



UNIVERSIDADE DA CORUÑA

FACULTADE DE INFORMÁTICA

TRABALLO FIN DE GRAO  
GRAO EN ENXEÑARÍA INFORMÁTICA

Mención en Computación

**Desenvolvemento de módulos para robótica  
educativa en App Inventor**

**Autor:** Vilar Rodríguez, Miguel Ángel

**Tutor:** Bellas Bouza, Francisco Javier

**Directores:** Varela Fernández, Gervasio

Paz López, Alejandro

A Coruña, 7 de setembro do 2017



## Resumo

A metodoloxía didáctica STEM (do inglés Science, Technology, Engineering and Mathematics), baséase no uso da tecnoloxía como catalizador da formación en destrezas básicas como autonomía, resolución de problemas, traballo en equipo, proactividade, etc. É unha metodoloxía que explota a ensinanza baseada en proxectos, onde se aprende facendo. Unha das ferramentas básicas de STEM é a robótica, xa que facilita a aprendizaxe no mundo real mediante unha plataforma atractiva para os alumnos, que integra conceptos de diferentes materias como programación, matemáticas ou física. Un exemplo neste sentido é o robot ROBOBO, desenvolvido e comercializado por unha spin-off da UDC chamada MINT (Manufactura de Ingenios Tecnológicos), que se basea na utilización dun smartphone sobre unha base robótica móvil. Este robot permite que os estudiantes empreguen o seu móvil nas clases como ferramenta de traballo. Por outra banda, unha das ferramentas software más potentes para que usuarios non expertos creen aplicacións móbiles Android é App Inventor, gracias á súa linguaxe visual por bloques moi intuitiva.

Actualmente o ROBOBO permite o uso de Scratch e ROS, ademais de contar cun framework que facilita o desenvolvemento de aplicacións, e dun conxunto de módulos que dan acceso a tódalas súas funcionalidades. O obxectivo deste Traballo de Fin de Grao é incrementar as capacidades do robot, posibilitando e simplificando o desenvolvemento de aplicacións para o robot en Android, así como que usuarios sen experiencia anterior en programación poidan iniciarse neste mundo a través desta plataforma dunha maneira intuitiva e atractiva, beneficiarse do emprego da metodoloxía didáctica STEM e a ensinanza baseada en proxectos, e que ademais sexa fácil de estender con novas capacidades. Para isto realizouse un desenvolvemento en dúas fases dunha API simplificada adaptada a este contexto coa intención de que os estudiantes se poidan iniciar na robótica e na programación a través dela primeiro empregando a linguaxe visual de App Inventor, para despois continuar con Java.



# Índice xeral

<b>1. INTRODUCIÓN</b>	<b>1</b>
1.1. Motivación . . . . .	1
1.2. Obxectivos . . . . .	3
1.3. Estrutura da memoria . . . . .	4
1.4. Ferramentas empregadas . . . . .	5
<b>2. ANTECEDENTES</b>	<b>7</b>
2.1. Programación como ferramenta didáctica . . . . .	7
2.1.1. Logo . . . . .	8
2.1.2. AgentSheets . . . . .	8
2.1.3. Etoys . . . . .	9
2.1.4. Alice . . . . .	10
2.1.5. NetLogo . . . . .	11
2.1.6. Scratch . . . . .	11
2.1.7. App Inventor . . . . .	12
2.2. Robótica educativa . . . . .	12
2.2.1. Bee-Bot . . . . .	13
2.2.2. BQ Zowi . . . . .	13
2.2.3. LEGO . . . . .	14
2.2.4. Fischertechnik Robotics . . . . .	15
2.2.5. MakeBlock . . . . .	16
2.2.6. WowWee . . . . .	16
2.2.7. NAO . . . . .	17
2.3. Conclusións . . . . .	18
<b>3. FUNDAMENTOS TECNOLÓXICOS</b>	<b>19</b>
3.1. Robobo . . . . .	19

---

3.1.1.	ROB . . . . .	19
3.1.2.	ROBOBO! Framework . . . . .	21
3.1.3.	Módulos do Robobo . . . . .	22
3.1.3.1.	Control da base motorizada (ROB) . . . . .	22
3.1.3.2.	Emocións . . . . .	23
3.1.3.3.	Son . . . . .	23
3.1.3.4.	Control da pantalla táctil . . . . .	24
3.1.3.5.	Visión . . . . .	24
3.1.3.6.	Producción e recoñecemento da fala . . . . .	24
3.1.3.7.	Interacción por mensaxes . . . . .	24
3.1.3.8.	Compatibilidade con ROS . . . . .	24
3.2.	App Inventor . . . . .	25
3.3.	Android . . . . .	27
<b>4. DESENVOLVEMENTO</b>		<b>29</b>
4.1.	Introdución . . . . .	29
4.2.	Aproximación conceptual . . . . .	31
4.3.	Arquitectura . . . . .	33
4.3.1.	Análise crítico e xustificación . . . . .	33
4.3.2.	Aquitectura de alto nivel . . . . .	37
4.3.3.	Principais componentes da arquitectura . . . . .	39
4.3.3.1.	Compoñentes da API simple . . . . .	39
4.3.3.2.	Xestor de componentes . . . . .	43
4.3.3.3.	Comunicación cliente-servidor . . . . .	45
4.3.3.4.	Compoñentes de App Inventor . . . . .	49
4.4.	Deseño e implementación . . . . .	53
4.4.1.	Metodoloxía . . . . .	53
4.4.2.	Primeira iteración . . . . .	54
4.4.2.1.	Expresión corporal . . . . .	54
4.4.2.1.1.	Deseño funcional . . . . .	54
4.4.2.1.2.	Primeiro nivel de abstracción . . . . .	56
4.4.2.1.3.	Segundo nivel de abstracción . . . . .	59
4.4.2.2.	Demostración . . . . .	62
4.4.3.	Segunda iteración . . . . .	67
4.4.3.1.	Sentido da vista . . . . .	67
4.4.3.1.1.	Deseño funcional . . . . .	67

