DCI

- How to get ahead in software architecture

About me

Andreas Söderlund

Independent consultant in northern Sweden.

Worked on larger web systems and applications for 10-15 years, specializing in system integration and architecture.

- E-commerce
- Banking
- Large-scale social platforms.

Haxe projects
Haxigniter, ufront, erazor, unject, umock, Haxedci, HaxeContracts

The roots of programming

FORTRAN - IBM Mathematical Formula Translating System

COBOL - COmmon Business-Oriented Language

"Formula translating", "Business-oriented"
Mapping from one language to another

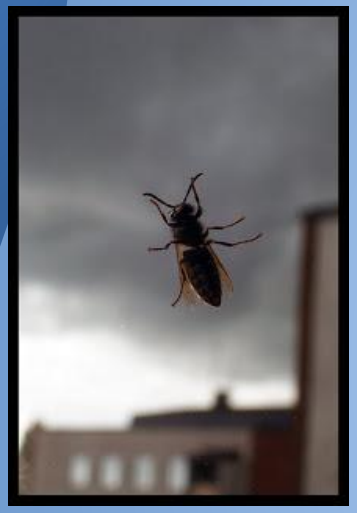
Mental Models

"An explanation of someone's thought process about how something works in the real world."

"Lift the receiver and call the person you want to reach by turning the circular wheel."

Object terminology (*receiver*, *wheel*)
- with a function (*call* a person)
- based on a form (the phone).

Form & Function



"I need a container to get the wasp out."

The form enables the function

Mental Model Matching

When the mental model matches, the objects you control becomes an extension of your mind.

It feels like you are directly manipulating the system.

Mental Model Matching...?

On the Insert tab, the galleries include items that are designed to coordinate with the overall look of

your document. You can use these galleries to insert tables, headers, footers, lists, cover pages, and other document building blocks. When you create pictures, charts, or diagrams, they also coordinate with your current document look.

When the mental model mismatches, you lose control.
The programmers mental model mismatched the users.

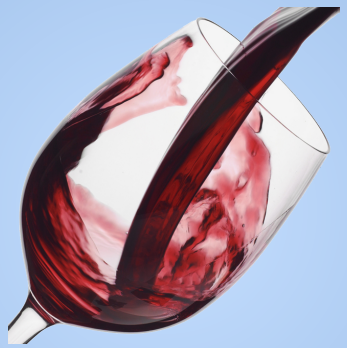
Note: All unit tests passed in the software above!

The System

"A set of interacting components forming an integrated whole"

Objects with <u>form</u>...

...that can have different types of <u>function</u>



Mental models: Thought processes about the behavior of the above

Maybe all this can be called a **system.**

And the form of the system, its **architecture**?

```
class Employee
{
	public var name : String;
	public var birth : Date;
}

class Waiter extends Employee
{
}

class Chef extends Employee
{
}

class Manager extends Employee
{
}
```

Designing a System

There must be a register with name and birthdate of all employees.

Making a digital ordering system for restaurants.

Use when:
- Serving food
- Cooking food
- Customer pays bill
- Manager pays salary

```
class Employee
{
      public var name : String;
      public var birth : Date;
}

class Waiter extends Employee
{
}

class Chef extends Employee
{
}

class Manager extends Employee
{
}
```

Designing a System

The mental model of serving guests at a Restaurant:

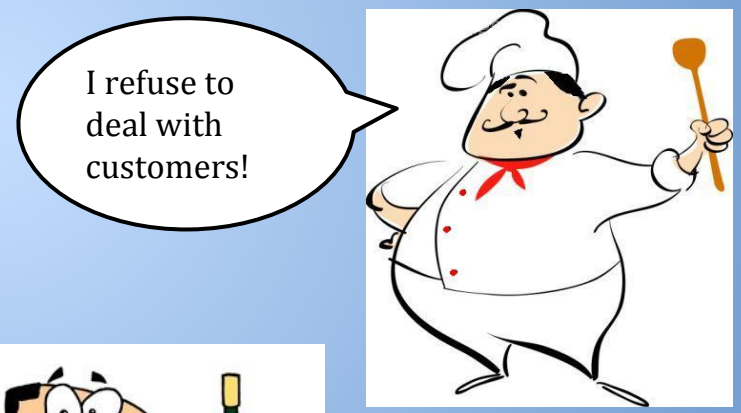
"The waiter walks up to the guests and shows them the menu. He takes the order and passes it on to the chef. The chef cooks the food, which is brought out to the guests by the waiter.

