NODE Technical Book Club

C++ Software Design - Klaus Iglberger

G1: Understand the Importance of Software Design

- Software design is the essential part of software development.
- Language features are just tools, the design is what makes the difference.
- Dependency is the key problem, and the design is the art of managing dependencies and abstractions.

G2: Design For Change

- Changes in software are inevitable so design for easy change.
- Two principles:
  - Adhere to **Single Responsibility Principle (SRP)** to separate concerns.
  - Follow the **Don't Repeat Yourself (DRY)** principle to minimize duplication.

Separation of Concerns

- **Single Responsibility (SRP):** A class should have only one reason to change.
- Group only those things that truly belong together, and separate those that don't.
- **Example:** A document class with serialize and ExportToJson methods.

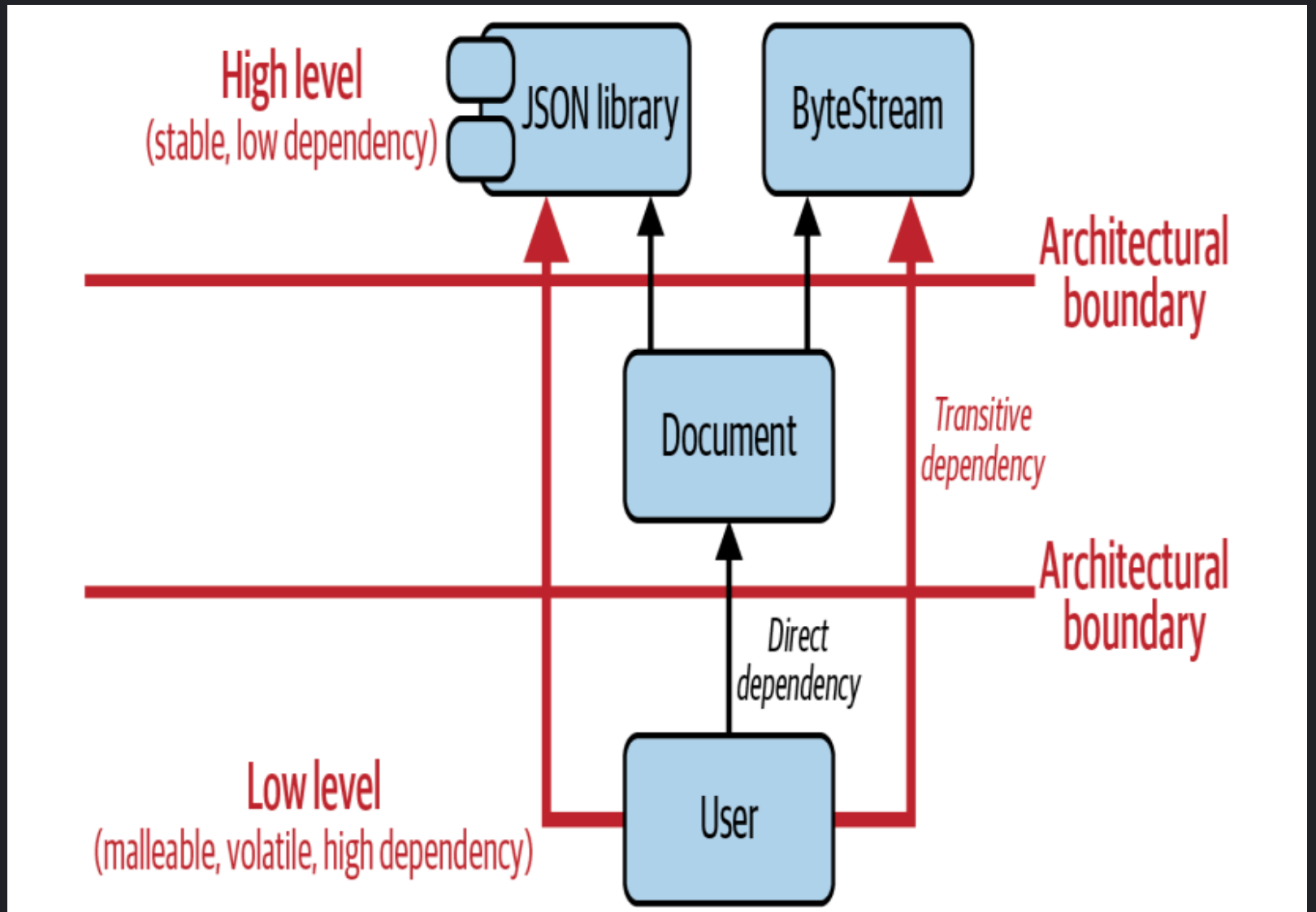


Figure 1-2. The strong transitive, physical coupling between `User` *and orthogonal aspects like JSON and serialization.*

