The Other Art of Computer Programming. Milestone 6: Swift.

2000s

Melanie Tarr

*Edith Cowan University*
Examine the 7 categories from planning and designing to prototyping, testing and coding

Learn about the product development life cycle

Look at programming paradigms

Memorise the Swift interface

Learn different design patterns that you can apply to app development

Swift

The Other Art of Computer Programming by Melanie Tarr

2000s

Ralph Johnson

John Vlissides

Erich Gamma

Grady Booch

John Vlissides

Swift

DOI 10.25958/5e3a777262fda
The “Gang of Four” wrote a seminal work detailing twenty-three software patterns. Model View Controller (MVC) is one of these. MVC is a frequently used iOS app design pattern.
# Contents

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Product Development Life cycle</td>
<td>3</td>
</tr>
<tr>
<td>Introduction</td>
<td>4</td>
</tr>
<tr>
<td>Programming Paradigms</td>
<td>5</td>
</tr>
<tr>
<td>Computable objects</td>
<td>6</td>
</tr>
<tr>
<td>Design brief</td>
<td>7</td>
</tr>
<tr>
<td>Brain Storming</td>
<td>8</td>
</tr>
<tr>
<td>Planning</td>
<td>9</td>
</tr>
<tr>
<td>Computable objects</td>
<td>9</td>
</tr>
<tr>
<td>Information architecture</td>
<td>10</td>
</tr>
<tr>
<td>Inheritance</td>
<td>11</td>
</tr>
<tr>
<td>Interface input</td>
<td>12</td>
</tr>
<tr>
<td>Interface cues and gesture</td>
<td>13</td>
</tr>
<tr>
<td>Inheritance blueprint</td>
<td>14</td>
</tr>
<tr>
<td>Classes in OOP</td>
<td>15</td>
</tr>
<tr>
<td>Model View Controller</td>
<td>16</td>
</tr>
<tr>
<td>Isometric diagram of MVC</td>
<td>17</td>
</tr>
<tr>
<td>Begin the application</td>
<td>18</td>
</tr>
<tr>
<td>Coding the transport class</td>
<td>19</td>
</tr>
<tr>
<td>Classes replicate through inheritance</td>
<td>20</td>
</tr>
<tr>
<td>Properties tell the what</td>
<td>21</td>
</tr>
<tr>
<td>Functions tell the how</td>
<td>22</td>
</tr>
<tr>
<td>Designing the car class</td>
<td>23</td>
</tr>
<tr>
<td>Debugging</td>
<td>24</td>
</tr>
<tr>
<td>Smalltalk example - MVC</td>
<td>26</td>
</tr>
<tr>
<td>Back to COBOL</td>
<td>27</td>
</tr>
<tr>
<td>Swift class types</td>
<td>28</td>
</tr>
<tr>
<td>Coding inheritance</td>
<td>29</td>
</tr>
<tr>
<td>Coding inheritance</td>
<td>30</td>
</tr>
<tr>
<td>Coding inheritance</td>
<td>31</td>
</tr>
<tr>
<td>Overriding functions</td>
<td>32</td>
</tr>
<tr>
<td>Overriding functions</td>
<td>33</td>
</tr>
<tr>
<td>Coding inheritance</td>
<td>34</td>
</tr>
<tr>
<td>Model View Controller</td>
<td>35</td>
</tr>
<tr>
<td>Data structure of the model</td>
<td>36</td>
</tr>
<tr>
<td>Entering the data</td>
<td>37</td>
</tr>
<tr>
<td>Setting and testing the model</td>
<td>38</td>
</tr>
<tr>
<td>Computed Properties</td>
<td>39</td>
</tr>
<tr>
<td>Testing the array or list</td>
<td>40</td>
</tr>
<tr>
<td>Testing the list or array selection</td>
<td>41</td>
</tr>
<tr>
<td>Connecting model and view</td>
<td>42</td>
</tr>
<tr>
<td>Expanding the abstraction of model and view</td>
<td>44</td>
</tr>
<tr>
<td>Cont' expanding the model and view</td>
<td>45</td>
</tr>
<tr>
<td>Subclasses in inheritance</td>
<td>46</td>
</tr>
<tr>
<td>Subclasses in inheritance</td>
<td>47</td>
</tr>
<tr>
<td>Connecting model and view</td>
<td>47</td>
</tr>
<tr>
<td>Use inheritance to</td>
<td>48</td>
</tr>
<tr>
<td>create a new bike class</td>
<td>48</td>
</tr>
<tr>
<td>Create a new bike class</td>
<td>50</td>
</tr>
<tr>
<td>Override the superclass</td>
<td>51</td>
</tr>
<tr>
<td>Test the bikes</td>
<td>53</td>
</tr>
<tr>
<td>Use inheritance to create a new truck class</td>
<td>53</td>
</tr>
<tr>
<td>Logic stays in the model</td>
<td>55</td>
</tr>
<tr>
<td>Logic and model</td>
<td>56</td>
</tr>
<tr>
<td>Customise the truck class</td>
<td>57</td>
</tr>
<tr>
<td>Customise the truck class</td>
<td>58</td>
</tr>
<tr>
<td>Add more truck classes</td>
<td>59</td>
</tr>
<tr>
<td>Test the truck classes</td>
<td>60</td>
</tr>
</tbody>
</table>