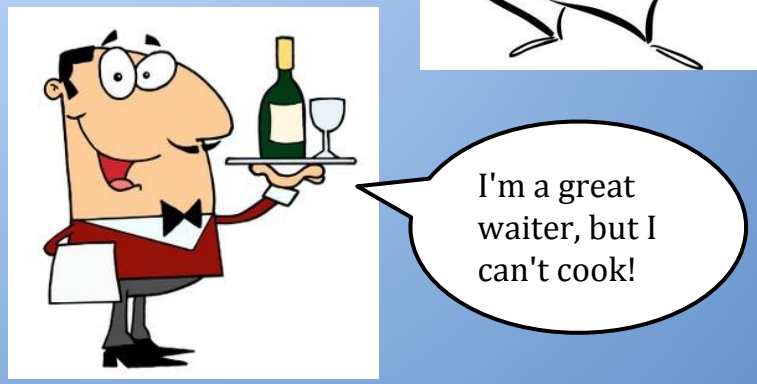
After the meal the waiter brings the bill to the guests, and collects the payment."

```
class Employee
{
	public var name : String;
	public var birth : Date;

	public function serveFood() {}
	public function cookFood() {}
	public function bringBill() {}
}

class Waiter extends Employee
{
}

class Chef extends Employee
{
}

class Manager extends Employee
{
	public function paySalaries() {}
}
```

Part I Inheritance

The Manager can also be a Chef and Waiter, if the restaurant is small.





```
class Employee
{
	public var name : String;
	public var birth : Date;
}

class Waiter extends Employee
{
	public function serveFood() {}
	public function bringBill() {}
}

class Chef extends Employee
{
	public function cookFood() {}
}

class Manager extends Employee
{
	public function paySalaries() {}
}

class ChefManager extends Chef extends Manager
{
}

class WaiterManager extends Waiter extends Manager
{
}
```

Part I.5: Multiple Inheritance

The Manager can also be a Chef and Waiter, if the restaurant is small.

Not all Managers are both Chefs and Waiters of course!

…what if even more employee types are added?

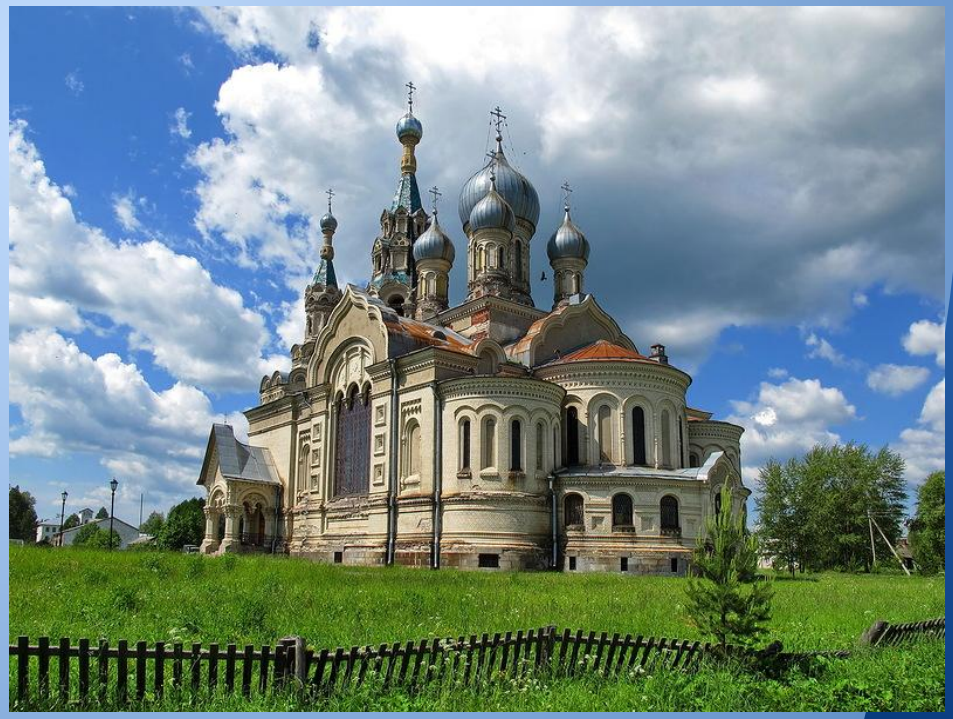
…and how to instantiate this hierarchy?

```
class Employee
{
	public var name : String;
	public var birth : Date;
}

interface IChef
{
	function cookFood() : Food;
}

interface IWait
{
	function serveFood(f : Food) : Void;
	function bringBill() : Money;
}

interface IManage
{
	function paySalaries() : Void;
}

class Chef extends Employee implements IChef
{
	public function cookFood() {}
}

class Waiter extends Employee implements IWait
{
	public function serveFood(f : Food) {}
	public function bringBill() {}
}

class Manager extends Employee implements IManage
{
	public function paySalaries() {}
}

class ChefManager extends Manager implements IChef
{
	public function cookFood() {}
}

class WaiterManager extends Manager ...
...
```

Part II Interfaces

This code doesn't have a beautiful form and architecture.