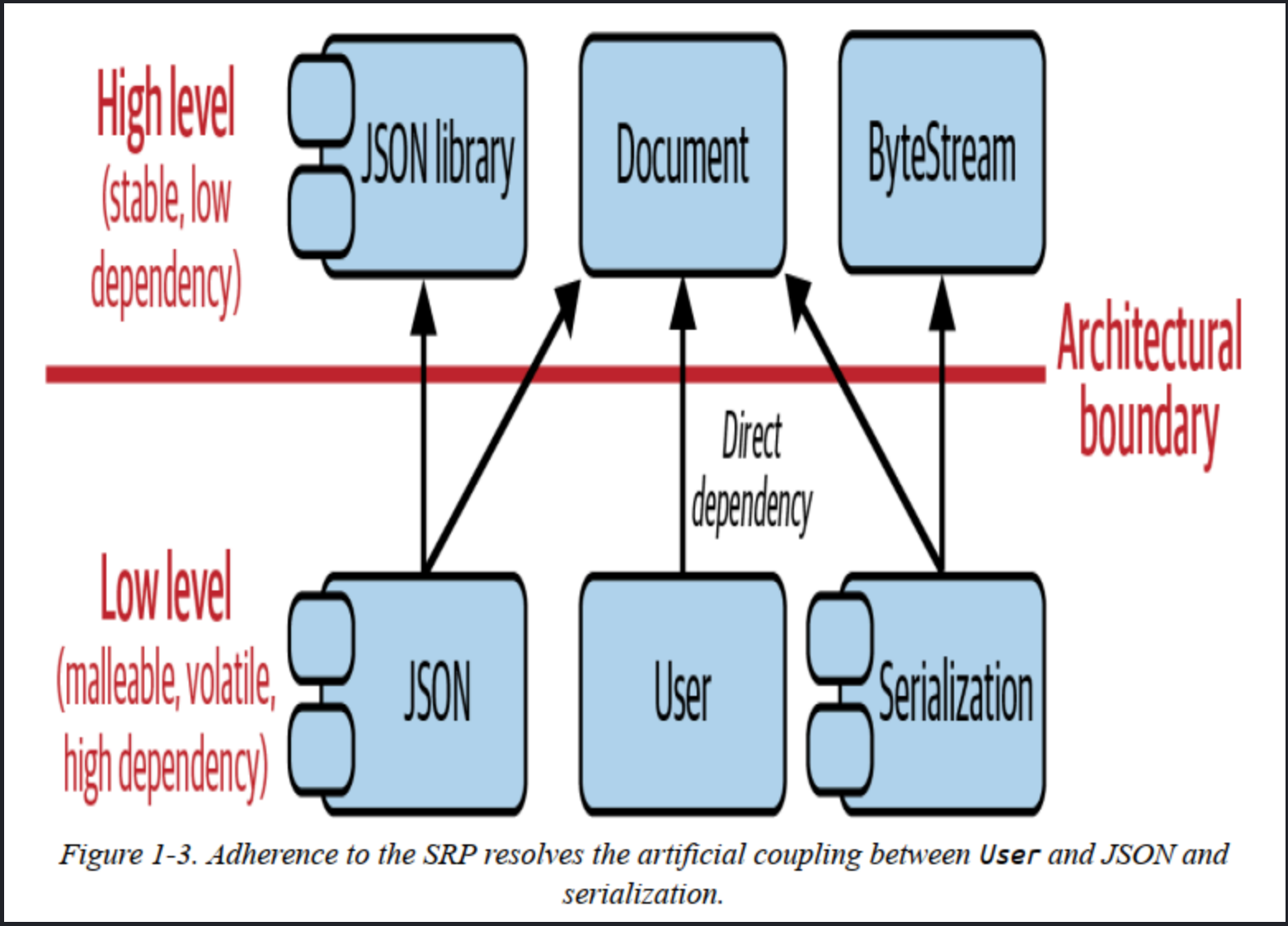


Figure 1-3. Adherence to the SRP resolves the artificial coupling between `User` *and JSON and serialization.*

Don't Repeat Yourself

- Do not duplicate some key information in many places.
- Design the system such that **we can make the change in only one place**.
  - **Example:** Tax calculation in different item types.

