Tool Rental Application

System Architecture and Design

Contents

System Overview

Requirement summary

To create a point-of-sale tool rental application that provides a single checkout() command that:

- Receives a request including tool code, rental days, discount %, checkout date
- Generates a response containing either a **1)** with a nicely formatted full rental agreement including a return due date, pricing information and general tool information OR **2)** a user-friendly error message if bad request values are received
- The application will be standalone and self-sufficient without dependency on a user interface or a database

Key features

- **Validation:** Ensures input constraints are met.
- **Charge Calculation:** Accurately calculates based on tool-specific rules.
- **Holiday Handling:** Handles observed holidays correctly.
- **Output Formatting:** Provides clear, user-friendly output in the specified format.
- **Scalability:** Easily extendable for additional tool types or new holidays.

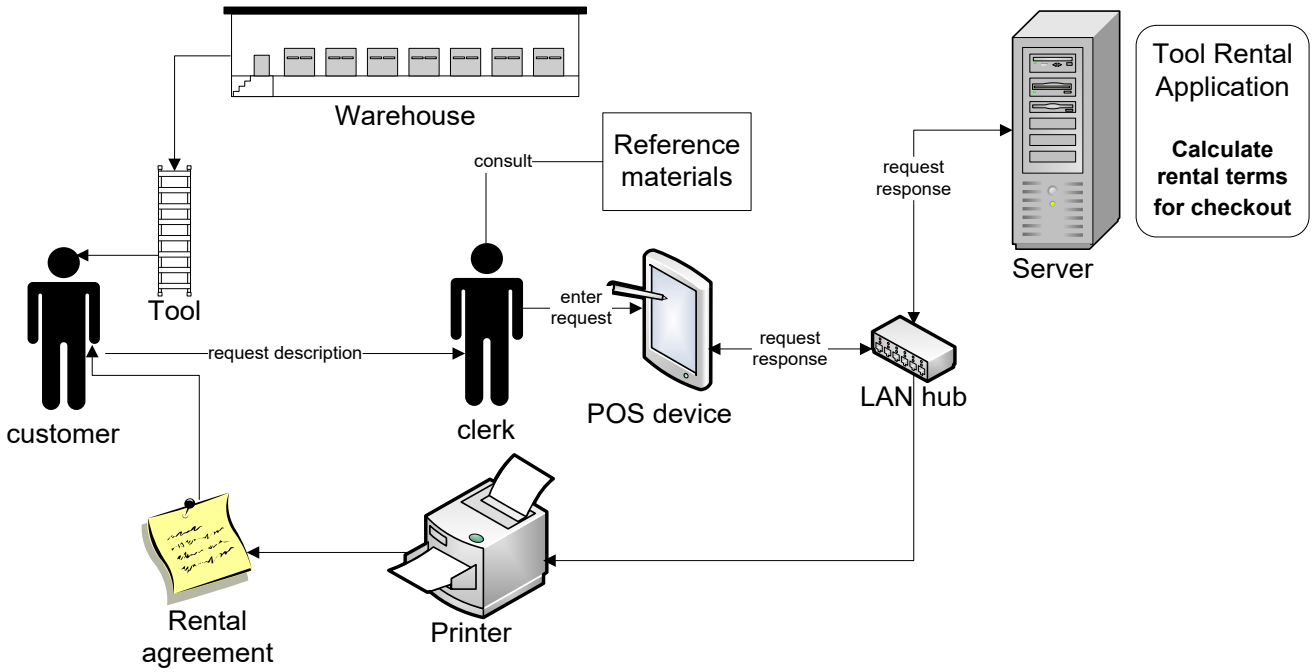
System Context

"Our system" = our Tool Reservation System
A point-of-sale tool rental application implies that:

- an in-store clerk gathers rental needs from a customer
- the clerk determines the correct tool code for the product described (*either from prior training or from reference materials such as a printed booklet or a lookup via an external system*)
- the clerk confirms that the requested tool is in-stock by checking inventory (*via a manual lookup or via an external system*)
- the clerk enters basic rental request information into a client device with its own UI that's external to our system
- that device sends the request to a server where our system resides (*the client device might be wired to an in-house server or to a remote cloud-based server*)
- our system calculates rental terms
- our system sends back a rental agreement to the client
- the clerk presents the customer with the physical tool and with the rental agreement

The following is a projected diagram that summarizes the steps mentioned above and tries to capture the main information flows:

The initial Tool Rental Application: physical representation



Assumptions and Constraints

These assumptions can be confirmed and validated upon meetings and discussions with stakeholders such as business analysts, user groups and management.

Business needs

The application need as is currently defined is a very simple one that performs a single task on a tool catalog of limited size. The volume of per-day tool rental requests in a single store might be in the hundreds, tens or even single digits. This will limit the initial size of computing equipment required and would be of low cost. The main cost would be developer time and that would be modest also due to the initially small nature of the application.

