Code Review Summary

Overall Quality Assessment:

The code **meets expectations** in terms of functionality and structure but has areas that require improvements, especially in error handling, efficiency, and adherence to SOLID principles.

Detailed Feedback:

Input Validation:

- **Tool code validation**: Not explicitly implemented. The assumption seems to be that invalid or nonexistent tool codes will be prevented at a higher level (such as the UI) or by failing to find the tool in the `toolInventory` map, leading to a `NullPointerException`.
- **Rental day count validation**: Properly implemented in `Cart` by computing end date from start date and total days.
- **Discount percent validation**: Correctly implemented in `CheckoutService`, with exceptions thrown for invalid values.

Comprehensive Functionality and Logic:

- **Due date calculation**: Correctly implemented in `Cart` by adding total days to the start date.
- **Daily charges based on tool type**: Implemented correctly but lacks flexibility for changing rates without modifying code.
- **"No charge" days logic**: Correctly computes chargeable days in `CheckoutService` but the process could be more efficient and readable.
- **Discount logic and rounding**: Correctly implemented in `CheckoutService`, meeting requirements for rounding and calculations.

Rental Agreement:

- All required details are generated and included in the agreement. However, the system only deals with one tool per transaction, which limits functionality contrary to the potentially multi-tool support suggested by `Cart`'s design.

Error Handling:

- User-friendly messages are provided for different exception scenarios; however, broader exception handling for unforeseen errors is not present.

Print Functionality:

- The `RentalAgreement.toString()` method correctly formats output, meeting the specification for dates, currency, and percent.

Code Quality (S.O.L.I.D Principles):

- **Single Responsibility**: Each class has a clear responsibility. However, `CheckoutService` might be overburdened with responsibility for both checkout logic and calculation of charge days.
- **Open-Closed**: The current design makes adding new tool types or changing holiday rules difficult without modifying existing code.
- **Liskov's Substitution**: Not applicable as there is no inheritance hierarchy among the entities.
- **Interface Segregation and Dependency Inversion**: Not fully applied. The use of concrete classes over interfaces or abstractions for dependency injection (e.g., in `CheckoutService`) limits flexibility for future changes.

Unit Testing:

- Covers specified scenarios but lacks depth in covering edge cases or invalid paths (e.g., attempting to rent a tool not in inventory).
- Test names accurately describe what they are testing.

Additional Coding Best Practices:

- **Comments and Documentation**: Adequate, providing clarity on functionality.
- **Code Reuse**: Present but could be optimized, especially in the handling of business rules around holidays and chargeable days.
- **Null Handling**: Direct null checks are used, but the use of `Optional` or similar patterns could make the code more robust.
- **Indentation and Spacing**: Consistent across the codebase.

Suggestions for Improvement:

- **Tool Validation**: Implement explicit validation for tool codes based on the available inventory.
- **Flexibility**: Introduce interfaces or abstract classes to abstract out tool information and pricing, improving adherence to SOLID principles.
- **Error Handling**: Broaden error handling to catch and handle unexpected exceptions, improving robustness.
- **Efficiency**: Optimize the calculation of chargeable days, possibly by using more efficient data structures or algorithms.
- **Testing**: Increase the test coverage to include more edge cases and invalid input scenarios.

The code is functional and largely meets the required specifications, with areas identified for improvements to maintainability, flexibility, and robustness.