G3: Seperate Interface to Avoid Artificial Coupling

- **Interface segregation principle(ISP):** Clients should not be forced to depend on interfaces they do not use.
- **Example:** `Document` class have both exportToJSON and serialize methods. `exportDocument` method only uses `exportToJSON` method but still also depends on `serialize` method.

Refactor to:

```
class Document
: public JSONExportable
, public Serializable
{
    public:
    // ...
};
```

and

```
void exportDocument( JSONExportable const& exportable )
{
    exportable.exportToJSON();
}
```

G4: Design for Testability

- Software is constantly changing and test are the safety net.
- Design the software such that it is testable, even **easily testable** in the best case.

```
class Widget
{
    public:
        // ...
    private:
     void updateCollection( /* some arguments needed to update the collection */
    );

    std::vector<Blob> blobs_;
    /* Potentially other data members */
};
```

Challenge: How would you test `updateCollection` ?

Possible Solutions:

- Test the public methods where `updateCollection` is called.
    - Making test friend of the class.
    - Make it protected and derive a test class.

The True Solution: Separate Concerns

- Extract the private method into a separate class or free function.

```
class BlobCollection
{
public:
    void updateCollection( /* some arguments needed to update the collection */);
private:
    std::vector<Blob> blobs_;
};
```

G5: Design for Extension

- Software always evolves and grows, so it should be easy to extend.
- **Open-Closed Principle (OCP):** Software entities should be open for extension but closed for modification.