If timing is not an issue, then a more robust approach can be taken with an eye on extensibility to coming features and system growth and expansion, scalability to future higher volume needs for increased throughput, optimization and speed of system performance.

If timing is an issue and the system needs to be live and deployed by a certain near-term date or there are other pressing needs and developers are only allotted a limited amount of time to work on the system then efforts can be scaled down to focus on only the most urgent and immediate needs in a more direct and brute-force approach which might require more refactoring in the future.

Charge calculation rules

Determining date type: Weekday vs. Weekend vs. Holiday

- By definition, weekday and weekend are distinct and never overlap.
- In the USA, holidays in practice fall on weekdays but historically and theoretically can occur on weekends eg. July 4 which leads to the concept of July 4 actual vs. date of public observance. For simplicity and for charging purposes, I am assuming that:
    - holidays will always occur on weekdays
    - "weekday charge rules" only apply to **regular non-holiday weekdays**

This produces three distinct date scenarios: **W** - regular weekday, **E** - weekend, **H** – holiday, which would imply the following truth table for pricing rules:

Weekday charge?	Weekend charge?	Holiday charge?	Resulting charge rule per date scenario
N	N	N	Always free
N	N	Y	H
N	Y	N	E
N	Y	Y	E or H
Y	N	N	W
Y	N	Y	W or H
Y	Y	N	W or E
Y	Y	Y	Always charge

Non-Functional Requirements

Potential for emerging needs should be taken into consideration. This can include performance, security, scalability, and extensibility for reuse elsewhere (*see **Future Roadmap** for more on this*).

Architectural Strategy

General approach to minimize technical debt

Traditional approaches have attempted to reduce future refactoring of code by anticipating coming needs in the development of the initial system codebase. The inherent problem is that sometimes the future is unknown and emerging needs my make past anticipations obsolete introducing its own technical debt.

In recent times with the growing adoption of Agile development a popular saying is **YAGNI** – *You Ain't Gonna Need It*. To code now only if there is a present need. This will avoid project creep and prevent overworking a solution past an absolutely necessary point. The code and coding staff should be nimble enough that when new system changes are needed, one will only then do a modest amount of refactoring.

I prefer to take a middle ground. If within a very familiar business domain where a modest amount of forethought and coding infrastructure now will eliminate a significant amount of coding in the future, then pursue it. When operating in unfamiliar areas, exercise caution against overpreparing in advance. Use a step-by-step approach in simple increments where minimal assumptions will be made about the future.

The timing issue mentioned earlier will also affect our approach. If the goal is to release a simple solution with a short turnaround time, then some elements of design impacting optimization and scalability will be released later.

The task at hand is a Point-of-Sale system which is a well-defined domain. Since the current goal is merely to checkout a tool rental by receiving some very simple input, doing some very simple product and pricing lookups and then performing a modest calculation, we would definitely want to keep things simple and to avoid overkill. That being said, where tweaks to prepare for extendibility and scalability are fairly simple, they should be employed now due to the high probability of their usefulness later.

Software technology and tools

This demonstration will be sticking with plain **Java** as its primary coding language and will used the **JUnit** framework for testing purposes. To avoid overkill for such a simple task, no third-party libraries or tools will be utilized at the present but some will be recommended in the **Future Roadmap** later.

