



GameDuell

Zwiebeln statt Schichten

Hexagonale Architektur als Alternative zur Schichten-Architektur

Dirk Ehms, GameDuell GmbH

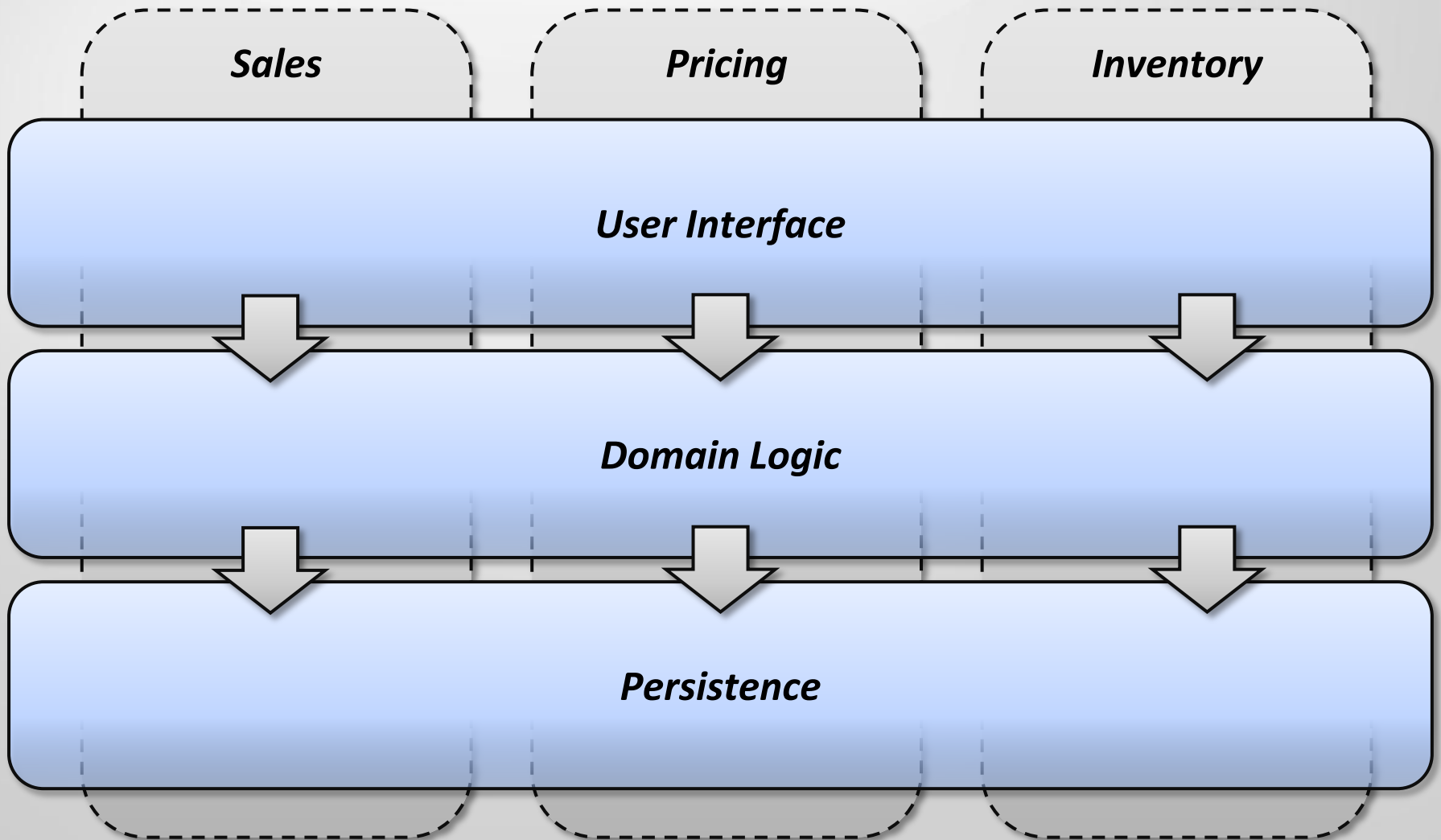


Agenda

- 1. Layered Architecture**
- 2. Example Application**
- 3. Test Automation**
- 4. Code Review**
- 5. Dependency Management**
- 6. Hexagonal Architecture**
- 7. Onion Architecture**
- 8. Conclusion and Discussion**

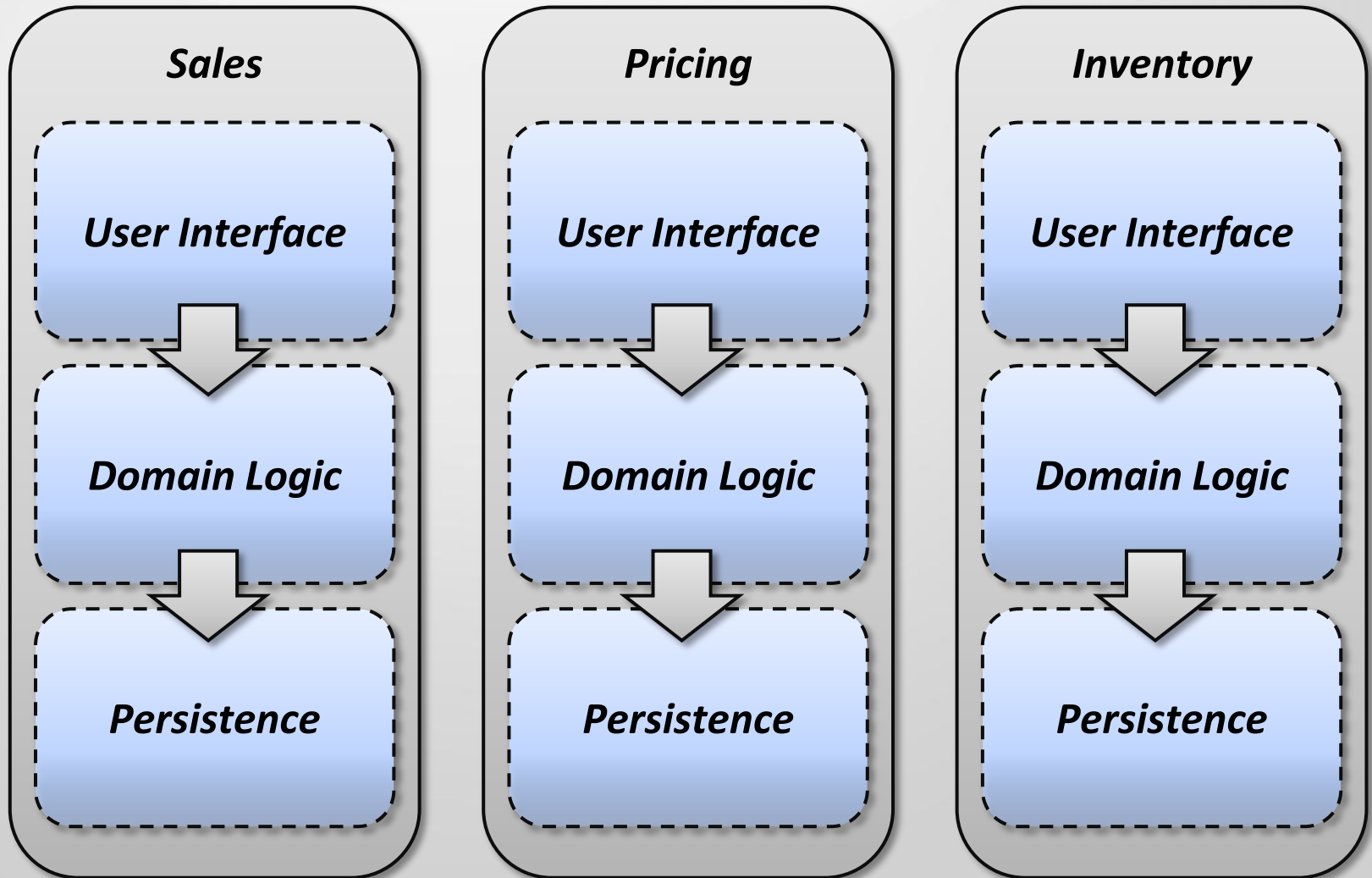


Layered Architecture





Layered Architecture (2)