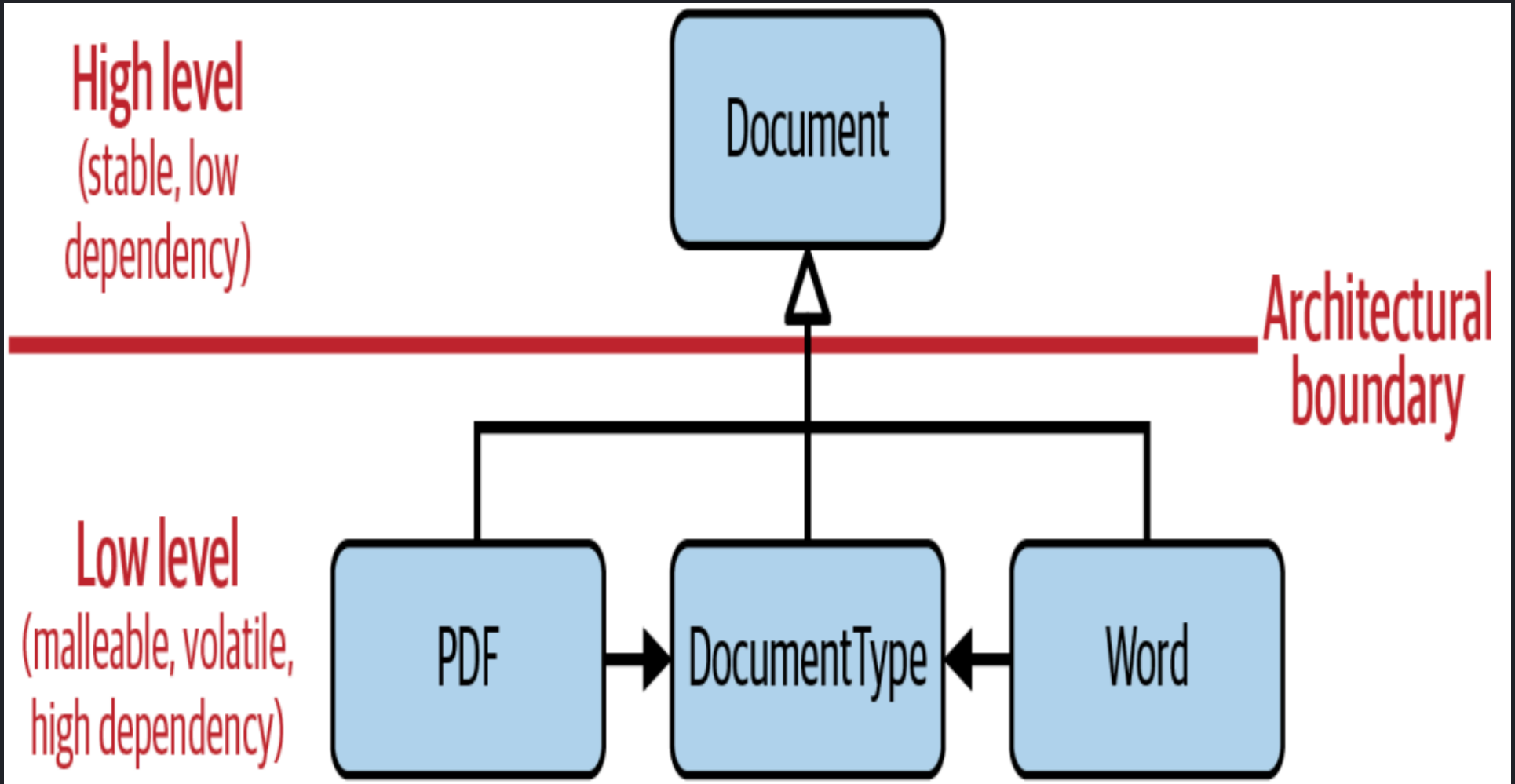


Figure 1-5. Artificial coupling of different document types via the `DocumentType` *enumeration.*