Colors:
- **Design**
- **Programming**
- **Interface description**
- **Testing**
The Product Development Life cycle

Planning concerns listening to the client on what their needs are for the product. Empathy for the user is a trait that helps in this stage of the cycle.

Creating a working prototype helps to collect further requirements from the stakeholders and also enables them to comprehend the system. It also helps in reassuring ideas and defining go to market (GTM) feasibility.

User testing within the cycle concerns feedback from the users and continuous debugging throughout.

Coding of the software application almost always concerns the developer and is a continual process that commences after the applications design.

The idea stage is the initial conception of requirements based upon the what the client wants. Tools: such as sketches, paper, pens etc.

Designing an application is an ongoing process that aligns with the users of the application’s requirements.
This lesson steps through the product development process to produce a software application about different types of transportation.
Programming languages belong to specific paradigms. So far, the following paradigms have been used in The Other Art of Computer Programming (TOAACP).

<table>
<thead>
<tr>
<th>Milestone</th>
<th>Paradigm</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>1950s</td>
<td>Machine code</td>
<td>Assembly</td>
</tr>
<tr>
<td>1960s</td>
<td>Functional</td>
<td>Logo</td>
</tr>
<tr>
<td>1970s</td>
<td>Imperative</td>
<td>COBOL, Fortran</td>
</tr>
<tr>
<td>1980s</td>
<td>OOP</td>
<td>Smalltalk</td>
</tr>
<tr>
<td>1990s</td>
<td>Declarative</td>
<td>HTML, Ruby, SQL</td>
</tr>
<tr>
<td>2000s</td>
<td>OOP</td>
<td>Swift</td>
</tr>
</tbody>
</table>
Object-oriented programming (OOP) is another programming paradigm based on a microworld.

In OOP, a real-life metaphor is programmed through its virtual object. The computable object is within the cube.
The design brief is to make an application that lists various types of transport (1). The transport can be anything that transports a client from one place to another. The app will list different types of transport and features that occur for each vehicle (2).
The features of different types of transport to be represented on the app are researched through design processes.

An initial brainstorm with the client for the app reveals they would like the following transport types included as choices within the app. These transport types are Porsche, truck, bike, mini, Tesla and convertible Porsche.

An affinity diagram is often used to gather and organise user requirements about a product. It is an outcome of brainstorming.
A meeting after the brainstorming session revealed that the client wanted the following features displayed on the app. These features were the number of wheels, engine type, whether or not the transport was a convertible as well as a host of activities the transport could do.

<table>
<thead>
<tr>
<th>properties</th>
<th>functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>model</td>
<td>sunroof</td>
</tr>
<tr>
<td>year</td>
<td>hatchback</td>
</tr>
<tr>
<td>power</td>
<td>convertible</td>
</tr>
<tr>
<td>wheels</td>
<td>No of doors</td>
</tr>
</tbody>
</table>
The app’s information architecture is mapped out in the affinity diagram below.