This is a beautiful architecture:



```
class Employee
{
	public var name : String;
	public var birth : Date;
}

interface IChef
{
	function cookFood() : Food;
}

interface IWait
{
	function serveFood(f : Food) : Void;
	function bringBill() : Money;
}

interface IManage
{
	function paySalaries() : Void;
}

class Chef extends Employee implements IChef
{
	public function cookFood() {}
}

class Waiter extends Employee implements IWait
{
	public function serveFood(f : Food) {}
	public function bringBill() {}
}

class Manager extends Employee implements IManage
{
	public function paySalaries() {}
}

class ChefManager extends Manager implements IChef
{
	public function cookFood() {}
}

class WaiterManager extends Manager ...
...
```

Part II Interfaces

The code looks more like this:

```
class Employee
{
	public var name : String;
	public var birth : Date;
}

interface IChef
{
	function cookFood() : Food;
}

interface IWait
{
	function serveFood(f : Food) : Void;
	function bringBill() : Money;
}

interface IManage
{
	function paySalaries() : Void;
}

class Chef extends Employee implements IChef
{
	public function cookFood() {}
}

class Waiter extends Employee implements IWait
{
	public function serveFood(f : Food) {}
	public function bringBill() {}
}

class Manager extends Employee implements IManage
{
	public function paySalaries() {}
}

class ChefManager extends Manager implements IChef
{
	public function cookFood() {}
}

class WaiterManager extends Manager ...
...
```

Part II Interfaces

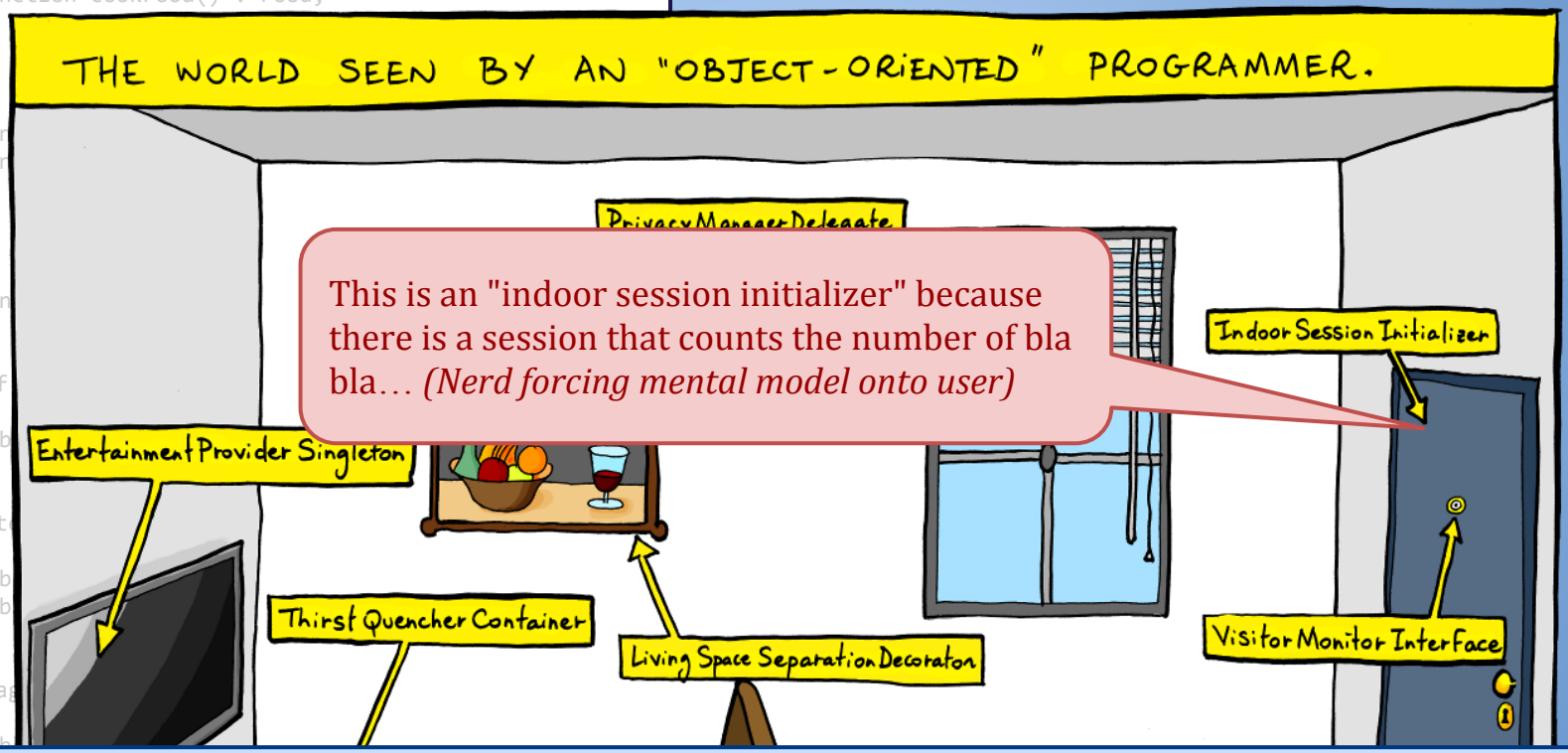
The system form is not a representation of the user mental model anymore (maybe somewhere deep within), it's more and more about engineering.