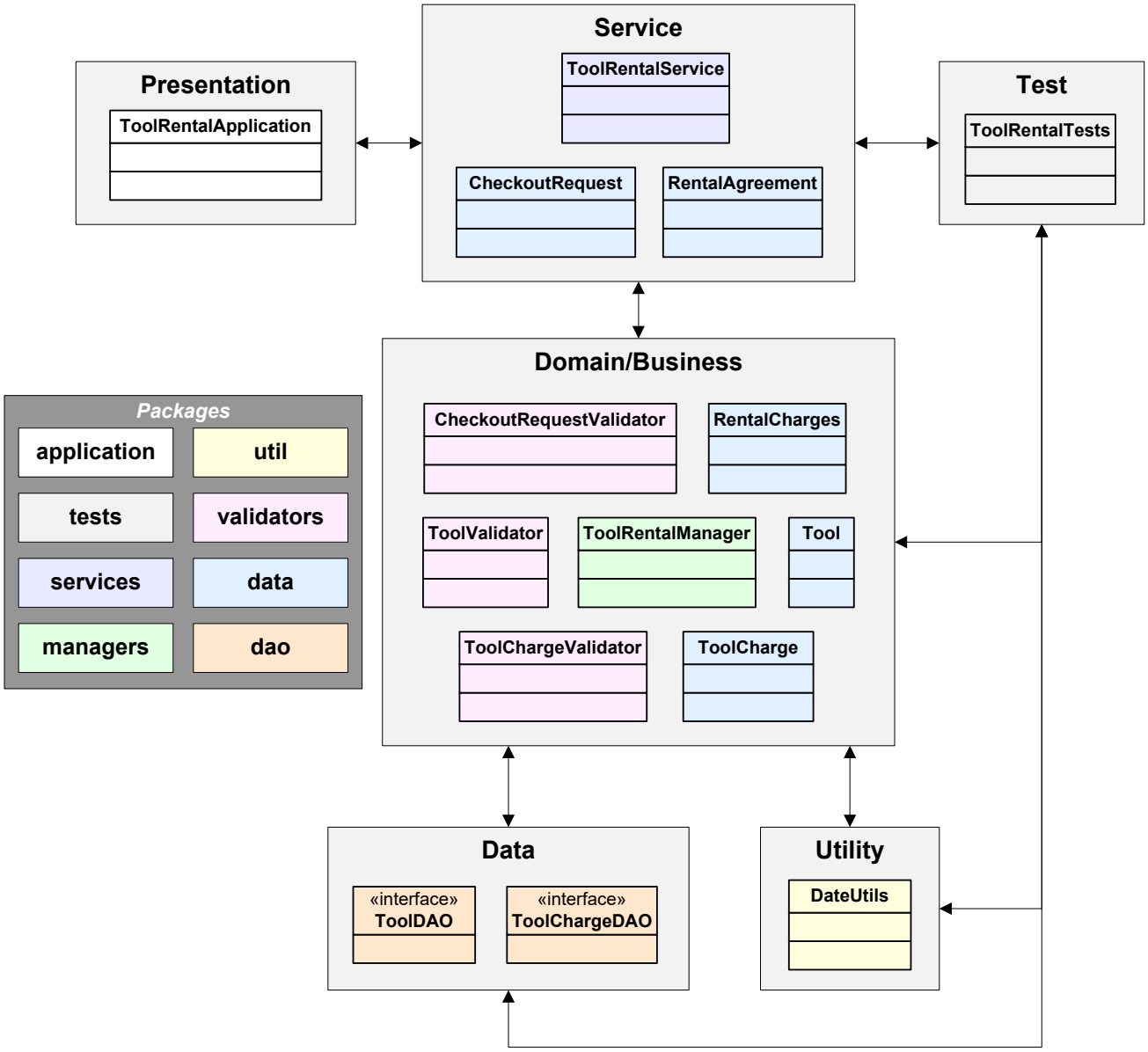
System layers

The system employs an object-oriented and modular architecture. Separation of concerns into system layers should be maintained in either case as this is fairly simple and will ensure modularity, maintainability, testability, and scalability. The key layers are:

- **Presentation Layer**: While there's currently no actual UI component, the Application class will serve as a simple placeholder for this layer as it makes a checkout request to receive a nicely formatted Rental Agreement from the Service layer.
- **Service Layer**: Receives requests and routes work to relevant business classes. Currently it will receive a checkout request from an external origin and then formulate a CheckoutRequest object to pass along to the Business layer with the expectation to receive a formatted Rental Agreement in return.
- **Domain/Business Layer**: This will currently receive a checkout call from the Service layer with a CheckoutRequest. With that information it will handles business logic, such as rental charge calculations based on defined rules. Models core business entities like Tool, RentalAgreement, and CheckoutRequest.
- **Utility Layer**: Provides helper classes for date and formatting operations.
- **Data Layer**: Holds predefined data for tool definition information and tool pricing schedules.
- **Test Layer**: Holds unit and regression test modules.

(see diagram on next page)

The following diagram shows the system layers and how the primary Java modules will fit into their respective layers. *For full details on how individual modules will be arranged into the system package structure, see* ***Module Organization / Package structure*** *section later.*



Implementation plan

The paragraphs below for ***Suggested initial optimization/enhancement*** *should be followed if time allows. They are deemed to be very likely needs for a real-world POS system. The later section* ***Future Roadmap*** *contains other suggestions for system improvements that will not be included in an initial implementation.*

As a stub in place of a **Presentation layer**, create a simple ToolRentalApplication class with a simple main() method. The main() method should call a checkout() method from a class in the Service layer with simple hardcoded inputs for a single tool rental use case and in turn receive a `String` response containing the formatted

text of a rental agreement (*see **System Overview/Requirement summary** above*). Wrap this within a try/catch block to report unexpected errors.
Suggested initial optimization: a multi-threaded approach will increase throughput in a real-world POS system in the event that simultaneous checkout() requests are allowed. A simple implementation has the response instead coming in the form of `CompletableFuture<String>` and then a shutdown() call would be issued to the multithreaded service in order to wrap things up.

The **Service layer** will initially consist of a class with a single checkout() method that will receive a request with basic tool rental parameters from either a Presentation layer client or a unit test method. It will build a **CheckoutRequest** bean object from the request parameters and then pass it along to the checkout() method of a manager class within the Business layer. It will receive a response with a **RentalAgreement** bean object where that object's getFormattedAgreement() method will return a nicely formatted text to the client while handling the styling of currency, percentages, and dates. The call to checkout() should be wrapped within a try/catch block so that Exceptions are handled cleanly with proper error messages which might include input and data validation errors generated by the lower-level checkout().
Suggested initial optimization: as mentioned earlier, a multi-threaded approach would be implemented here. Within the service class, an `ExecutorService` would be instantiated and a public wrapper to access to its `shutdown()` method would be provided. The service's `return` statement that would wrap the main body of the checkout() method within a `CompletableFuture.supplyAsync()` that uses the `ExecutorService` object.