Example Application: *Promotion Messenger*

Functional Requirements

1. Read customer records from the database
2. Filter customers who have a specific interest
3. Send a personalized promotion message by email





Example Application: *Promotion Messenger*

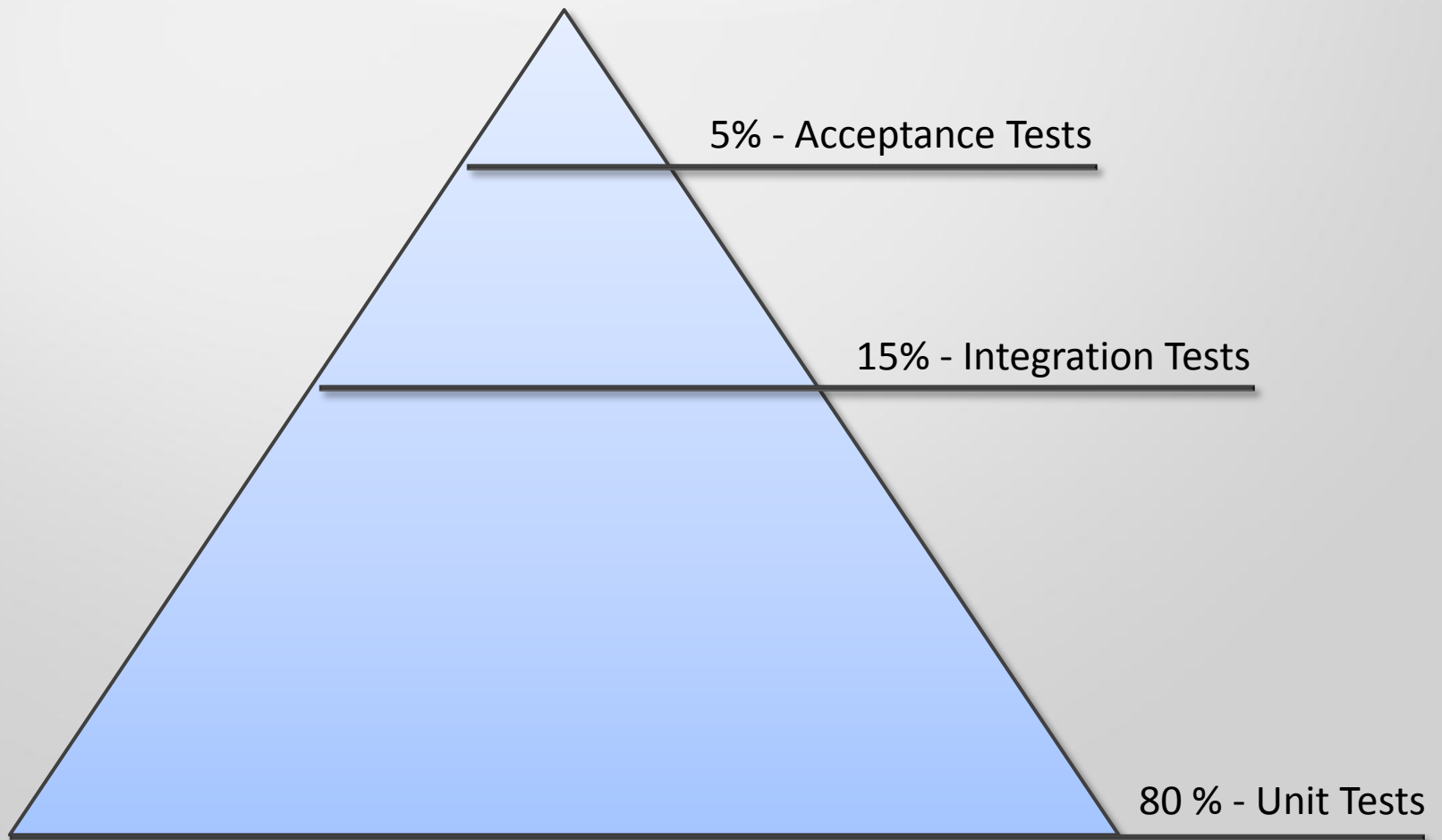
Non-functional Requirements

1. Application must be easy to change and extend
2. Application must be easy to test
3. Unit tests must be very fast and reliable
4. Domain logic must not depend on low level APIs
5. Domain logic must be clearly separated from external systems



