17-423/723: Software System Design

Design for Change II

Feb 4, 2026

Leaning Goals

- Apply principles for improving modularity: Single responsibility, interface segregation, and dependency inversion
- Describe the benefits & limitations of each principle
- Evaluate possible costs of modularity and its impact on other quality attributes

Last Class

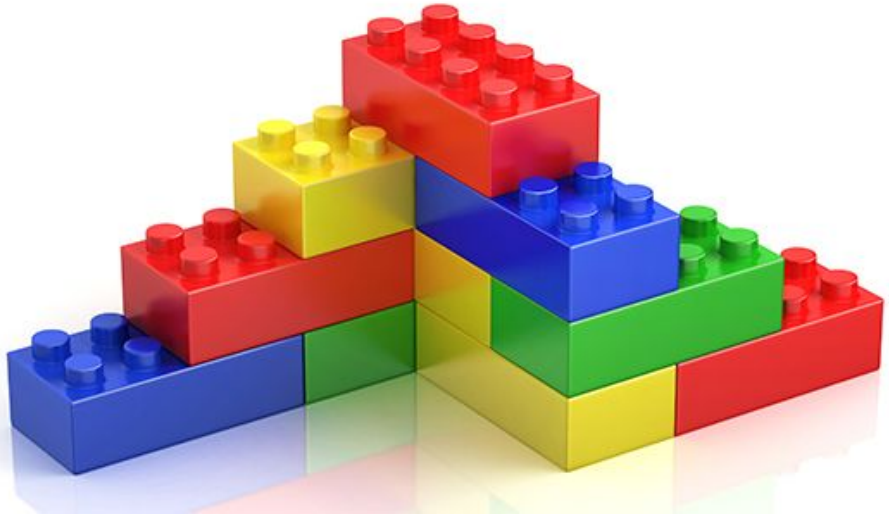
- Changeability
- Information hiding
- Data abstraction
- Interface abstraction
- Encapsulation

Recall: Changeability

- A measure of the amount of effort involved in making a change to a system
- Usually qualitative (i.e., yes/no), but sometimes quantified in terms of numerical metrics (e.g., lines of code changed)
- Quality attribute specifications – examples:
  - “A new publisher can be added without having to change any of the existing subscribers”
  - “New types of stocks can be added without changing the format of how each stock is displayed”
  - “Improving the performance of the C++ compiler does not affect the parser”
  - “Adding a new type of sensor in a self-driving vehicle requires changing only the image processing module”

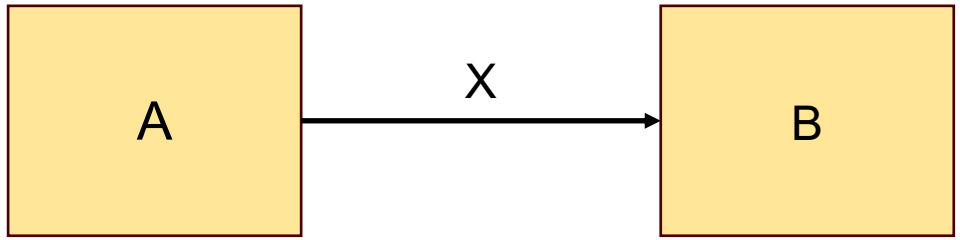
Related Concepts

- Modularity
  - Degree to which different parts of the system can be substituted with alternative parts without affecting the rest of the system
  - Closely related to changeability: Modularity supports changeability!



Dependency

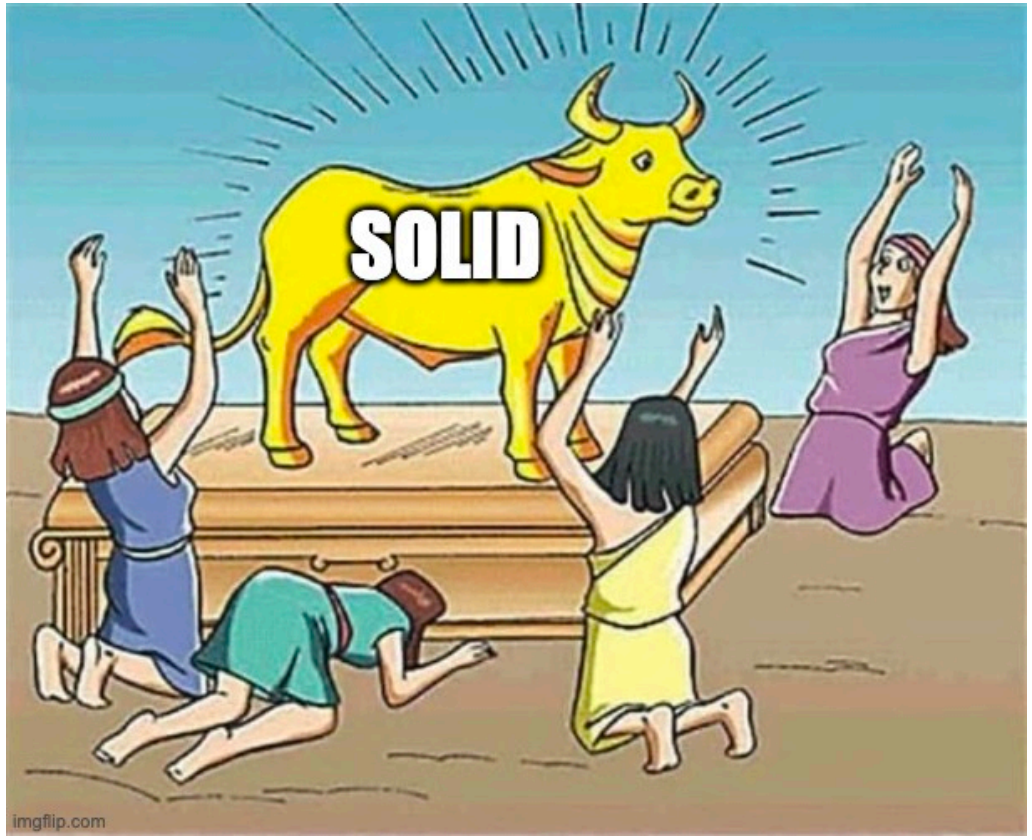
- Degree to which one component relies on another component to fulfill its responsibility
- To fulfill its responsibility, component A depends on B through connection X
- If B changes in a way that affects X, A may need to change to continue fulfilling its responsibility
- **Information hiding**: Hide secrets (design decisions) that are likely to change from other components



Today's Class

- Other principles for improving the modularity and changeability of the system
- SOLID principles
  - **Single-responsibility principle**
  - Open-closed principle
  - Liskov substitution principle
  - **Interface segregation principle**
  - **Dependency inversion principle**

A word of caution…



- People tend to get attached to trendy/popular ideas
- SOLID encodes good design practices, but are NOT a solution to every design problem
- Even good ideas, when applied blindly, can result in harmful outcome
- Think of these as tools! Ultimately, you need to apply your own judgement on when these are helpful or not

Single Responsibility Principle

Single Responsibility Principle (SRP)

Single Responsibility Principle (SRP)

- **Each component should be responsible for fulling a single purpose only**
  - Purpose: A unit of functionality, a use case, or a quality attribute
  - A purpose is associated with one or more design decision(s)
- **Corollary**: A component should **not** be designed to serve multiple purposes
  - Such a component may contain multiple secrets (i.e., design decisions) for different purposes
  - This encourages those secrets to become intermingled & dependent on each other; harder to change independently!
  - Such a component should be separated into multiple components

SRP:Example

```
public class InvoiceService {
    public void generateInvoice(Order order) {
        // 1. Compute totals
        double subtotal = order.subtotal();
        double tax = subtotal * 0.08;
        double total = subtotal + tax;

        // 2. Format invoice for display
        String invoiceText =
            "Order: " + order.id() + "\n" +
            "Subtotal: $" + subtotal + "\n" +
            "Tax: $" + tax + "\n" +
            "Total: $" + total;

        // 3. Save invoice
        FileWriter writer = new FileWriter("invoice.txt");
        writer.write(invoiceText);
        writer.close();
    }
}
```

- **Q. What purpose(s) does this program serve? What could go wrong?**
- **Possible change**: Change the totals calculation by rounding up the tax amount

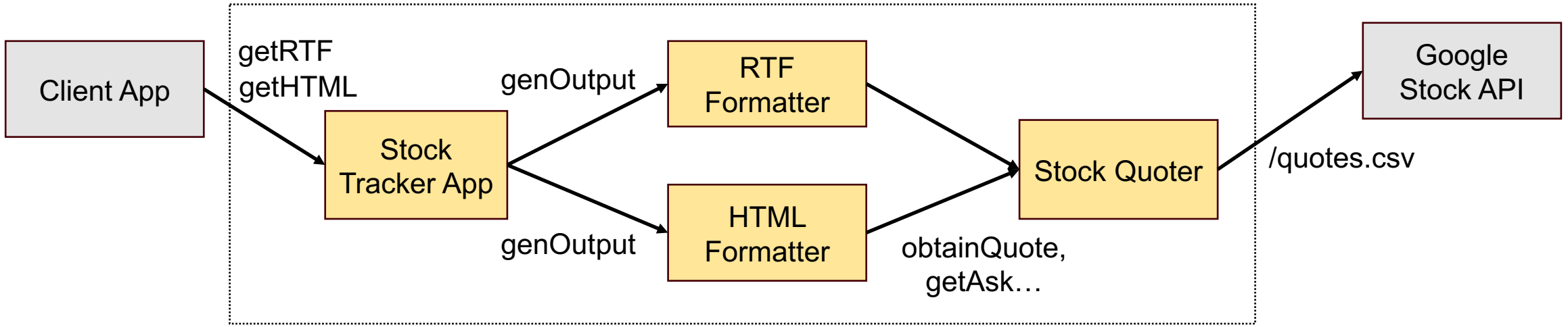
```
double total = subtotal +
    round(tax);
```

Recall: Stock Tracker App

- Get a list of stock quotes (prices) from an external source (e.g., Google)
- Produce output in HTML or RTF format
- Put the quote in **bold** if the change since the opening is > 1%

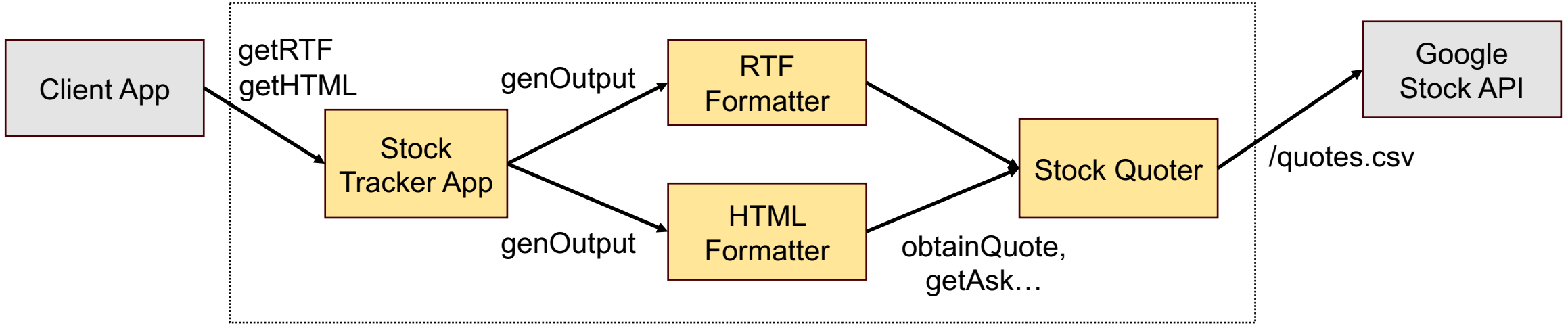
Based on an example by Daniel Jackson & Rob Miller

Stock Tracker: Violation of SRP?