The **Business layer** will have all code that contains business logic and will consist of:

- A manager class with a checkout() method which receives a **CheckoutRequest** from the Service layer's checkout() and responds with a **RentalAgreement** if no validation exceptions are thrown. This checkout() method will call other methods which issue validation and a call to calculate charges
- Validator classes which will initially validate the input of the **CheckoutRequest** and also include other validator classes if time allows (*see **enhancements** below*). At a minimum, rental day count and discount percentage should be validated
- Simple Data/bean classes to represent a Tool, a ToolCharge and calculated RentalCharges.

The **Tool** and **ToolCharge** derived from a **CheckoutRequest** will be based on Data layer lookups via DAO classes. As already stated in the functional specification for the application, the charge calculation (calculateCharges()) will be a function of a pricing lookup based on the tool type less a discount percent. The greatest complexity lies in determining how many days to charge a given daily rate for a tool type based on the rental start and end dates. A separate method calculateChargeDays() will loop between the rental start date and end date while checking against daily pricing rules for the tool type to determine total days to be charged (*see **Future Roadmap** for a more optimal calculation method*).
Suggested initial enhancements: as mentioned earlier, other validators can be added to verify that **Tool** and **ToolCharge** objects contain valid information that might arise from bad data entry, imports and transcription errors in case we cannot fully trust the data received from the Data layer. Also add a sanity check to additionally validate the **CheckoutRequest** against a maximum rental day limit of 4000 days will prevent unrealistic inputs and also prevent performance hits during calculations if a very large date range was entered accidentally.

The **Data layer** in lieu of a regular database will maintain hardcoded data stores of all **Tool** and **ToolCharge** Objects in DAO classes within key-value pairs in HashMaps. A **ToolDAO** will have a getTool() method to fetch **Tool** info based on a tool code key and a **ToolChargeDAO** will have a getToolCharge() method to fetch **ToolCharge** info based on a tool type key. Since real-world systems typically employ SQL or noSQL databases within their data layers a wise choice would be for Business layer calls to directly hit DAO interfaces which will in turn be implemented by DAOMapImpl classes for HashMap-based lookups. This provides future flexibility to write database-oriented implementations while keeping the access methods the same from outside the Data layer.

The **Utility layer** will initially consist of isWeekend() and isHoliday() methods called by the calculations in the Business layer in order to determine whether specific dates occur on weekends or holidays.

Module Organization

Package structure

Java source code will be organized under the **src\main\java\com\toolrental** folder (***com.toolrental*** *package*) in the following way according to the nature of its activity:

- **application** –will contain the top level application class
- **dao** – data access classes
- **data** – low-level, simple bean classes for data. Mostly with just setter/getter methods and some with special helper methods for formatting, etc.
- **managers** – classes containing high-level business logic and might pass off low level work to utility classes or calculators. These also will typically access its data needs from resident DAO classes
- **services** – controller classes that receive requests and act as traffic cops to pass along work business classes within **managers**
- **util** – for utility classes
- **validators** - for classes containing validation logic for the business classes within **managers**

Java test code will be organized under the **src\test\java\com\toolrental\tests** folder (***com.toolrental.tests*** *package*) and for starters will have a unit test class for five of the packages (*for diagram purposes in this document, they are all lumped together as "**ToolRentalTests**"*):

- dao**ToolRentalDAOTest** - for testing methods in two DAO interfaces/classes
- managers**ToolRentalManagersTest** – for testing methods in one manager class
- services**ToolRentalServicesTest** - for testing methods in one service class
- util**ToolRentalUtilTest** - for testing methods in one util class
- validators**ToolRentalValidatorsTest** - for testing methods in three validator classes