4.4.3.1.2.	Primeiro nivel de abstracción . . . . .	69
4.4.3.1.3.	Segundo nivel de abstracción . . . . .	71
4.4.3.2.	Expresión facial . . . . .	74
4.4.3.2.1.	Deseño funcional . . . . .	74
4.4.3.2.2.	Primeiro nivel de abstracción . . . . .	75
4.4.3.2.3.	Segundo nivel de abstracción . . . . .	76
4.4.3.3.	Sentido do tacto . . . . .	78
4.4.3.3.1.	Deseño funcional . . . . .	78
4.4.3.3.2.	Primeiro nivel de abstracción . . . . .	79
4.4.3.3.3.	Segundo nivel de abstracción . . . . .	81
4.4.3.4.	Demostración . . . . .	82
4.4.4.	Terceira iteración . . . . .	86
4.4.4.1.	Expresión sonora . . . . .	86
4.4.4.1.1.	Deseño funcional . . . . .	86
4.4.4.1.2.	Primeiro nivel de abstracción . . . . .	87
4.4.4.1.3.	Segundo nivel de abstracción . . . . .	89
4.4.4.2.	Percepción sonora . . . . .	92
4.4.4.2.1.	Deseño funcional . . . . .	92
4.4.4.2.2.	Primeiro nivel de abstracción . . . . .	93
4.4.4.2.3.	Segundo nivel de abstracción . . . . .	95
4.4.4.3.	Demostración . . . . .	96
4.4.5.	Cuarta iteración . . . . .	99
4.4.5.1.	Equilibriocepción . . . . .	99
4.4.5.1.1.	Deseño funcional . . . . .	99
4.4.5.1.2.	Primeiro nivel de abstracción . . . . .	100
4.4.5.1.3.	Segundo nivel de abstracción . . . . .	101
4.4.5.2.	Interocepción . . . . .	102
4.4.5.2.1.	Deseño funcional . . . . .	103
4.4.5.2.2.	Primeiro nivel de abstracción . . . . .	104
4.4.5.2.3.	Segundo nivel de abstracción . . . . .	104
4.4.5.3.	Demostración . . . . .	105
<b>5. CONCLUSIÓNS</b>		<b>109</b>
<b>A. Manual de usuario</b>		<b>113</b>
A.1. Instalación . . . . .		113

---

A.2. Java . . . . .	113
A.2.1. Resolución de dependencias . . . . .	113
A.2.2. Obtención dos componentes . . . . .	114
A.3. App Inventor . . . . .	115
A.3.1. Importar a extensión . . . . .	115
A.3.2. Engadir os componentes . . . . .	116
A.3.3. Uso dos bloques . . . . .	117
<b>Bibliografía</b>	<b>119</b>

# Índice de figuras

1.1.	ROBOBO na súa última versión. . . . .	2
2.1.	Agent Sheets. . . . .	9
2.2.	Alice. . . . .	10
2.3.	Scratch. . . . .	12
2.5.	BQ Zowi. . . . .	14
2.7.	Fischertechnik. . . . .	15
2.8.	MakeBlock. . . . .	16
2.10.	Robot NAO. . . . .	17
3.1.	ROB. . . . .	20
3.2.	Esquema de situación dos actuadores e perceptores. . . . .	21
3.3.	Interfaz IModule. . . . .	22
3.4.	IRob interface para o control do ROB. . . . .	23
3.5.	Entorno de traballo de App Inventor. . . . .	26
4.1.	Deseño conceptual. . . . .	32
4.2.	Diagrama dos niveis de abstracción. . . . .	35
4.3.	Diagrama de disposición física dos elementos da API simple. . . . .	38
4.4.	Diagrama de clases do cliente e servidor para o compoñente de expresión corporal. . . . .	40
4.5.	Diagrama básico da comunicación entre compoñentes e handlers. . . . .	41
4.6.	Clase AActionHandler. . . . .	42
4.7.	Clase ASimpleAPIComponent. . . . .	42
4.8.	Interface IComponentListener para a notificación do estado dun compoñente. . . . .	43
4.9.	Diagrama UML do xestor de compoñentes. . . . .	44

4.10. Interface ISimpleRoboboManagerListener para a notificación do estado dos componentes. . . . .	44
4.11. Servizo: RoboboSimpleAPIService. . . . .	47
4.12. Secuencia de inicio dun componente. . . . .	48
4.13. Clase Parameters. . . . .	49
4.14. Bloques dalgúns dos componentes de App Inventor para Robobo. . . . .	50
4.15. Clase de App Inventor RoboboComponent. . . . .	50
4.16. Comunicación handler-componente na primeira iteración. . . . .	55
4.17. Accións de expresión corporal. . . . .	56
4.18. Interfaces para a recepción da información de sensores. . . . .	58
4.19. Expresión corporal no primeiro nivel de abstracción. . . . .	58
4.20. Primeira iteración. . . . .	59
4.21. Expresión corporal no segundo nivel de abstracción. . . . .	60
4.22. Bloques de expresión corporal. . . . .	62
4.23. Axuste da velocidade do sensor central dianteiro en App Inventor. . . . .	65
4.24. Axuste da velocidade dos motores en App Inventor. . . . .	65
4.25. Acción periódica do movemento dos motores en App Inventor. . . . .	66
4.26. Comportamento do robot ao achegarse a unha parede. . . . .	67
4.27. Accións do sentido da vista. . . . .	68
4.28. UML de Blob e Blobcolor. . . . .	69
4.29. Interface para recibir eventos da vista. . . . .	70
4.30. Sentido da vista no primeiro nivel de abstracción. . . . .	71
4.31. Sentido da vista no segundo nivel de abstracción. . . . .	72
4.32. Bloques das cores para o sentido da vista. . . . .	73
4.33. Bloques do sentido da vista. . . . .	74
4.34. Accións de expresión facial. . . . .	74
4.35. Interface para eventos da expresión facial. . . . .	76
4.36. Expresión facial no primeiro nivel de abstracción. . . . .	76
4.37. Expresión facial no segundo nivel de abstracción. . . . .	77
4.38. Bloques das emocións para a expresión facial. . . . .	78
4.39. Bloques de expresión facial. . . . .	78
4.40. Accións do sentido do tacto. . . . .	79
4.41. Interfaces do framework actual para a detección de accións sobre a pantalla táctil. . . . .	80
4.42. Interface para eventos do sentido do tacto. . . . .	80