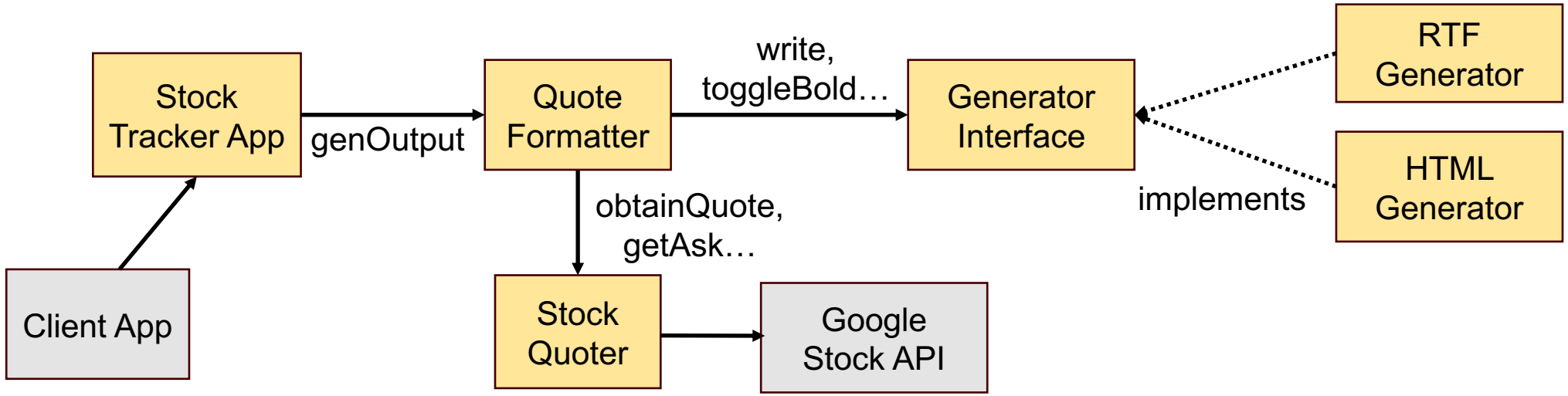
- Stock Trakcer App: Fulfills requests from a client for a quote in a certain format
- RTF/HTML formatter: Get quote from Stock Quoter & generate output in the right format
- Stock Quoter: Invoke Google API to get quote & return the result to Formatter
- **Q. Does this design violate SRP?**

Stock Tracker: Violation of SRP?



- **Problem**: HTML/RTF Formatters know (1) **how to** generate HTML/RTF elements in different formats and (2) **what** should be bolded, underlined, etc.,
- (2) is a design decision that can be separated & hidden from components that generate HTML/RTF!

Stock Tracker App: New Design



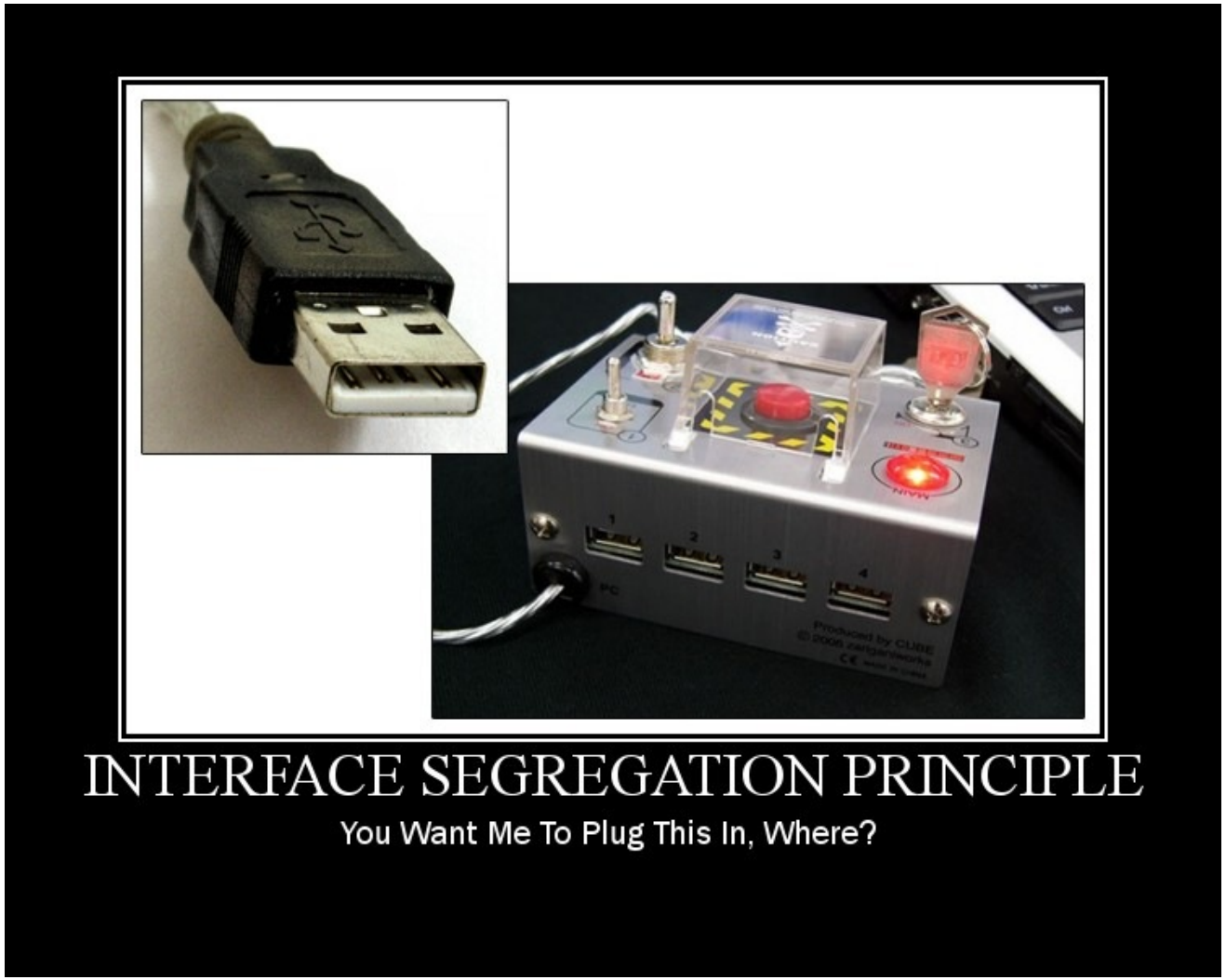
- HTML/RTF Generator: Writes & formats a given string using HTML/RTF tags
- Formatter: Encodes which part of the quote should be bolded, italicized
- Generators and Formatter now serve separate responsibilities!

Single Responsibility Principle (SRP)

- **Each component should be responsible for fulling a single purpose only**
- **Benefits**: Single-responsibility (SR) components
  - Reduce dependency between design decisions; make it easier to change them independently
  - Are more reusable: Provide a distinct unit of purpose that can be reused in other contexts
  - Are easier to understand & test
- **Q. Limitations or dangers of SRP?**

Interface Segregation Principle

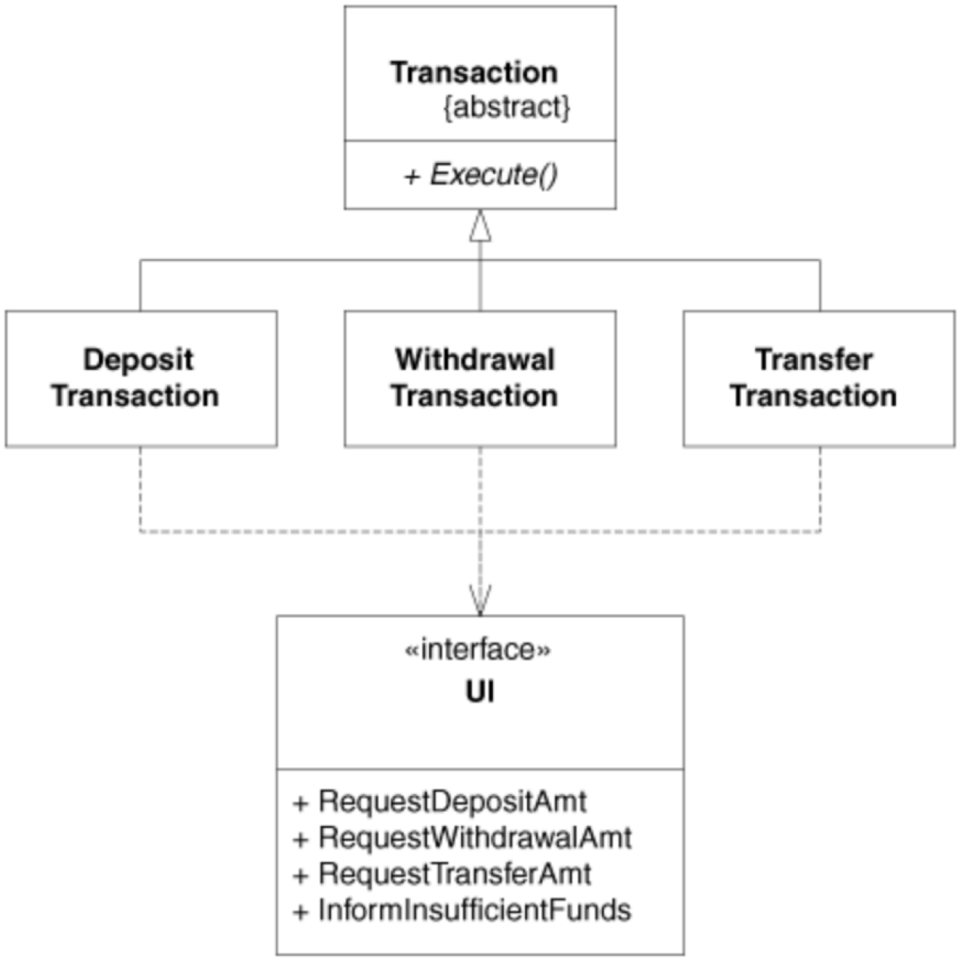
Interface Segregation Principle (ISP)