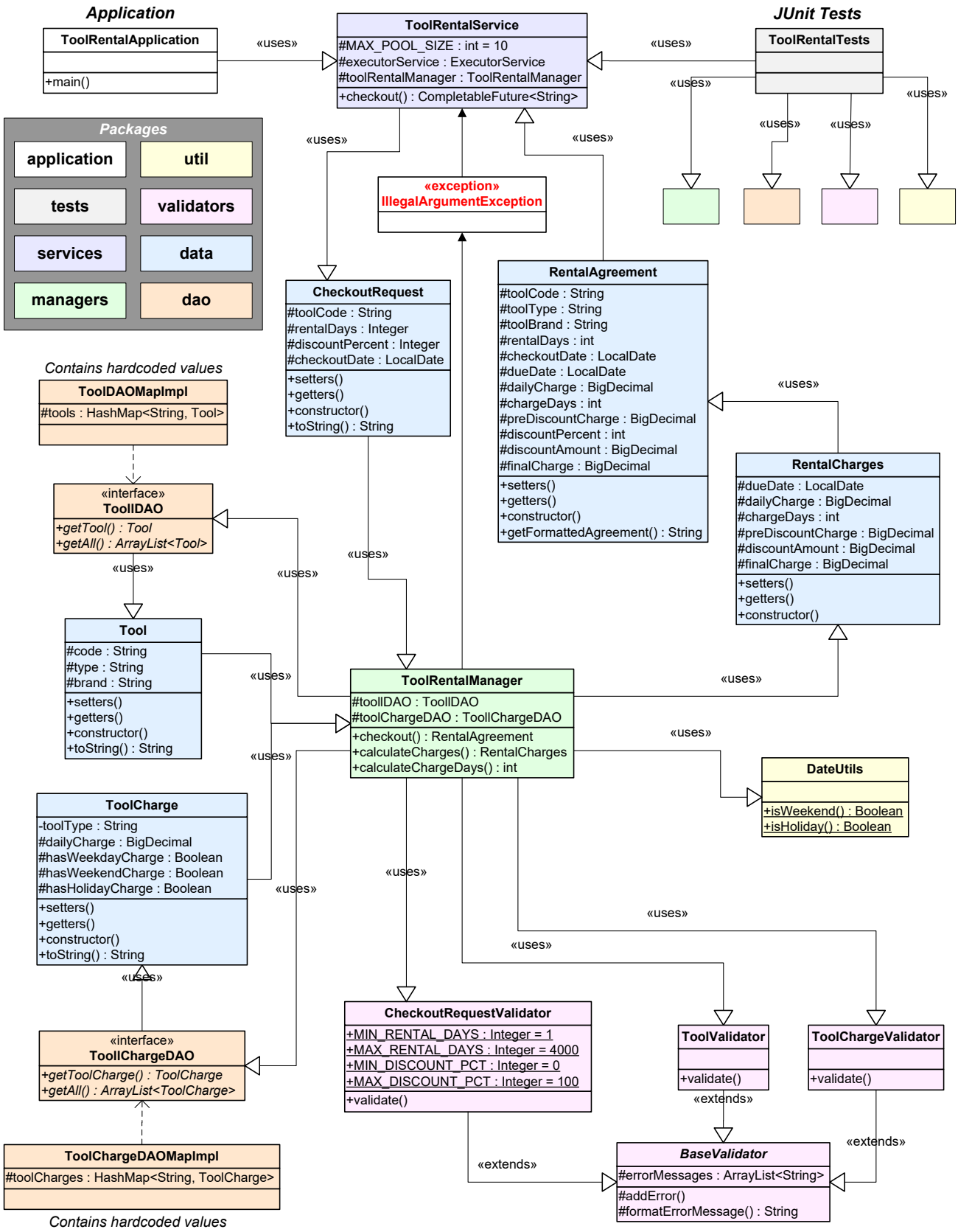
As the codebase grows more robust and complex over time, other packages and subpackages can be added (*eg. calculators*) and the structure can be refactored for the purpose of organizing things better.

*Source code API documentation can be accessed from **javadoc\index.html***

Components

The following is a diagram of the application components. It utilizes the basic structure of a UML diagram with adjustments to better show data flow and displays each Java class and interface in the system along with their primary attributes and methods. It is color-coded to show which components belong to the various packages described above. (*Stubs for many of these classes have already appeared in the earlier **System layers diagram***).

(see diagram on next page)


