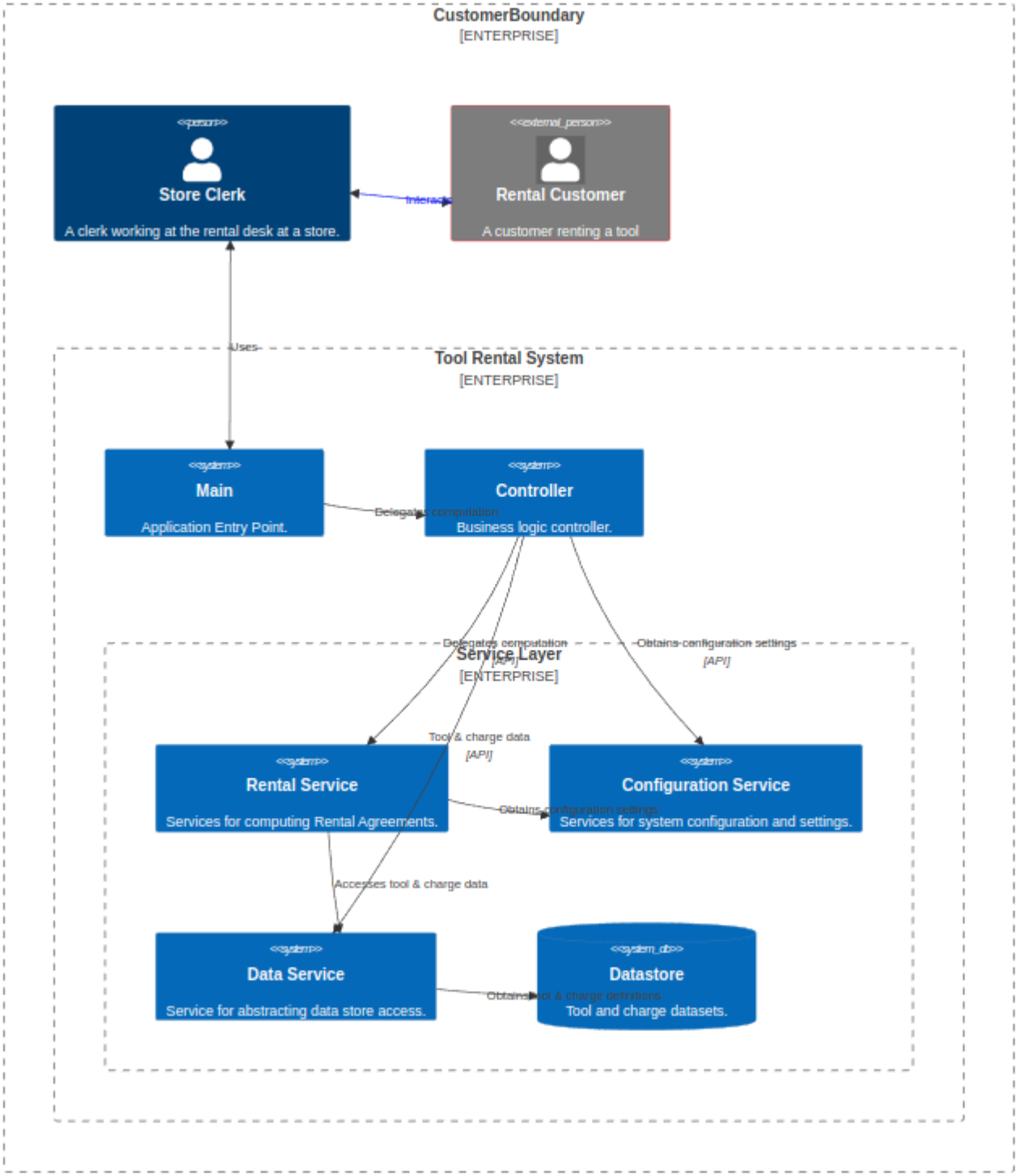
1. **Rental Service** - computes Rental Agreements
2. **Data Service** - abstracts access to supporting application data: Tools and Charges.
3. **Configuration Service** - facilitates access to configuration and settings.

Additional design items include:

- The service objects in the Service Layer will be made accessible to the Controller.
- Service objects should be effectively stateless, such that subsequent calls do not depend upon previous calls.
    - This doesn't mean the objects cannot contain state variables.
- Computations should be absolutely deterministic. The same inputs should always produce the same outputs.