Interface Segregation Principle (ISP)

- **An interface should not force clients to depend on unnecessary details**
- **Interface pollution**: A common issue that arises when an interface grows & serves tasks for different types of clients

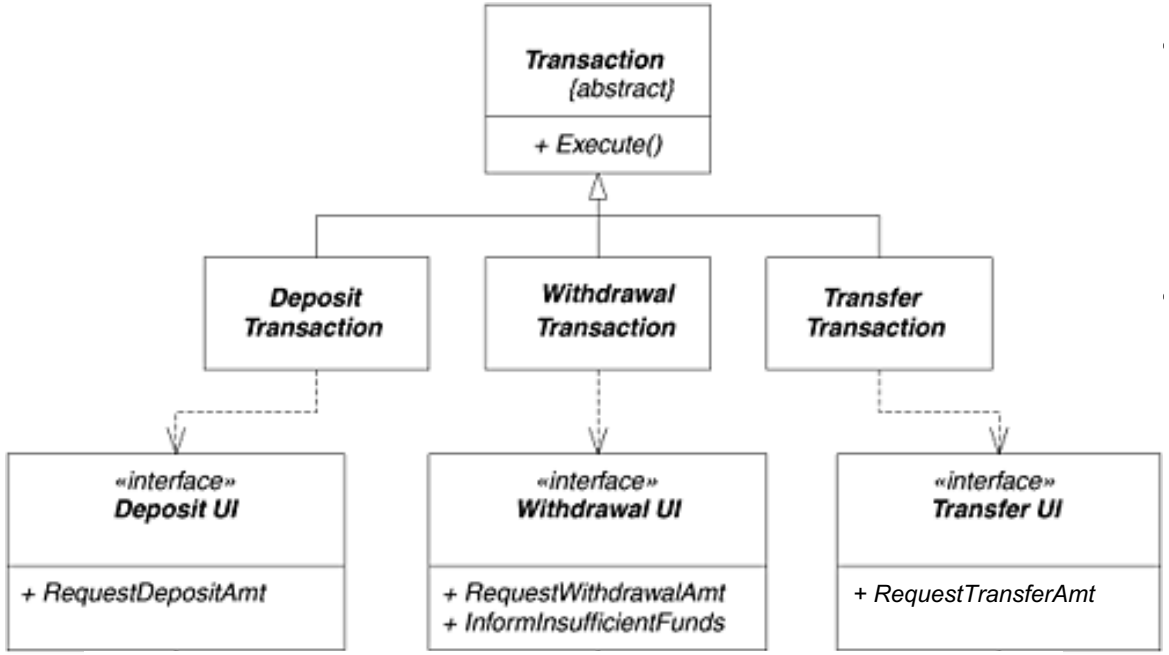
Example: ATM User Interface



- Different types of transactions require different user interactions
- Some UI methods are only used by a single transaction
- **Q. What could go wrong here?**
- **Unnecessary dependencies between the interface & clients!**
  - e.g., A change in UI can cause changes across *all* transactions
- **Q. What can we do to mitigate this issue?**

Example from: *Agile Principles, Patterns, and Practices in C#* by Martin & Matin (2007)

Example: ATM User Interface



- An alternative design: Decompose the bloated interface into multiple, separate interfaces
- **Benefits**:
  - Each interface serves one particular type of client
  - Each interface does not force the client to depend on unnecessary details
  - Each interface (and its client) can change independently from other interfaces

e from: *Agile Principles, Patterns, and Practices in C#* by Martin & Matin (2007)

Another Example: Stock Tracker

```
public interface Generator {
    public void open () throws Exception;
    public void close ();
    public void newLine ();
    public void toggleBold ();
    public void toggleItalic ();
    public void write (String s);
}

public class RTFGenerator implements Generator {
    public void open() throws FileNotFoundException { ... }
    ...}
public class HTMLGenerator implements Generator {
    public void open() throws FileNotFoundException { ... }
    ...}
public class JSONGenerator implements Generator {
    public void open() throws FileNotFoundException { ... }
    ...}
}
```

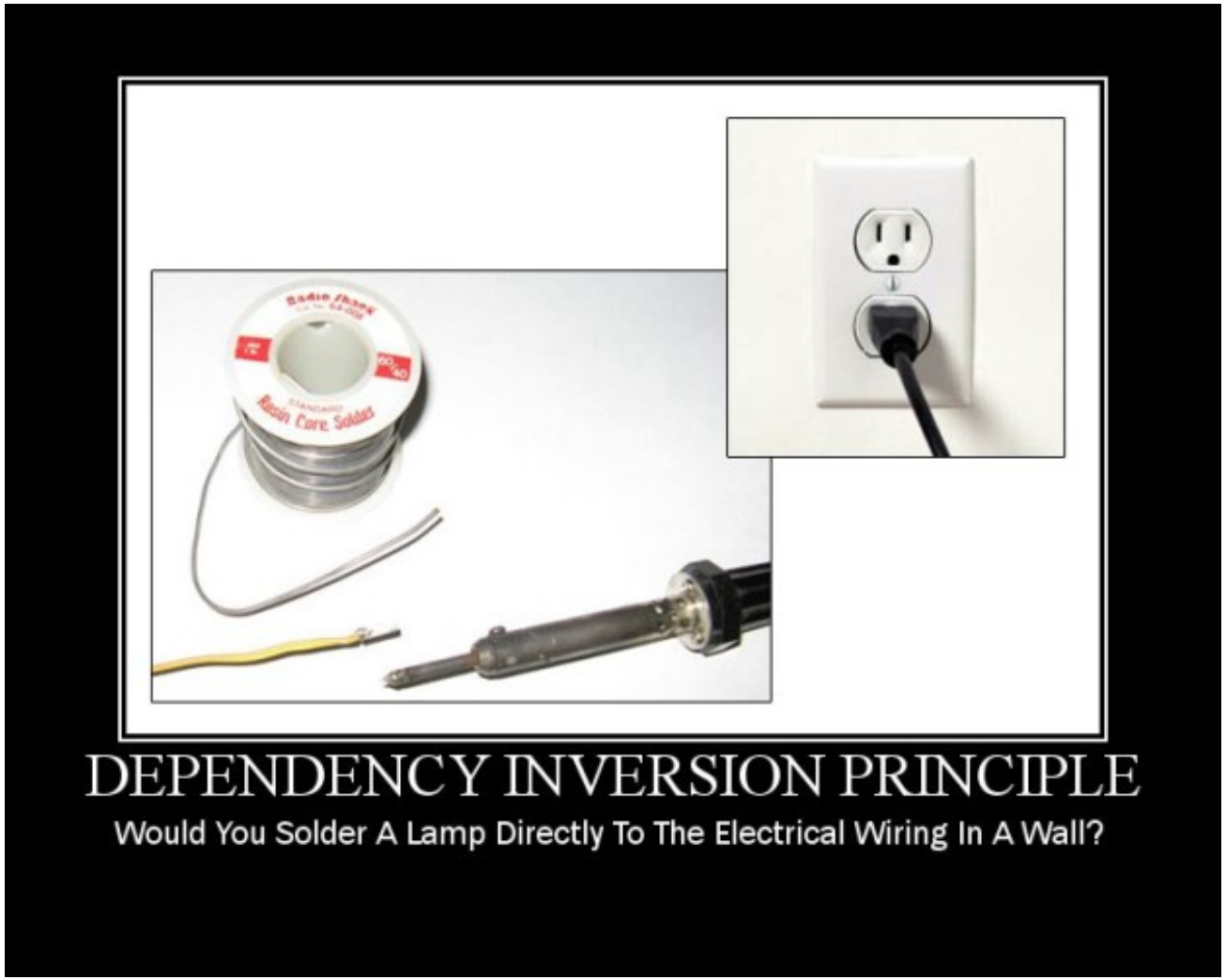
Suppose we want to add a new type of generator: JSON
Q. What can go wrong?
Q. How can we do better?

Interface Segregation Principle (ISP)

- **An interface should not force clients to depend on unnecessary details**
- **Interface pollution**: A common issue that arises when an interface grows & serves tasks for different types of clients
- Decompose the bloated interface into separate interfaces, each exposing details that are needed only by a single client
- **Q. What is the relationship between ISP and single responsibility principle (SRP)?**

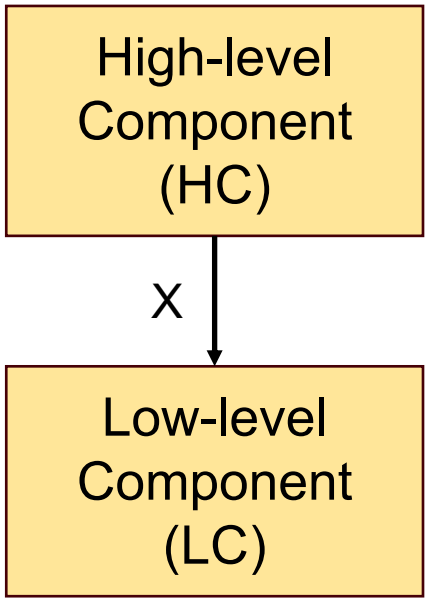
Dependency Inversion Principle

Dependency Inversion Principle (DIP)



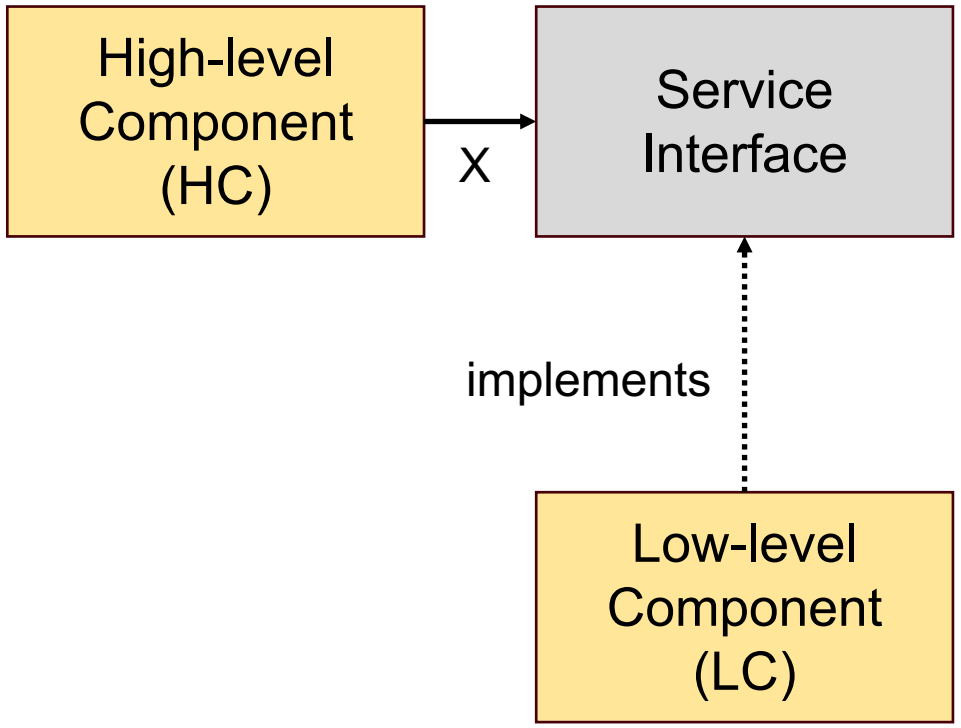
Dependency Inversion Principle (DIP)

