Tool Rental Application High-Level Design

Author:	A S
Publish Date:	1/24/2024
Revision:	1.0

Contents

Overview

This document presents a High-Level Design of a simple Tool Rental application that has a separate functional requirements specification (located elsewhere). The solution will be coded in Java.

The purpose of this document is to guide a developer in the general construction of the solution: overall development principles, the object model and class specification, the application logic flow, basic configuration, and testing strategies.

General Solution Development Principles

These are some general principles to be followed in the construction of the solution.

Configuration is Preferable to Hardcoding

Wherever possible, reference data and locale-specific behaviors (date/time formatting, currency formatting, etc.) should be set by external configuration, rather than by hardcoding the values in the class definitions themselves. This allows the solution to be easily configured to support additional tools for rent, tool types and their associated prices and day charging rules, and additional holidays.

Reference Data Definition

There is no need to create multiple copies of reference and configuration data. It should be loaded one time (either eagerly or upon first use), and then accessed as “read-only”—especially if any of it is exposed outside of the implementing package.

Accessibility Rules

Classes should limit their accessible scope to the smallest possible. It is easy to simply declare all classes “public” and to make accessor methods public. However, consider if the entity really needs to be accessible outside of the package scope. Also consider creating immutable class instances to return reference data, as by its nature it isn’t changeable at runtime—so don’t even provide the methods for altering the state of the class instance.

Decimal Number Handling

When using numbers that represent monetary values, don’t use classes derived from floating point (or “double”) types. All currencies have associated scale and rounding rules that are applied to financial calculations. Java has appropriate classes to represent monetary values (e.g.: java.math.BigDecimal).

Requirements Analysis

After review of the functional spec, there are a few important takeaways:

No Persistence

There is no persistence. If this were a “real” tool rental application, there would be persistence layer to deal with. Specifically, the definitions of tools, tool types, and holidays would be in some form of database, with at least read-only access available to the reference data. There would also be a rental inventory management capability so that one could not rent tools that aren’t available at the requested date.

- Assume there is “infinite” inventory available.

Agreements are One Per Tool Being Rented

In an actual rental implementation, one might consider a rental agreement to be able to handle more than one tool being rented at a time.

- A completed agreement will reference only one tool.

The Association Between Tools and the Tool Types is Relational

Tools available to rent have a “tool code” as a key, as well as a “tool type” and “brand” property. However, the general rental rules are tied to “tool types” that hold the rental price, the weekday / weekend / holiday charging rules. In other words, the “brand” of a tool has no influence on its rental price or charging rules.

- Don’t encapsulate an instance of a “tool type” rules within a “tool” definition.

