Interacção Humana com o Computador

Aula 5

Departamento de Informática
UBI 2024/2025

João Cordeiro
jpcc@ubi.pt

Chapter 3

The Interaction

The Interaction

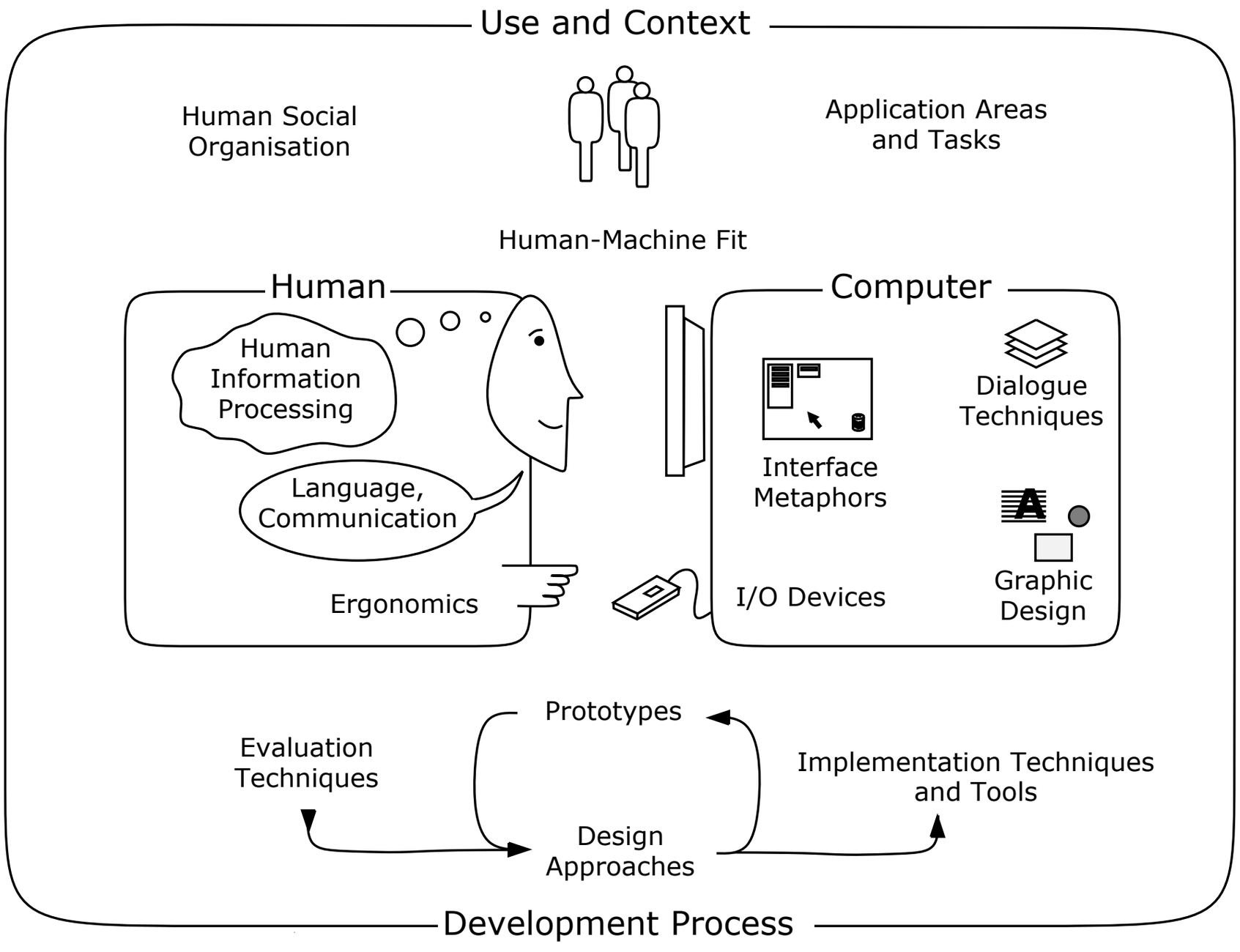


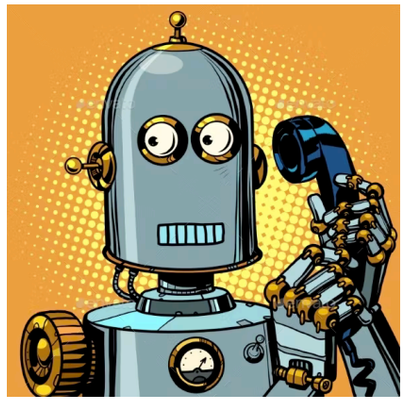
Figure 1.1: The nature of Human-Computer Interaction. Adapted from Figure 1 of the ACM SIGCHI Curricula for Human-Computer Interaction [Hewett et al., 2002]

What is interaction?