- **Idea: A "high-level" component should not depend on a "low-level" component**
- High-level components (**HC**): Responsible for the core application/business logic and use cases
- Low-level components (**LC**): Services or libraries that serve the core logic



Dependency Inversion Principle (DIP)

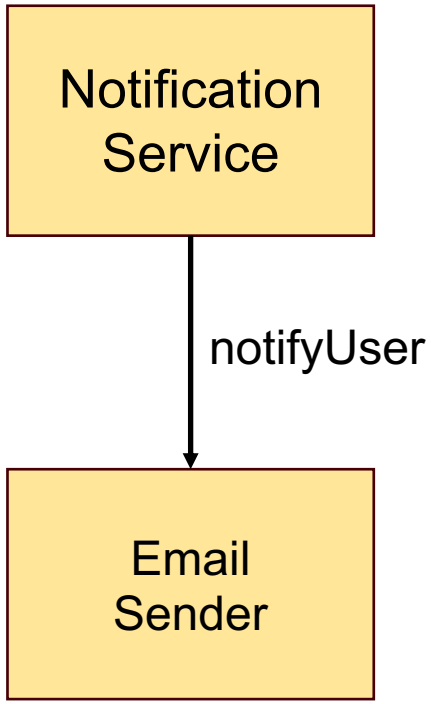
- "Invert" the dependency from HC to LC by introducing an intermediate abstraction (e.g., an interface)
  - **Recall**: Interface abstraction!
- Interface provides abstract operations & data that are needed by HC
- HC & LC both depend this abstraction
- HC does not directly interact with LC
- **Goal**: When LC changes, minimize its impact on HC



Dependency Inversion: Example

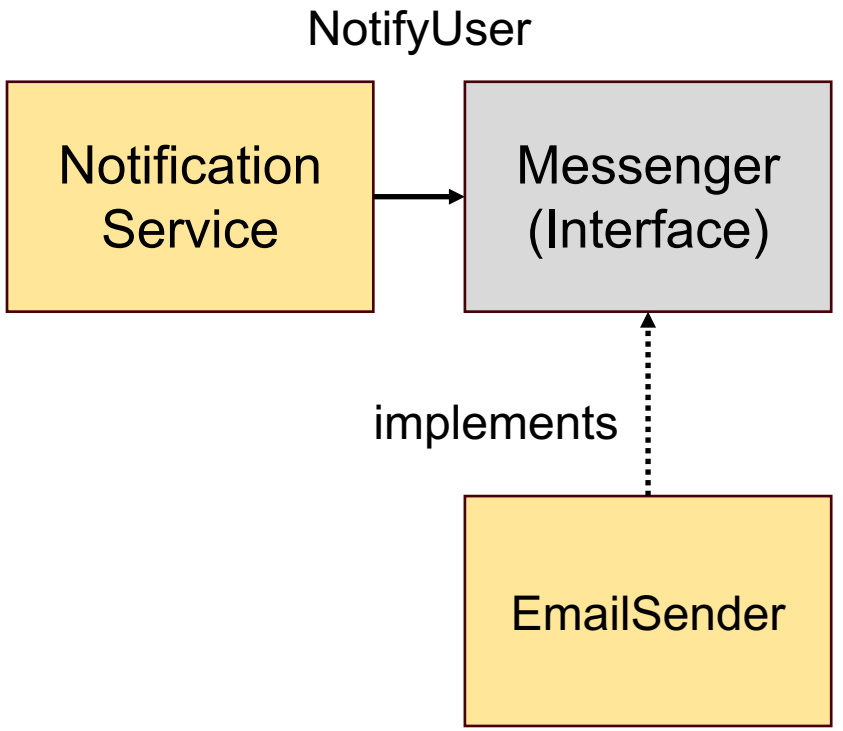
```
// High-level module
public class NotificationService {
  private EmailSender sender;
  public NotificationService(EmailSender sender) {
    this.sender = sender;
  }
  public void NotifyUser(String user, String message) {
    string fullMessage = $"To: {user}\nMessage: {message}";
    sender.SendMessage(user, fullMessage);
  }
}

// Low-level module
public class EmailSender {
  public void SendMessage(String user, String message) {
    // Implements sending an e-mail message to user
  }
}
```



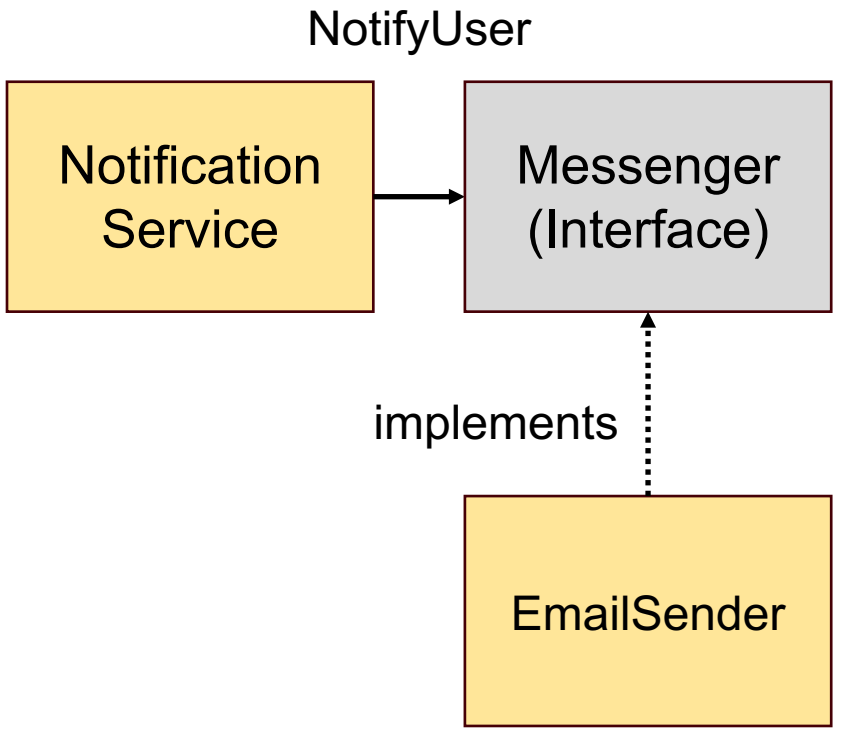
Dependency Inversion: Example

```
// Abstraction
public interface Messenger {
    void SendMessage(String user, String message);
}
// High-level module
public class NotificationService {
  private Messenger sender;
  public NotificationService(Messenger sender) {
    this.sender = sender;
  }
  public void NotifyUser(String user, String message) {
    string fullMessage = $"To: {user}\nMessage: {message}";
    sender.SendMessage(user, fullMessage);
  }
}
// Low-level module
public class EmailSender implements Messenger {
  public void SendMessage(string user, string message) {
    // Implements sending an e-mail message to user
  }
}
```



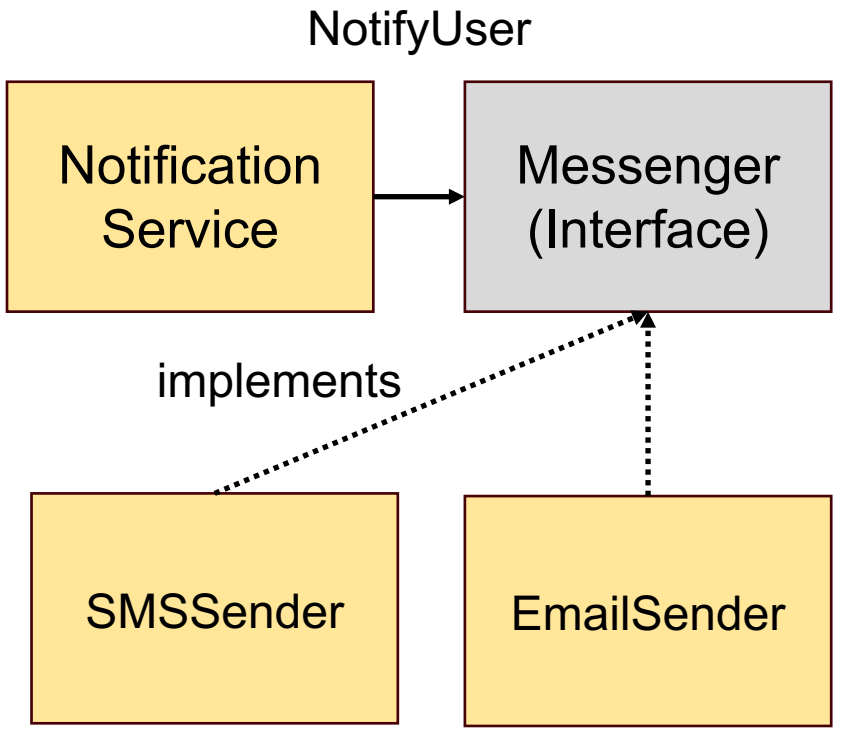
Dependency Inversion: Example

```
// Abstraction
public interface Messenger {
    void SendMessage(String user, String message);
}
// High-level module
public class NotificationService {
  private Messenger sender;
  public NotificationService(Messenger sender) {
    this.sender = sender;
  }
  public void NotifyUser(String user, String message) {
    string fullMessage = $"To: {user}\nMessage: {message}";
    sender.SendMessage(user, fullMessage);
  }
}
// Low-level module
public class EmailSender implements Messenger {
  public void SendMessage(string user, string message) {
    // Implements sending an e-mail message to user
  }
}
```