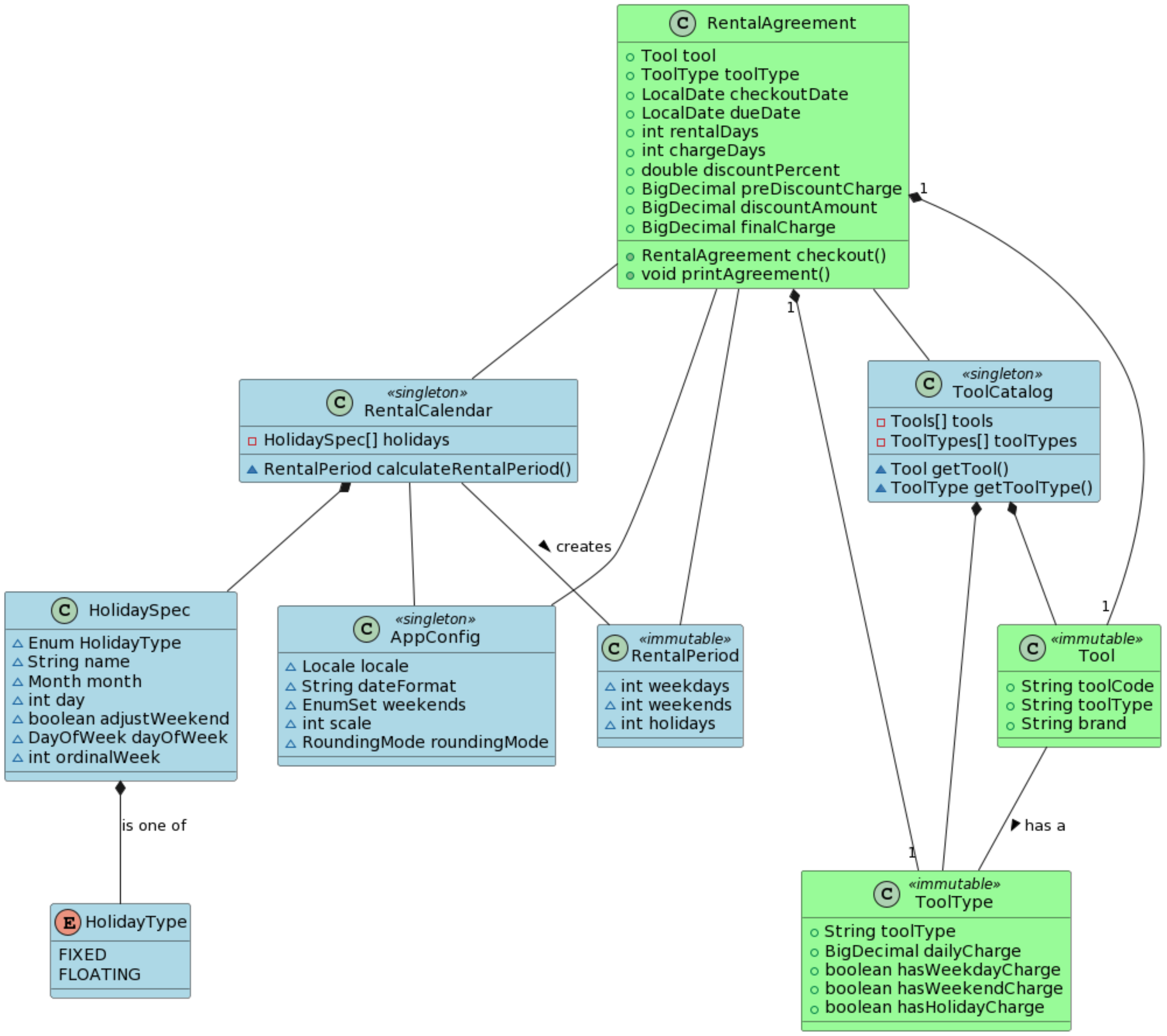
Formatting Rules are Aligned with US Locale Conventions

All the date/time, currency, and percentage formatting rules are aligned with common U.S. conventions. However, there is a general principle (stated above) to allow these types of properties to be configured.

- Set the locale-specific properties to US conventions as described in the spec by default.
- Allow the locale-specific properties to be overridden by external configuration.

Class Model

There are only a few "public" classes exposed by the solution. The remainder are package protected—assisting with the creation of the main Agreement class. Classes in green are public, those in blue are protected.



Class: Application

This is the main public class of the solution. It is the only one to be exposed outside of the package except for two immutable data classes that represent the tool being rented (Tool) and the rental rules for the type of tool (ToolType).

Class Type/Pattern:	*Static Factory*	Creates a new Application instance upon invocation of the static factory method.
Public Methods:	*RentalAgreement checkout()*	Factory method. Validates input and throws IllegalArgumentException if any input is not valid
	void printAgreement()	Prints the agreement properties to the console, formatted using locale-based formatting rules.

	"getters" for exposed Agreement properties	Allow the consumer to read (but not modify) the values of a completed agreement.

Class: Tool

Public immutable class containing an instance of a tool that may be rented. The properties are as per the functional spec.

Class Type/Pattern:	*POJO (immutable)*	Instantiated from the constructor only.
Public Methods:	*"getters" for exposed properties*	Allow the consumer to read (but not modify) the values of a Tool.