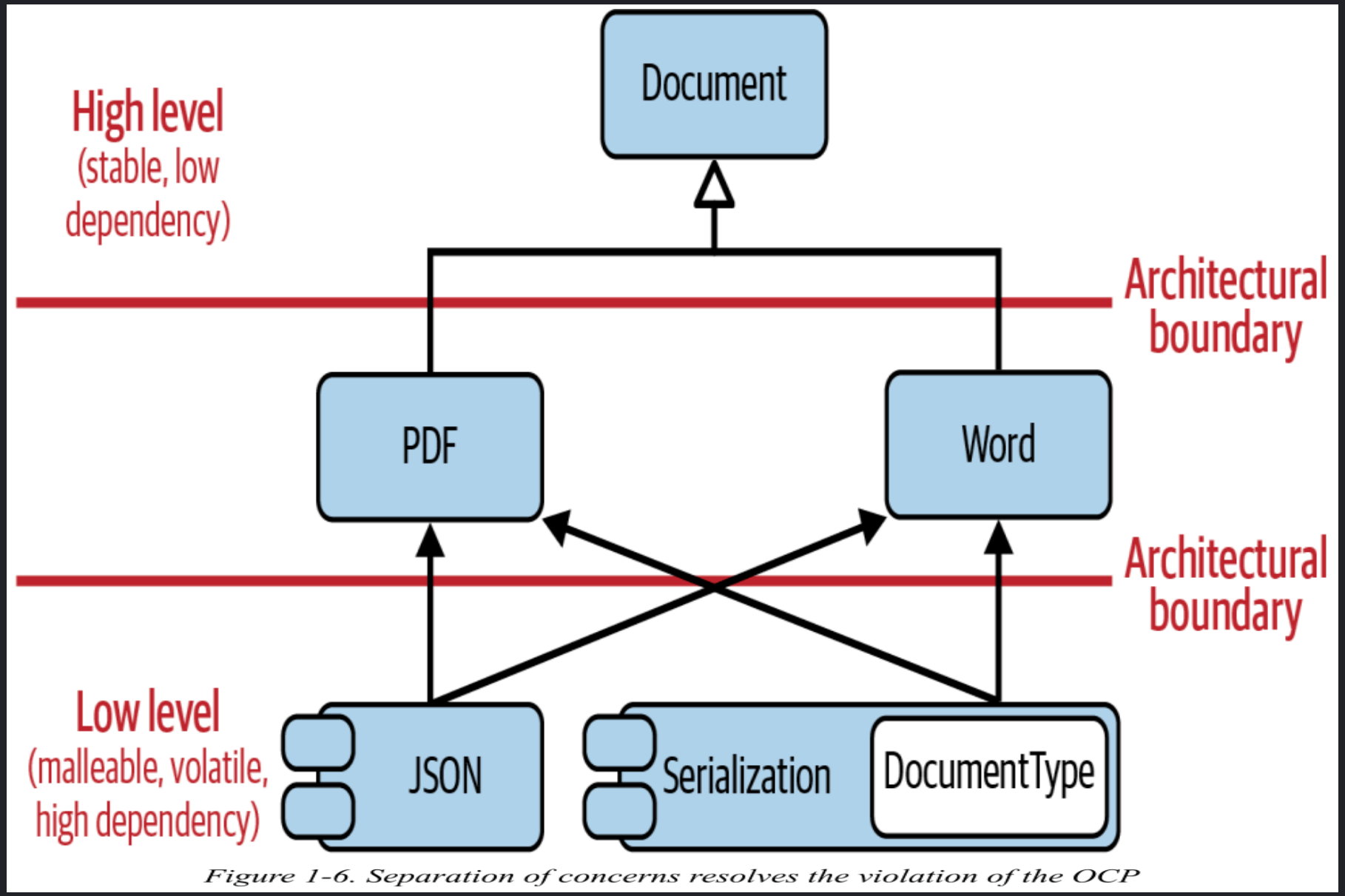


Figure 1-6. Separation of concerns resolves the violation of the OCP

Compile-Time Extensibility

- The Standard Library is designed for extensibity. But it builds on function overloading, templates, and (class) template specialization instead of inheritance.
- **Example:** `std::swap`
  - It is a template function so can be used with any type.
  - It can be specialized if needed.

G6: Adhere to the Expected Behavior of Abstractions

- The classical example: Is square a rectangle?
  - Geometrically, yes.
  - But in software, no.

```
void transform( Rectangle& rectangle) {
    rectangle.setWidth ( 7 );
    rectangle.setHeight( 4 );
    assert( rectangle.getArea() == 28 );
    // ...
```

- **Liskov Substitution Principle (LSP):** Expectations in an abstraction, must be adhered to in a subtype.
- Preconditions cannot be strengthened in a subtype.
  - Postconditions cannot be weakened in a subtype.
- Function return types in a subtype must be covariant.
- Function parameters in a subtype must be contravariant.
- Invariants of the supertype must be preserved in a subtype.

G7: Understand the Similarities Between Base Classes and Concepts

- LSP is not limited to dynamic polymorphism and inheritance.
- Also can be applied to compile-time polymorphism and templates.
- Adhere to the expected behavior of concepts when using templates.

G8: Understand the Semantic Requirements of Overload Sets

- Every abstraction represents a set of semantic requirements.
- Free functions represent a compile-time abstraction.
- Free functions perfectly live up to the OCP.
  - Easy to extend by adding new functions without modifying existing ones.

```
template<typename Range>
void traverseRange(Range const& range)
{
    for(auto pos=range.begin(); pos!=range.end(); ++pos) {
    // ...
    }
}
```

vs.

```
template<typename Range>
void traverseRange(Range const& range)
{
    for(auto pos=std::begin(range); pos!=std::end(range); ++pos) {
    // ...
    }
}
```

STL Philosophy

- Loose coupling and reuse by separating concerns as free functions is one part of the STL philosophy.
- Containers and algorithms are two separate concepts within the STL.
- The abstraction between them is accomplished via iterators.