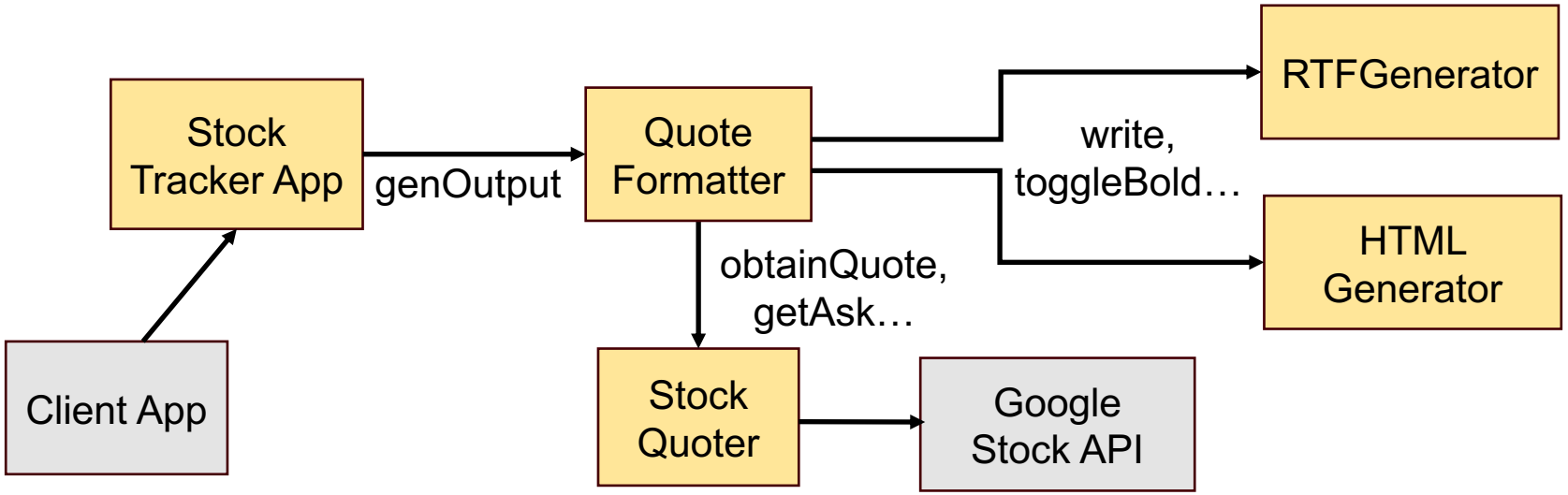


Dependency Inversion: Example

```
// Abstraction
public interface Messenger {
    void SendMessage(String user, String message);
}
// High-level module
public class NotificationService {
  private Messenger sender;
  public NotificationService(Messenger sender) {
    this.sender = sender;
  }
  public void NotifyUser(String user, String message) {
    string fullMessage = $"To: {user}\nMessage: {message}";
    sender.SendMessage(user, fullMessage);
  }
}
// Low-level module
public class SmsSender implements Messenger {
  public void SendMessage(string user, string message) {
    // Implements sending an SMS message to user
  }
}
```

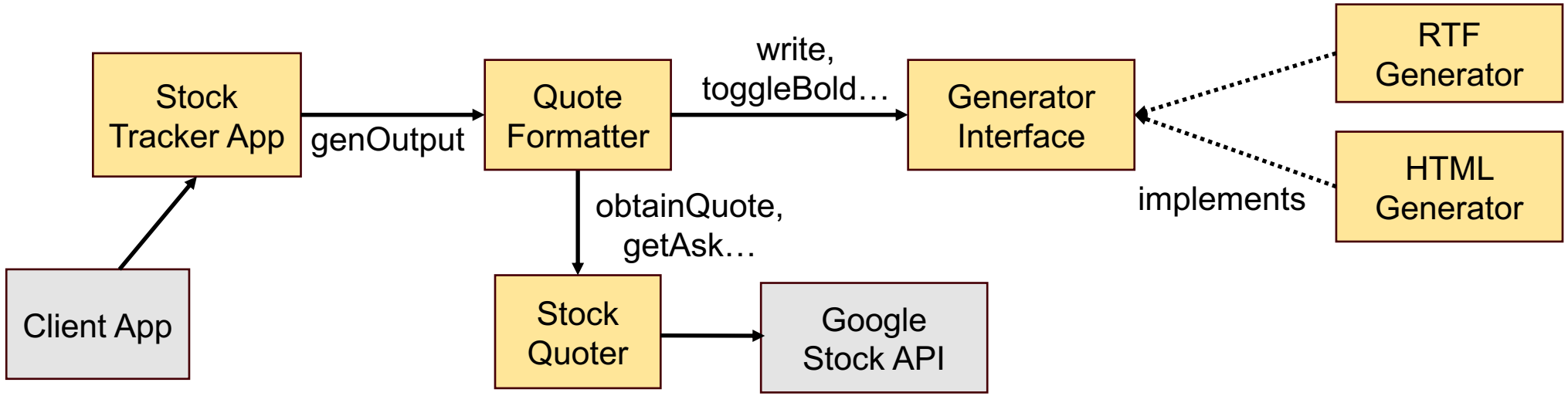


Stock Tracker App: New Design



- **Q. Does this design violate DIP?**

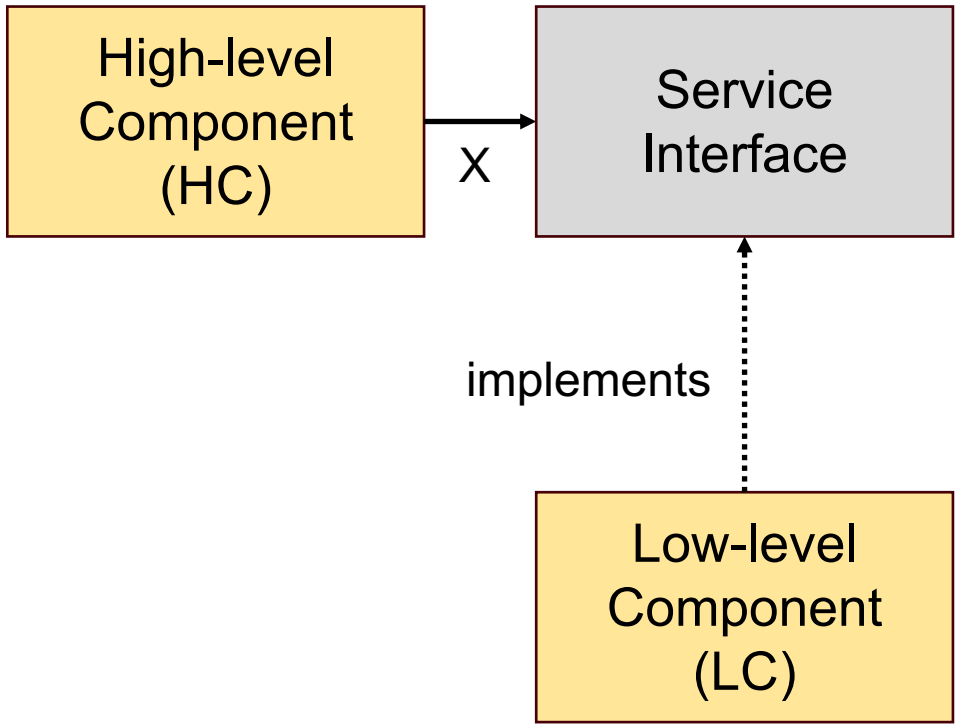
Stock Tracker App: New Design



- Generator interface: Hides the type of output file (HTML/RTF) from Formatter
- Formatter: Encodes which part of the quote should be bolded, italicized; does not know anything about HTML/RTF!
- Formatter (**HC component**) no longer depends on the generators (**LC** components)

Dependency Inversion Principle (DIP)

- Invert the dependency from HC to LC by introducing an intermediate abstraction (e.g., an interface)
- HC & LC both depend this abstraction
- HC does not know anything about LC
- **Goal**: When LC changes, minimize its impact on the high-level component
- **Q. What assumption is this principle making? Do they always hold in practice?**



```
interface Writer {
    void writeHeader();
    void writeLine(String text);
    void writeBold(String text);
}
public class PdfWriter implements Writer { … }
public class CsvWriter implements Writer { … }

// Financial report service
public class ReportService {
    public void generateMonthlyReport() {
        MySqlConnection conn = new MySqlConnection("prod-db");
        ResultSet rs = conn.executeQuery("SELECT * FROM sales");

        Writer writer = new PdfWriter();
        writer.writeHeader();
        while (rs.next()) {
            double revenue = rs.getDouble("revenue");
            if (revenue > 10000) {
                writer.writeBold(rs.getString("region"));
            } else {
                writer.writeLine(rs.getString("region"));
            }
        }
    }
}
```

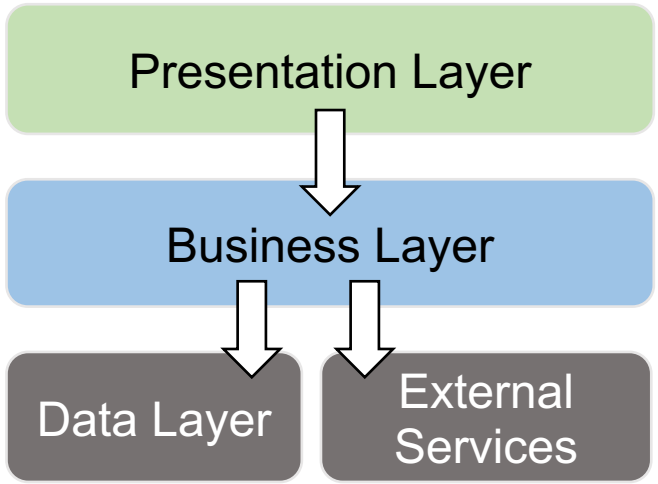
Exercise:

Which of the three principles (SRP, ISP, DIP) are violated in this code?

How do would improve the design?