Information architecture (IA) is the structural design of shared information environments. IA brings principles of design and architecture to the digital landscape. Each characteristic will have a specific of storage in memory.
Now that we have our basic list of objects the coding can begin.

The first virtual object within the program is a virtual transport object. The object represents all common characteristics within the different types of transport. The transport object is the parent object in the program or the superclass. Objects are replicated through a process called inheritance.
There are three ways to enter information on an XCode interface.

- **Form**
- **Gesture**
- **String**
The three ways to enter information can be classified through gesture.

- **GESTURAL GROSS MOTOR**: Mouse interaction is a gross motor action. The action on the interface occurs by clicking on icons and menu commands.

- **SHORT TERM FINE MOTOR**: Gross motor and typing form input make up the short-term fine motor group of commands. These actions combine menu interface interaction and form input.

- **SUSTAINED FINE MOTOR**: Typing and coding is a sustained fine motor action. This ‘coding’ occurs in the editor section of the interface. The recommended maximum time of coding is fifteen minutes long.
A blueprint is used to create copies of plans like a class. In object orientated programming, once a class is created more objects can be created as instances of the class.

To create a car class, a class of a transport object is constructed called car. As many class copies are produced as required to represent different types of transport. Car is the first class we are creating.
Model View Controller (MVC)

The model for the transport object has now been programmed. However, the app is far from finished. The view and the controller parts of the design pattern have not been started.

MVC is a popular design pattern used in mobile applications. The pattern consists of the programming for the model, the view, and the controller.
Model View Controller (MVC)

The exploded isometric figure represents the app to be programmed. The views are indicated with the view icon, the model with the model icon and the areas the controller operates with the controller icon.
Open Swift and enter the generic transport class into the file transport.swift.

Click on Swift and open a file called transport.swift.

Click Swift
create a new file

click Swift

click Swift

Click on Swift and open a file called transport.swift

var modelName = "null"
var modelYear = 0
var powerSource = "null"
var wheelNumber = 0
The design features of the object can be directly entered into the file transport.swift now as variables and properties. The variables (var) are the primary features in the affinity diagram and the functions (func) are the secondary features and represent the action that takes place.

```swift
class transport {
    var modelName = "null"
    var modelYear = 0
    var powerSource = "null"
    var wheelNumber = 0

    func forward() -> String {
        return "null"
    }

    func backward() -> String {
        return "null"
    }

    func stop() -> String {
        return "null"
    }

    func turn(degrees: Int) -> String {
        var normalizedDegrees = degrees
        let degreesInACircle = 360
        if (normalizedDegrees > degreesInACircle || normalizedDegrees < -degreesInACircle) {
            normalizedDegrees = normalizedDegrees % degreesInACircle
        }
        return String(format: "Turn %d degrees.", normalizedDegrees)
    }

    func gearChange(newGearName: String) -> String {
        return String(format: "Put %@ into %@ gear.", selfmodelName, newGearName)
    }

    func honkHorn() -> String {
        return "null"
    }
}
```
Classes replicate in object orientated programming through a process called inheritance. Every class below the superclass inherits the variables and properties of the class above it.
The properties of the transport class explain what the vehicle looks like.
What does transport do?

Within this application design, regardless of what the transportation looks like (properties), all transport moves people from A to B, has gears, turns corners and makes a noise returned through a function.
Designing the car class
(data structure)

The car class has been designed to inherit properties and features of the transport class.
By entering the following code into the file AppDelegate.swift you can check the execution of your program. So far if the returns null values and an integer value for the turning degrees, your code is working.

```
func application(application: UIApplication, didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
    var transport = Transport()
    println("Transport turn: ",(transport.turn(700)))
    var changeGearResult = transport.gearChange("Test")
    println("Transport change gears: ",(changeGearResult))
    println("Transport make noise: ",(transport.honkHorn()))
    println("Transport go forward: ",(transport.forward()))
    println("Transport go backward: ",(transport.backward()))
    println("Transport stop moving: ",(transport.stop()))
    return true
}
```
What is a design pattern?