Communication

User ⟺ **System**

Task language



Core language

Some terms of interaction

Domain: The **area** of work under study

e.g. graphic design

Goal: **What** you want to achieve

e.g. create a solid red triangle

Task: **How** you go about doing it
ultimately in terms of operations or actions

e.g. … select fill tool, click over triangle

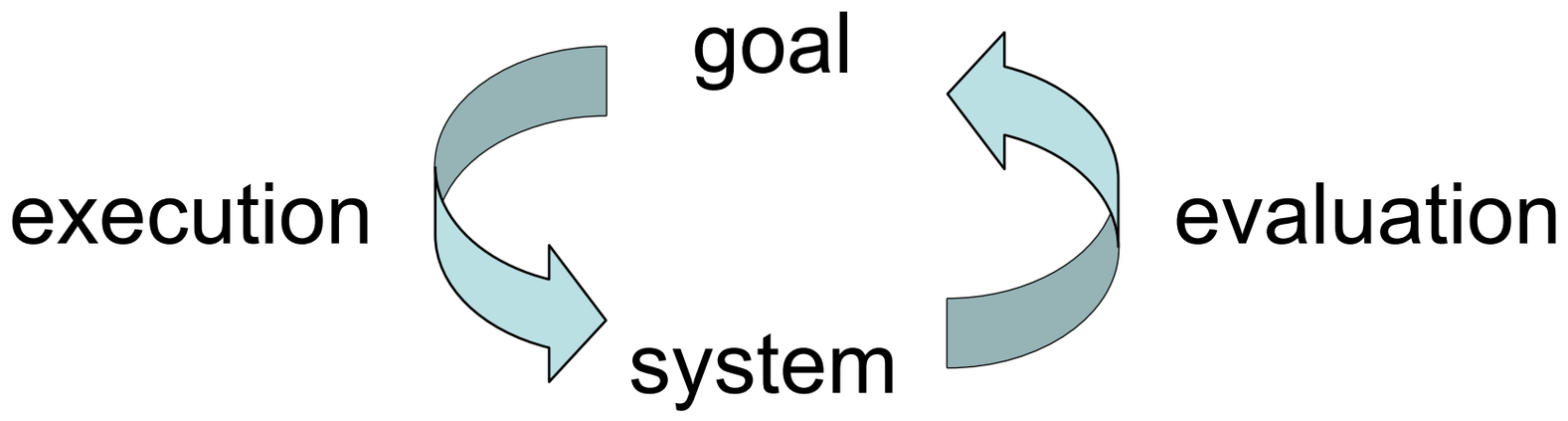
Donald Norman's Model

- **Seven stages**
  1. User establishes the goal
  2. Formulates intention
  3. Specifies actions at interface
  4. Executes action
  5. Perceives system state
  6. Interprets system state
  7. Evaluates system state with respect to goal

- **Norman's model concentrates on user's view of the interface**

Donald Norman's Model

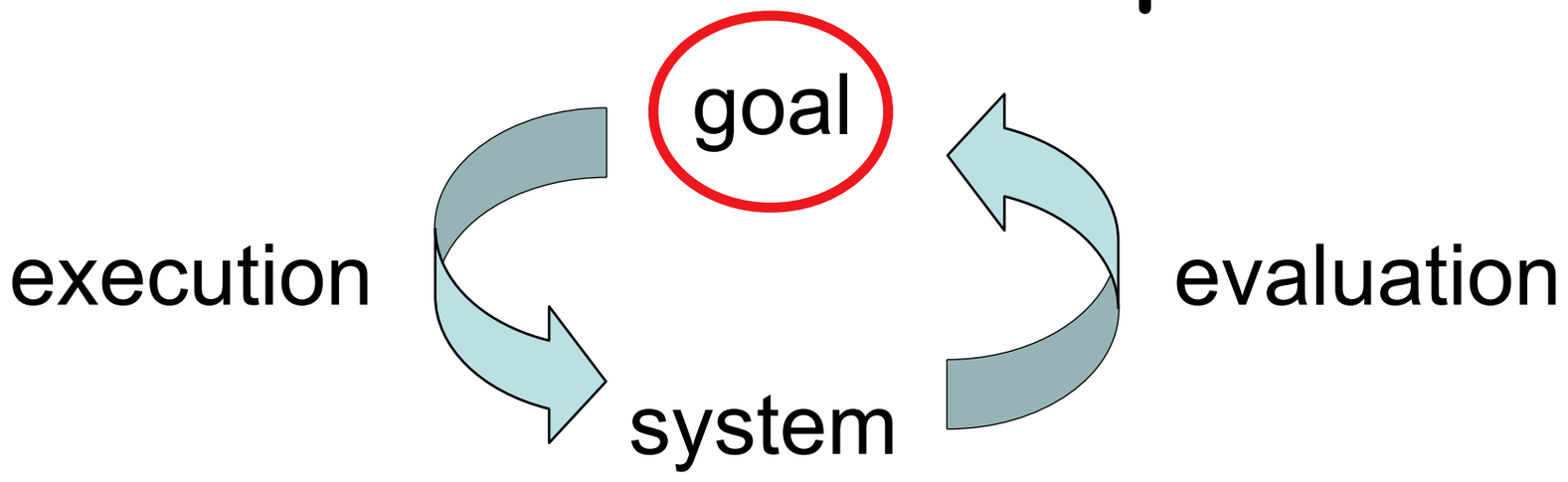
execution/evaluation loop



- user establishes the goal
- formulates intention
- specifies actions at interface
- executes action
- perceives system state
- interprets system state
- evaluates system state with respect to goal

Donald Norman's Model

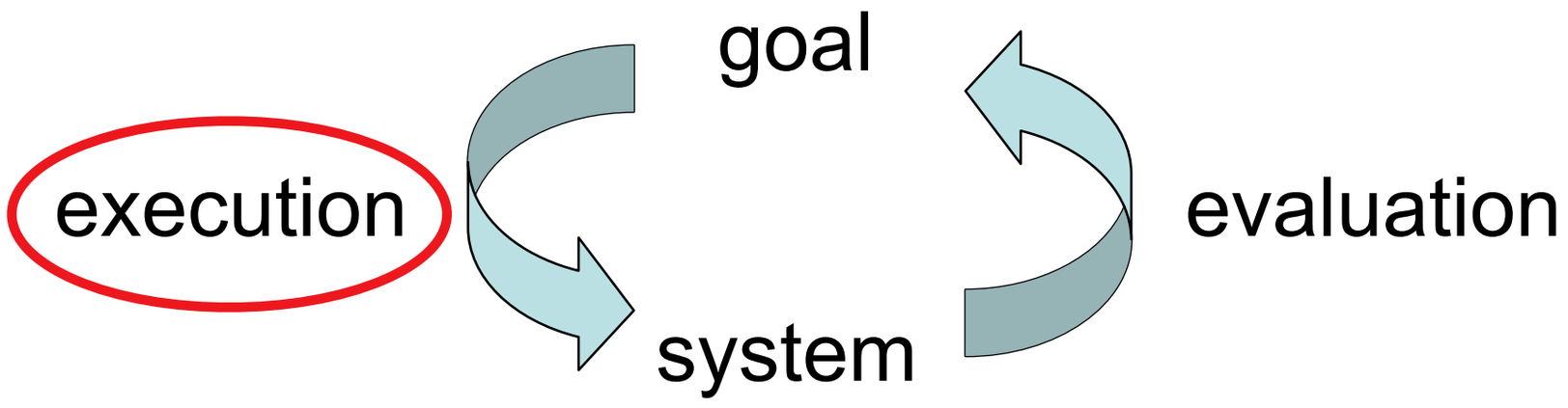
execution/evaluation loop



- user establishes the goal
- formulates intention
- specifies actions at interface
- executes action
- perceives system state
- interprets system state
- evaluates system state with respect to goal