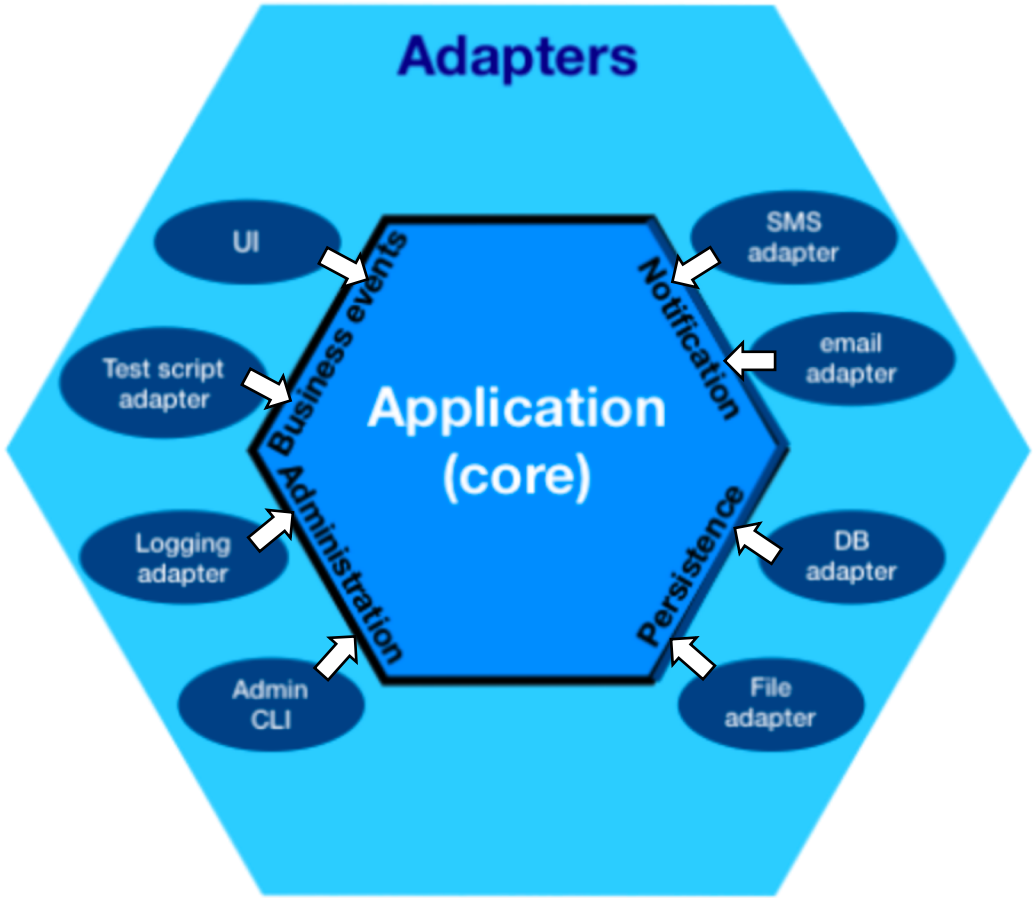
Dependency Inversion in Practice

Traditional Layered Architecture



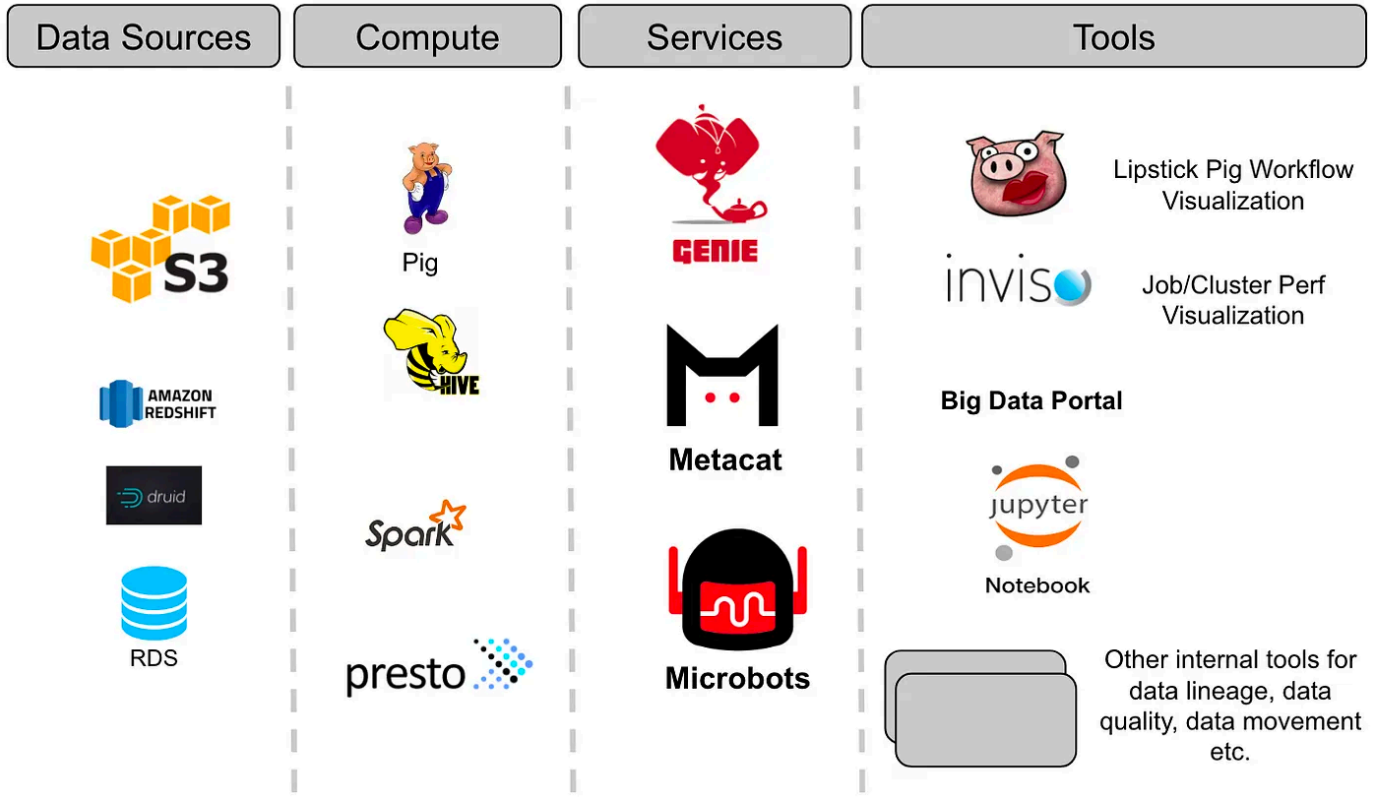
- Common 3-layer pattern for application architecture
- **Top-down dependency**: Higher-level components depend on lower-level ones
- Presentation layer: User facing components (UI, APIs, command line…)
- Business layer: Implements the core application logic
- **Q. Potential downside (w.r.t. changeability?)**

Alternative Design: Hexagonal Architecture



- **Inward** dependency only: All components depend on core business logic (**dependency inversion**!)
- **Adapter**: An implementation of an interface in the core logic
  - Link between an external component & the interface
- **Input adapters:** Allow users, external actors, and client services to interact with the core logic
- **Output adapters:** Wrappers for services used by the core logic (e.g., database engine)

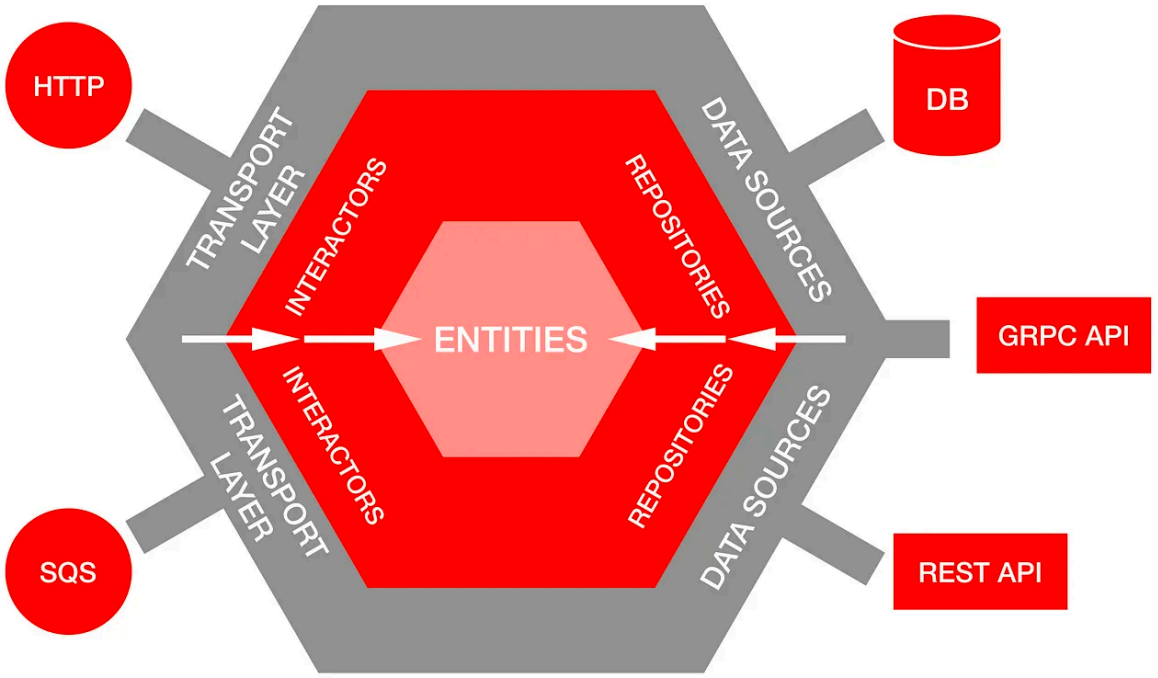
Example: Netflix Architecture



- Many different data sources, external services, tools
- Data about movies, production dates, employees, shooting locations (> 300 DB tables)
- Multiple protocols: gRPC, JSON API, GraphQL…
- **Challenge**: Swap data sources without affecting the core business logic

https://netflixtechblog.com/metacat-making-big-data-discoverable-and-meaningful-at-netflix-56fb36a53520

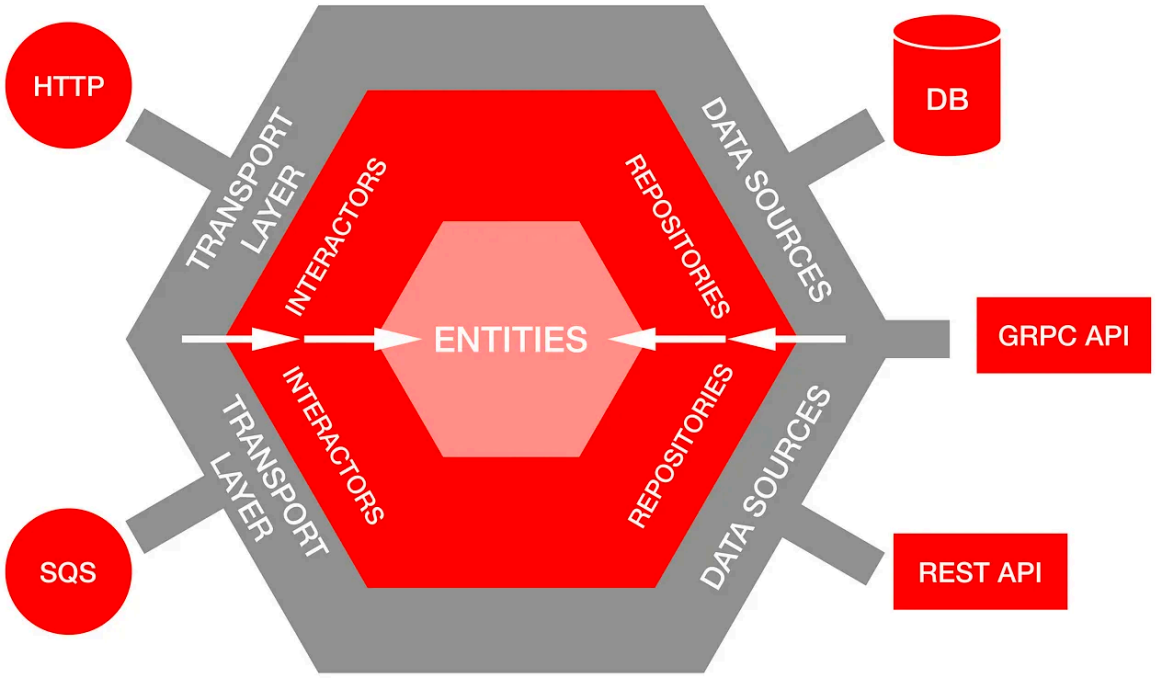
Hexagonal Architecture at Netflix



- **Entities**: Business objects (e.g., Movie or Shooting Location); knows nothing about how they are stored
- **Repositories:** Interfaces to create, retrieve, and modify entities from a data source
- **Interactors:** Components that implement business logic (e.g., initiate a new movie production)

https://netflixtechblog.com/ready-for-changes-with-hexagonal-architecture-b315ec967749