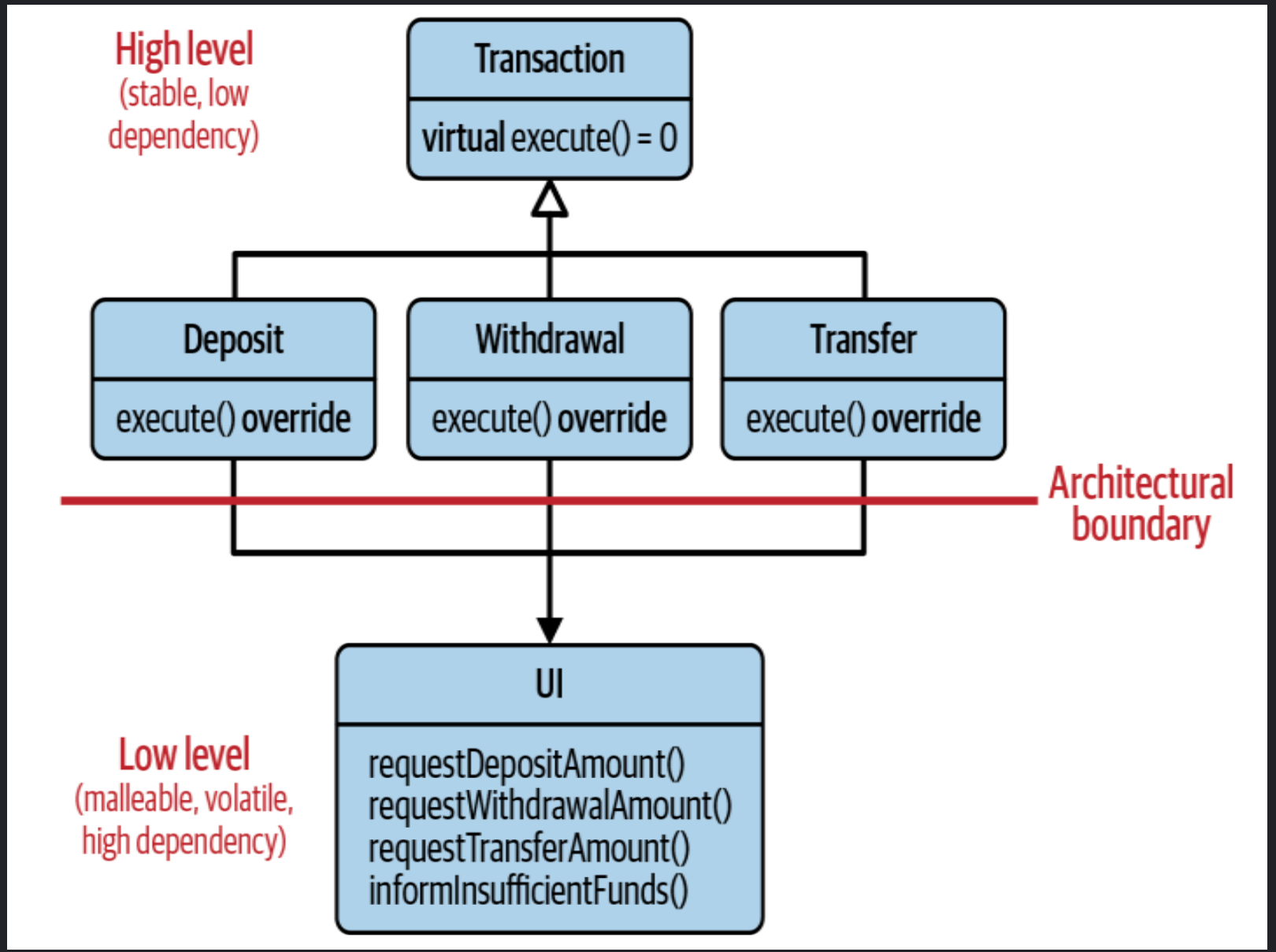
> There was never any question that the STL represented a breakthrough in efficient and extensible design.