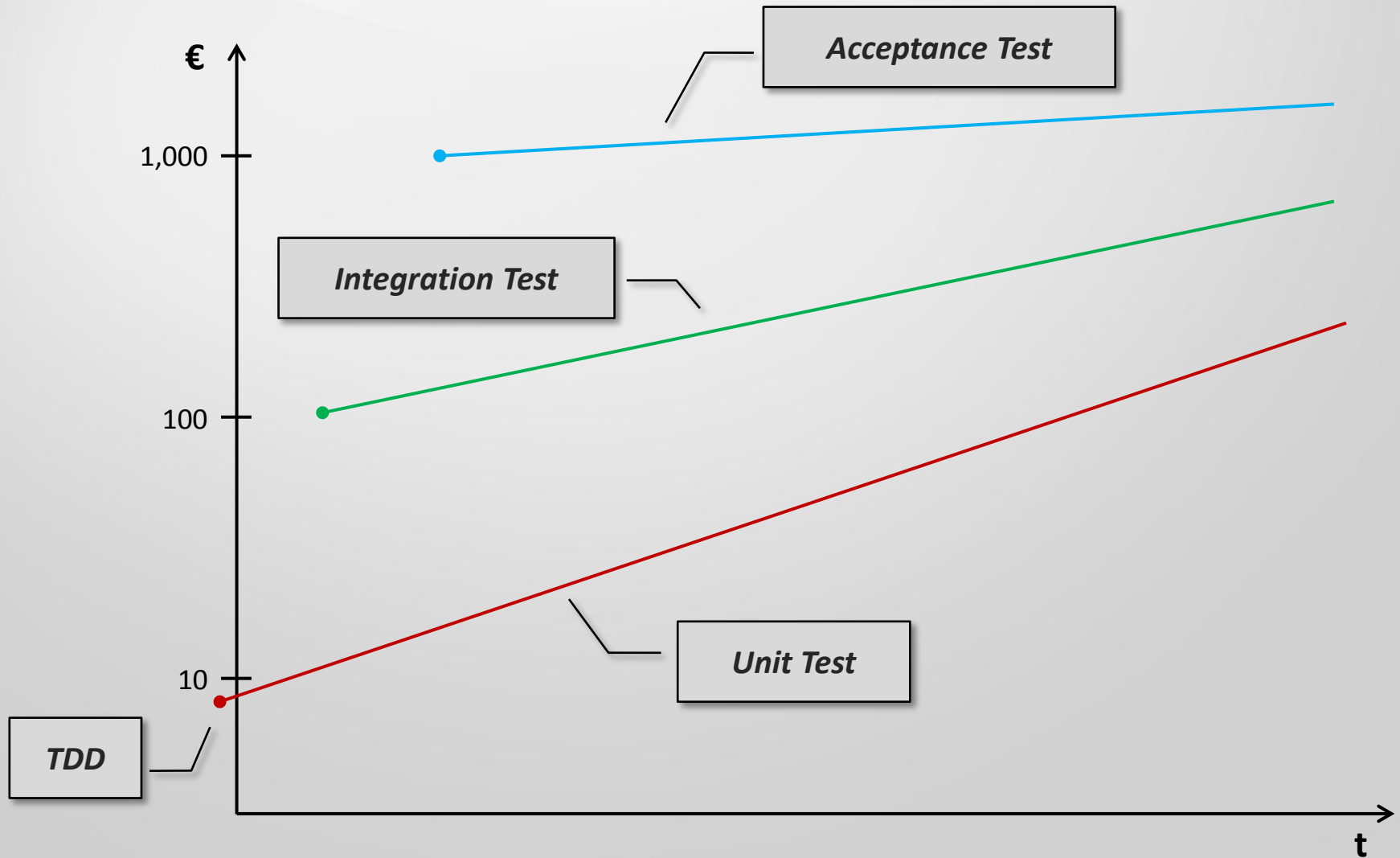


Test Automation Pyramid





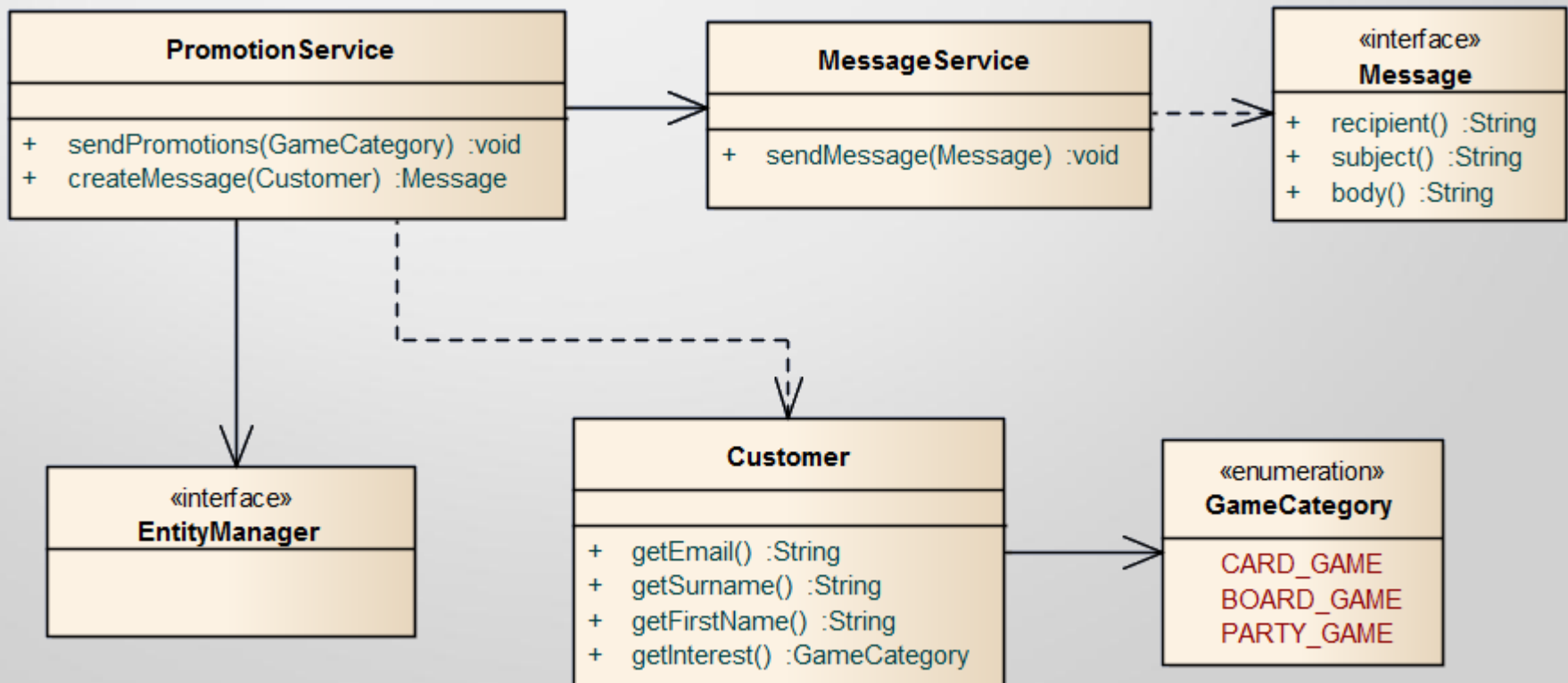
Cost of writing automated tests





First Approach (KISS)

(Keep It Simple, Stupid)





PromotionService Implementation (KISS)

```
public class PromotionService {

    @PersistenceContext
    private EntityManager em;

    @Inject
    private MessageService messageService;

    public void sendPromotions(GameCategory category) {

        List<Customer> customers = em
            .createNamedQuery("findByInterest", Customer.class)
            .setParameter("interest", category)
            .getResultList();

        for (Customer customer: customers) {
            messageService.sendMessage(createMessage(customer));
        }
    }

    Message createMessage(final Customer customer) {
        return new Message() {...};
    }
}
```



PromotionService Implementation (2)

```
public class PromotionService {

    @PersistenceContext
    private EntityManager em;

    @Inject
    private MessageService messageService;

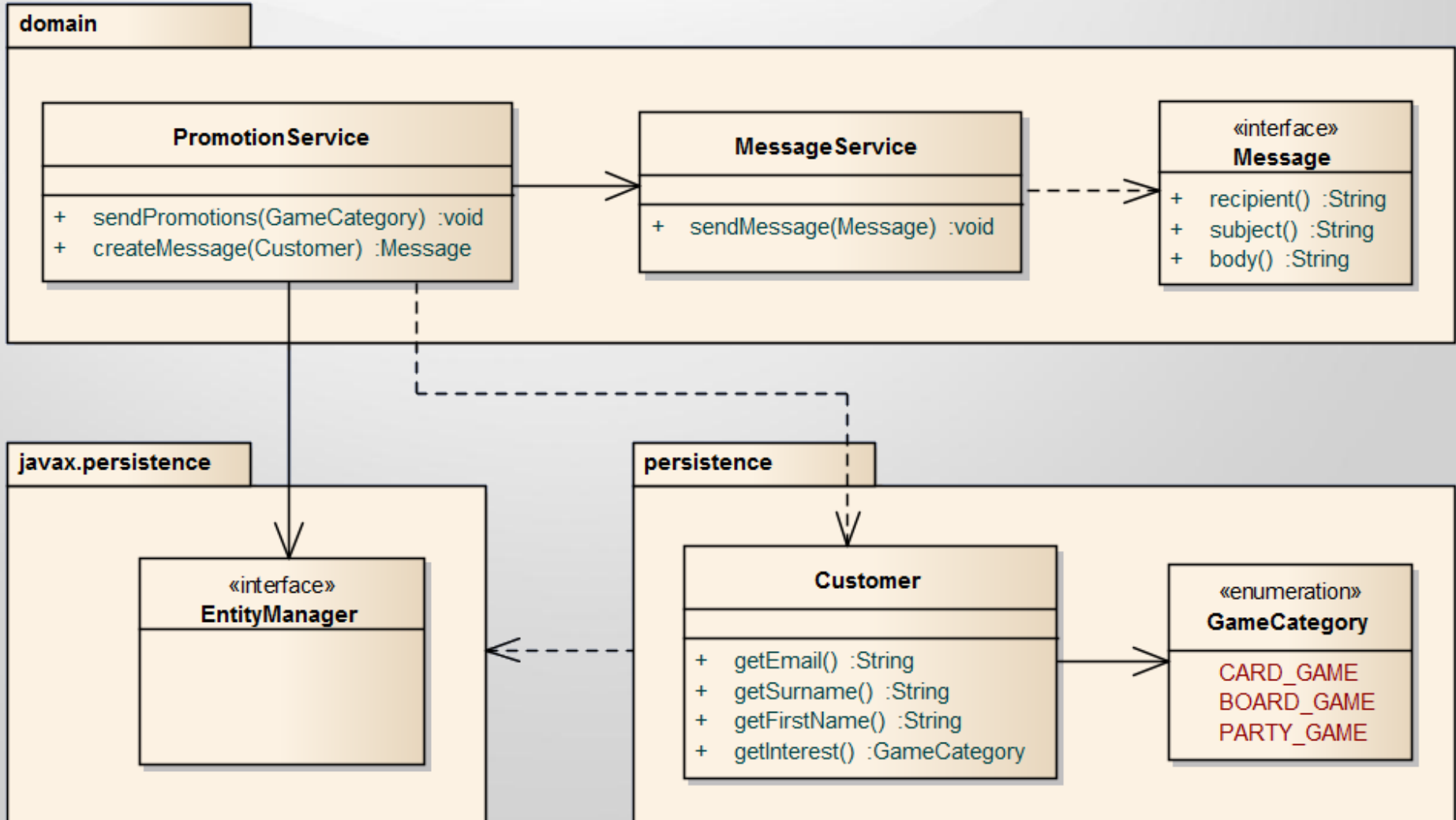
    public void sendPromotions(GameCategory category) {
        for (Customer customer: findCustomersByInterest(category)) {
            messageService.sendMessage(createMessage(customer));
        }
    }

    Collection<Customer> findCustomersByInterest(GameCategory interest) {
        return em.createNamedQuery("findByInterest", Customer.class)
            .setParameter("interest", interest)
            .getResultList();
    }

    Message createMessage(final Customer customer) {
        return new Message() {...};
    }
}
```