Each pattern describes a problem that happens over and over again in our environment. This also applies to object orientated design.
A visual Smalltalk example of MVC

Design patterns were invented by Christopher Alexander and in general are made up of four elements that are:
- The pattern name
- The problem
- The solution
- The consequences

The purpose of a design pattern is either creational, structural or behavioural. The patterns vary in their level of abstraction and granularity. For example, the singleton pattern has a creational purpose while the adaptor and decorator pattern have a structural purpose.
When the program compiles into source code the header file, and implementation file are appended to one long list in Swift.

Compiling an iOS application takes seconds as opposed to the three days a program took Grace Hopper to compile. Today the technology is much faster although the coding techniques have changed far less.

The `println` statement is still used today in debugging software.
To code the design for inheritance, we first create a new subclass of the transport class. This new car subclass is coded in a new file called `car.swift`.
Coding inheritance

The superclass contains most of the properties and functions you need for the car without adding any more code. The default the car class has inherited from transport are enough. The number of wheels on the average car is four, so the first thing that needs to be coded is the number of wheels. The superclass is overridden in the `car.swift` file, and the wheel number is set to four.

```swift
class Car : Transport {
    override init() {
        super.init()
        wheelNumber = 4
    }
}
```
Coding inheritance

If we want a third set of properties to describe the car, they can easily be added to the car class. These properties added to the car class will now be available to all car objects in addition to the properties inherited from the transport class.

```
car.swift
var isConvertible: Bool = false
var isHatchback: Bool = false
var hasSunroof: Bool = false
var numberOfDoors: Int = 0
```
The properties for the car subclass are now complete. However, the methods for the superclass must be overridden to provide their full implementations.

When you override a superclass, you create a custom implementation for that class. A call to the super method inherits everything in the object above.

In `car.swift`, override the superclass with the following code.

```swift
private func start() -> String {
    return String(format: "Start power source %@.", powerSource)
}
```
Overriding functions

The properties for the car subclass are now complete. However, the methods for the superclass must be overridden to provide full implementations.

```swift
override func forward() -> String {
    return String(format: "%@ %@ Depress gas pedal.", start(), gearChange("Forward"))
}

override func backward() -> String {
    return String(format: "%@ %@ Look backwards, then depress gas pedal.", start(), gearChange("Reverse"))
}

override func stop() -> String {
    return String(format: "Depress brake pedal. %@", gearChange("Park"))
}

override func honkHorn() -> String {
    return "Beep beep!"
}
```
Coding inheritance

The superclass contains most of the properties and functions you need for the car without adding any more code. The default the car class has inherited from transport are enough. The number of wheels on the average car is four, so the first thing that needs to be coded is the number of wheels. The superclass is overridden in the `car.swift` file, and the wheel number is set to four.

```swift
class Car : Transport {
    override init() {
        super.init()
        wheelNumber = 4
    }
}
```
The Porsche instance of the car class inherits properties and features to ensure that less code is written for the Porsche instance to be complete.

The properties and features of the porsche are stores in an array data structure within the computer's memory.

The array structure of the model component within the Model View Controller (MVC) design pattern.
Model View Controller (MVC)

To make programming manageable the Gang of Four set out to categorise 23 patterns used commonly in computer programming.