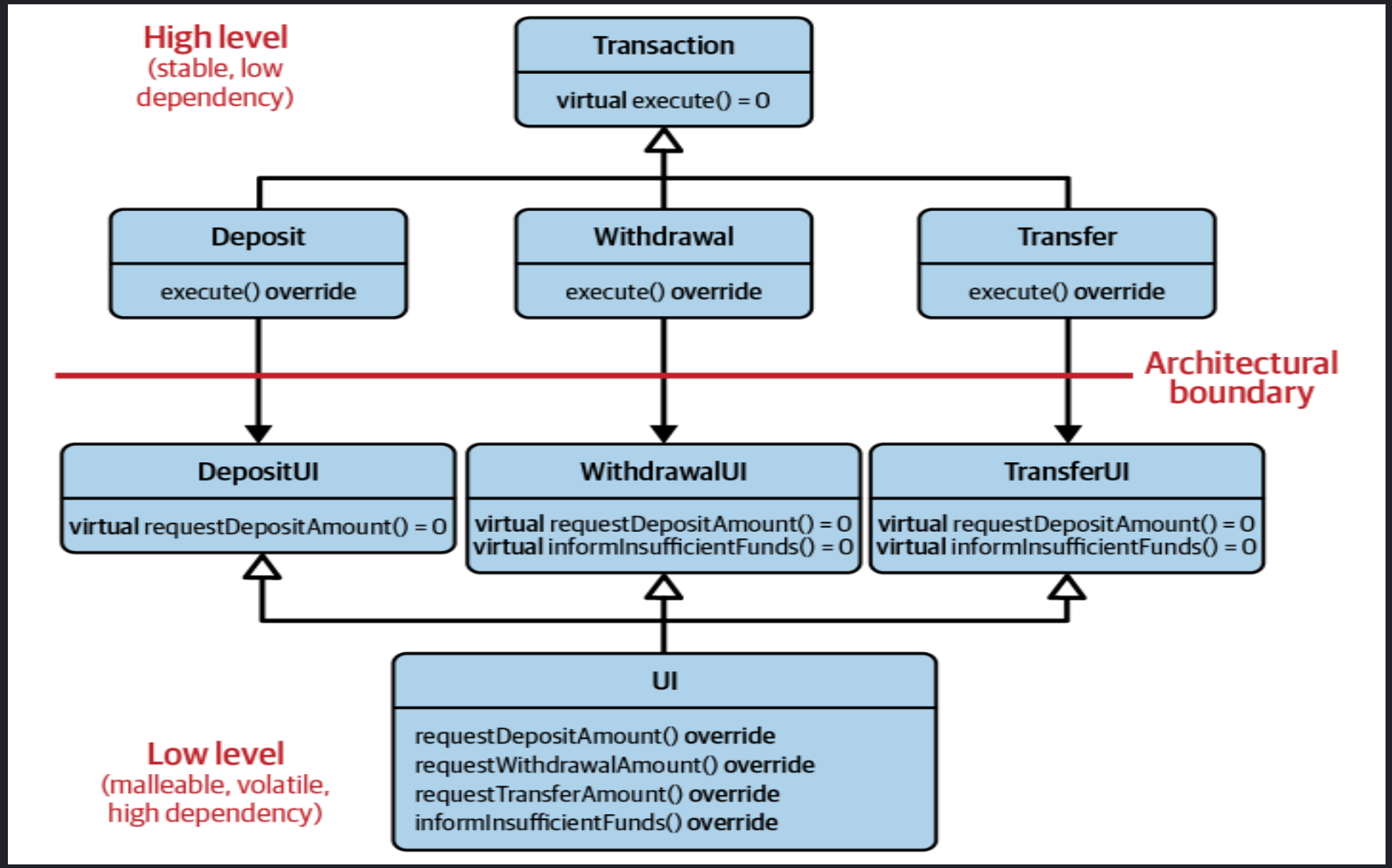
The Problem of Free Functions: Expectations on the Behavior