Donald Norman's Model

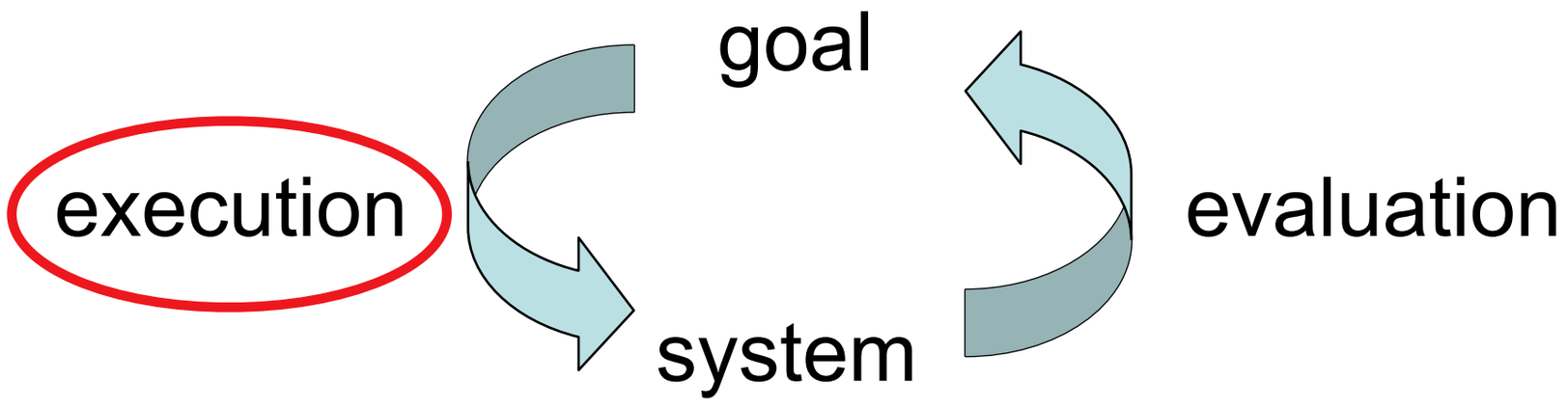
execution/evaluation loop



- user establishes the goal
- formulates intention
- specifies actions at interface
- executes action
- perceives system state
- interprets system state
- evaluates system state with respect to goal

Donald Norman's Model

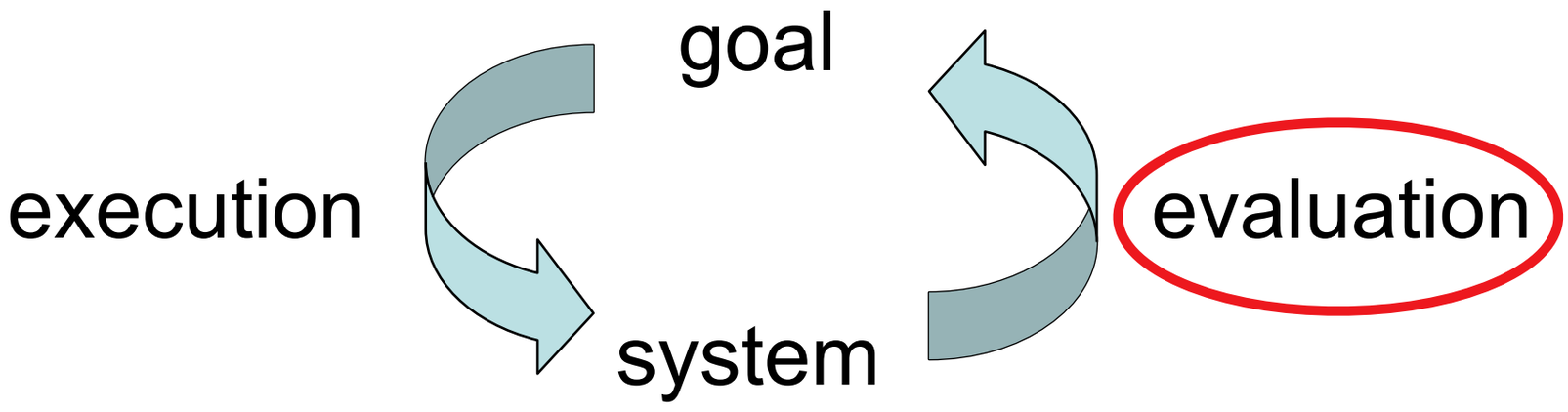
execution/evaluation loop



- user establishes the goal
- formulates intention
- specifies actions at interface
- executes action
- perceives system state
- interprets system state
- evaluates system state with respect to goal

Donald Norman's Model

execution/evaluation loop



- user establishes the goal
- formulates intention
- specifies actions at interface
- executes action
- perceives system state
- interprets system state
- evaluates system state with respect to goal

Using Norman's Model

Some systems are harder to use than others

Why?

Gulf of Execution

user's formulation of actions

≠ actions allowed by the system

Gulf of Evaluation

user's expectation of changed system state

≠ actual presentation of this state

Using Norman's Model

Human error —> Slips and Mistakes

Slip

- ☺ understand system and goal
- ☺ correct formulation of action
- ☹ incorrect action

Mistake

- ☹ may not even have right goal!

Fixing things?