Hexagonal Architecture at Netflix



- **Data sources**: Output adapters; interface with different storage implementations (e.g., SQL, REST API, gRPC)
- **Transport layer**: Input adapters; triggers a business use case; separates input modes (e.g., HTTP) from the interactors

https://netflixtechblog.com/ready-for-changes-with-hexagonal-architecture-b315ec967749

Adapters: Example

```
public interface OrderRepository {
  Optional<Order> findById(UUID id);
  void save(Order order);
}
```

Repository (interface) used by the business logic; doesn't know anything about the DB engine

```
public class MongoDbOrderRepository implements
OrderRepository {
  public Optional<Order> findById(UUID id) {
    // MongoDB-specific implementation
  }
  public void save(Order order) {
    // MongoDB-specific implementation
  }
}
```

An adapter that implements the repository interface; wraps details specific to a data source (e.g., MongoDB)

https://netflixtechblog.com/ready-for-changes-with-hexagonal-architecture-b315ec967749

Adapters: Example

```
public interface OrderRepository {
  Optional<Order> findById(UUID id);
  void save(Order order);
}
```

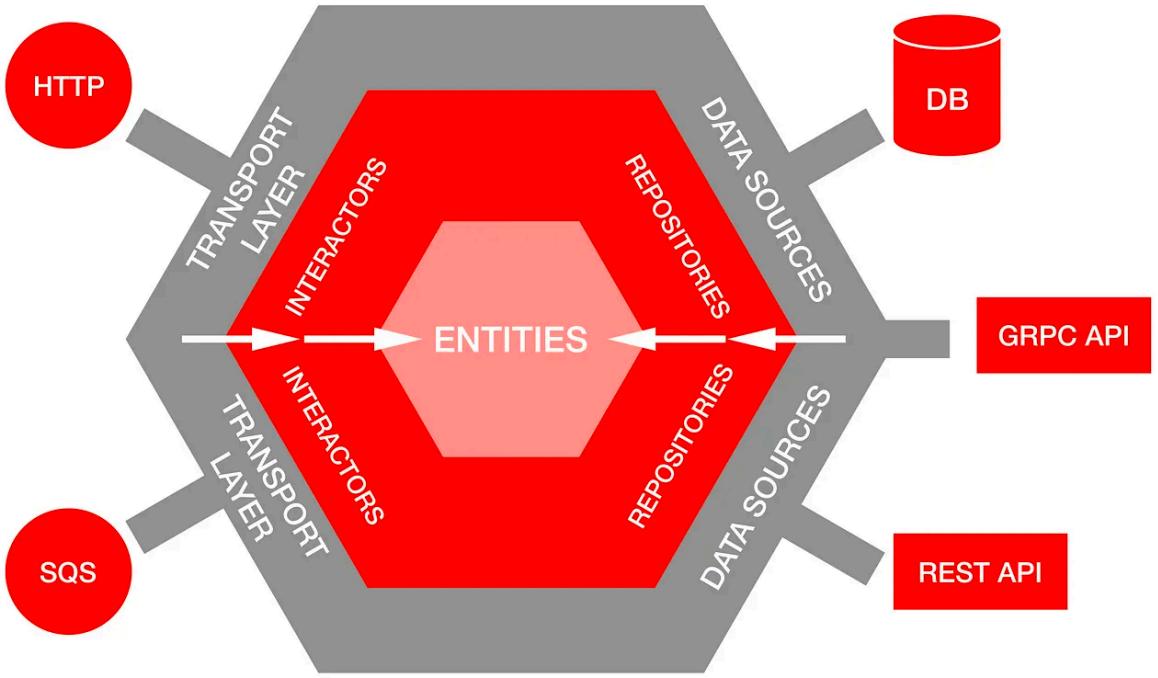
Repository (interface) used by the business logic; doesn't know anything about the DB engine

```
public class CassandraDbOrderRepository
implements OrderRepository {
  public Optional<Order> findById(UUID id) {
    // Cassandra-specific implementation
  }
  public void save(Order order) {
    // Cassandra-specific implementation
  }
}
```

Can swap in and out different data sources without affecting the business logic!

https://netflixtechblog.com/ready-for-changes-with-hexagonal-architecture-b315ec967749

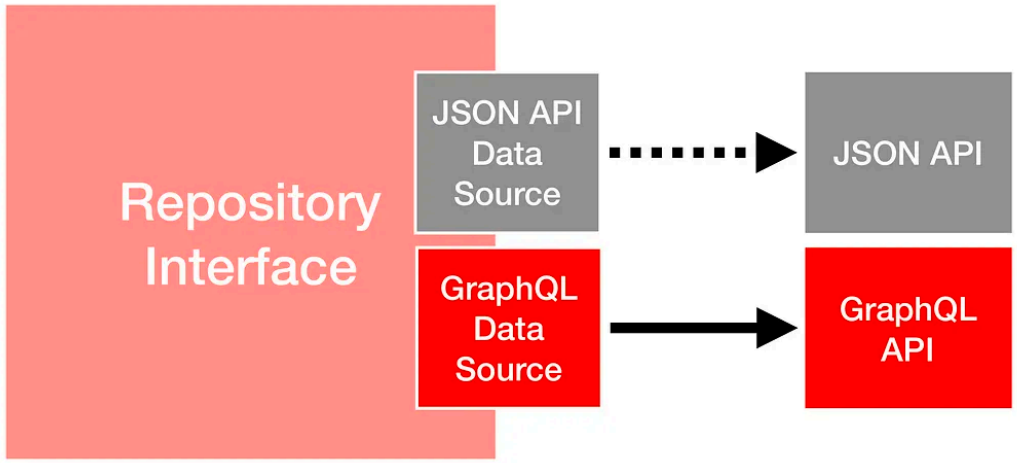
Example: Netflix Architecture



- **Q. What are benefits of this architecture?**
- Core logic does not know anything about transport layer or data sources
- Can add a new user interaction (e.g., command line) or data sources without changing the business logic

https://netflixtechblog.com/ready-for-changes-with-hexagonal-architecture-b315ec967749

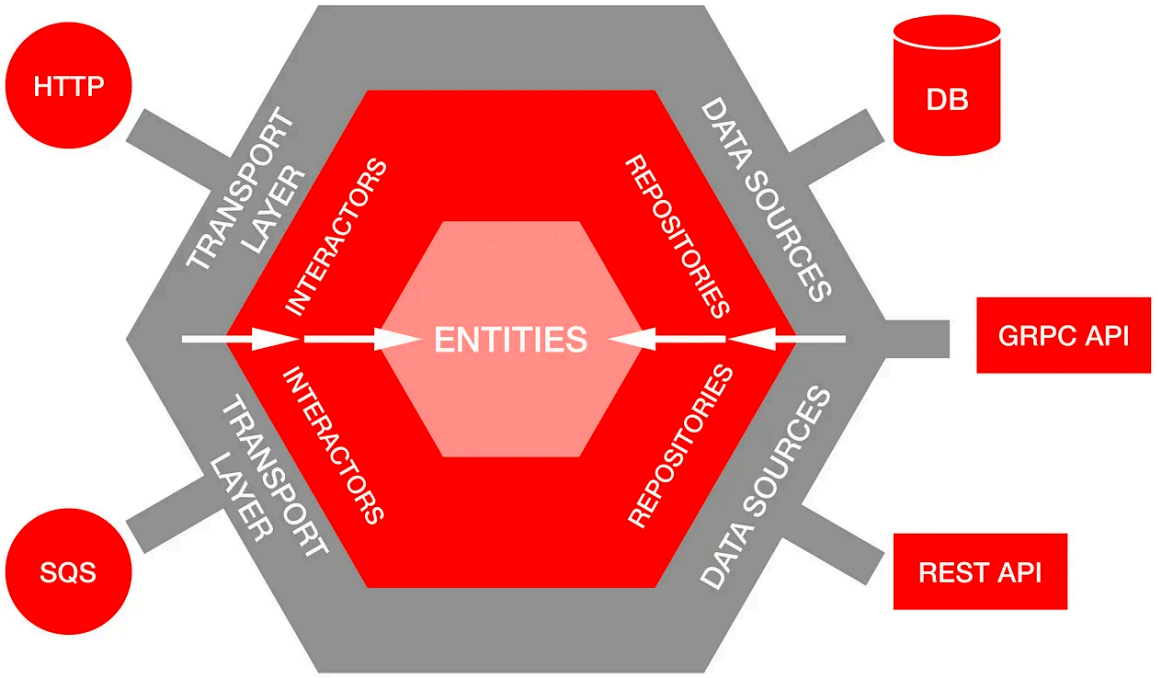
Example: Netflix Architecture



- Can change data sources without impacting core logic, as long as they conform to repositories
- “We managed to transfer reads from JSON API to GraphQL data source within 2 hours.”
- No leakage of secrets about data persistence into the business logic!
- Also improves scalability & testability (**Q. how so?)**

https://netflixtechblog.com/ready-for-changes-with-hexagonal-architecture-b315ec967749

