# Banking Application – API Documentation

## Overview

This document describes the internal API (class interface) for a simple banking application built with Python and Tkinter, following Design‑by‑Contract (DbC).

## Architecture Component Diagram

## Class Diagram

## 

## BankAccount Class

Encapsulates core banking operations with DbC enforcement.

### deposit(amount: float) → float

Adds funds to the account and logs the transaction.

* Preconditions:
* amount > 0
* Postconditions:
* Returned value equals updated balance

Returns: New balance as float

### withdraw(amount: float) → float

Removes funds from the account if sufficient balance exists.

* Preconditions:
* amount > 0
* balance ≥ amount
* Postconditions:
* Returned value equals updated balance

Returns: New balance as float

## BankingApp GUI

Tkinter-based GUI that interacts with a single BankAccount instance, handles user input, and shows contract violations.

### create\_widgets()

Builds all GUI widgets.

### update\_display()

Refreshes balance display and color coding.

### do\_deposit()

Triggers a deposit via user input.

### do\_withdraw()

Triggers a withdrawal via user input.

### show\_violations()

Displays logged contract violations in a new window.

# Strategies for Implementing Correctness in Software Quality Management

Correctness ensures software adheres to specified requirements and behaves as intended without errors. Below are evidence-based strategies from literature, tailored for programming contexts, along with practical implementation details.

## 1. Formal Methods & Mathematical Proofs

Description: Use mathematical models to verify algorithms and logic before implementation.

Implementation:

- Tools: Coq, Isabelle, TLA+, Z notation

- Steps:

1. Write formal specifications for critical functions.

2. Construct mathematical proofs of correctness.

3. Translate verified designs into code.

Benefits: Eliminates logic errors in critical systems (e.g., aerospace, blockchain).

## 2. Test-Driven Development (TDD)

Description: Write tests before coding to enforce correctness by design.

Implementation:

- Workflow:

1. Red: Write a failing unit test.

2. Green: Write minimal code to pass the test.

3. Refactor: Optimize code, keeping tests green.

- Tools: JUnit (Java), pytest (Python), Jest (JavaScript)

Example (pytest):

# pytest example  
def test\_factorial():  
 assert factorial(0) == 1  
 assert factorial(5) == 120

Evidence: TDD reduces defects by 40–90% (Nagappan et al., 2008).

## 3. Static Analysis & Linting

Description: Automatically detect flaws in source code without executing it.

Implementation:

- Tools: ESLint, Pylint, SonarQube, Coverity

- Common Checks: Null pointer dereferences, Type mismatches, Dead/unreachable code

CI Integration:

# GitHub Actions with SonarCloud  
jobs:  
 analyze:  
 runs-on: ubuntu-latest  
 steps:  
 - uses: sonarsource/sonarcloud-github-action@master

Benefit: Detects ~30% of correctness issues early (SAST tools evaluation, 2020).

## 4. Design by Contract (DbC)

Description: Use contracts to define runtime expectations—preconditions, postconditions, and invariants.

Implementation:

- Tools: icontract (Python), Contracts for Java, Eiffel

- Process: Preconditions, Postconditions, Invariants

Example (Java):

@Requires("x >= 0")  
@Ensures("Math.abs(result \* result - x) < 0.001")  
public double sqrt(double x) { ... }

Impact: Reduces runtime errors by 25% (Meyer, 1997).

## 5. Property-Based Testing (PBT)

Description: Test code with automatically generated inputs to validate properties.

Implementation:

- Tools: QuickCheck (Haskell), Hypothesis (Python), jqwik (Java)

- Workflow:

1. Define general properties (e.g., reversibility).

2. Auto-generate 100–1,000 test cases.

3. Minimize failure inputs for debugging.

Example (Hypothesis):

from hypothesis import given  
from hypothesis.strategies import lists, integers  
  
@given(lists(integers()))  
def test\_reverse\_twice(l):  
 assert reverse(reverse(l)) == l

Benefit: Finds edge cases missed by unit tests.

## 6. Model Checking

Description: Exhaustively verify state transitions for correctness, especially in concurrent systems.

Implementation:

- Tools: TLA+, Java PathFinder (JPF), SPIN

- Process:

1. Define state models (e.g., mutex protocol).

2. Specify invariants (e.g., no deadlocks).

3. Explore all paths to check violations.

Example (TLA+):

\* Mutual exclusion  
Invariant == \A p \in Procs: ~(pc[p] = "critical" /\ pc[q] = "critical" for q ≠ p)

Use Case: Critical for concurrent systems like AWS DynamoDB.

## 7. Fuzz Testing

Description: Inject random/malformed inputs to discover crashes or unexpected behaviors.

Implementation:

- Tools: AFL, libFuzzer (C/C++), Jazzer (Java)

- Steps:

1. Define fuzz targets (e.g., parsers).

2. Run fuzzers over time.

3. Analyze crash logs.

Example (libFuzzer):

$ clang -fsanitize=fuzzer parser.c -o fuzzer  
$ ./fuzzer -max\_total\_time=3600

Evidence: Google found 30,000+ bugs via fuzzing (2023).

## Key Recommendations for Implementation

1. Risk-Based Prioritization: Apply formal methods to high-risk modules.

2. Automation in CI/CD: Integrate static analysis, PBT, and fuzzing.

3. Hybrid Strategy: Combine TDD (feature correctness) + PBT (edge cases) + DbC (invariants).

4. Track Quality Metrics: Monitor defect density, test coverage, and analysis warnings.

## References

- Nagappan, N. (2008). Realizing quality improvement through test driven development.

- Meyer, B. (1997). Object-Oriented Software Construction.

- Google Fuzzing Report (2023)