Slip: better interface design

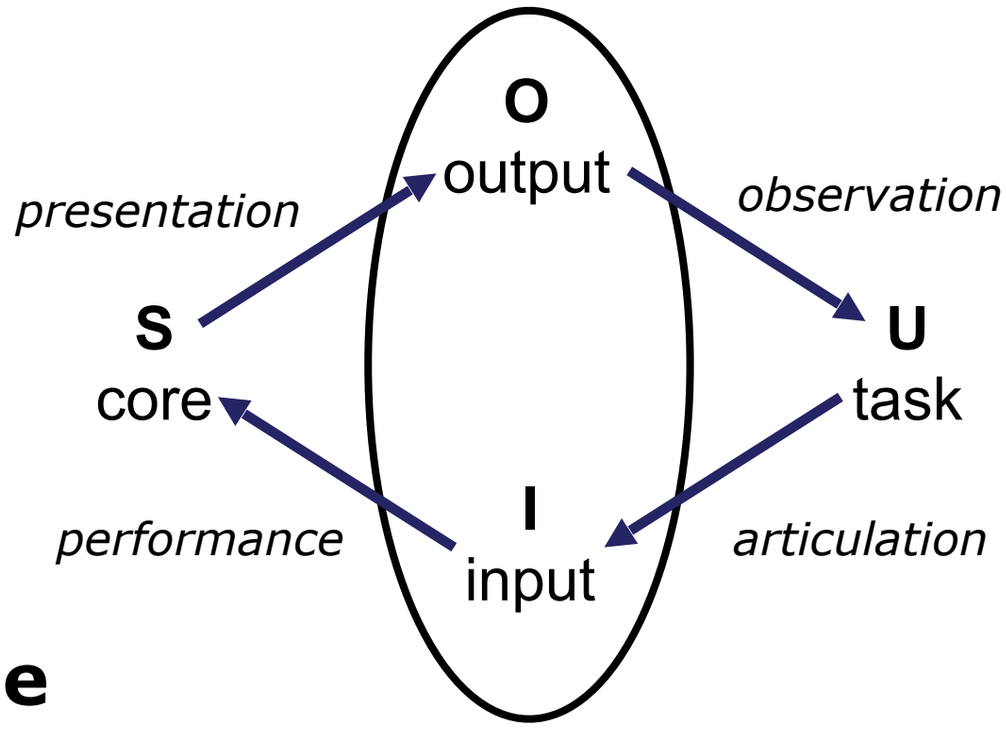
Mistake: better understanding of system

Abowd and Beale framework

The Interaction Framework

Extension of Norman's Model their interaction framework has **4 parts**

- **U**ser
- **I**nput
- **S**ystem
- **O**utput



Each has its own unique language

Interaction ⇒ Translation between languages

Problems in interaction == Problems in translation

Using Abowd & Beale's Model

User intentions

→ translated into **actions** at the interface

→ translated into **alterations** of system state

→ reflected in the **output** display

→ **interpreted** by the user

General framework for understanding interaction

- **not restricted** to electronic computer systems
- identifies **all** major components involved in interaction
- allows comparative assessment of systems
- an abstraction

The Ergonomics of Interaction

- Study of the **physical characteristics** of interaction (ex: how controls are designed)

- Also known as **human factors** – but this can also be used to mean much of HCI!

- **Ergonomics** good at defining standards and guidelines for constraining the way we design certain aspects of systems

The Ergonomics of Interaction

Examples:

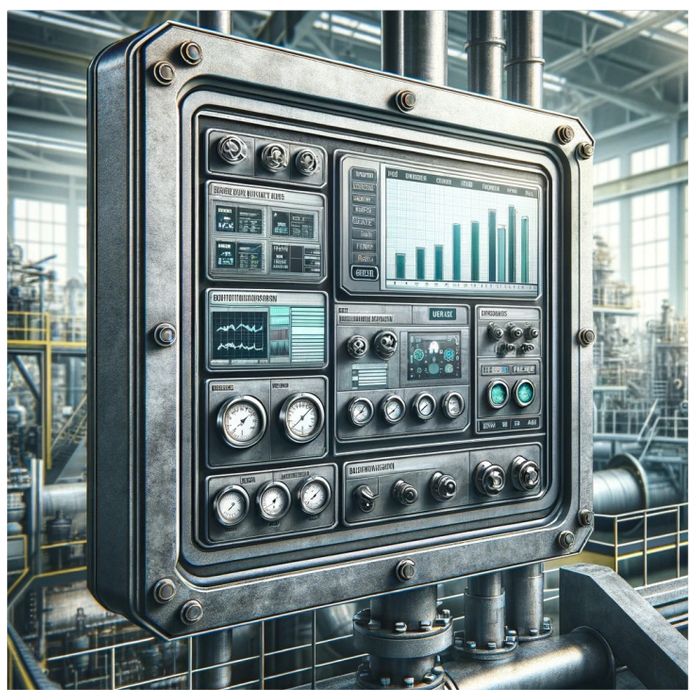
- **Arrangement of controls and displays**
  e.g. controls grouped according to function or frequency of use, or sequentially
- **Surrounding environment**
  e.g. seating arrangements adaptable to cope with all sizes of user
- **Health issues**
  e.g. physical position, environmental conditions (temperature, humidity), lighting, noise,
- **Use of color**
  e.g. use of red for warning, green for okay, awareness of colour-blindness and culture etc.

The Ergonomics of Interaction

- **Arrangement of controls and displays**
  - e.g. controls grouped according to function or frequency of use, or sequentially

- **Functional**
- **Sequential**
- **Frequency**

Industrial interfaces



Are they different?

Office interface vs. industrial interface?