Example: Netflix Architecture

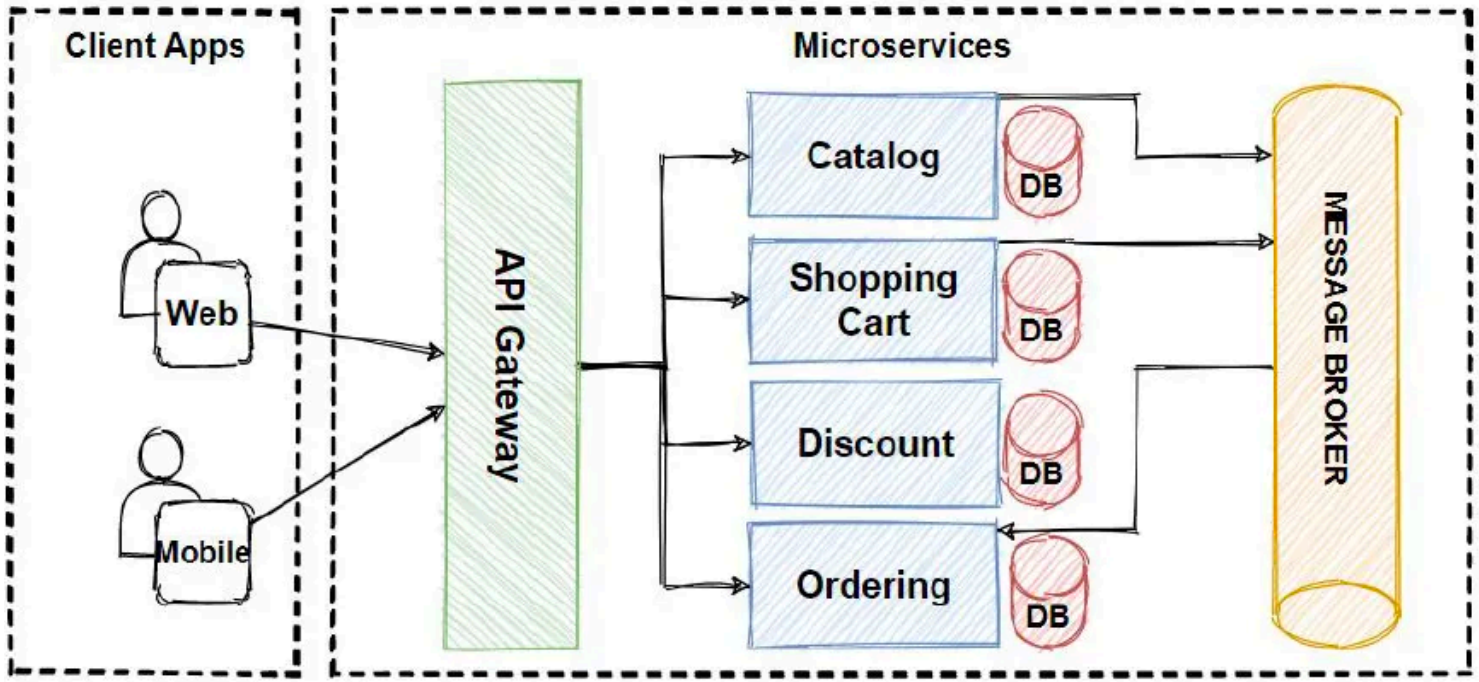


- Q. What are benefits of this architecture?
- **Q. What are some limitations? What assumption does this architecture rely on?**

https://netflixtechblog.com/ready-for-changes-with-hexagonal-architecture-b315ec967749

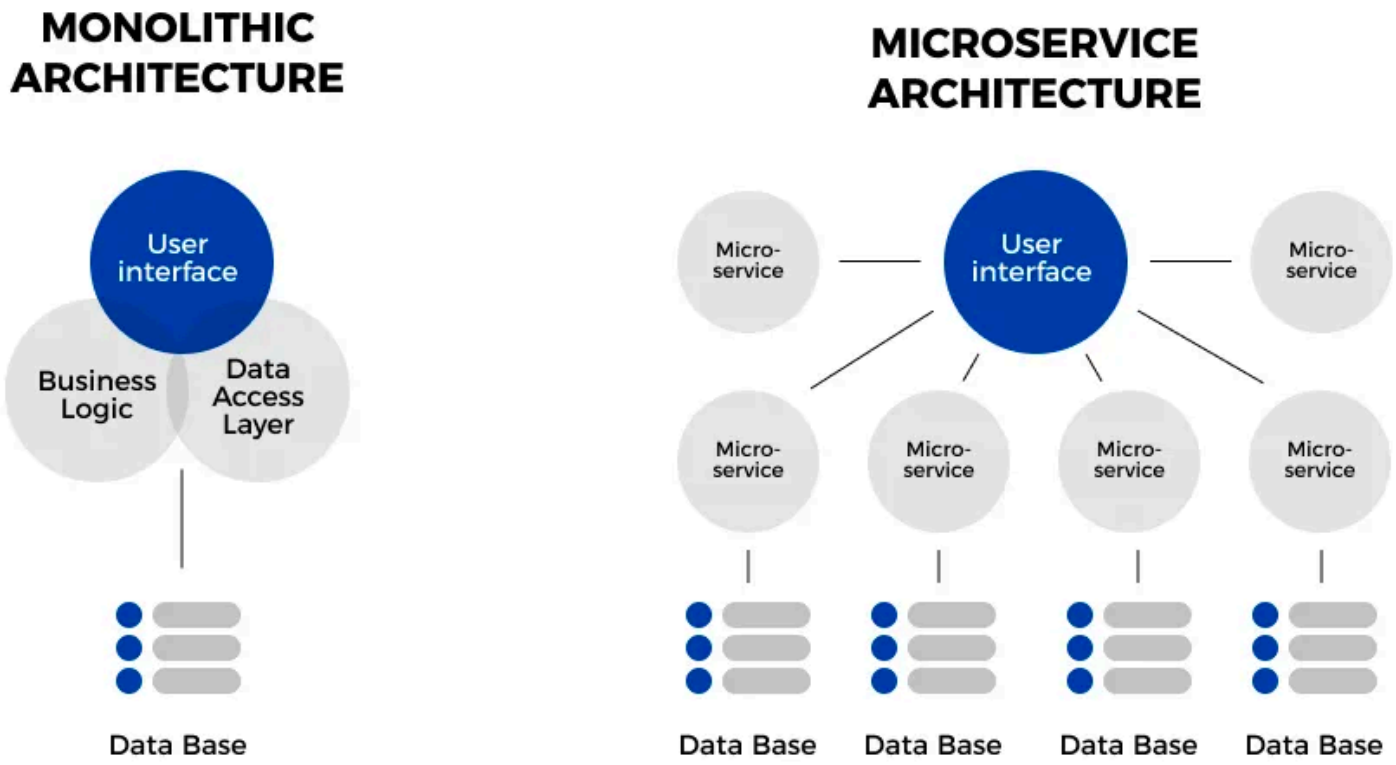
Cost of Modularization

Microservice Architecture



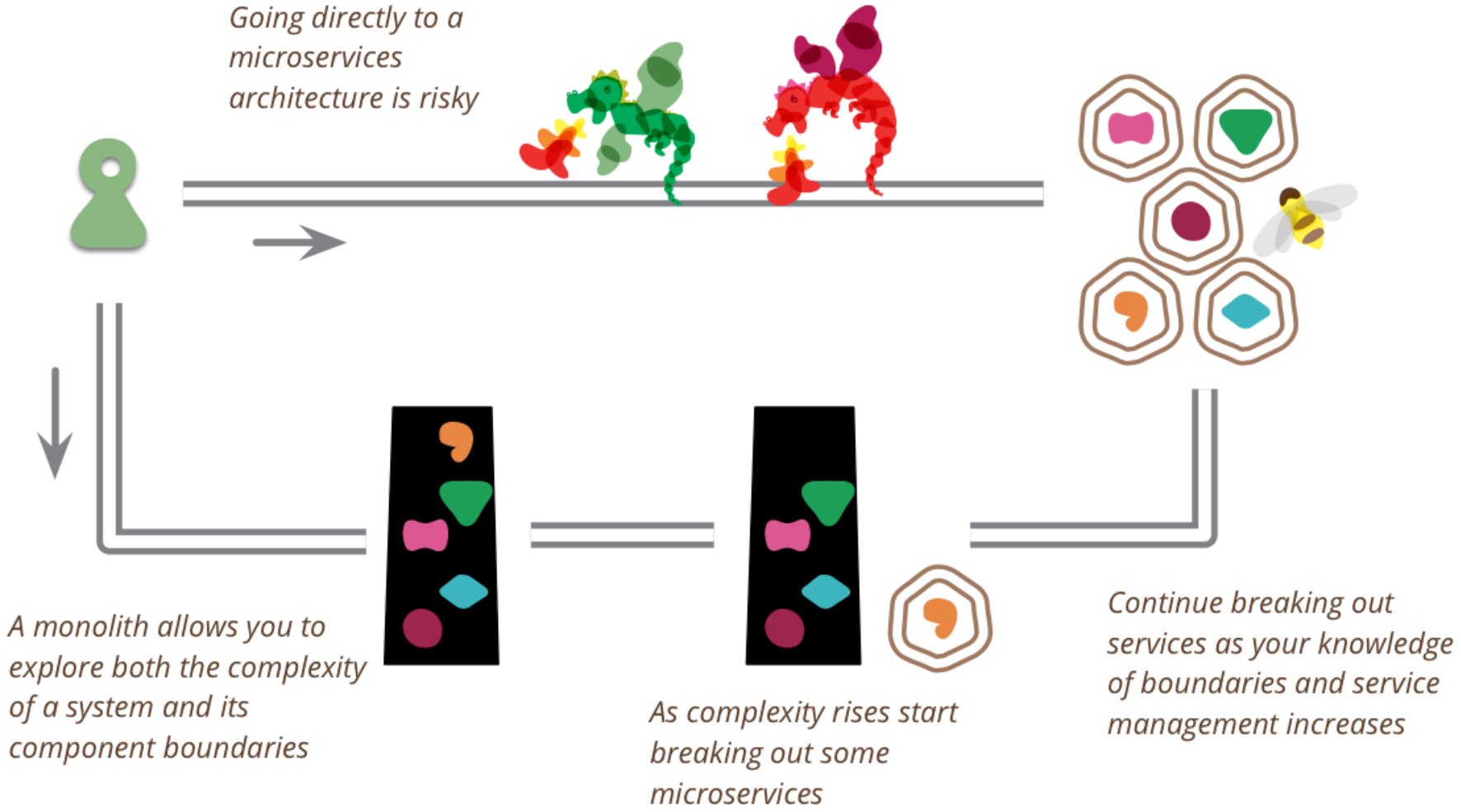
- Decompose system into multiple, deployable units of services, typically developed by independent teams
- User requests are routed to the appropriate service
- Services communicate directly or through a message broker

Microservice Architecture



- **Q. What are the benefits of a microservice architecture?**
- **Q. What are its potential downsides?**

“Monolith First”



https://martinfowler.com/bliki/MonolithFirst.html#footnote-typical-monolith

Cost of Modularization: Takeaway

- Like other quality attributes, changeability comes with costs and trade-offs
- Modularization & abstraction, in general, are good practices
- But too much modularization can be harmful
  - Can increase complexity, add development costs, affect performance, and make certain changes even harder to make
- **Recall**: Risk-driven design!
  - What are likely changes in my system that I need to be ready for?
  - How important is the flexibility to adapt to these changes?
  - Is the lack of flexibility the most significant risk to my product right now?

Summary of Principles & Methods

- Information Hiding: Secrets that are likely to change should be hidden from other components
- Single Responsibility: A component should be responsible for fulling a single purpose only
- Interface Segregation: An interface should not force clients to depend on unnecessary details
- Dependency Inversion: A high-level component should not depend directly on a low-level component
- Data Abstraction: Hide details of a data representation
- Interface Abstraction: Hide details of a service implementation
- Encapsulation: Isolate & hide a secret in one place within the system

Summary

- Exit ticket!