Want some Design Patterns with that?

- Abstract Factory
- Adapter
- Command
- Decorator
- Flyweight
- Memento
- Strategy
- Visitor
- ...

```
class Employee
{
        public var name : String;
        public var birth : Date;
}

interface IChef
{
        function cookFood() : Food;
}

interface
{
        fun
        fun
}

interface
{
        fun
}

class Chef
{
        pub
}

class Wait
{
        pub
        pub
}

class Manag
{
        pub
```

Part II Interfaces



Design patterns can be useful as implementation, but not as a user mental model and architecture. However, there are patterns that are used to align a user mental model with the computer, like MVC.

```
class Employee
{
	public var name : String;
	public var birth : Date;
}

interface IChefService
{
	function cookFood(Employee e) : Food;
}

class ChefService implements IChefService
{
	public Food cookFood(Employee e) { … }
}


// The program
static function main()
{
	var chefService = Services.Get(IChefService);
	var chef = Employee.Get(2);

	chefService.cookFood(chef);
}
```

Part III Services

Object model mismatch. We move from “who does what” to “what happens”. Too imperative and all-knowing.

Information is abstracted away through interfaces.

Part IV
DCI

Data, Context, Interaction

Inventors of DCI

Trygve Reenskaug

Inventor of MVC
Wrote CAD-software in the 60's
Professor Emeritus, Oslo

James "Cope" Coplien

Author of several influential C++ books
Created most cited patent in software history
Chair CS 2003-2004, Brussels
Patterns, Scrum, Agile, Lean...

Data

```
class Employee
{
    public var name : String;
    public var birth : Date;
}
```

Function

- Chef - Cook food
- Waiter - Serve food, bring the bill
- Manager - Pay salary

Data

```
class Employee
{
    public var name : String;
    public var birth : Date;
}
```

Function

Chef — `public function cookFood()`

Waiter — `public function serveFood()`

Waiter — `public function bringBill()`

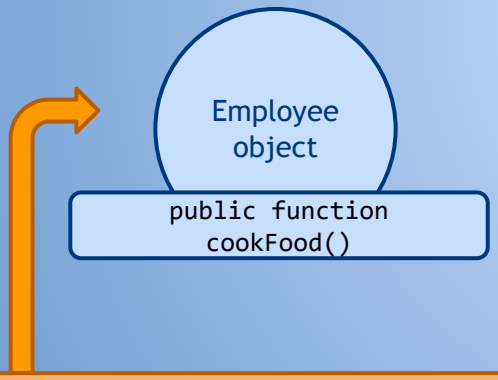
Manager — `public function paySalaries()`

If an Employee object could attach the cookFood() method... then it could be a Chef!

We cannot do it in classes (we tried in part **I–III**), so it seems like it must be a more dynamic operation.

Data

```
class Employee
{
    public var name : String;
    public var birth : Date;
}
```



Function

Chef	`public function cookFood()`
Waiter	`public function serveFood()`
Waiter	`public function bringBill()`
Manager	`public function paySalaries()`

This Employee **object** can now play the Role of a Chef!

Data

```
class Employee
{
    public var name : String;
    public var birth : Date;
}
```

Function

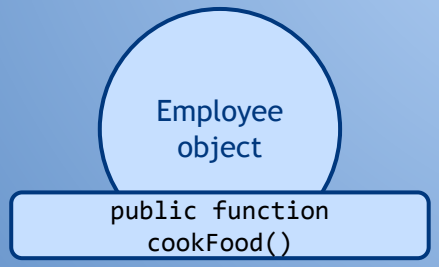
Chef — `public function cookFood()`

Waiter — `public function serveFood()`

Waiter — `public function bringBill()`

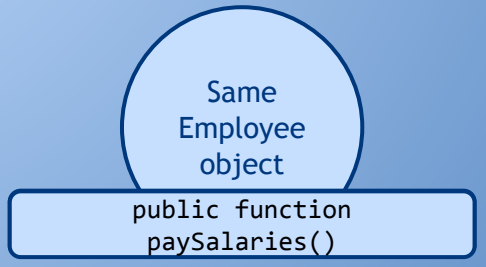
Manager — `public function paySalaries()`

Now *a Chef*



… a little later in the program…

Later *a Manager*



Roles and Contexts

A Role is only relevant in a specific Context.