4.43. Sentido do tacto no segundo nivel de abstracción. . . . .	81
4.44. Bloques das direccións para o sentido do tacto. . . . .	82
4.45. Bloques do sentido do tacto. . . . .	83
4.46. Interruptor do movemento en App Inventor empregando o tacto. . . . .	84
4.47. Cambio de caras por obxecto vermello en App Inventor. . . . .	85
4.49. Accións de expresión sonora. . . . .	86
4.50. Interface para eventos da expresión sonora. . . . .	89
4.51. Expresión sonora no primerio nivel de abstracción. . . . .	90
4.52. Expresión sonora no segundo nivel de abstracción. . . . .	90
4.53. Octava das notas dispoñibles para a expresión sonora. . . . .	92
4.54. Bloques da expresión sonora. . . . .	92
4.55. Accións de percepción sonora. . . . .	93
4.56. Interface para eventos do sentido do oído. . . . .	94
4.57. Sentido do oído no segundo nivel de abstracción. . . . .	95
4.58. Octava das notas dispoñibles para a percepción sonora. . . . .	96
4.59. Bloques da percepción sonora. . . . .	97
4.60. Saúdo do robot en App Inventor cando está listo para funcionar. . . . .	98
4.61. Interruptor do movemento co son en App Inventor. . . . .	99
4.62. Accións de equilibriocepción. . . . .	100
4.63. Equilibriocepción no segundo nivel de abstracción. . . . .	101
4.64. Bloques de equilibriocepción. . . . .	102
4.65. Accións de interocepción. . . . .	103
4.66. Interocepción no segundo nivel de abstracción. . . . .	104
4.67. Bloques de interocepción. . . . .	105
4.68. Queixas do robor por movelo. . . . .	106
A.1. Como importar unha extensión de App Inventor. . . . .	116
A.2. Vista dos compoñentes engadidos na vista de deseño de App Inventor.	117
A.3. Pasos para empezar a empregar os compoñentes. . . . .	118



# Capítulo 1

## INTRODUCCIÓN

### 1.1. Motivación

O GII (siglas en castelán de Grupo Integrado de Ingeniería) leva desenvolvendo, desde a súa creación no ano 1999 na Universidade da Coruña (UDC), unha liña de investigación en Robótica e Cognición como principal obxectivo de traballo. A meta desta liña consiste no desenvolvemento de sistemas reais capaces de responder co maior grao de autonomía posible ás condicións cambiantes do medio, sen ter que precisar da axuda dun operador ou programador para establecer as novas estratexias necesarias para a consecución da tarefa encomendada.

Dentro desta liña de investigación debemos diferenciar dúas derivadas: unha chamada Robótica Autónoma, na que se engloban os proxectos relacionados co desenvolvemento de controladores para robots autónomos e o deseño e fabricación dos propios robots; e outra chamada Mecanismos Cognitivos, máis centrada no estudio e implementación de funcionalidades cognitivas de alto nivel en robots autónomos, que emprega os procesos cognitivos humanos como inspiración para desenvolver un mecanismo cognitivo completo para un robot.

Un dos proxectos asociados á liña de Mecanismos Cognitivos é DREAM [1] (Deferred Restructuring of Experience in Autonomous Machines); está financiado pola Unión Europea e incorpora o sono e procesos semellantes aos soños nunha arquitectura cognitiva, permitindo que un robot ou grupo de robots consoliden a experiencia obtida empregando un formato máis xenérico e útil, mellorando a súa capacidade para aprender e adaptarse, e facendo ao robot máis autónomo na adquisición, organización e uso do seu coñecemento. A idea subxacente consiste en aproveitar a fase de "sono" do robot dunha maneira semellante a como xa se teñ observado que ocorre

no cerebro humano, onde sabemos que durmir é unha fase crítica para o aprendizaxe [2], para analizar e procesar a gran cantidade de información obtida da experiencia noutra escala temporal, e facer que o robot posúa un mellor entendemento do mundo que o rodea.

A raíz deste proxecto, xorde a iniciativa *adopt a robot*; cuxos principais obxectivos son a obtención de experiencia coa arquitectura cognitiva en situacíons da vida real, e a interacción entre un robot real e os humanos; así como permitir que usuarios non expertos –por exemplo pequenos– empecen a súa aventura cos robots.

Para alcanzar estes obxectivos, o GII vai contar con ROBOBO; un proxecto de robótica educativa que permite, ademais do citado no parágrafo anterior, que os estudantes interactúen con robots más realistas, e empreguen funcións más avanzadas, como as requiridas nunha arquitectura cognitiva. Este proxecto está baseado nunha plataforma hardware que recibe o mesmo nome: ROBOBO. Actualmente comercializado por unha spin-off da UDC chamada MINT (Manufactura de Ingenios Tecnolóxicos), e que podemos observar na figura 1.1.



Figura 1.1: ROBOBO na súa última versión.

O ROBOBO consiste nunha base motorizada, que recibe o nome ROB, e un smartphone, denominado OBO. O ROB sería as pernas e os brazos do robot; permiten o seu movemento, e danlle as capacidades básicas para sentir a súa contorna mediante sensores infravermellos. E o OBO é o seu cerebro; proporciona capacidade de proceso e habilidades de alto nivel para actuar e sentir, como poden ser a fala, a visión e o oído. O robot estará dispoñible para as escolas en novembro de 2017 para

servir como ferramenta educativa ou de entretenimento para ensinar programación, robótica, e disciplinas STEM (Science, Technology, Engineering and Mathematics) nun amplio rango de idades. Ademais, a súa arquitectura modular permite estendelo con novas características, e facilita que usuarios de diferentes niveis educativos programen o robot empregando ROS (Robot Operating System), Java ou programación por bloques. Pero tamén ten outras vantaxes, como o baixo custo da plataforma; a gran capacidade de sensorización e conexión que brinda o emprego dun smartphone, xa que calquera de gama media conta con acelerómetro, xiroskopio, GPS, cámara, micrófono, WIFI, GSM e Bluetooth; escalabilidade, xa que mediante o cambio do smartphone pódese actualizar dependendo das necesidades; e a gran disponibilidade de teléfonos, pois actualmente case todo o mundo, independentemente da idade, conta co seu propio terminal. Por último, cabe destacar que existen versións libres do hardware e do software do robot que están dispoñibles na Web do proxecto [3] para a súa descarga, así como tamén están dispoñibles os modelos 3D cos que poderemos imprimir a plataforma do robot.