Application
ToolRentalApplication
+main()
«uses»
ToolRentalService
#MAX_POOL_SIZE : int = 10
#executorService : ExecutorService
#toolRentalManager : ToolRentalManager
+checkout() : CompletableFuture<String>
«uses»
JUnit Tests
ToolRentalTests
Packages
application
util
tests
validators
services
data
managers
dao
«exception»
IllegalArgumentException
CheckoutRequest
#toolCode : String
#rentalDays : Integer
#discountPercent : Integer
#checkoutDate : LocalDate
+setters()
+getters()
+constructor()
+toString() : String
RentalAgreement
#toolCode : String
#toolType : String
#toolBrand : String
#rentalDays : int
#checkoutDate : LocalDate
#dueDate : LocalDate
#dailyCharge : BigDecimal
#chargeDays : int
#preDiscountCharge : BigDecimal
#discountPercent : int
#discountAmount : BigDecimal
#finalCharge : BigDecimal
+setters()
+getters()
+constructor()
+getFormattedAgreement() : String
Contains hardcoded values
ToolDAOMapImpl
#tools : HashMap<String, Tool>
«interface»
ToolIDAO
+getTool() : Tool
+getAll() : ArrayList<Tool>
RentalCharges
#dueDate : LocalDate
#dailyCharge : BigDecimal
#chargeDays : int
#preDiscountCharge : BigDecimal
#discountAmount : BigDecimal
#finalCharge : BigDecimal
+setters()
+getters()
+constructor()
Tool
#code : String
#type : String
#brand : String
+setters()
+getters()
+constructor()
+toString() : String
ToolRentalManager
#toolIDAO : ToolIDAO
#toolChargeDAO : ToolIChargeDAO
+checkout() : RentalAgreement
+calculateCharges() : RentalCharges
+calculateChargeDays() : int
DateUtils
+isWeekend() : Boolean
+isHoliday() : Boolean
ToolCharge
-toolType : String
#dailyCharge : BigDecimal
#hasWeekdayCharge : Boolean
#hasWeekendCharge : Boolean
#hasHolidayCharge : Boolean
+setters()
+getters()
+constructor()
+toString() : String
CheckoutRequestValidator
+MIN_RENTAL_DAYS : Integer = 1
+MAX_RENTAL_DAYS : Integer = 4000
+MIN_DISCOUNT_PCT : Integer = 0
+MAX_DISCOUNT_PCT : Integer = 100
+validate()
ToolValidator
+validate()
ToolChargeValidator
+validate()
«extends»
«interface»
ToolIChargeDAO
+getToolCharge() : ToolCharge
+getAll() : ArrayList<ToolCharge>
BaseValidator
#errorMessages : ArrayList<String>
#addError()
#formatErrorMessage() : String
ToolChargeDAOMapImpl
#toolCharges : HashMap<String, ToolCharge>
Contains hardcoded values

Technical considerations and trade-offs

- Class variables and methods are typically set as **protected** instead of **private** to allow for extensibility.
- While it's expected that **tool type** will eventually be fleshed out to the point where it has additional attributes and merits its own bean class, it will be kept to a simple **String** for now. The same goes for **brand** which might evolve into some type of vendor information class. Once those things happen these entities and entries would require their own validation rules.
- Holiday definitions will now be hardcoded rules within price calculations but they might eventually evolve into a List of Java lambda expressions with individual code blocks for each separate holiday rule (*with the caveat that lambdas might involve performance hits unless these are executed as a one-time calculation*)
- **ToolCharge** can theoretically be abstracted without the tool type property which might make sense if the same pricing rule might apply to whole swaths of tool types, but keeping a flatter structure for now which does include the tool type.
- Java's slower **BigDecimal** is used for dollar values as opposed to faster **double** since the former offers the precision required for currency calculations.
- Even though it isn't a normal practice to clutter code with so many "todo" comments, I have used them for this exercise to inform possibilities for future improvements and expansion.

Future Roadmap

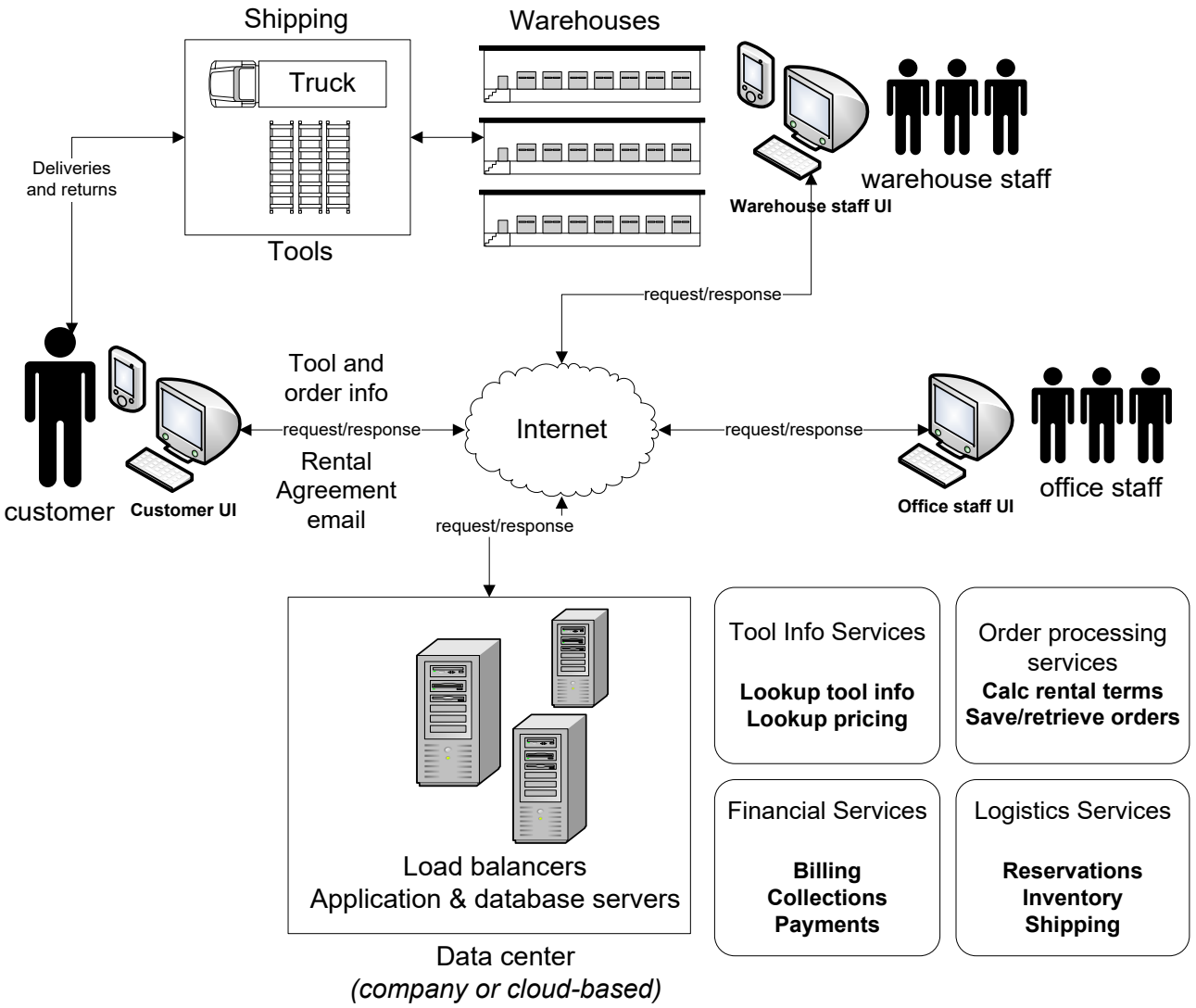
Challenges inherent with a likely expansion scenario