Context matters!

	office	industrial
type of data	textual	numeric
rate of change	slow	fast
environment	clean	dirty

not only this …

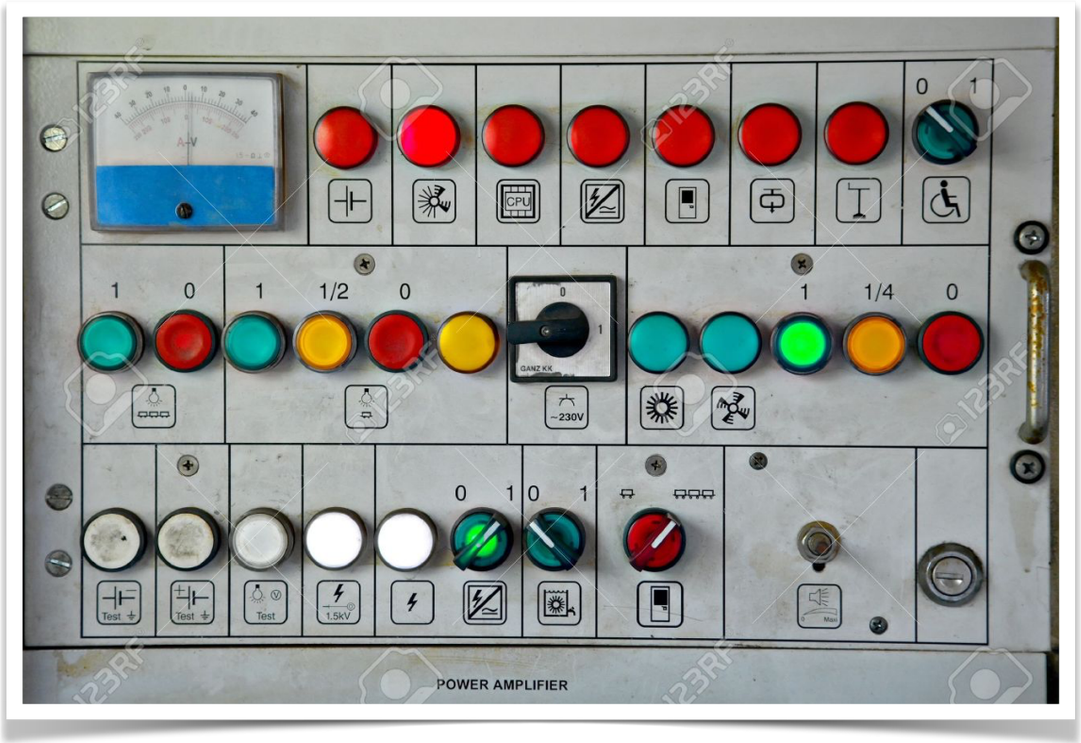
Industrial interfaces

MULTIVAC Marking & Inspection

Office interface vs. industrial interface?

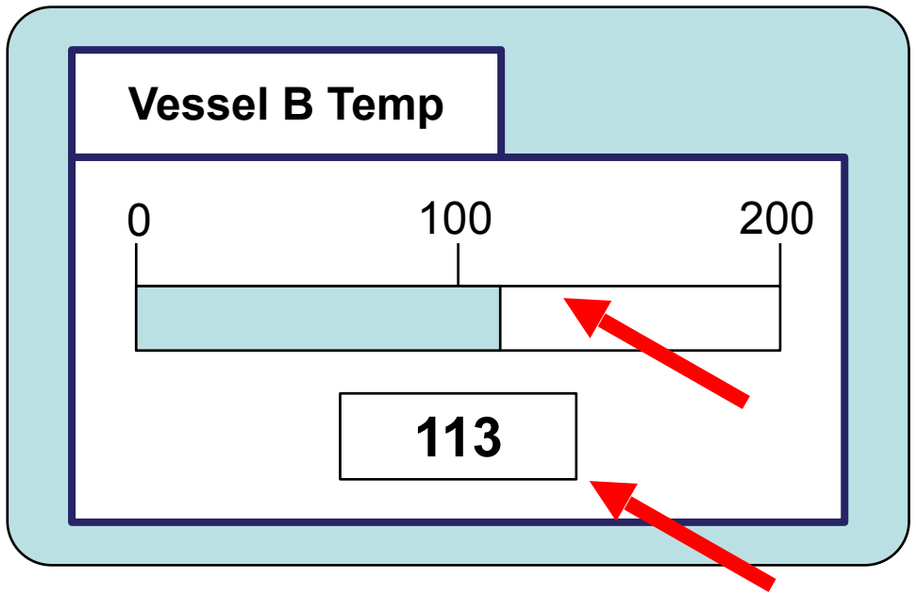
Context matters!

	office	industrial
type of data	textual	numeric
rate of change	slow	fast
environment	clean	dirty



Glass interfaces

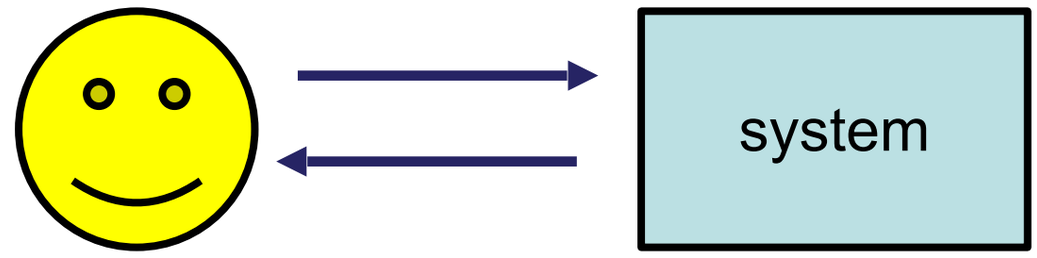
- **Industrial interface:**
  - traditional … dials and knobs
  - now … screens and keypads
- **Glass interface**
  + cheaper, more flexible, multiple representations, precise values
  - not physically located, loss of context, complex interfaces
- **May need both**