3.2 Context Diagram

C4 Model - Context Diagram

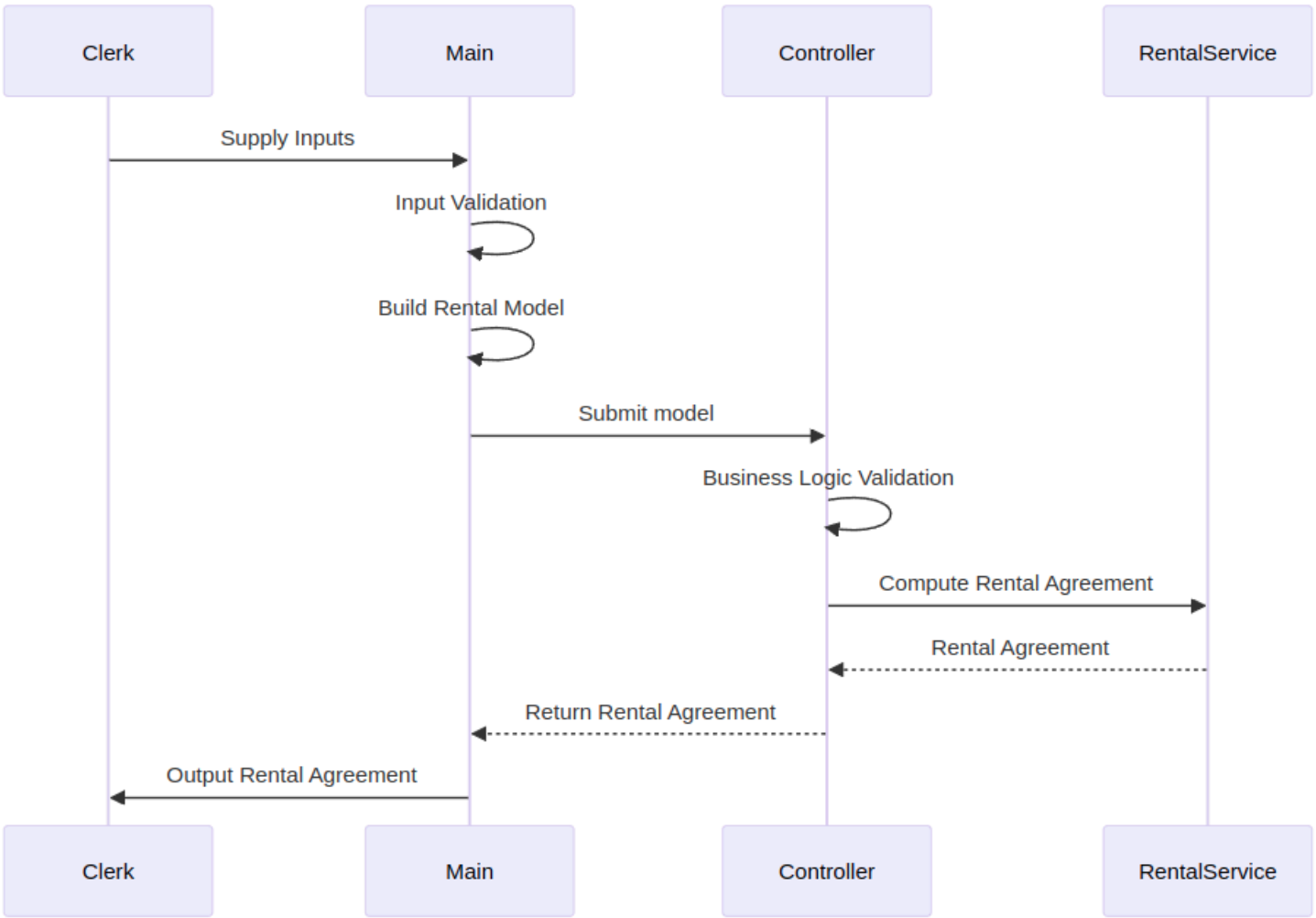


4 Behavioral Design

Basic application flow is proposed as:

1. Validate Inputs
2. Build Rental Model
3. Submit model to Controller
4. Controller does business logic validation
5. Controller computes Rental Agreement with the assistance of a Rental Service
6. Controller returns Rental Agreement model object
7. Output Rental Agreement to standard output

4.1 Sequence Diagram



4.2 Configuration

The system will utilize file-based configuration settings. For simplicity, the definition of tools and charges can be held in this configuration file and read by the application.

4.3 Error Handling Strategy

The following principles will guide the error handling approach for the application:

1. Using our modular and layered structure, exceptions arising during computation shall be handled within the manifesting component.
2. Recoverable failures shall be subject to a sensible retry operation (e.g. backoff, circuit breaking, etc).
3. Clear and obvious propagation of unrecoverable errors will occur in the back-to-front direction (i.e. Service relays error to Controller, Controller relays error to Main, etc).
4. Each layer will present a normalized (with possible variety) set of error objects to the adjacent layer (caller).
5. Errors resulting in user-facing messages shall be subject to approval and review of the product team.

A simple proposal for error normalization would have two general categories of errors:

1. Validation errors - input data failing basic or business logic validation.
2. Computation errors - errors arising during the computation of a Rental Agreement.

Allowing the Controller to relay these two types of errors provides a simple and normalized approach to handling them, all in a single place (Main).

5 Architecture & Engineering

5.1 Architectural Decisions

There are no items in the decision log.

5.2 Crosscutting Concerns

The design does not address any crosscutting concerns.

5.3 Non-Functional Requirements

The application shall be performant. Computation of the Rental Agreement should take no longer than several seconds to run.

5.4 Deployment & Operations

The application shall manifest as a single binary executable file. Likely an JAR file with a Main-class manifest entry. This should be portable and include a mechanism to obtain the software version.

5.6 Issues & Risks

Possible issues may include extensibility - how are new tools added to the application, how are changes in charges or tools handled?

5.6.1 Statement on Financial Arithmetic

Of course, computers use the binary (base-2) number system to represent numbers, including floating point numbers. Problems arise when representing decimal (base-10) fractions in binary, as these fractions often cannot be represented exactly in a finite number of binary digits. These days, most systems use the IEEE-754 standard for floating-point arithmetic, which represents numbers using scientific notation with an exponent and mantissa. This approach still has issues that are important to be aware of.

A basic illustration is that the representation of $0.10 is a repeating binary fraction and cannot be exactly represented using IEEE 754 without error. The error is very small, but in fields like finance, where quantities can be large, this error can accumulate with repeated operations such as multiplication.

For this reason, systems that perform pricing and currency calculations avoid using floating point primitives. For example, trading systems on Wall Street often use BigDecimal for its arbitrary precision or employ proprietary *fixed-point arithmetic operations* on integers to ensure accuracy and performance.

For these reasons, the application shall not use floating point primitives for pricing calculations.