GUICE anyone?

lungu.cristian@gmai.com

Dependency injection

"[...] a software design pattern that allows removing hard-coded dependencies and making it possible to change them, whether at run-time or compile-time."

Wikipedia

How I found out about GUICE?

Problem: *Write a code that doesn't uses **if** statements*

"How to Write Clean, Testable Code"

"GUICE is the new NEW!"

Google

Motivation

```
public interface BillingService {

  /**
   * Attempts to charge the order to the credit card. Both successful and
   * failed transactions will be recorded.
   *
   * @return a receipt of the transaction. If the charge was successful, the
   *      receipt will be successful. Otherwise, the receipt will contain a
   *      decline note describing why the charge failed.
   */
  Receipt chargeOrder(PizzaOrder order, CreditCard creditCard);
}
```

Direct constructor calls

```
public class RealBillingService implements BillingService {
  public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {
    CreditCardProcessor processor = new PaypalCreditCardProcessor();
    TransactionLog transactionLog = new DatabaseTransactionLog();

    try {
      ChargeResult result = processor.charge(creditCard, order.getAmount());
      transactionLog.logChargeResult(result);

      return result.wasSuccessful()
          ? Receipt.forSuccessfulCharge(order.getAmount())
          : Receipt.forDeclinedCharge(result.getDeclineMessage());
    } catch (UnreachableException e) {
      transactionLog.logConnectException(e);
      return Receipt.forSystemFailure(e.getMessage());
    }
  }
}
```

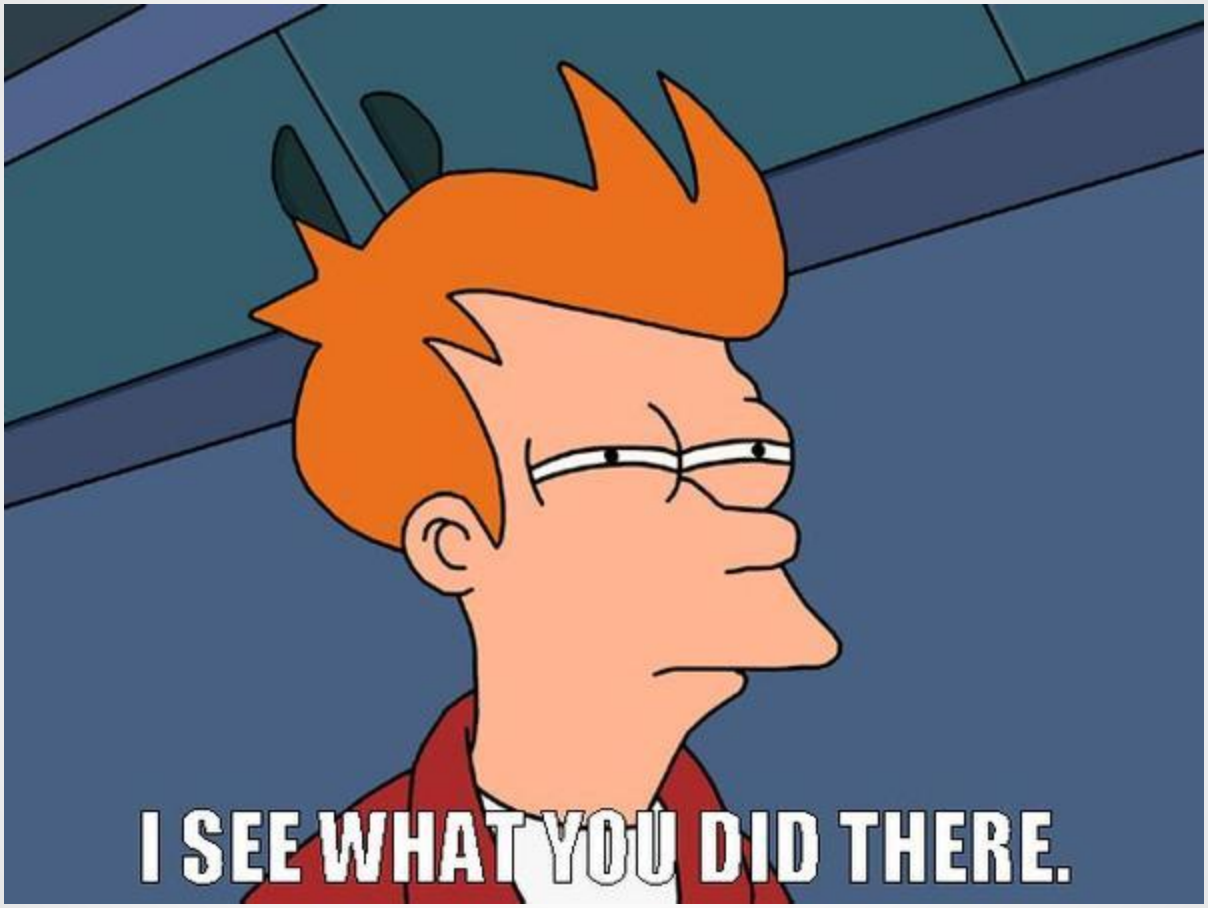
Factories

```
public class CreditCardProcessorFactory {

  private static CreditCardProcessor instance;

  public static void setInstance(CreditCardProcessor creditCardProcessor) {
    instance = creditCardProcessor;
  }

  public static CreditCardProcessor getInstance() {
    if (instance == null) {
      return new SquareCreditCardProcessor();
    }

    return instance;
  }
}
```

```
public class RealBillingService implements BillingService {
  public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {
    CreditCardProcessor processor = CreditCardProcessorFactory.getInstance();
    TransactionLog transactionLog = TransactionLogFactory.getInstance();

    try {
      ChargeResult result = processor.charge(creditCard, order.getAmount());
      transactionLog.logChargeResult(result);

      return result.wasSuccessful()
          ? Receipt.forSuccessfulCharge(order.getAmount())
          : Receipt.forDeclinedCharge(result.getDeclineMessage());
     } catch (UnreachableException e) {
      transactionLog.logConnectException(e);
      return Receipt.forSystemFailure(e.getMessage());
    }
  }
}
```