Chef.cookFood()

vs.

Chef.cookFood()

A Mental Model can be represented as a Context!

```
// Data (Form)
class Employee
{
      public var name : String;
      public var birth : Date;
}
```

Part IV
DCI

The Context (Mental Model) of serving guests at a Restaurant:

"The waiter walks up to the guests and shows them the menu. He takes the order and passes it on to the chef. The chef cooks the food, which is brought out to the guests by the waiter.

After the meal the waiter brings the bill to the guests, and collects the payment."

```
// Data (Form)
class Employee
{
      public var name : String;
      public var birth : Date;
}
```

Part IV
DCI

Remember from part I-III:

The methods cannot go in the *data* class, they should be attached by some means.

Where to put the methods, the *functionality*?

```
// Data (Form)
class Employee
{
      public var name : String;
      public var birth : Date;
}

// Context (Mental Model)
class ServeGuests implements haxedci.Context
{
      @role var waiter = {}
      @role var guests = {}
      @role var chef = {}
}
```

Part IV
DCI

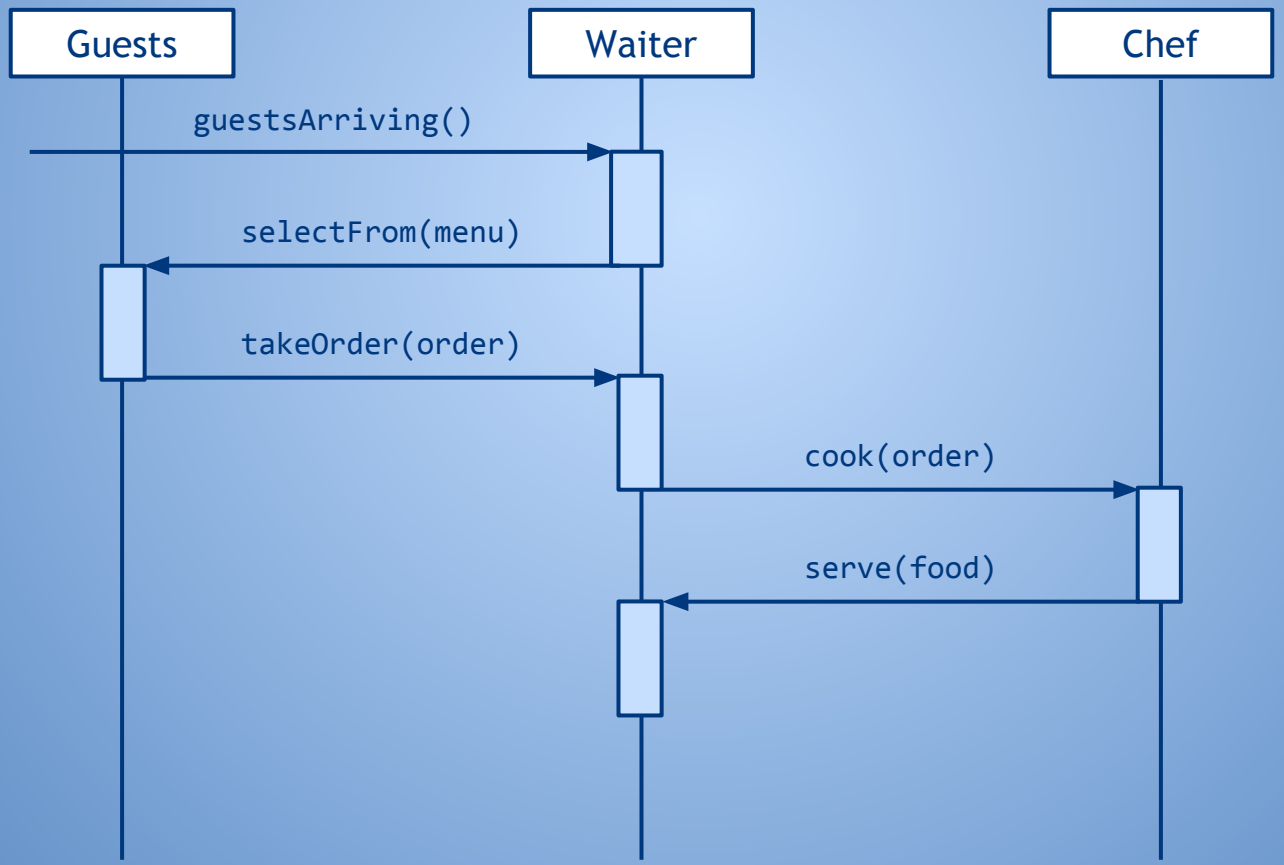
The Context (Mental Model) of serving guests at a Restaurant:

"The waiter walks up to the guests and shows them the menu. He takes the order and passes it on to the chef. The chef cooks the food, which is brought out to the guests by the waiter.