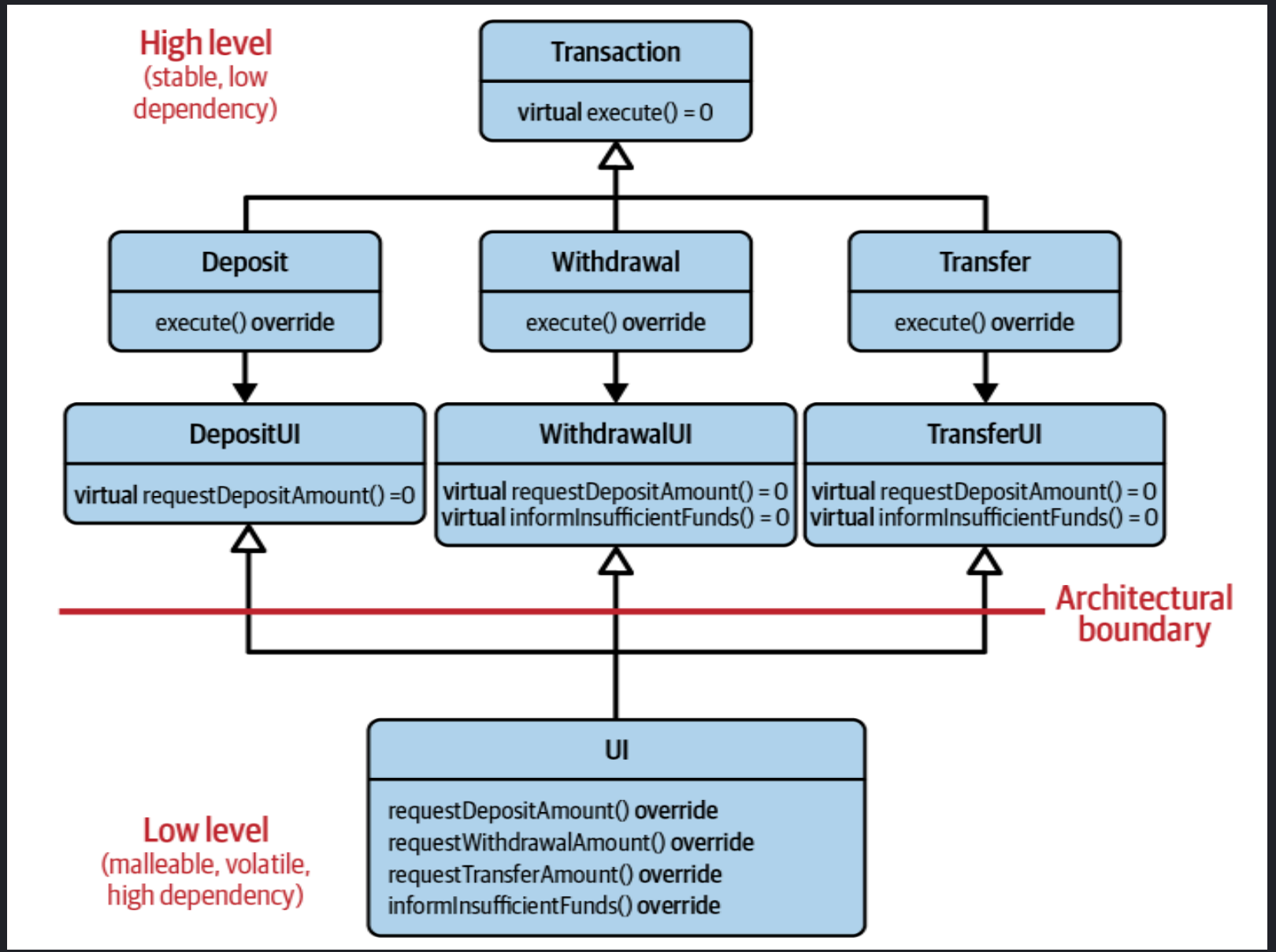
- It is not guaranteed that the special implementation of a free function adheres to the expected behavior.
- It may not always be clear what the expected behavior is.
- We need to be careful and pay attention to existing conventions.

G9: Pay Attention to the Ownership of Abstractions

- **Dependency Inverson Principle (DIP):** You should depend on abstractions, not on concretions.
- In class diagrams, dependency arrows should be from low-level to high-level modules.


High level
(stable, low dependency)
Transaction
virtual execute() = 0
Deposit
execute() override
Withdrawal
execute() override
Transfer
execute() override
Architectural boundary
UI
requestDepositAmount()
requestWithdrawalAmount()
requestTransferAmount()
informInsufficientFunds()
Low level
(malleable, volatile, high dependency)


High level
(stable, low
dependency)
Transaction
virtual execute() = 0
Deposit
execute() override
Withdrawal
execute() override
Transfer
execute() override
Architectural
boundary
DepositUI
virtual requestDepositAmount() = 0
WithdrawalUI
virtual requestDepositAmount() = 0
virtual informInsufficientFunds() = 0
TransferUI
virtual requestDepositAmount() = 0
virtual informInsufficientFunds() = 0
UI
requestDepositAmount() override
requestWithdrawalAmount() override
requestTransferAmount() override
informInsufficientFunds() override
Low level
(malleable, volatile,
high dependency)


High level
(stable, low
dependency)
Transaction
virtual execute() = 0
Deposit
execute() override
Withdrawal
execute() override
Transfer
execute() override
DepositUI
virtual requestDepositAmount() =0
WithdrawalUI
virtual requestDepositAmount() = 0
virtual informInsufficientFunds() = 0
TransferUI
virtual requestDepositAmount() = 0
virtual informInsufficientFunds() = 0
Architectural
boundary
UI
requestDepositAmount() override
requestWithdrawalAmount() override
requestTransferAmount() override
informInsufficientFunds() override
Low level
(malleable, volatile,
high dependency)

G10: Consider Creating an Architectural Document

> In most successful software projects, the expert developers working on that project have a shared understanding of the system design. This shared understanding is called ‘architecture.’

- Architectural document is needed to maintain and communicate the architecture.

Final Comments?

See you in part 2!