Unha vez neste punto, o seguinte paso a seguir é incrementar as capacidades do ROBOBO para empregalo como ferramenta educativa nun rango de idades o máis amplio posible, e independentemente dos coñecementos técnicos en materia de programación e robótica que teñan os usuarios; de maneira que os alumnos poidan participar no proceso de desenvolvemento da tecnoloxía a través da creación de novas funcións de máis alto nivel, e no proceso de ensinanza propoñendo exercicios para outros estudiantes baseados na súa propia aprendizaxe.

Ademais, unha das principais virtudes do ROBOBO é a súa capacidade tecnolóxica grazas ao uso do smartphone. Por isto, de cara a espremer ao máximo as posibilidades que este aporta, e como complemento ao que xa existe en Scratch, proponse o desenvolvemento dunha extensión de App Inventor que mediante os seus bloques achegue ademais aos alumnos ao desenvolvemento de aplicacións móbiles, e engada tamén unha nova faceta ao ROBOBO.

## 1.2. Obxectivos

Segundo o camiño exposto na sección anterior, o obxectivo deste traballo será definir un conxunto de funcións de alto nivel adaptado a unha contorna didáctica, e dotar ao robot ROBOBO das ferramentas necesarias para permitir que usuarios non expertos programen o robot mediante o uso dunha linguaxe de programación

baseada en bloques –App Inventor–, e mediante unha API en Java que simplifique a actual. Deste xeito, un novo usuario poderíase iniciar no mundo da robótica e da programación empregando unha sinxela linguaxe gráfica moi intuitiva, e despois progresar cara a programación Java a medida que van aumentando os seus coñecementos, primeiro empregando a API que se vai desenvolver neste traballo coa intención de facilitar a iniciación a novos programadores, e despois traballando directamente cos módulos que xa están dispoñibles actualmente.

Este obxectivo pódese dividir en tres fases:

- Definición de funcións de alto nivel enfocadas a unha contorna didáctica procurando sobre todo que sexan intuitivas e fáciles de empregar ata por rapaces sen experiencia previa en programación.
- Desenvolvemento da API Java simplificada que permita o acceso as funcións definidas no punto anterior procurando evitar detalles técnicos que complicaran innecesariamente o seu uso por parte de programadores sen moita experiencia, e facendo que a maneira de traballar sexa o máis parecida posible ás librerías actuais do robot para facilitar a transición entre ambas.
- Integración de App Inventor coa API Java simplificada tentando manter o xeito de traballar igual ca se facía no punto anterior, e procurando que os algoritmos se poidan transformar entre bloques e código sen moito esforzo.

### 1.3. Estrutura da memoria

Tras este primeiro capítulo de **Introducción**, a presente memoria está estruturada do seguinte xeito:

- **Antecedentes**, no que se vai falar dos campos da programación e robótica educativa; empezando polos primeiros intentos de levar a programación as escolas, e continuando con outros robots programables semellantes que se poden atopar agora no mercado.
- **Fundamentos tecnolóxicos**, onde se describe detidamente o robot ROBO-BO e as características coas que conta actualmente, o framework que permite o seu desenvolvemento, os módulos cos que actualmente conta o robot, e App Inventor.

- **Desenvolvemento**, no que se describe de forma detallada o proceso levado a cabo para a realización do proxecto, e se expoñen algunhas aplicacións de proba que permiten ilustrar os conceptos propostos e o desenvolvemento realizado, ao mesmo tempo que permiten avaliar como se alcanzaron os obxectivos do proxecto.
- **Conclusíons**, no que se comenta o resultado final do proxecto e se propón traballo futuro a desenvolver.

## 1.4. Ferramentas empregadas

- **App Inventor** como software para programar o robot mediante unha linguaxe visual baseada en bloques.
- **Android Studio** como IDE, xa que o desenvolvemento se vai facer en Android debido a App Inventor; ademais, o ROBOBO só soporta teléfonos con este sistema operativo.
- Unha unidade do **robot ROBOBO** para realizar as probas que consta motores para o movemento da base e do móbil, así como capacidade básica para sentir a contorna a través de sensores.
- Un smartphone **BQ Aquaris M5** con cámara, micrófono, altofalante, conexión Bluetooth e sensores como o acelerómetro ou o xiroscopio.



# Capítulo 2

## ANTECEDENTES

Neste capítulo introduciremos a robótica e programación educativas facendo un pequeno recorrido dende as primeiras linguaxes de programación deseñadas para ser accesibles a usuarios non científicos, ata algúns dos robots educativos programables que podemos atopar actualmente no mercado.

### 2.1. Programación como ferramenta didáctica

No ano 1941 o enxeñeiro civil alemán Konrad Zuse constrúe o primeiro ordenador electromecánico programable completamente automático, o Z3 [4], que se programaba por medio de tarxetas perforadas. Soamente tres anos máis tarde, os británicos atacan a primeira mensaxe militar alemá o 5 de febreiro de 1944 empregando o *Colossus* [4]; o primeiro ordenador dixital electrónico programable. Uns anos despois, Estados Unidos desenvolve o *ENIAC* [4], que se podía programar a través de interruptores e cables do mesmo xeito que o *Colossus*.