A design pattern

The programming for each component within the design pattern, can be identified and separated.
The data structure used in the application is an array for each transport type that contains the transport’s properties.
Getting data into the model

The model array is first created in the memory space of the computer (1). Then another array is created that holds all the data for the mini's properties. The car object or array is called “mini” (2). Finally, the mini array is stored within the transport array. The data structure is now a two dimensional array (3). The array was drawn on the previous page.

```swift
func setupTransportArray() {
    // 1
    transport.removeAll(keepCapacity: true)

    // 2
    var mini = Car()
    minimodelName = "X200"
    miniModelYear = 2005
    mini.isConvertible = true
    mini.isHatchback = false
    mini.hasSunroof = false
    mini.numberOfDoors = 2
    mini.powerSource = "gas engine"

    // 3
    transports.append(mini)
}
```
Setting and testing the model

TransportListTableViewController.swift

The array is appended as more cars are added to it.

```swift
setupTransportArray()
title = "Transport"
```

Building and running the application will show the instances of transport in the array.
Computed properties

A stored property is a variable stored within a class or structure. The “var” keyword signals a stored value. The value has a default assigned when the class is first initialised. Computed properties are read-only properties and generate a new string each time the code is run inside the function `tableView` below.

```swift
override func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCellWithIdentifier("Cell", forIndexPath: indexPath) as! UITableViewCell

    let transport = transport[indexPath.row] as Transport
    cell.textLabel?.text = transport.transportTitle
    return cell
}
```
Testing the list or array selection

Building and running the application will show the detailed instances of transport in the array.
Testing the list selection

Selecting a car will not return any details about the transport as the model and the view are not connected.
Connecting model and view

Since you have declared the view array as a transport object, now pipe the data from the model (basicDetails) into the view (detailDescriptionLabel) through a segue. The segue connects the two parts in the pattern that are model and view, for the data to flow through.

func configureView() {
    if let transport = detailTransport {
        title = transport.transportTitle

        var basicDetails = "Basic transport details:
        basicDetails += "Model name: \(transport.modelName)\n        basicDetails += "Model year: \(transport.modelYear)\n        basicDetails += "Power source: \(transport.powerSource)\n        basicDetails += "# of wheels: \(transport.numberOfWheels)\n
        detailDescriptionLabel?.text = basicDetails
    }
}
Expanding the abstraction of model and view

Since you have declared the view array as a

What is happening here?
Assigning the model variable to the view variable.

The model variable points to an array containing the properties of the object.

`basicDetails`

The view variable is assigned to the array containing the properties of the object.

`detailDescriptionLabel?.text`

```javascript

```
Connecting model and view

To add more specific details about the car add the following lines to the function configureView:

```
func configureView() {
...
    basicDetails += "\n\nCar-Specific Details:\n\n"
    basicDetails += "Number of doors: $(transport.numberOfDoors)"
}
```
To keep the view controller unaware of specific details in the subclasses we create a details string that returns the details of the transport class.

```swift
var transportDetails: String {
    var details = "Basic transport details:\n\n" + "Model name: \(modelName)\n" + "Model year: \(modelYear)\n" + "Power source: \(powerSource)\n" + "# of wheels: \(numberOfWheels)\n" + "\n";
    return details
}
```
Subclasses in inheritance

By adding the transport string and the car class secondary properties together. You can retrieve the data you need for the detailed view on that type of transport. The data is just a string of appended properties. If the properties exist (yes) then the string is appended to the carDetails for the view.

```car.swift
override var transportDetails: String {
    let basicDetails = super.transportDetails
    var carDetailsBuilder = "\n\nCar-Specific Details:\n\n"
    let yes = "Yes\n"
    let no = "No\n"
    carDetailsBuilder += "Has sunroof: "
    carDetailsBuilder += hasSunroof ? yes : no
    carDetailsBuilder += "Is Hatchback: "
    carDetailsBuilder += isHatchback ? yes : no
    carDetailsBuilder += "Is Convertible: "
    carDetailsBuilder += isConvertible ? yes : no
    carDetailsBuilder += "Number of doors: \(numberOfDoors)"
    return carDetailsBuilder
}
```
Connecting model and view

To add more specific details about the car add the following lines to the function `configureView`

```
func configureView() {
    ...
    basicDetails += "\n\nCar–Specific Details:\n\n"
    basicDetails += "Number of doors: \(transport.numberOfDoors)"
}
```
Use inheritance to create a new bike class

The application has been designed to use inheritance, so very little code needs to be added to create a new virtual bike class.