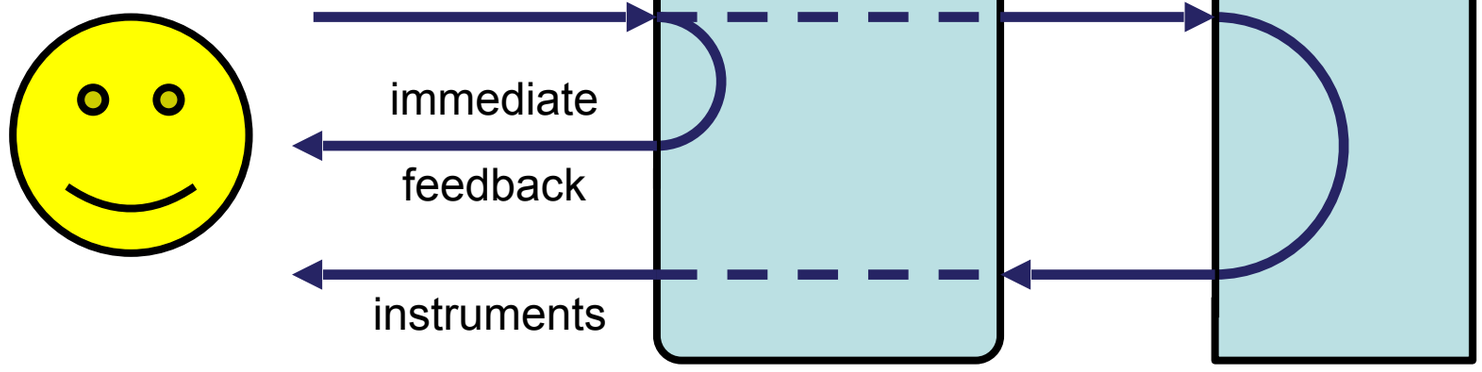
Multiple representations of same information

Direct & Indirect Manipulation

- **Office** – direct manipulation
  - User interacts with an artificial world



- **Industrial** – indirect manipulation
  - User interacts *with* the real world, but *through* an interface
- **Issues** …
  - Feedback
  - Delays



Propriedades e Ligações Dinâmicas em JavaFX

Java Interface

- Na forma mais simplista podemos dizer que uma **interface** é um conjunto de métodos *abstratos* agregados sob uma estrutura.
- Ultrapassa a limitação da herança múltipla em POO.
- Define um protocolo geral de ação/comportamento a ser detalhado e implementado por classes (java class) no futuro. Estas comprometem-se a garantir esse comportamento.
- Amplamente utilizado no tratamento de eventos, nas interfaces gráficas, quer do **Swing** quer do **JavaFX**.

Java Interface

```
interface <NomeI> {
    … metodo1(…);
    … metodo2(…);
    … … …
    … metodoN(…);
}
```

Definidos com prefixo opcional “abstract” ou em alternativa “default”. Neste último caso teremos de fornecer uma implementação.

```
class <NomeC> implements <NomeI> {
    … … …
    … metodo1(…) {…}
    … metodo2(…) {…}
    … … …
    … metodoN(…) {…}
    … … …
}
```

Utilidade?

Java Interface — Exemplo:

Definidos com prefixo opcional “abstract” ou em alternativa “default”. Neste último caso teremos de fornecer uma implementação.

```
public interface FiguraGeometrica
{
    abstract double perimetro();

    abstract double area();


    default double racio() {
        return   perimetro() / area();
    }

    default String getNome() {
        return this.getClass().getSimpleName();
    }

    default boolean maiorQue(FiguraGeometrica other) {
        return this.area() > other.area();
    }
}
```

Java Interface — Exemplo:

```java
public class Circulo implements FiguraGeometrica
{
    private final double raio;


    public Circulo(double raio) {
        if ( raio < 0 )  raio = 0;
        this.raio= raio;
    }

    public double getRaio() {
        return raio;
    }


    @Override
    public double perimetro() {
        return 2.0*Math.PI*raio;
    }

    @Override
    public double area() {
        return Math.PI * raio*raio;
    }
}
```

```java
public class Retangulo implements FiguraGeometrica
{
    private final double comprimento;
    private final double largura;


    public Retangulo(double comprimento, double largura) {
        this.comprimento= comprimento;
        this.largura= largura;
    }

    public double getComprimento() {
        return comprimento;
    }

    public double getLargura() {
        return largura;
    }


    @Override
    public double perimetro() {
        return 2*comprimento + 2*largura;
    }

    @Override
    public double area() {
        return comprimento * largura;
    }
}
```

Java Interface & GUI Events

- No tratamento de eventos de interfaces gráficas, uma component (ex: `Button`) informa um *listener* de algo que sucedeu.
- Um *listener* é a implementação (`implements`) de uma `Java interface`, (ex: `ActionListener`) especificando a resposta ao evento correspondente.
- O *listener* terá de ficar "registado" na component correspondente: `componente.addListener(…)`
- Vejamos a seguir um pequeno exemplo …

Java Interface & GUI Events

Java Swing

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;

public class ButCuriosidade
{
    public static void main(String[] args) {
        JButton myBut= new JButton("Curiosidade");

        myBut.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent ae) {
                System.out.println("A curiosidade é perigosa!");
            }
        });

        JFrame window= new JFrame("Frame Curiosidade (IHC 2015/16)");
        window.getContentPane().add(myBut);
        window.setSize(300, 182);
        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        window.setLocationRelativeTo(null);
        window.setVisible(true);
    }
}
```

java.awt.event

Class ActionEvent

java.lang.Object
- java.util.EventObject
  - java.awt.AWTEvent
    - java.awt.event.ActionEvent

java.awt.event

Interface ActionListener

All Superinterfaces:

EventListener

Java Interface & GUI Events

JavaFX

```
public class SimpleEvents extends Application {
    @Override
    public void start(Stage primaryStage) {
        Button myButton = new Button();
        myButton.setText("Say 'Hello World'");
        myButton.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                System.out.println("Hello World!");
            }
        });

        StackPane root = new StackPane();
        root.getChildren().add(myButton);
        Scene scene = new Scene(root, 300, 250);
        primaryStage.setTitle("Hello World!");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

javafx.event

Interface EventHandler<T extends Event>

Type Parameters:

T - the event class this handler can handle

All Superinterfaces:

java.util.EventListener

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;
```

Java Lambdas

- Nova funcionalidade do Java 8 que implementa aspetos de *programação funcional*.

- Simplifica o código, nomeadamente em situações de *classes anónimas interiores*.

- Permite uma definição dinâmica de funções e sua passagem como argumento de outra função/método.

- Assim, um método pode receber tipos primitivos, objetos e funções (*lambdas*).

Java Lambdas

- Em Java, uma expressão lambda é composta por três partes: (1) a lista de argumentos; (2) o símbolo “->” e (3) o corpo do lambda, por exemplo:

(1)	(2)	(3)
Argument List	**Arrow Token**	**Body**
`(int x, int y)`	`->`	`x + y`