Class: ToolType

Public immutable class containing an instance of a tool that may be rented. The properties are as per the functional spec.

Class Type/Pattern:	*POJO (immutable)*	Instantiated from the constructor only.
Public Methods:	*"getters" for exposed properties*	Allow the consumer to read (but not modify) the values of a Tool.

Class: ToolCatalog

Protected class that holds all the Tool and ToolType entities. Provides methods for retrieving Tool and ToolType by their keys. This class should be populated by externalized reference data (not static constants).

Class Type/Pattern:	*Singleton*	Lazy initialization—reads tool definitions from external configuration file
Protected Methods:	*getInstance()*	Returns a reference to the instantiated instance of the class.
	Tool getTool ()	Retrieves the Tool by the supplied toolCode. Returns **null** if not found.
	ToolType getToolType()	Retrieves the ToolType by the supplied toolCode. Returns **null** if not found.

Note: if there is an error in the initialization of the reference data, then an exception should be thrown. It is not possible to calculate a RentalAgreement if the reference data is missing or misconfigured.

Enumeration: HolidayType

Defines the types of holidays that can be expressed as a "HolidaySpec" class.

FIXED	The holiday is on a "fixed" day of the month. May "slide" if on weekends (per HolidaySpec)
FLOATING	The holiday is on a specific weekday of a specific week of a month

Class: HolidaySpec

Package protected data class that holds a holiday specification. Not necessarily immutable because it's not exposed outside of the package. Only those attributes relevant to a specific HolidayType are populated. The attributes correspond to those expressed in the functional spec for holidays.

Class Type/Pattern:	*POJO*	Instantiated from the constructor only.

Protected Methods:	*"getters" for exposed properties*	Allow the consumer to read (but not modify) the values of a HolidaySpec.

Class: RentalPeriod

Package protected immutable class containing a rental period's properties. The properties are the number of weekdays, number of weekends, and number of holidays for a given date and number of days.

Class Type/Pattern:	*POJO (immutable)*	Instantiated from the constructor only.
Public Methods:	*"getters" for exposed properties*	Allow the consumer to read (but not modify) the values of a RentalPeriod.

Class: RentalCalendar

Package Protected class that calculates a rental period based on an input date and number of days. Contains a collection of HolidaySpecs that are loaded from external configuration. Weekends days are also specified by configuration (in the AppConfig class).

Class Type/Pattern:	*Singleton*	Lazy initialization—reads holiday definitions from external configuration file
Protected Methods:	*getInstance()*	Returns a reference to the instantiated instance of the class.
	RentalPeriod calculateRentalPeriod()	Factory method that creates a RentalPeriod instance based on the supplied date and duration. Uses holiday specs and Weekend day definitions in the calculation of the period.

Note: if there is an error in the initialization of the reference data, then an exception should be thrown. It is not possible to calculate a RentalAgreement if the reference data is missing or misconfigured.

Class: AppConfig

Package Protected class that holds "general" configuration that is used throughout the application: the locale, date format, weekend days, and scale and rounding modes for currency. Is preconfigured with default values according to the functional spec. These values may be overridden by configuration in an external properties file.

Class Type/Pattern:	*Singleton*	Lazy initialization—reads configurations from external configuration file
Protected Methods:	*getInstance()*	Returns a reference to the instantiated instance of the class.
	"getters" for exposed properties	Allow the consumer to read (but not modify) the values of an AppConfig.

Note: if there is an error in the reading of the configuration data, then an exception should be thrown. It is not possible to calculate a RentalAgreement if the tool is misconfigured.