The **subclass** bike is derived from the **superclass** transport.
Create a new bike class

Create a new bike class with features to view on your app.

```swift
class Bike: Transport {
    override init() {
        super.init()
        wheelNumber = 2
        powerSource = "none"
    }
}
```
Create a new bike class with properties to view on your app.

```swift
Bike.swift

override func Forward() -> String {
    return String(format: "%@ Pedal forward",
                   gearChange("Forward"))
}

override func Backward() -> String {
    return String(format: "%@ Walk %@ backwards using feet.",
                     modelName)
}

override func stop() -> String {
    return "Squeeze brakes"
}

override func beep() -> String {
    return self.engineNoise
}
```
Add more bikes

Add more bikes to the data structure array that have different types. First, create the new bike (1), then add it to the array (2) and repeat the process for another bike (3 and 4).

```swift
var shimano = Bike()
shimano.modelName = "Softail"
shimano.modelYear = 2010
shimano.engineNoise = "None"

transport.append(shimano)

var bmx = Bike()
bmx.modelName = "Curse 20"
bmx.modelYear = 2005
bxm.engineNoise = "None"

self.transport.append(bmx)
```
Test the bikes

By running the app, you can see the two new classes in the list.

Select transport to see the features

By selecting one of the bikes the attributes and properties come up.

Model: Curse 20
Year: 2005
PowerSource: None
No. of wheels: 2
Use inheritance to create a new truck class. The unique attributes of trucks are:

- their capacity to carry a large weight;
- make different sounds according to truck type;
- they trigger an alarm when reversing.
Logic stays in the Model

Create a new truck class with features to view on your app.

```swift
class Truck: Transport {
    var cargoCapacity: Int = 0

    override func Forward() -> String {
        return String(format: "%@ Press accelerator.", gearChange("Drive"))
    }

    override func stop() -> String {
        return String(format: "Press brakes. %@", gearChange("Park"))
    }
}
```
Logic and Model

Customise the different types of trucks by creating an alarm when the truck reverses. The alarm will only sound if the trucks capacity is over 150 units.

```swift
private func Alarm() -> String {
    return "Beep! Beep! Beep! Beep!"
}
```

```swift
override func Backward() -> String {
    if capacity > 150 {
        return String(format:"Wait for \"%@\",
                 then %@", soundBackupAlarm(),
                 gearChange("Reverse"))
    } else {
        return String(format:"%@ Depress gas pedal.", gearChange("Reverse"))
    }
}
```
The case statement further customises the class. When there are different types of trucks, the software queries what type of truck class has sounded. If a large truck has been called, then a large sounding horn will be matched to the class.

```swift
override func beep() -> String {
    switch wheelNumber{
    case Int.min...4:
        return "Beep!"
    case 5...8:
        return "Honk!"
    default:
        return "LOUDER HONK"
    }
}
```
The truck class has the basic transport details, and then an extra property called capacity is added. The new class is then returned to the program with all details included.

Customise the truck class

**Truck.swift**

```swift
override var transportDetails: String {
    let basicDetails = super.transportDetails
    var newtruckDetails = "\n\nTruck-Specific Details: \n\n"
    newtruckDetails += "Cargo Capacity: \(cargoCapacityCubicFeet) cubic feet"
    let truckDetails = basicDetails + newtruckDetails
    return truckDetails
}
```
Add more trucks to the app that have different types. First, create the new truck (1), then add it to the array (2).

```swift
// 1
var dump = Truck()
dump.model = "Silverado"
dump.Year = 2011
dump.wheelNumber= 4
dump.Capacity = 53
dump.powerSource = "gas engine"

// 2
transport.append(dump)

// 3
var garbage = Truck()
garbage.model = "Chevrolet"
garbage.Year = 2013
garbage.wheelNumber= 10
garbage.Capacity = 408
garbage.powerSource = "diesel engine"

// 4
transport.append(garbage)
```
Test the truck classes

By running the app you can see the two new objects in the list.

By selecting one of the trucks the details come up

Model: Silverado
Year: 2011
PowerSource: gas
No. of wheels: 4
Capacity: 53