```
(String s) -> { System.out.println(s); }

() -> 47
```

Java Lambdas

Exemplo concreto:

```
public class RunnableTest {

    public static void main(String[] args) {
        System.out.println("=== RunnableTest ===");

        Runnable r1 = new Runnable() {
            @Override
            public void run() {
                System.out.println("Hello world one!");
            }

        };

        Runnable r2 = () -> System.out.println("Hello world two!");

        r1.run();
        r2.run();
    }
}
```

java.lang

Interface Runnable

All Known Subinterfaces:

RunnableFuture<V>,
RunnableScheduledFuture<V>

All Known Implementing Classes:

AsyncBoxView.ChildState,
ForkJoinWorkerThread, FutureTask,
RenderableImageProducer, SwingWorker,
Thread, TimerTask

`public interface` **`Runnable`**

The `Runnable` interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called `run`.

This interface is designed to provide a common protocol for objects that wish to execute code while

Java Events with Lambdas

Java Swing

```
import java.awt.event.ActionEvent;
import javax.swing.JButton;
import javax.swing.JFrame;

public class ButCuriosidade
{
    public static void main(String[] args) {
        JButton myBut= new JButton("Curiosidade");

        myBut.addActionListener((ActionEvent ae) -> {
            System.out.println("A curiosidade é perigosa!");
        });

        JFrame window= new JFrame("Frame Curiosidade (IHC 2015/16)");
        window.getContentPane().add(myBut);
        window.setSize(300, 182);
        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        window.setLocationRelativeTo(null);
        window.setVisible(true);
    }
}
```

Java Events com Lambdas

Java Swing

Functional Interface !

```
public static void main(String[] args) {
    JButton myBut= new JButton("Curiosidade");

    myBut.addActionListener(new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent ae) {
            System.out.println("A curiosidade é perigosa!");
        }
    });
```

```
public static void main(String[] args) {
    JButton myBut= new JButton("Curiosidade");

    myBut.addActionListener((ActionEvent ae) -> {
        System.out.println("A curiosidade é perigosa!");
    });
```

Java Events com Lambdas

JavaFX

Functional Interface !

```
Button myButton = new Button();

myButton.setText("Say 'Hello World'");
myButton.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
        System.out.println("Hello World!");
    }
});
```

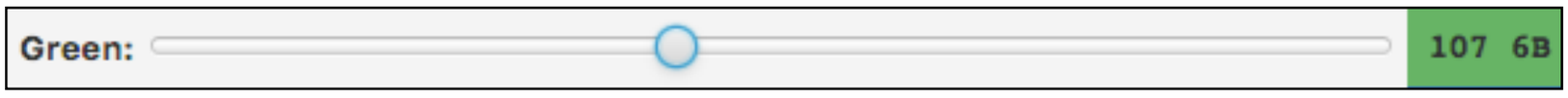
```
Button myButton = new Button();

myButton.setText("Say 'Hello World'");
myButton.setOnAction((ActionEvent event) -> {
    System.out.println("Hello World!");
});
```

Java Events com Lambdas

JavaFX

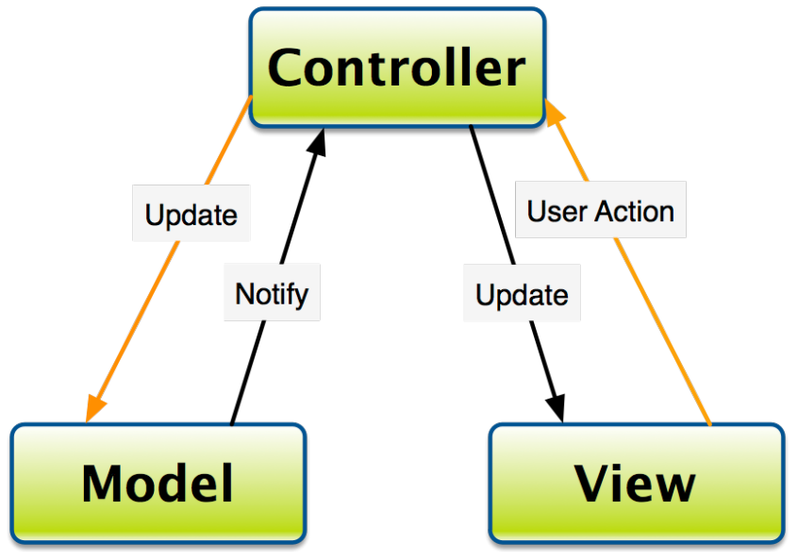
```
@Override
public void initialize(URL url, ResourceBundle rb) {
    slideRed.valueProperty().addListener(new ChangeListener() {
        @Override
        public void changed(ObservableValue observable, Object oldValue, Object newValue) {
            pintar();
        }
    });
    slideGreen.valueProperty().addListener((observable, oldValue, newValue) -> pintar());
    slideBlue.valueProperty ().addListener((observable, oldValue, newValue) -> pintar());
}
```



Slider

Observadores e Observados

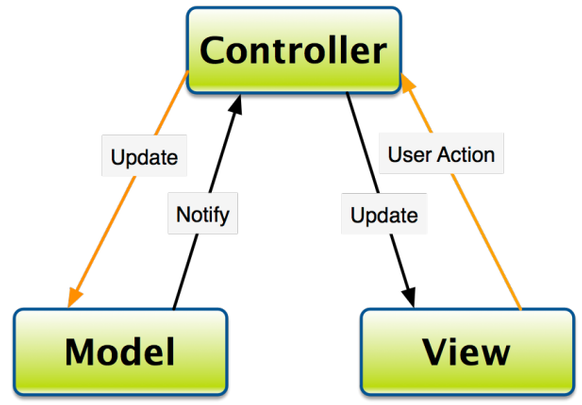
- O Java oferece infraestrutura de suporte ao desenvolvimento **MVC**, ideal para projetos com GUI



- Os pares Observados–Observadores são uma das técnicas de suporte em JavaFX, em particular nas propriedades

Observadores e Observados

- O Java oferece infraestrutura de suporte ao desenvolvimento **MVC**, ideal para projetos com GUI



- Os pares Observados—Observadores são uma das técnicas de suporte em JavaFX, em particular nas propriedades

- Existe um objeto com característica de observado e outro com característica de observador, emparelhado com o primeiro.

- Quando o estado do observado é alterado, o observador é notificado diretamente.

Observadores e Observados

O `Observer` é uma *interface*
e o `Observable` é uma *class*

