CODE Refactoring Comparison Report

1. Code Structure

Original Code:

- **onCreate method:** Holds nearly all initialization and business logic.
- **Unorganized variables:** Many class member variables, some redundant or not clearly named.
- **Repeated and deeply nested logic:** Complicated conditional checks.
- Uses repetitive if-else chains and string concatenations.

Refactored Code:

- Clear methods for initialization, event setup, date validation, and calculation.
- **Meaningful variable names:** Improved naming for views and logic variables.
- **Modular helper methods:** Dedicated methods for leap year check, day/month addition, day adjustment for month length.
- **Reduced redundancy:** Month days and names held in arrays; reusable methods handle repeating logic

---

2. Performance

Original Code:

- **Repeated calculations and conditions** inside loops, e.g., leap year calculation inside big conditional blocks.
- **String concatenation in loops** for month name assembly, inefficient and risky with potential bugs.
- **Excessive nested while loops** for day/month/year adjustments that lack clean stop conditions or optimization.
- **Validation checks scattered**

Refactored Code:

- **Leap year computed once per calculation,** reused when needed.
- **Array lookup for month names and days,** O(1) access.
- **Iteration for day addition/subtraction done efficiently in a single loop.**
- **Centralized validation and adjustment methods** reduce error surface and speed checks.
- **Avoid unnecessary string concatenation in loops,** uses concise formatting.

- **Early exits on invalid input,** reducing wasted computation cycles.

CODE Refactoring Comparison Report

3. Readability & Maintainability

Original Code:

- **Hard to follow flow** due to inline complex logic in **onCreate**.
- **Variable naming lacks clarity** (e.g., **resu**, **dayint**, **monthcounter**), detracts understanding.

Refactored Code:

- **Clear, semantic method names** for readability and self-documentation.
- **Single responsibility principle adhered to:** each method does one well-defined task.
- **Use of comments and Javadoc-style method docs** clarify intent.
- **calculateNewDate** reads more like a recipe—input → validate → compute → output.

---

4. User Interface Interaction

Original Code:

- Toasts used for feedback but scattered and inconsistent messages.
- Clears inputs and resets states in one place but mixed with logic.

Refactored Code:

- Toast messages are consistent and appear at logical flow points.
- Clear inputs method distinct and easily extendable.
- UI responses separated clearly from logic, improving future UI changes.

---

5. Error Handling and Validation

Original Code:

- Out of range checks present but mixed with logic.

- Some catch blocks present but limited.

Refactored Code:

- Clear user feedback on errors via Toasts.
- Guard clauses to minimize checking complexity.
- Proper catch for parsing exceptions

---

6. Scalability and Extensibility

Original Code:

- Rigid Logic: New rules/time units require large edits to nested conditionals
- Difficult to track how date is changed.

Refactored Code:

- Adding new time units is easier: just add a radio button and corresponding method if needed.
- Date mutation centralized in **addDays()** and **addMonths()** helpers.
- Logic can be extended with minimal impact—high cohesion and low coupling.

Summary:

Aspect	Original Code	Refactored Code
Structure	Monolithic, inline logic	Modular, well-organized methods
Performance	Repetitive, inefficient	Optimized loops, reduced recalculations
Readability	Hard to read & maintain	Clean, semantic names and flow
UI Handling	Mixed UI and logic	Clear separation and consistent feedback
Error Handling	Minimal	Proper try-catch and checks
Extensibility	Difficult	Designed for easy enhancement