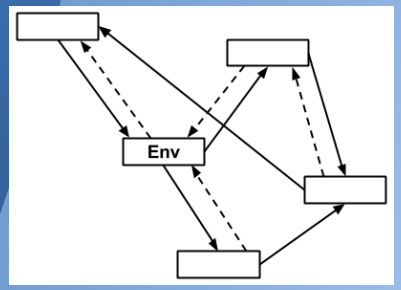
After the meal the waiter brings the bill to the guests, and collects the payment."

Part IV
DCI

"The waiter walks up to the guests and shows them the menu. He takes the order and passes it on to the chef. The chef cooks the food, which is brought out to the guests by the waiter.

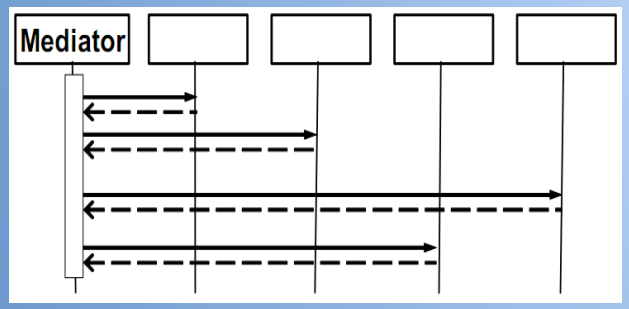


Part IV
DCI



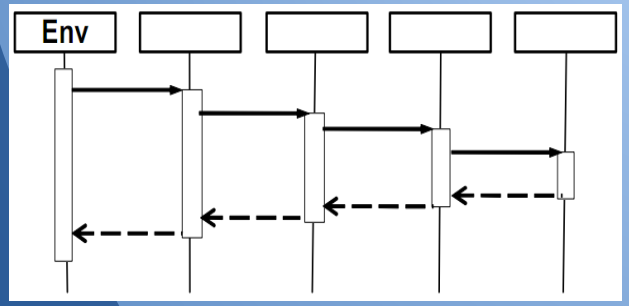
The current “OO” way of creating object communication: Passing messages between objects wherever and whenever it is needed.

- Too chaotic
- No separation of data and behavior
- Can only see one object at a time
- Reasoning about system behavior and functionality is very hard.



Procedural thinking, implemented as the Mediator pattern.

- Prevents object communication
- Creates a complex all-knowing algorithm
- Basically a step back to Pascal and Fortran.