In the **Non-functional Requirements** section earlier, we briefly touched upon potential future issues. These can come into play with a possible expansion of our small POS checkout system that could realistically include the following:

- the customer's interaction with a physical store and clerk moves to a web or mobile app that lets the customer create a rental order/shopping cart
    - for a start date which is either immediate or scheduled for specified future date
    - for a specified number of days
    - with one-to-many tool product line items where for each item the user will:
        - perform a tool lookup by description and select a tool
        - enter tool quantity >= 1
        - for each quantity change a request will be fired to our system with similar information to a **CheckoutRequest** and will receive an item price generated by our system
        - a request will be issued to an inventory and reservation system to receive information on whether any or all of the line items quantity will be available for the requested date range
    - for each change to rental days, a new pricing lookup will need to be done for each line item
    - the system auto-calcs customer-based discount based on certain customer criteria, time of year and store policies
    - user submits and pays for the order
    - order is shipped or available for in-store or warehouse pickup
- At a later point the customer will ship or physically return the item(s) and be assessed a possible fee for late returns. For each item that gets returned, the inventory system needs to be updated

And this doesn't even take into account user/customer management, billing, collections and payment processing.

Possible expansion of Tool Rental Application: physical representation



Refactoring to address such an expansion

While this future expansion can be handled with a separation of concerns and the creation of new systems or services that are external to our system, many of these needs would require the following technical changes to our system to address the needs of performance, security, scalability, and extensibility for reuse elsewhere:

- The service would likely be converted to a REST service and access only granted to requests accompanied by a valid API key. Spring Web Services can help here and restricting service access can be accomplished with Spring Security by creating an API Key Authentication Filter.
- The service will no longer merely have a single checkout() command. In fact, checkout would be handled by a separate order tracking system while our system still can keep the elements needed to perform the following operation:
    - lookupTool(toolDescription) which based on a full or partially entered description will return a list of zero to many {tool code, type, brand} for all matching tool catalog entries. AI tools can be employed to match free-form descriptions to our tool catalog.

- lookupPricing(toolCode) which will return **RentalCharges** (*discount percent will now need to be externally calculated*)

Optimization opportunities

Determining a tool type's total days to charge over the rental period

This is now accomplished by the brute-force method within **ToolRentalManager.calculateChargeDays()** of looping day by day through the rental period and, categorizing the day as a weekday, weekend or holiday and then analyzing via the ToolCharge flags whether to add that day to the total chargeable days. **This produces O(n) performance**.
In theory, this can blow-up if the customer requests an extremely large rental period. To mitigate this a bit, we already enforce a MAX_RENTAL_DAYS validation so this doesn't get out of hand. Additionally, I have performed basic stress-testing on the current brute-force looping algorithm and it only starts to blow up with poor performance upon extremely long rentals which are decades in length and way past MAX_RENTAL_DAYS.