Como vemos, nesta época os ordenadores non contaban nin con compiladores nin con linguaxes de programación de alto nivel, polo que a tarefa de programalos era un traballo de gran dificultade que requiría un gran esforzo intelectual e era propenso a erros; por exemplo, a programación do *ENIAC* era un proceso tan complexo que podía durar semanas [5]. Pero non foi ata 1945 que Konrad Zuse deseña o *Plankalkül*, a primeira linguaxe de programación imperativa de alto nível, que non foi implementada ata febreiro do ano 2000 por investigadores da Universidade de Berlín [6]. De feito, para ver a primeira implementación dunha linguaxe deste tipo houbo que esperar ata a década de 1950, coa aparición de *Fortran* en 1957 [7], enfocada a aplicacións científicas e de enxeñaría; e a de *LISP* un ano despois [8], que

rapidamente se converteu no preferido para a investigación en intelixencia artificial.

Estas dúas linguaxes estaban enfocadas ao traballo en campos moi específicos que facían que a súa aprendizaxe por persoas fóra deses ámbitos fora unha tarefa complicada. Neste sentido, unha das primeiras linguaxes enfocadas a estudantes doutros campos que non foran as ciencias ou as matemáticas, xa que eran os únicos que adoitaban aprender programación, foi *BASIC* (Beginner's All-purpose Symbolic Instruction Code) en 1964 [9], que posteriormente se estendería a partires de mediados dos 70 e os 80 como ferramenta para programar microcomputadores. Por outra banda, en 1967 aparecía *Logo*, unha linguaxe de programación enfocada a nenos [10].

### **2.1.1. Logo**

Como xa se introducía ao final do parágrafo anterior, *Logo* é unha linguaxe de programación de propósito xeral enfocada a nenos; é unha adaptación multiparadigma e un dialecto de *LISP* cuxo obxectivo era crear unha contorna onde os nenos puideran xogar con palabras e frases.

A característica máis coñecida de *Logo* é a tartaruga; un cursor en pantalla tradicionalmente con forma triangular que amosaba a saída de comandos de movemento. A tartaruga foi incluída na linguaxe por Seymour Papert a finais dos 60 para dar soporte a súa versión da tartaruga robot; un simple robot controlado polo usuario e deseñado para levar a cabo funcións de debuxo empregando un lapis retráctil unido ao corpo do robot, e que se move a través de comandos relativos á súa propia posición. Ademais, algunas implementacións da linguaxe soportaban detección de colisións e permitían ao usuario redefinir a apariencia do cursor, facendo que a tartaruga se comporte coma un sprite.

Ao longo da historia houbo varias implementacións de *Logo* para varios sistemas, como *Apple Logo*, *Atari Logo* ou *Color Logo* –Tandy–; versións orientadas a obxectos coma *ObjectLOGO*; ou a versión tridimensional *Logo3D*. Pero ademais, esta linguaxe tamén serviu de inspiración a outras posteriores coma *Smalltalk*, *AgentSheets*, *Etoys* ou *Kojo*.

### **2.1.2. AgentSheets**

*AgentSheets* é unha ferramenta de ciberaprendizaxe para ensinar programación e habilidades propias das tecnoloxías da información a través do desenvolvemento de xogos e simulacións baseadas en axentes a través dunha interface de arrastrar e

soltar [11][12]. O primeiro prototipo apareceu en 1989 na Universidade de Colorado da man de Alexander Repenning, o inventor da programación por bloques de arrastrar e soltar [13], que foi incluída en *AgentSheets* en 1996, cando se converteu nun produto comercial. Esta enfocada a rapaces en educación primaria ou secundaria, pero tamén pode ser empregado por profesionais en investigación para modelar complexos problemas científicos con miles de axentes ou explorar axentes intelixentes ou cooperativos. Ademais, existe unha versión que permite traballar en tres dimensíons chamada *AgentCubes*.

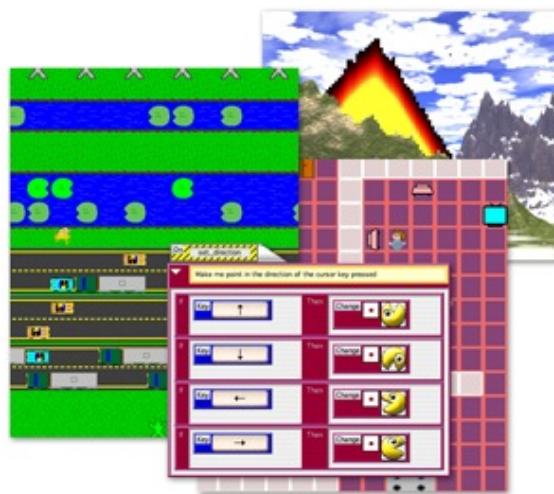


Figura 2.1: Agent Sheets.

### 2.1.3. Etoys

*Etoys* é un framework de programación visual para a linguaxe *Squeak* [14]; un dialecto de *Smalltalk-80* desenvolvida por Apple Computer.

Trátase dunha linguaxe de programación baseada en prototipos e orientada a obxectos enfocada ao uso en educación desenvolvida por Alan Kay en Walt Disney Imagineering para dar soporte ao aprendizaxe construtivista. A primeira implementación data do ano 1996, e conta cunha interface de arrastrar e soltar fácil de usar e que segue unha estratexia moi semellante á de *AgentSheets*. Ademais esta licenciado baixo as licencias MIT e Apache, e conta cunha ampla comunidade internacional que o emprega coa que poder compartir proxectos a través de internet.

### 2.1.4. Alice

*Alice* é unha contorna de programación baseado en bloques que permite crear animacións, historias interactivas ou pequenos xogos 3D facilmente [15]. Está deseñado para ensinar habilidades de pensamento lóxico e computacional, os fundamentos da programación, e para ser a primeira exposición á programación orientada a obxectos. Ademais, o proxecto *Alice* tamén conta con ferramentas e materiais complementarios para empregalo en diferentes espectros de idades e materias.