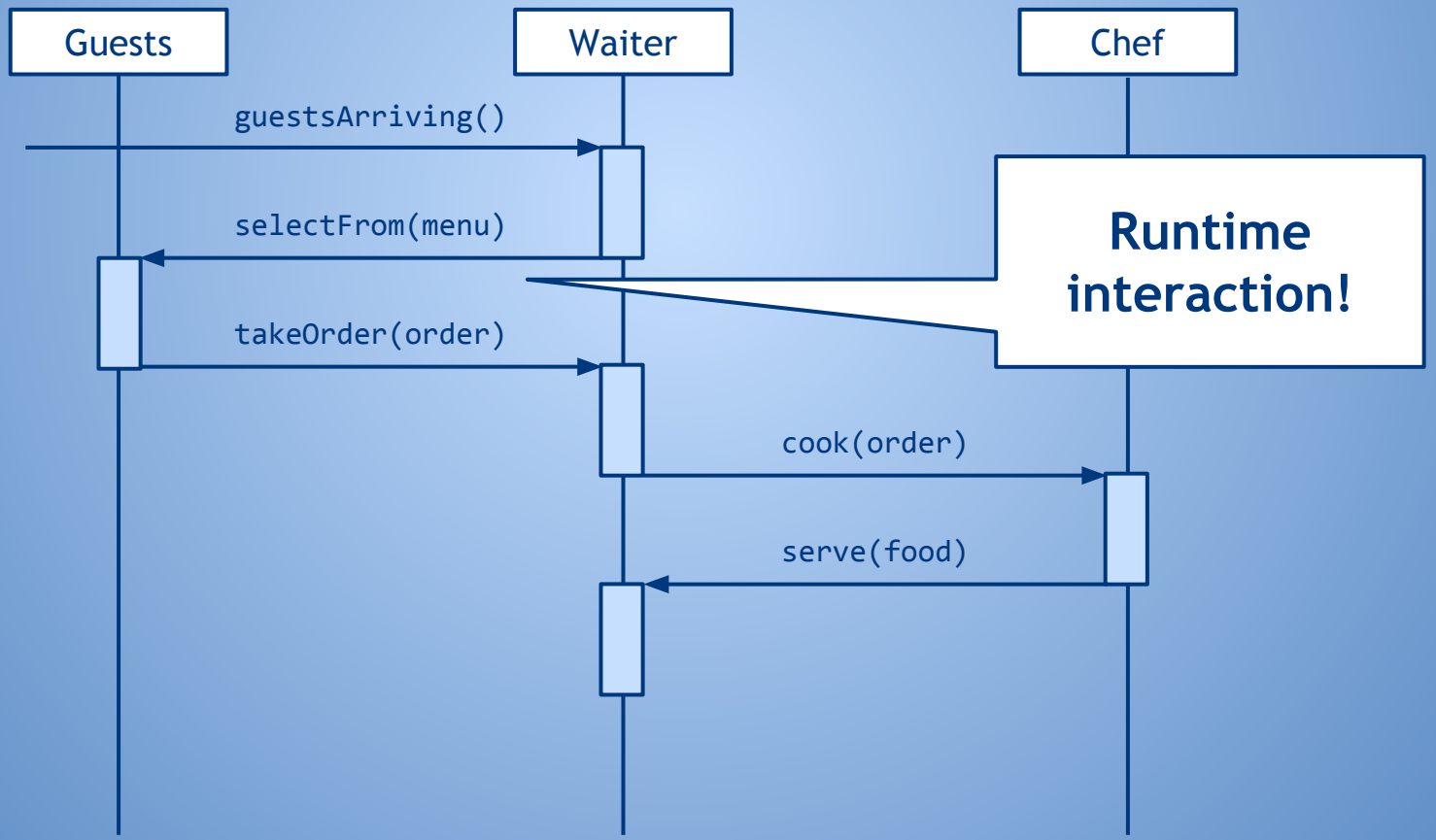


DCI: A middle road, distributed logic under full control, encapsulated in a Context

- State and behavior is distributed among the objects
- Creates a network of collaborating objects
- **We can reason about runtime system behavior!**

Part IV
DCI

"The waiter walks up to the guests and shows them the menu. He takes the order and passes it on to the chef. The chef cooks the food, which is brought out to the guests by the waiter.



```
// Data (Form)
class Employee
{
      public var name : String;
      public var birth : Date;
}

// Context (Mental Model)
class ServeGuests implements haxedci.Context
{
      @role var waiter = {}
      @role var guests = {}
      @role var chef = {}

      public function new(employeeWaiter,
                          employeeChef,
                          guestList)
      {
            this.waiter = employeeWaiter;
            this.chef = employeeChef;
            this.guests = guestList;
      }

      // System operation (Event)
      public function guestsArriving()
      {
            waiter.guestsArriving();
      }
}
```

Part IV
DCI

The Context (Mental Model) of serving guests at a Restaurant:

"The waiter walks up to the guests and shows them the menu. He takes the order and passes it on to the chef. The chef cooks the food, which is brought out to the guests by the waiter.

After the meal the waiter brings the bill to the guests, and collects the payment."

```
// Context (Mental Model)
class ServeGuests implements haxedci.Context
{
  @role var waiter =
  {
    function guestsArriving() : Void
    {
      var menu = “1: Roast beef. 2: ...”;
      guests.selectFrom(menu);
    }
  }
}
```

Part IV
DCI

The Context (Mental Model) of serving guests at a Restaurant:

"The waiter walks up to the guests and shows them the menu. He takes the order and passes it on to the chef. The chef cooks the food, which is brought out to the guests by the waiter.

After the meal the waiter brings the bill to the guests, and collects the payment."

```
// Context (Mental Model)
class ServeGuests implements haxedci.Context
{
  @role var waiter =
  {
    function guestsArriving() : Void
    {
      var menu = “1: Roast beef. 2: ...”;
      guests.selectFrom(menu);
    }
  }

  @role var guests =
  {
    function selectFrom(menu) : Void
    {
      Sys.println(menu);
      order = Sys.stdin().readLine();
      waiter.takeOrder(order);
    }
  }
}
```

Part IV
DCI

The Context (Mental Model) of serving guests at a Restaurant:

"The waiter walks up to the guests and shows them the menu. He takes the order and passes it on to the chef. The chef cooks the food, which is brought out to the guests by the waiter.