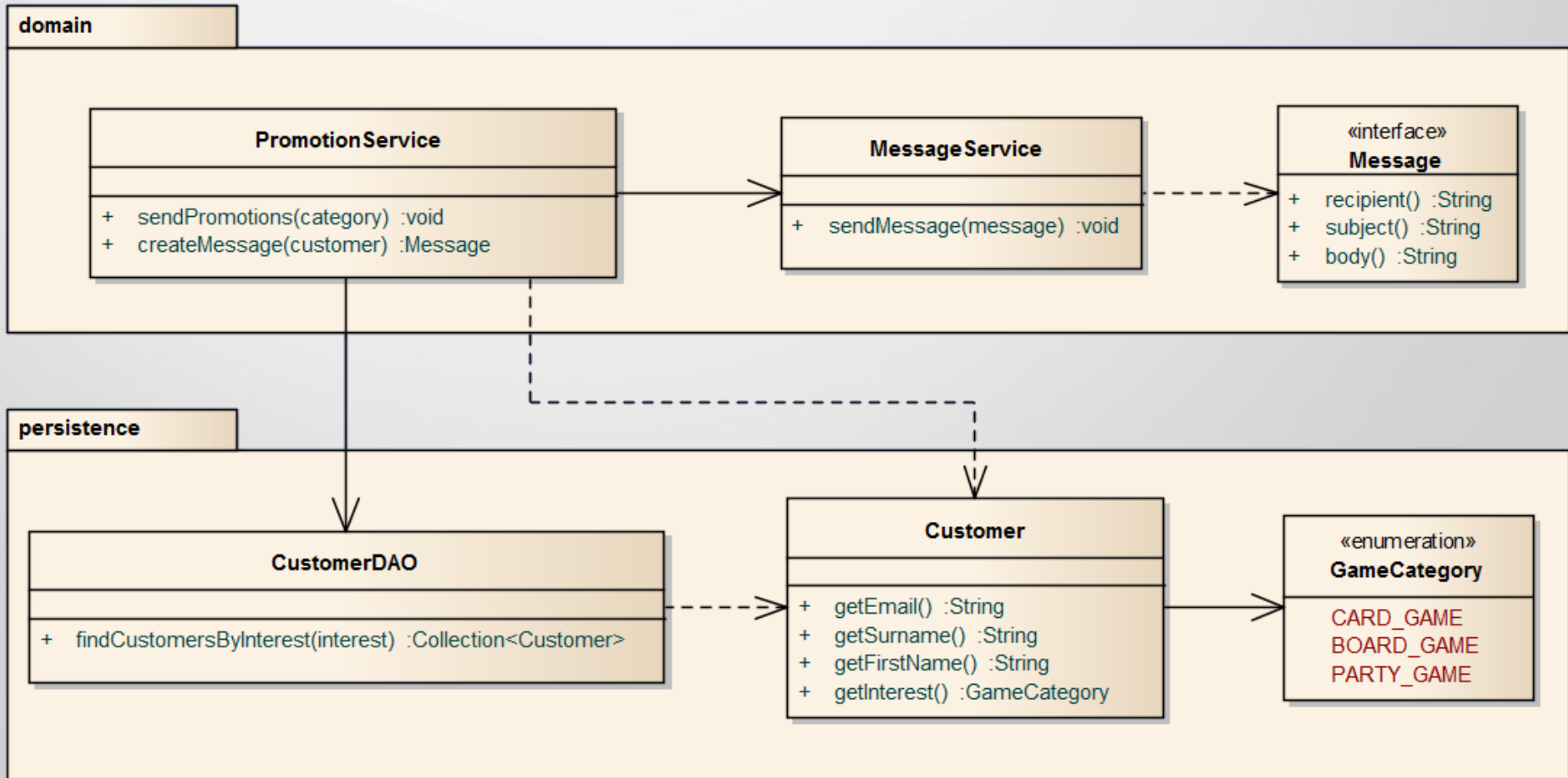


KISS Approach (Layered Architecture?)





Layered Architecture Approach





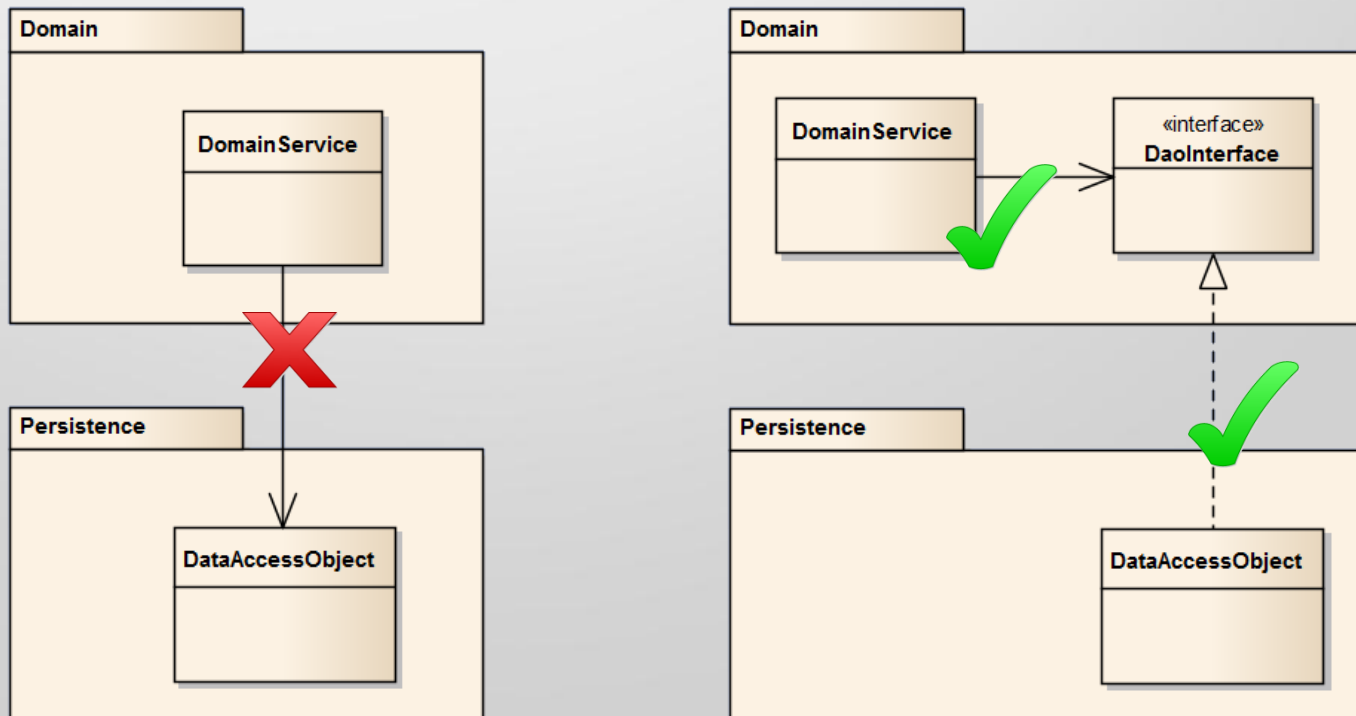
PromotionService (Layered Architecture)

```
public class PromotionService {  
    @Inject  
    private CustomerDAO customerDAO;  
  
    @Inject  
    private MessageService messageService;  
  
    public void sendPromotions(GameCategory category) {  
        for (Customer customer: customerDAO.findCustomersByInterest(category)) {  
            messageService.sendMessage(createMessage(customer));  
        }  
    }  
  
    Message createMessage(final Customer customer) {  
        return new Message() {...};  
    }  
}
```