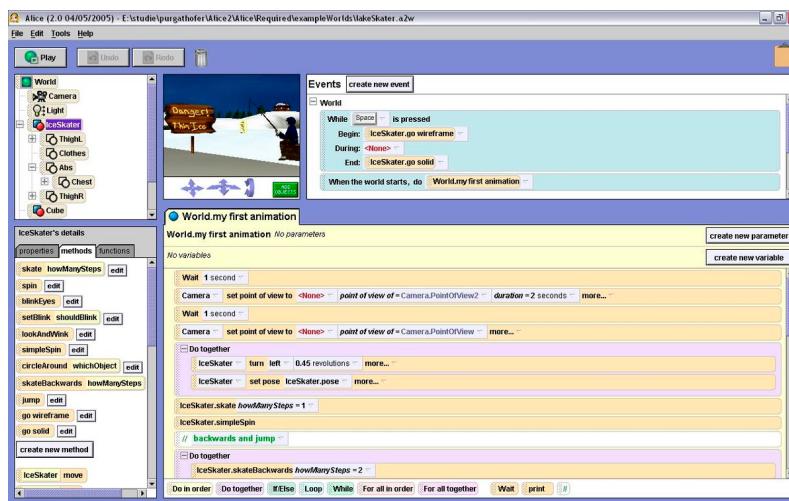


Figura 2.2: Alice.

O proxecto nace en 1995, desenvolvido por un grupo de investigación da Universidade de Virginia liderado por Randy Pausch, como unha ferramenta de prototipado rápido en realidade virtual; e que permite dende 1999 a creación de modelos 3D empregando unha interface de arrastrar e soltar.

Ademais, dende 2007, existe unha versión chamada *Story Telling Alice* creada por Caitlin Kelleher, posteriormente coñecida co nome de *Looking Glass*, coa intención de facer *Alice* aínda máis intuitivo, e poñendo especial énfase en como atraer más nenas ao mundo da informática. As principais diferencias entre estas dúas versións de *Alice* son [16]:

- Animacións de alto nivel que permiten aos usuarios programar interaccións sociais entre personaxes.
- Un tutorial en forma de historia que introduce aos usuarios á programación a través da construcción dunha historia.

- Unha galería de personaxes e escenarios 3D con animacións á medida deseñadas para inspirar ideas.

### **2.1.5. NetLogo**

*NetLogo* é unha linguaxe de programación de código libre baseada en axentes e unha contorna integrada de modelado deseñado por Uri Wilensky no espírito da linguaxe *Logo* [17]. Permite ensinar conceptos de programación empregando axentes na forma de *tartarugas, parches, enlaces e o observador*.

Esta linguaxe foi deseñada con varios públicos en mente, dende nenos en idade escolar ata expertos sen formación en programación para modelar fenómenos naturais. De feito, a contorna conta cunha extensa librería de modelos de diferentes dominios, coma economía, bioloxía, física, química ou psicoloxía; e é empregado por numerosos estudos científicos.

Outra das características destacables de *NetLogo* é que permite que calquera poida crear extensións da linguaxe [18] para proporcionarlle novas características empregando Java, Scala, ou calquera outra linguaxe baseada na máquina virtual de Java. Ademais, conta cunha comunidade activa en todo o mundo coa que poder compartir modelos ou extensións.

### **2.1.6. Scratch**

*Scratch* [19] é unha linguaxe libre de programación visual desenvolvida polo MIT coa intención de axudar á xente nova a aprender a pensar de maneira creativa, razoar sistematicamente, e traballar de xeito colaboracionista. Está deseñada especialmente para nenos de entre 8 e 16 anos, pero é empregado por xente de tódalas idades [20].

Os estudiantes están a aprender con *Scratch* en tódolos niveis, dende a ensinanza primaria ata a universidade; e a través de diferentes disciplinas, como as matemáticas, informática, artes ou ciencias sociais.

Igual que ocorría co *NetLogo*, *Scratch* permite tamén a creación de extensións para proporcionar novas funcionalidades e para poder conectar os seus proxectos con hardware externo, como poden ser PicoBoard, LEGO WeDo, LEGO Mindstorms, ou Arduino. Ademais, tamén conta cunha comunidade activa en todo o mundo coa que poder compartir proxectos e extensións.

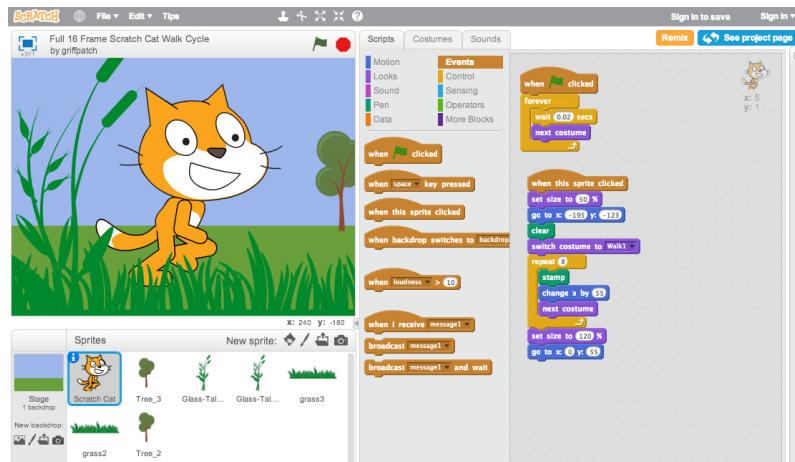


Figura 2.3: Scratch.

### 2.1.7. App Inventor

*App Inventor* é unha aplicación web de código libre orixinalmente construída por Google, e mantida actualmente polo Massachusetts Institute of Technology (MIT), que permite que programadores novatos poidan crear aplicacíons para Android mediante unha interface gráfica moi semellante a Scratch, na que os usuarios poden arrastrar e soltar obxectos visuais.

Non obstante, dado que forma parte da base sobre a que se desenvolve este traballo, describirase en máis detalle na sección 3.2.