Still, if due to high-volume use this causes a concern of an extra burden to our servers, **the following can be done to convert this from a looped search to a mere arithmetic calculation** after very limited and speedy HashMap lookups which would typically result in an **improvement to O(1) instead of O(n).**

At application startup, create a cached HashMap which would contain a **Date key entry** for every day between the current system date and a maximum allowed lookahead date. Building this information would require a one-time loop through those dates and to insert to the HashMap the **date key** plus a "**value**" which would be a Java object (*ie. called **DayCounts***) containing a running count of **{regularWeekdaysSinceStart, weekendDaysSinceStart, holidayDaysSinceStart}** (*using the pre-existing **DateUtils** methods for each date based upon **isWeekend()** and **isHoliday()***). Once the HashMap is built, then the **calculateChargeDays()** method would merely look up the two **DateCounts** entries corresponding to the rental start and end dates and then capture their differences for weekday/weekend/holiday and use arithmetic to apply it to the rules defined in the **ToolCharge** flags. Additionally, the **isHoliday()** determination itself can be shipped out to **holiday calc APIs** (*see https://www.timeanddate.com/services/api/ and others*) which will ease coding and potentially speed up the initial HashMap build. In addition to the option of a one-time startup build, a lazy initialization approach can be used where if rental dates, don't yet exist in the HashMap, the **DateCounts** values can be inserted for those missing dates on the fly.

Flattening the DAO call to lookup Tool and ToolCharges values from a tool code

Some possible performance improvement plus coding simplicity can be gained from caching the results of the current two-step **Tool** and **ToolCharges** DAO lookups into a single HashMap with the key being a **tool code** and the value being a new **ToolInfo** object which would contain **{toolType, brand, dailyCharge, hasWeekdayCharge, hasWeekendCharge, hasHolidayCharge, cacheInsertDateTime}** which are all the values needed to begin a rental charge calculation. The HashMap can be lazy initialized so that if the tool code is found, it returns ToolInfo and if not, then the old two step lookup is performed, AND if the size of the HashMap is less than an assigned MAX_CACHE_SIZE then a new ToolInfo object will be built and added to the HashMap for the tool code. A schedule daemon process will be created that will remove expired entries from the HashMap based upon aging rules – this will ensure most recent/popular tool products are available in the cache.

Robustness for handling increase in scale

- **Data Access Layer** (DAO classes): it is now hardcoded. Normal systems require data sets that are flexible and that can be quickly updated for changes. A **SQL** or **noSQL** database would likely be used in the future

for this purpose. At that point, new DAOImpl classes would implement the DAO interfaces to tie in their functionality to their respective database schemas.

- **System logging**: Java frameworks such as **Log4j** should be implemented for robust system logging which will make it easier to track down issues. Logging levels can be set to DEBUG, INFO, WARN, ERROR, etc. based upon need for whether a message appears in a DEV, TEST or PROD environment.
- **Dependency injection**: **Spring annotations** can be used for implementing **auto-wiring** so that classes be initialized with default objects (*eg. Manager classes can be instantiated with DAO objects which are connected to different databases depending on their environment*). This makes configurations much easier for different environments (eg. DEV, TEST, PROD) and results in decreased coupling, better testability and reusability.
- **Application settings handling**: For greater flexibility in changing application settings (*eg. **MAX_CACHE_SIZE, MAX_RENTAL_DAYS**, etc.*), instead of hardcoding settings like the present, the settings can be derived from one of the following:
    - a **properties file** (*pros: doesn't depend on a database connection*)
    - **settings table** in a database (*pros: changes can be recognized even without a system restart*)
    - an **admin module** that hits a service with requests to alter in-memory system settings (*pros: advantages of both but would still need to persist changes somewhere to make this permanent*)
- **System health and status monitoring**: All sorts of system monitoring tools eg. Nagios, Splunk; Java profiling; and database profiling can be implemented. These can generate alerts and also help identify problems.

How AI tools and frameworks can help

- **Natural Language Processing**: Use NLP models to:
    - Execute tool description lookups to retrieve tool codes based on fuzzy descriptions
    - Enhance error messages and make them more intuitive and friendly by understanding the user input context and offering tailored suggestions (*e.g., **ChatGPT API** or **BERT** for user interactions*)
- **Test case generation**: Tools like **Diffblue Cover** and even **ChatGPT** can generate test cases automatically for the core business logic, reducing the time needed to write unit tests.
- **Automated Javadoc generation and even automated code generation**: this that's where things are heading and **ChatGPT** does a fairly good job to generate the bulk work where careful review and proofreading still need to be done and re-tinkering with the seed-wording which generates the code.

To keep enterprise proprietary information private when using ChatGPT

Avoid directly inputting sensitive details, anonymize data where possible, use the "temporary chat" feature to prevent conversation history storage, consider opting out of data sharing in settings, and explore enterprise-specific solutions like OpenAI's private models or dedicated enterprise plans (see https://openai.com/index/data-partnerships/) which offer stricter data privacy controls. Essentially, only use public ChatGPT for general insights while carefully guarding any confidential information. If AI needs are deemed to be extensive and financing allows, an in-house, enterprise AI framework can be implemented to improve security of proprietary information.

Other add-ins and third-party frameworks and APIs

- **Mockito:** for automating unit testing.
- **Apache Commons and other Java software libraries:** to expand the Java language and speed and ease the software development process