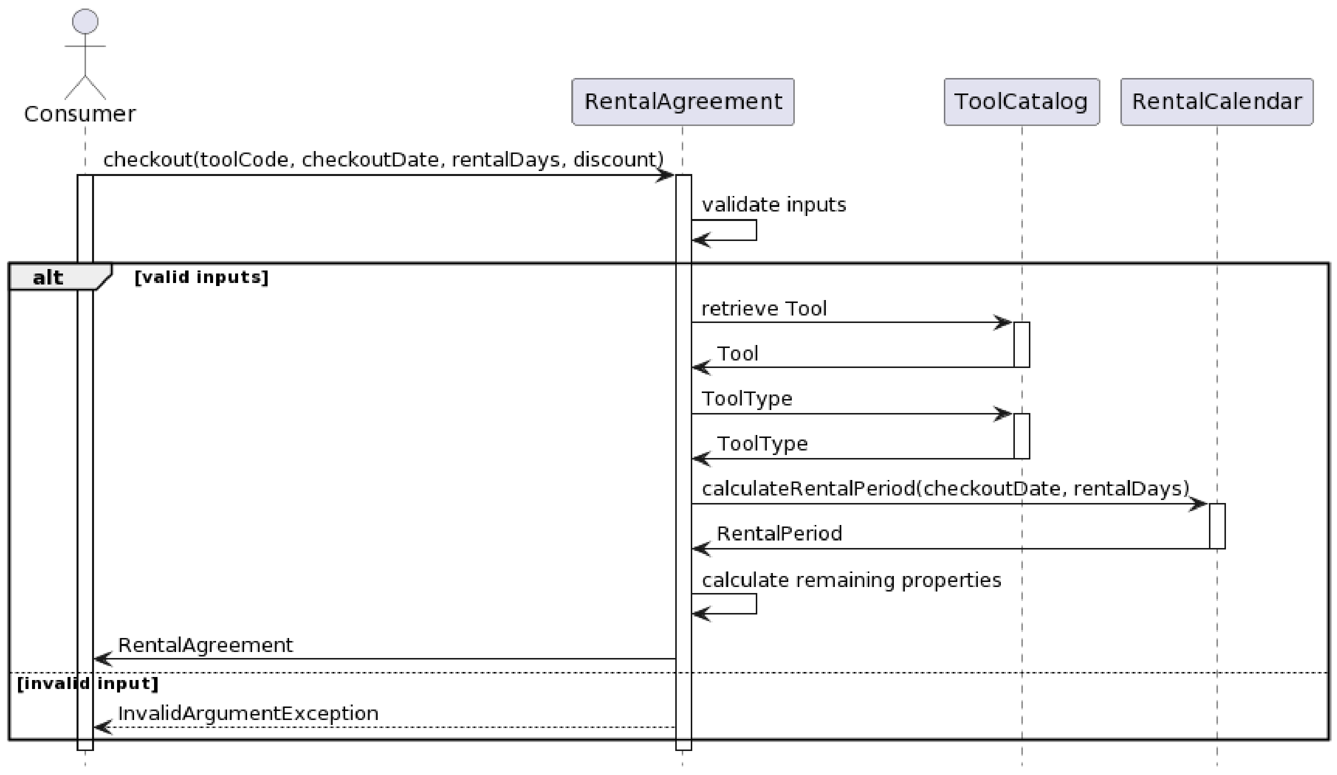
Additional Classes

The implementer may create additional package protected classes to assist with managing the reading of configuration as necessary.