Dependency Injection

```
public class RealBillingService implements BillingService {
  private final CreditCardProcessor processor;
  private final TransactionLog transactionLog;

  public RealBillingService(CreditCardProcessor processor,
                            TransactionLog transactionLog) {
    this.processor = processor;
    this.transactionLog = transactionLog;
  }

  public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {
      ...
  }
}
```


I SEE WHAT YOU DID THERE.

What happened?

The dependency is exposed in the **API signature**.

```
public static void main(String[] args) {
    CreditCardProcessor processor = new PaypalCreditCardProcessor();
    TransactionLog transactionLog = new DatabaseTransactionLog();
    BillingService billingService
        = new RealBillingService(creditCardProcessor, transactionLog);
    ...
  }
```

Dependency Injection with Guice

```
public class BillingModule extends AbstractModule {
  @Override
  protected void configure() {
    bind(TransactionLog.class).to(DatabaseTransactionLog.class);
    bind(CreditCardProcessor.class).to(PaypalCreditCardProcessor.class);
    bind(BillingService.class).to(RealBillingService.class);
  }
}

  public static void main(String[] args) {
    Injector injector = Guice.createInjector(new BillingModule());
    BillingService billingService = injector.getInstance(BillingService.class);
    ...
  }
```

```
public class RealBillingService implements BillingService {
  private final CreditCardProcessor processor;
  private final TransactionLog transactionLog;

  @Inject
  public RealBillingService(CreditCardProcessor processor,
                            TransactionLog transactionLog) {
    this.processor = processor;
    this.transactionLog = transactionLog;
  }

  public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {
      ...
}
```

Why?

- Testing
- Debugging
- Code reuse
- Modules

Demo

@Inject
@AssistedInject
@Named
@Provides
@Singleton

Benefits

- Scopes
- Avaliable for Android too!
- Search for RoboGuice

Disadvantages

- Overhead
- A **lot** of interfaces
- A **lot** of annotations

Conclusions

- Very usefull tool
- Must be used with measure!
- Helps to a clean design

Source

https://github.com/clungu/geekmeet.git

Thank you!

Questions?