Dependency Inversion Principle

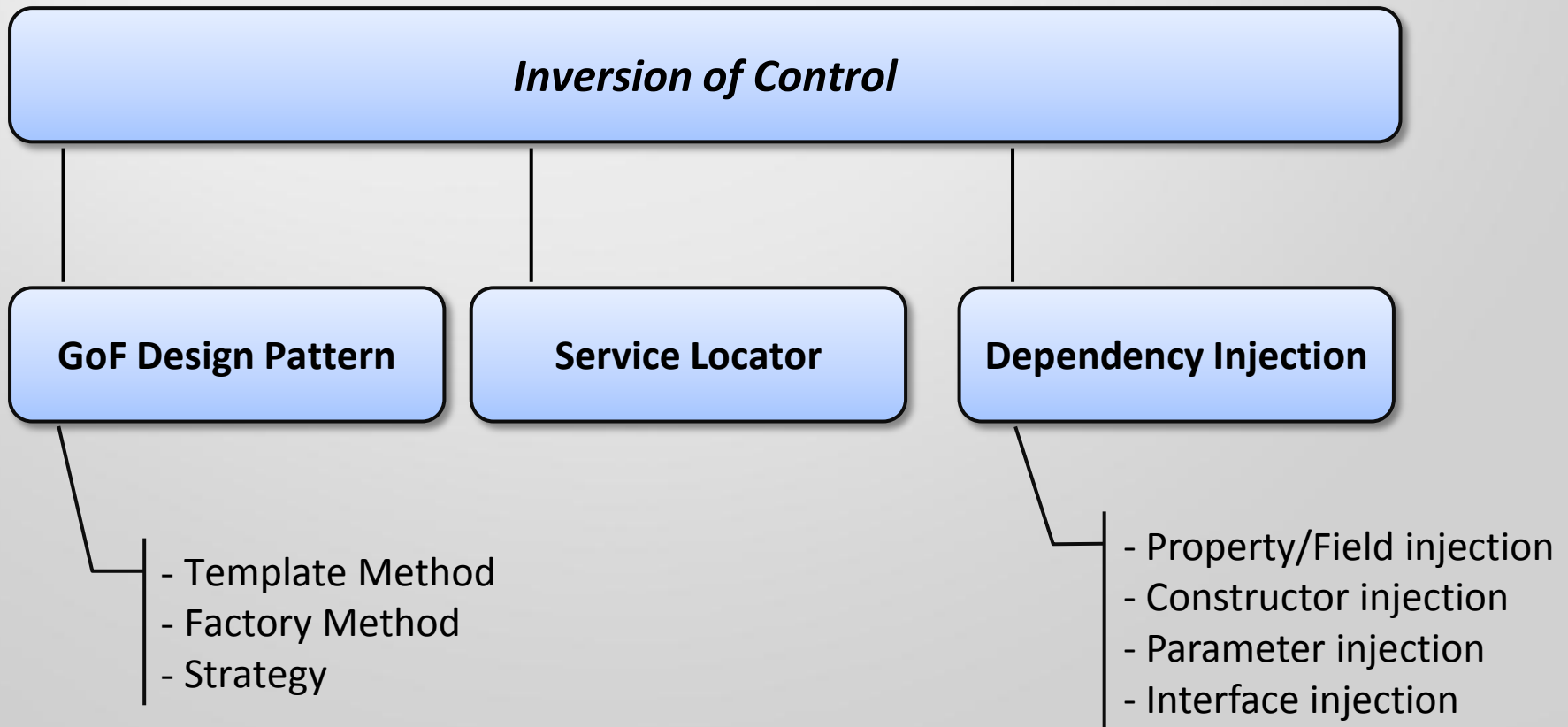
1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.





Inversion of Control (IoC)

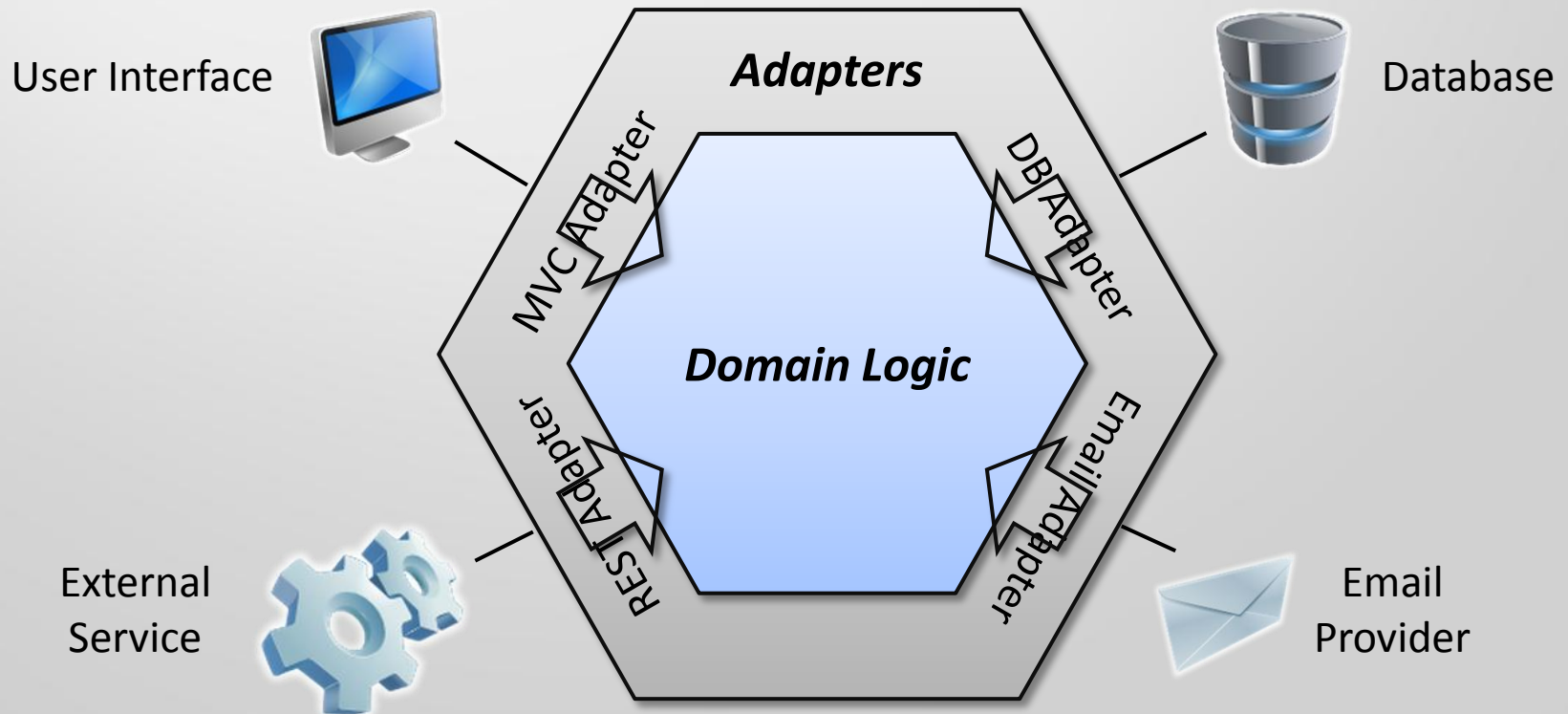
- **Hollywood principle:** *Don't call us, we'll call you.*