Application Logic Flow

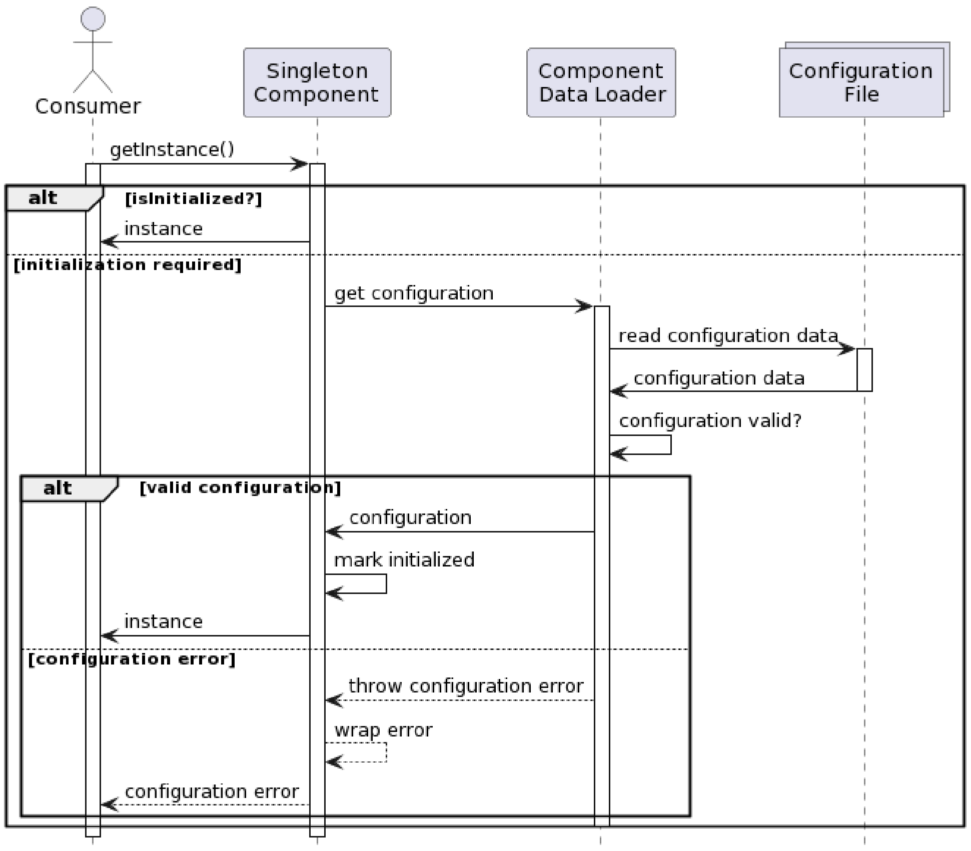
RentalApplication

The flow of the core logic of the application (RentalAgreement.checkout()) is straightforward, as illustrated by the sequence diagram, below:



External Configuration

Much of the complexity in the solution lies around the external initialization of the reference data encapsulated in singleton classes. However, if this is coded well, then the solution can be easily extended by amending the external reference data. This is a general pattern to follow—choice of configuration file type and loading is up to the implementer.



Baseline Configuration

Provide a set of initial configuration files that contain the “default” configurations outlined in the functional spec. These can be implemented as “properties” files, or some other form of easy deserialization (JSON documents, etc.).

- AppConfig: locale, date format, weekend days, and currency scaling and rounding properties.
- Tool: set of tool definitions corresponding to Tools table in the spec.
- ToolType: set of tool types associated properties corresponding to the ToolTypes table in the spec.
- HolidaySpec: set of holiday definitions corresponding to the holidays defined in the spec.

These configurations should be validated upon loading. An appropriate error (with corrective action message) should be thrown if there is a validation issue.

Testing Considerations

Configuration Validation

Positive and negative test cases should be created to verify that the externalized configuration readers are robust. Test for:

- File missing
- File contents malformed
- Out of range values
- Logical inconsistencies (e.g.: Tool refers to ToolType that doesn't exist)
- Missing Attributes, etc.

RentalAgreement Functional Validation

Positive and negative test cases should be created to verify the functionality of the "checkout" method. Test for:

- Invalid input values (tool code, checkout date, rental days, and rental percentage)
- Valid inputs in combinations that test:
  - The baseline tool codes
  - Various checkout dates that span holidays/weekends
  - Calculated values are correct (and rounded according to the configured rounding rule):
    - Due Date
    - Number of charge days
    - Total charge before discount
    - Discount percentage amount
    - Final charge

Extensibility Validation

Since the tool externalizes the definition of tool types, tool codes, and holidays, create additional values of each entity and repeat the configuration and functional validations.