## 2.2. Robótica educativa

Como vimos na sección anterior, xa dende os anos 60 existen multitude de proxectos enfocados a facilitar que cada vez máis xente, e cada vez máis nova, se inicie no mundo da programación. As razóns que argumentan dita postura son amplamente coñecidas e van dende fomentar a curiosidade dos alumnos pola ciencia ou a tecnoloxía e afacelos á resolución de problemas mediante actividades divertidas [21], ata favorecer o traballo en equipo e a capacidade de liderado [22].

Actualmente podemos atopar no mercado numerosos robots para empregar como ferramenta didáctica nun amplo rango de idades; presentaranse a continuación algúns deles.

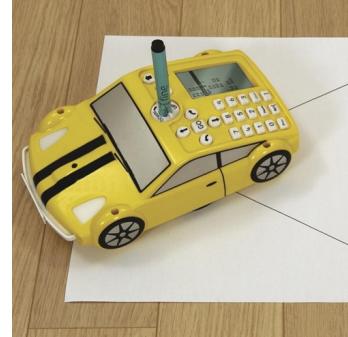
### 2.2.1. Bee-Bot

*Bee-Bot* [23] é un robot deseñado para nenos de entre 3 e 7 anos con teclas de dirección na parte superior para inserir ata 40 comandos que moverán o robot cara adiante, atrás, esquerda e dereita. Unha vez se preme o botón *GO*, *Bee-Bot* empezarase a mover seguindo os comandos inseridos e ha emitir un asubío coa conclusión de cada un deles para que o usuario sexa quen de seguir a execución; cando remata o programa, o robot notifíca a través de sons e luces.

O *Bee-Bot* realmente compórtase dunha maneira moi semellante ao *Logo*; de feito, existe unha versión chamada *Blue-Bot* que se pode conectar cun ordenador a través dunha conexión bluetooth, e descargar o programa desenvolvido coa linguaxe *Logo* para observar o seu comportamento cun robot real.



(a) Bee-Bot.



(b) Pro-Bot.

Tamén existe a versión *Pro-Bot*, que ten forma de coche de carreiras e é coma o irmán máis vello do *Bee-Bot*. Ten dous modos de funcionamento; un que traballa da mesma maneira que o seu irmán, e outro no que as instrucións permiten introducir números, de maneira que se poden facer xiros de 45 graos. Ademais, conta cunha pantalla LCD na que se visualizan o equivalente en *Logo* aos comandos introducidos, resaltando tamén durante o proceso de execución o comando que se está a realizar.

### 2.2.2. BQ Zowi

*Zowi* é un robot deseñado pola marca española BQ en colaboración con Clan de RTVE para o público de menos idade [24]. Nada máis sacalo da caixa convértese nunha especie de mascota de maneira que os nenos se sintan atraídos dende o primeiro momento, e xa programalo conta con dous botóns que o fan bailar ou esquivar os obxectos detectados polos sensores de ultrasóns que ten por ollos. Non obstante,

unha vez instalada a aplicación Android, o usuario poderá controlar o robot por control remoto ou mediante a asignación dunha secuencia de movementos gracias aos seus modos de funcionamento. Ademais esta baseado en Arduino e todo o hardware está baixo unha licencia Creative Commons, polo que é facilmente personalizable empregando unha impresora 3D. Por último, *Zowi* tamén se pode programar a través dunha linguaxe visual creada por BQ moi semellante en concepto a Scratch: Bitbloq.

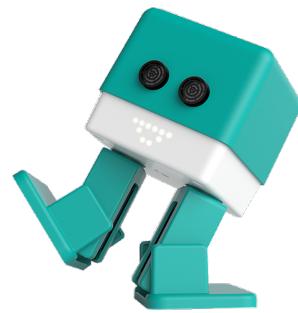


Figura 2.5: BQ Zowi.

### 2.2.3. LEGO

*LEGO* é unha empresa de xoguetes danesa recoñecida principalmente polos seus bloques de plástico interconectables, pero que ademais conta con dúas liñas destinadas ao mundo da robótica educativa: *LEGO WeDo* e *LEGO Mindstorms*.



(a) LEGO WeDo.



(b) LEGO Mindstorms.

*LEGO WeDo* é a proposta para os máis novos, estando enfocado exclusivamente á ensinanza primaria [25]. A través das clásicas pezas de LEGO que todos coñecemos

os nenos poden construír diferentes modelos con sensores e motores que se conectan ao ordenador e poden ser programados mediante unha ferramenta gráfica fácil e sinxela, ou tamén empregando *Scratch*.

Por outra banda, *LEGO* completa o seu catálogo en robótica educativa a través de *LEGO Mindstorms*, a súa proposta para alumnos a partires dos 10 anos que pode ser empregada como ferramenta nas aulas dende os últimos cursos da ensinanza primaria ata a universidade [26]. Está composto por un bloque que actúa como centro de control e fonte de potencia do robot, o *EV3*, que conta con portos de entrada e saída onde se conectan os motores e os sensores, miniUSB para a conexión co ordenador, USB para conectar un conector wifi, porto para tarxetas SD para ampliar a capacidade de memoria, altofalante integrado, receptores de sinais infravermellas para recibir comandos a través dun mando, e Bluetooth; ademais das clásicas pezas interconectables e outras especiais flexibles, de fixación ou móbiles [27], todo isto programable a través dunha contorna de programación visual baseado en bloques.

#### 2.2.4. Fischertechnik Robotics

A marca *Fischertechnik* conta tamén cunha liña de produtos destinada á robótica educativa. Pode ser empregada como xoguete educativo a partires dos 7 anos, como vehículo para experimentar con fenómenos físicos e científicos dende os 8 anos, e para aprender a construír e programar robots dende os 8-10 anos empregando un sistema de montaxe realista mediante pezas mecánicas ensamblables [28].



Figura 2.7: Fischertechnik.

Tódolos produtos desta liña contan cun controlador con pantalla táctil, entradas e saídas para sensores e motores, miniUSB para conectar co ordenador, USB para a conexión doutros accesorios coma pode ser unha cámara, porto para tarxetas SD, un

altofalante, WiFi e Bluetooth. Ademais, conta cunha ferramenta software que permite programar comportamentos de maneira intuitiva empregando unha contorna gráfica moi semellante aos diagramas de fluxo [29].