Hexagonal Architecture (aka *Port and Adapters*)

1. **Domain Logic** has no external dependencies.
2. **Adapters** depend on the **Domain Logic**.





Ports and Adapters

Ports are **entry points**, provided by the domain logic and define a set of functions.

- **Primary Port:** API of the domain logic
- **Secondary Port:** interface for a secondary adapter

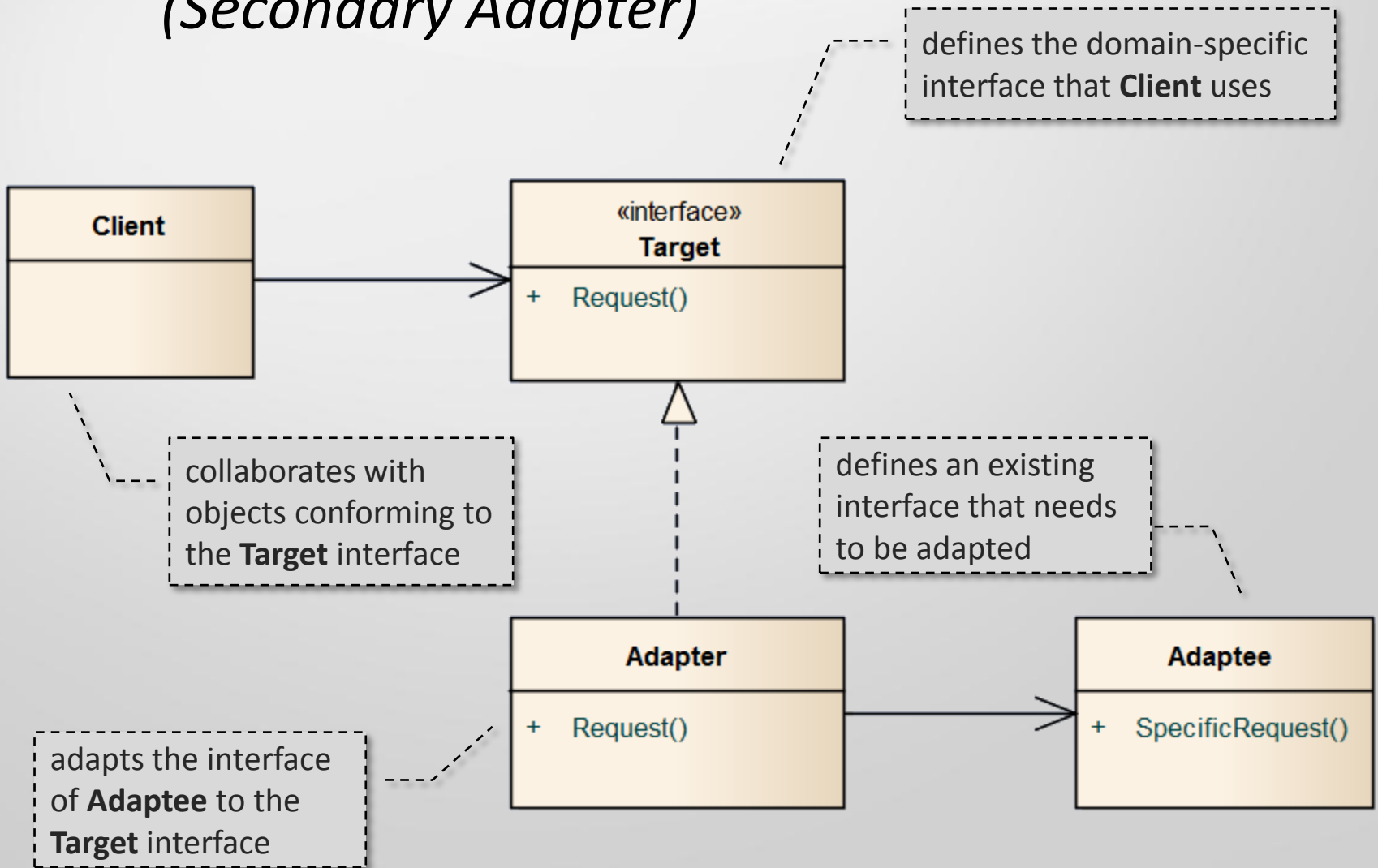
An **adapter** is a **bridge** between the application and an external service. It is assigned to one specific port.

- **Primary Adapter:** calls the API functions of the domain
- **Secondary Adapter:** implementation of a secondary port