```
public class Main
{
    public static void main(String[] args) {
        ObservableValue ovalue= new ObservableValue(1);
        ovalue.addObserver(new GeneralObserver());

        ovalue.setValor(2);
        ovalue.setValor(2*ovalue.getValor());
    }
}
```

```
import java.util.Observable;

public class ObservableValue extends Observable
{
    private int valor= 0;

    public ObservableValue(int n) {
        this.valor= n;
    }

    public void setValor(int n) {
        this.valor= n;
        setChanged();
        notifyObservers();
    }

    public int getValor() {
        return valor;
    }
}
```

```
import java.util.Observable;
import java.util.Observer;


public class GeneralObserver implements Observer
{
    @Override
    public void update(Observable o, Object arg) {
        System.out.printf("I saw a %s object changing its state!\n", o.getClass().getName());
        System.out.printf("Argument: %s\n\n", arg);
    }
}
```

Observadores e Observados

```java
public class Main2
{
    public static void main(String[] args) {
        ObservableValue x= new ObservableValue(1);
        Totoloto totA= new Totoloto("Euromilhões ...");
        Totoloto totB= new Totoloto("Lotaria .......");
        DetailedObserver policia= new DetailedObserver();

        x.addObserver(policia);
        totA.addObserver(policia);
        totB.addObserver(policia);

        x.setValor(3457);
        totA.jogar();
        totB.jogar();
    }
}
```

```java
import java.util.Observable;
import java.util.Observer;

public class DetailedObserver implements Observer
{
    @Override
    public void update(Observable ob, Object o) {
        if ( ob.getClass().getName().equals("Totoloto") ) {
            Totoloto loto= (Totoloto)o;
            System.out.printf("Saw Totoloto %s with value %d\n",
                    loto.getNome(),
                    loto.getNumero()
            );
        }
    }
}
```

```java
public class Totoloto extends Observable
{
    private final String nome;
    private final Random sorte;
    private int numero;

    public Totoloto(String nome) {
        this.nome= nome;
        this.numero= 0;
        this.sorte= new Random();
    }

    public void jogar() {
        numero= sorte.nextInt(49) + 1;
        setChanged();
        notifyObservers(this);
    }

    public String getNome() {
        return nome;
    }

    public int getNumero() {
        return numero;
    }
}
```

Propriedades e ligações (bindings)

- Baseado nos princípios anteriores, a **JavaFX** disponibiliza os conceitos de “propriedade observável” e “ligação entre propriedades”
- Elementos disponíveis nas packages:

```
javafx.beans.property.*
javafx.beans.binding.*
```

- Assim, podemos criar dependências automáticas entre as propriedades de diferentes objetos.
- É introduzida uma **nova convenção de nomes**, em especial para a definição de métodos de acesso às propriedades de objetos.

Propriedades — Exemplo:

```java
import javafx.beans.property.DoubleProperty;
import javafx.beans.property.SimpleDoubleProperty;

class Fatura
{
    private DoubleProperty valor = new SimpleDoubleProperty(0);

    public final double getValor() {
        return valor.get();
    }
    public final void setValor(double value) {
        valor.set(value);
    }

    public DoubleProperty valorProperty() {
        return valor;
    }
}
```

Novidade!

Propriedades — Exemplo:

```
import javafx.beans.value.ObservableValue;
import javafx.beans.value.ChangeListener;

public class MainFatura
{
    public static void main(String[] args) {
        Fatura electricidade = new Fatura();

        electricidade.valorProperty().addListener(new ChangeListener() {
            @Override
            public void changed(ObservableValue o, Object oldV, Object newV) {
                System.out.println("A fatura elétrica mudou!");
                System.out.println("Tipologia ..... "+o.getClass().getName());
                System.out.println("Valor antigo:... "+oldV);
                System.out.println("Valor novo:..... "+newV);
            }
        });

        electricidade.setValor(100.00);
    }
}
```

Ligações entre Propriedades

```
import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleIntegerProperty;


public class BindingExample
{
    public static void main(String[] args) {
        IntegerProperty a = new SimpleIntegerProperty(0);
        IntegerProperty b = new SimpleIntegerProperty(0);

        a.bind(b);
        b.set(4);
        System.out.println("a: "+a.getValue());
        System.out.println("b: "+b.getValue());

        a.unbind();
        a.set(3);
        System.out.println("a: "+a.getValue());
        System.out.println("b: "+b.getValue());

        a.bindBidirectional(b);
        b.set(7);
        System.out.println("a: "+a.getValue());
        System.out.println("b: "+b.getValue());

        a.set(9);
        System.out.println("a: "+a.getValue());
        System.out.println("b: "+b.getValue());
    }
}
```

Ligações entre Propriedades

```
import javafx.beans.binding.Bindings;
import javafx.beans.binding.NumberBinding;
import javafx.beans.property.IntegerProperty;
import javafx.beans.property.SimpleIntegerProperty;


public class MainSumBinding
{
public static void main(String[] args) {
        IntegerProperty a = new SimpleIntegerProperty(1);
        IntegerProperty b = new SimpleIntegerProperty(2);
        IntegerProperty c = new SimpleIntegerProperty(3);

        NumberBinding sum = a.add(b);
        NumberBinding som = Bindings.add(sum,c).multiply(3.75);
        System.out.println("Sum is: "+sum.getValue());
        System.out.println("Som is: "+som.getValue());
        a.set(3);
        b.set(-7);
        c.set(5);
        System.out.println("Sum is: "+sum.getValue());
        System.out.println("Som is: "+som.getValue());
    }
}
```