After the meal the waiter brings the bill to the guests, and collects the payment."

```
// Context (Mental Model)
class ServeGuests implements haxedci.Context
{
  @role var waiter =
  {
    function guestsArriving() : Void
    {
      var menu = “1: Roast beef. 2: ...”;
      guests.selectFrom(menu);
    }

    function takeOrder(order) : Void
    {
      Sys.println("Thank you.");
      chef.cook(order);
    }
  }

  @role var guests =
  {
    function selectFrom(menu) : Void
    {
      Sys.println(menu);
      var order = Sys.stdin().readLine();
      waiter.takeOrder(order);
    }
  }
}
```

Part IV
DCI

The Context (Mental Model) of serving guests at a Restaurant:

"The waiter walks up to the guests and shows them the menu. He takes the order and passes it on to the chef. The chef cooks the food, which is brought out to the guests by the waiter.

After the meal the waiter brings the bill to the guests, and collects the payment."

```
// Data (Form)
class Employee
{
	public var name : String;
	public var birth : Date;
}

// Context (Mental Model)
class ServeGuests implements haxedci.Context
{
	public function new(...)
	{
		this.waiter = employeeWaiter;
		this.guests = listGuests;
		this.chef = employeeChef;
	}
	// System operation (Event)
	public function guestsArriving()
	{
		waiter.guestsArriving();
	}
	// System operation (Event)
	public function guestsPaying()
	{
		waiter.bringBill();
	}
}
```

Part IV
DCI

The Context (Mental Model) of serving guests at a Restaurant:

"The waiter walks up to the guests and shows them the menu. He takes the order and passes it on to the chef. The chef cooks the food, which is brought out to the guests by the waiter.

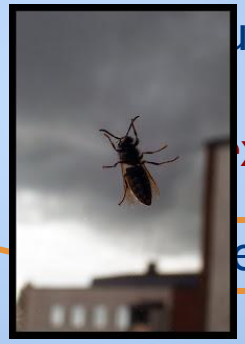
After the meal the waiter brings the bill to the guests, and collects the payment."

```
// Data (Form)
class Employee
{
      public var name : String;
      public var birth : Date;
}

// Role
@role var waiter =
{
      // RoleInterface
      var roleInterface : {
          var name : String;
      }

      // RoleMethods
      function guestsArriving() : Void
      {
            Sys.println(self.name);
            context.guests.selectFrom(menu);
      }
      ...
}
```

The Employee Data (Form) matches the RoleInterface, so it can play the waiter Role.

Part IV

A DCI Role has a RoleInterface and RoleMethods. The Role is private to its Context.



Form matches the RoleInterface for the Role “container” in the Context “Remove Insect”

If "self" in the role doesn't reference the object itself, it's not DCI.

```
// Data (Form)
class Employee
{
  public var name : String;
  public var birth : Date;
}

// Context (Mental model/Use case/Algorithm)
class ServeGuests implements haxedci.Context
{
  @role var waiter =
  {
    // RoleInterface (Compatible Form)
    var roleInterface : {
      var name : String;
    }

    // RoleMethods (Function)
    function guestsArriving() : Void
    {
      var menu = new List<String>();

      // Interaction (System behavior)
      guests.selectFrom(menu);
    }
  }

  // System operation (Events)
  public function guestsArriving() : Void
  {
    waiter.guestsArriving();
  }
}
```

So what is DCI?

A method of reflecting user mental models in code.

A way of separating what the system IS (**Data**, slowly changing) from what it DOES (**Function**, quickly changing).

Allows reasoning about the correctness of system functionality, not just class/method functionality. We can observe exactly what happens at runtime. No Polymorphism.

Behavior is put back within the boundaries of a human mental model, not spread across classes, services, patterns...

```
class Employee
{
  public var name : String;
  public var birth : Date;
}

class ServeGuests implements haxedci.Context
{
  @role var waiter =
  {
    var roleInterface : {
      var name : String;
    }

    function guestsArriving() : Void
    {
      var menu = new List<String>();
      guests.selectFrom(menu);
    }
  }

  public function guestsArriving() : Void
  {
    waiter.guestsArriving();
  }
}
```

So what is DCI?

A method of reflecting user mental models in code.

A way of separating what the system IS (**Data**, slowly changing) from what it DOES (**Function**, quickly changing).

Allows reasoning about the correctness of system functionality, not just class/method functionality. We can observe exactly what happens at runtime. No Polymorphism.

Behavior is put back within the boundaries of a human mental model, not spread across classes, services, patterns...

Questions?

DCI Headquarters:
http://fulloo.info

Google “haxedci”

Thank you!

andreas@ivento.se
github.com/ciscoheat
google “haxedci”