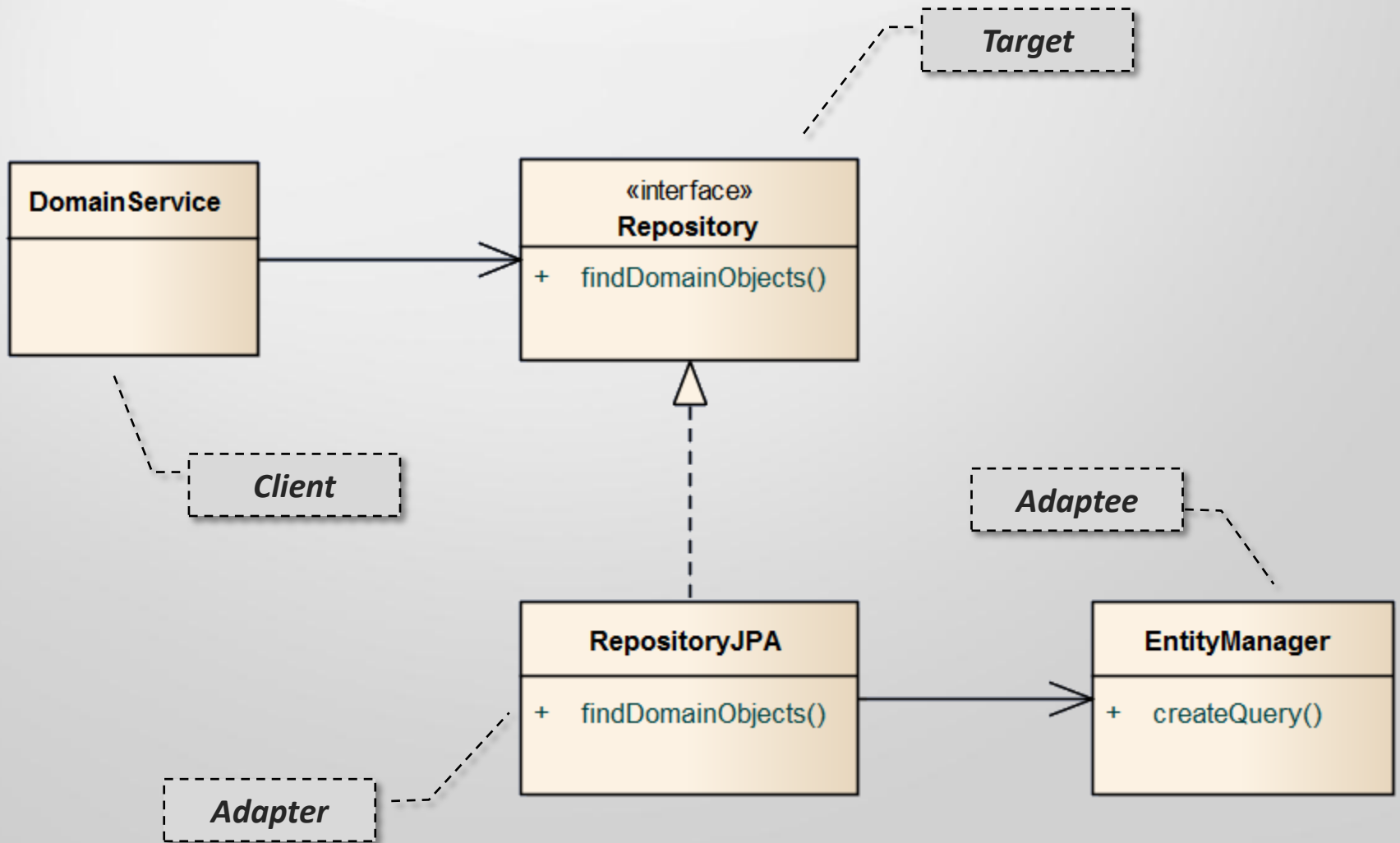
Adapter (Design Patterns - E. Gamma et al)

(Secondary Adapter)



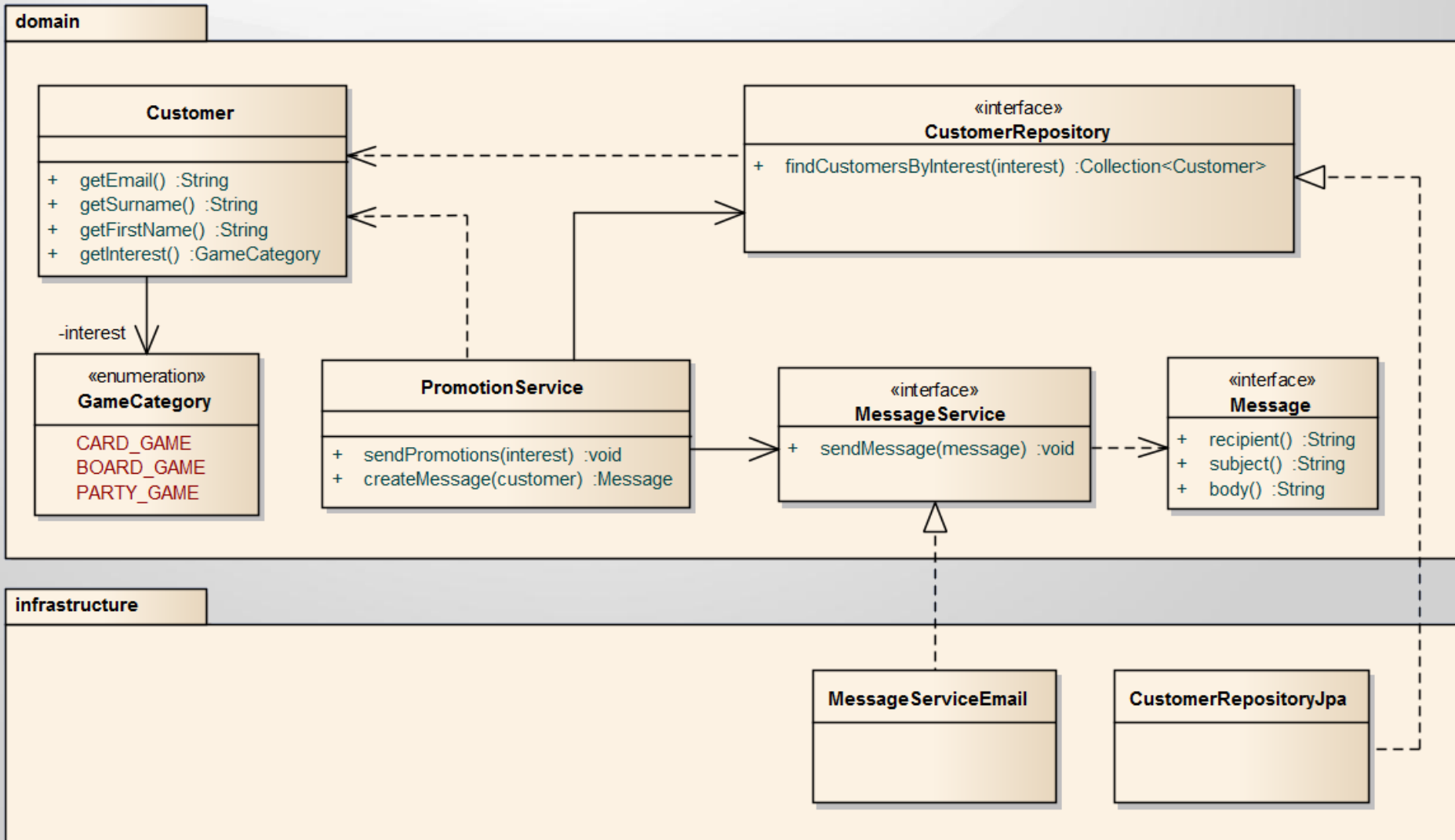


Hexagonal Architecture: Adapter Example





Hexagonal Architecture Approach





PromotionService (Hexagonal Architecture)

```
public class PromotionService {  
    @Inject  
    private CustomerRepository customerRepo;  
  
    @Inject  
    private MessageService messageService;  
  
    public void sendPromotions(GameCategory category) {  
        for (Customer customer: customerRepo.findCustomersByInterest(category)) {  
            messageService.sendMessage(createMessage(customer));  
        }  
    }  
  
    Message createMessage(final Customer customer) {  
        return new Message() {...};  
    }  
}
```



PromotionServiceTest (Hexagonal Architecture)

```
@RunWith(MockitoJUnitRunner.class)
public class PromotionServiceTest {

    @Mock
    private MessageService messageService;

    @Mock
    private CustomerRepository customerRepository;

    @Test
    public void sendPromotion_NoMatchingCustomers_NothingSent() {

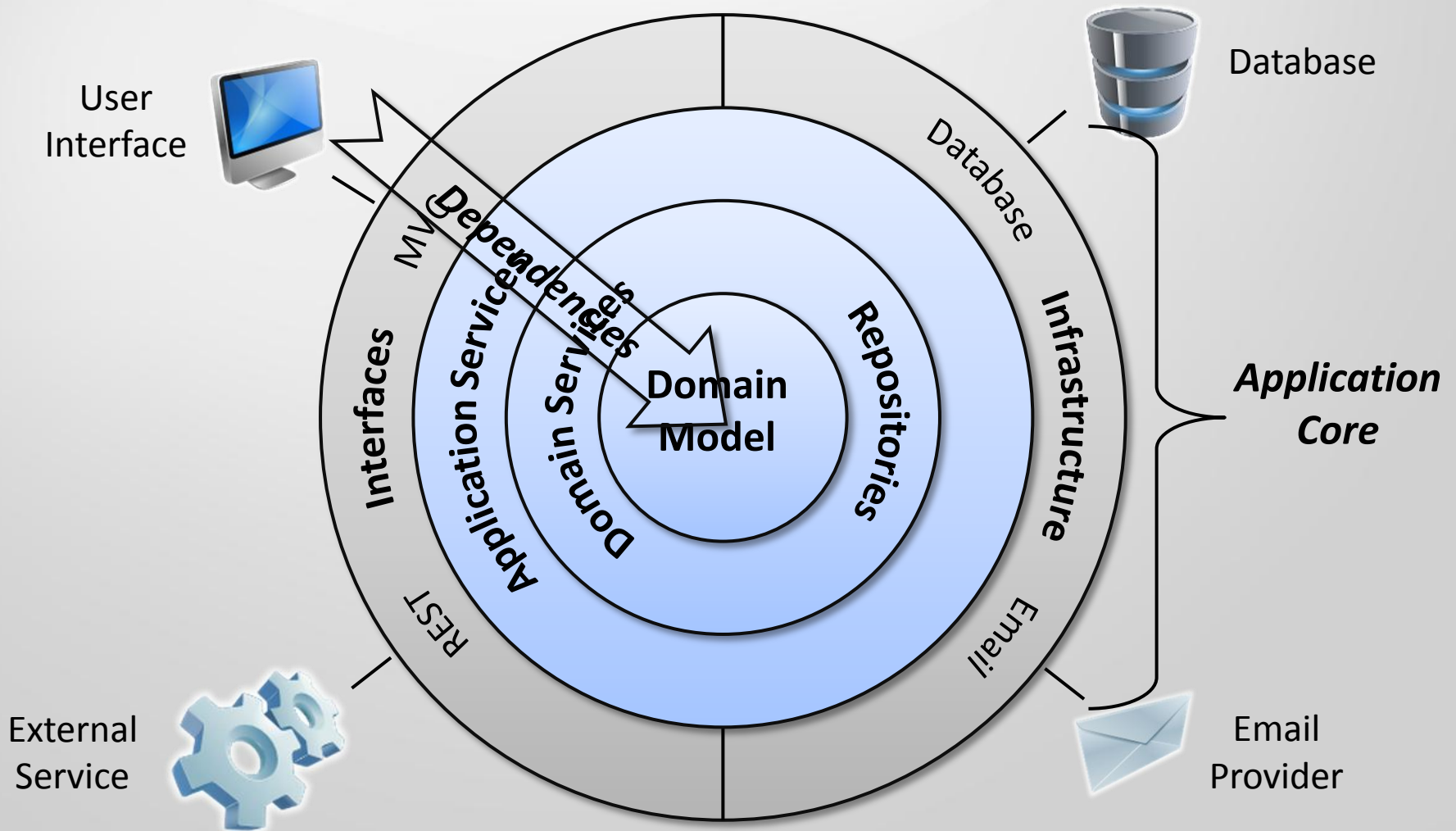
        PromotionService promotionService =
            new PromotionService(messageService, customerRepository);

        promotionService.sendPromotions(GameCategory.CARD_GAME);

        verify(customerRepository, times(1))
            .findCustomersByInterest(GameCategory.CARD_GAME);
        verify(messageService, never()).sendMessage(any(Message.class));
        verifyNoMoreInteractions(customerRepository, messageService);
    }
}
```

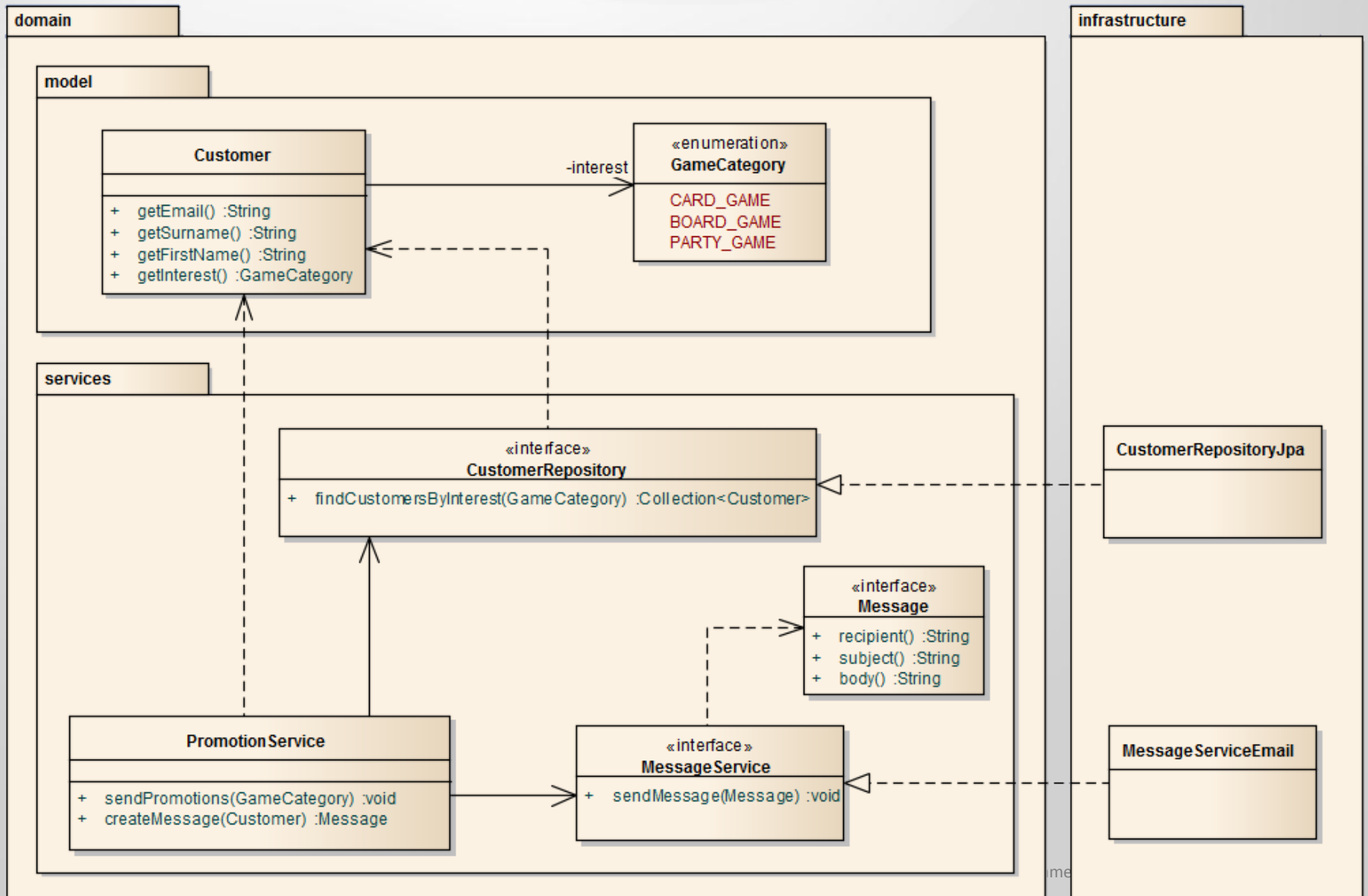


Onion Architecture





Onion Architecture Approach





Conclusion

- **Favor vertical slices over horizontal layers**
- **Avoid dependencies from the domain layer to low level APIs**
- **Build in testability from the very beginning**
- **Design for replacement instead of reuse**
- **Use the *Hexagonal Architecture* approach for complex domains (DDD)**



Thank You!

Any Questions?