### 2.2.5. MakeBlock

*MakeBlock* é unha empresa que comercializa kits de construcción de robots mediante pezas de aluminio que encaixan entre sí de maneira sinxela e flexible por medio de raís roscados e buratos [30]. Esta baseado en Arduino e tanto o hardware coma o software son libres e están dispoñibles na web, podendo ata imprimir as pezas a partir dos modelos 3D. As conexións entre a placa base e os demais compoñentes fanse de maneira sinxela empregando conectores RJ25, conta con diferentes tipos de motores e con multitud de sensores como poden ser o de son, ultrasóns, gas, humidade, temperatura, acelerómetro e máis, e outros compoñentes electrónicos como LEDs, potenciómetros, pantallas ou módulos wifi e bluetooth; todo isto programable dende unha contorna visual empregando Scratch ou Bitbloq.



Figura 2.8: MakeBlock.

### 2.2.6. WowWee

*WowWee* é unha marca que comercializa robots educativos, entre os que podemos atopar ao *COJI* e ao *Coder MIP*; ambos programables.

*COJI* é a proposta en robótica educativa programable de *WowWee* para os máis pequenos [31] –a partires dos 4 anos–. Conta cunha aplicación disponible para Android e iOS [32] que permite interactuar con el por medio de 5 xogos diferentes, que van dende un xogo de memoria ou control remoto, ata os que permiten programalo a través dunha linguaxe baseada en emoticonos.

Por outra banda, *Coder MIP* está enfocado a un público de maior idade que o *COJI* [33]. Conta con dúas aplicacións ambas dispoñibles para Android e iOS: a



(a) WowWee COJI.



(b) WowWee Coder MIP.

primeira permite controlar o robot remotamente, e a segunda proporciona unha interface para programar comportamentos a través dunha linguaxe visual por bloques de arrastrar e soltar.

### 2.2.7. NAO

*NAO* é un robot humanoide de 58 centímetros capaz de interactuar de forma natural con todo tipo de público [34]. Conta con dúas cámaras, catro micrófonos, nove sensores táctiles, dous de ultrasóns, oito de presión, acelerómetro e xiroscopio; pero ademais inclúe outros elementos de expresión que lle aportan mellores capacidades de interacción, como os seus 53 LEDs RGB, o sintetizador de voz e os seus dous alto-falantes. Ademais inclúe un software gráfico de programación chamado *Choreograph* compatible con Windows, Linux e MAC, que permite programalo sen necesidade de ter coñecementos previos de programación. Non obstante, para usuarios avanzados tamén é posible empregar linguaxes como C++, Python, Java, .NET e MATLAB.

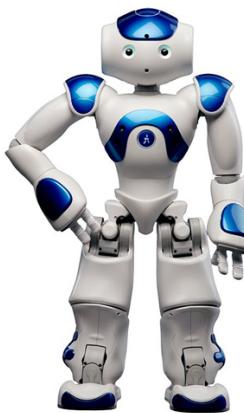


Figura 2.10: Robot NAO.

Actualmente este robot emprégase tanto nunha contorna educativa como de in-

vestigación ao redor de todo o mundo [35], e como ferramenta para ensinar a nenos autistas, xa que algúns destes nenos atopan ao robot máis amigable que a outros seres humanos.

## 2.3. Conclusíons

Como se pode comprobar co exposto no último capítulo, os intentos de achegar a programación a un público máis amplio son unha tónica constante dende os anos 60, empezando con BASIC e Logo na mesma década, e seguindo coa chegada das linguaxes visuais por bloques coma Scratch ou App Inventor. A pesar de todo isto, a día de hoxe a programación segue a ser unha práctica que poucos dominan fóra dos ámbitos das ciencias e da enxeñería. Non obstante, dado o gran número de produtos en robótica educativa que podemos atopar no mercado, parece que esta realidade está cambiando.

Unha posible razón deste cambio pode ser a gran cantidade de ordenadores e outros dispositivos coma smartphones e tabletas da que dispoñemos na actualidade, xa que gracias ao avance, e na consecuente baixada de prezo da tecnoloxía, cada vez máis persoas están expostas a esta dende unha idade máis temprá. Por isto, é o noso deber crear as ferramentas necesarias para introducir máis xente, e máis nova, ao mundo da programación; sendo a robótica educativa unha ferramenta prometedora que está a ser apoiada tanto por escolas como por empresas de xoguetes, e que ademais axudaría a mellorar outras capacidades nos estudantes como o traballo en grupo ou a creatividade.

Con esta mesma intención, o presente traballo pretende dota ao robot Robobo, un produto comercial existente que actualmente só conta cun framework de desenvolvemento, das ferramentas necesarias para convertelo nun robot educativo adaptado a diferentes niveis de experiencia, ofrecendo a posibilidade de programalo a través dunha linguaxe de bloques mediante App Inventor, e dunha sinxela API de robótica empregando Java.

# Capítulo 3

## FUNDAMENTOS TECNOLÓXICOS

Neste capítulo hanse expoñer as características das diferentes tecnoloxías involucradas no desenvolvemento deste traballo; que van dende a base do robot e a linguaxe visual por bloques de App Inventor, ata algúns mecanismos de Android.

### 3.1. Robobo

Como xa comentabamos na introdución, a plataforma Robobo que se emprega para o desenvolvemento deste traballo está composta por unha base motorizada (ROB) con capacidade básica de sensorización, e un smartphone (OBO) como base para a execución de comportamentos de alto nivel que estende as capacidades de interacción e sensorización. Nas seguintes subseccións hanse expoñer máis detidamente as características do ROB, así como do framework e das librerías.

#### 3.1.1. ROB

O ROB, cuxa foto podemos ver na Figura 3.1, é o corpo do robot; permite que se move e proporcionalle a capacidade para sentir o entorno a través dos seus infravermellos. A continuación, as súas características principais:

- Soporte universal para móbiles: inclúe un adaptador de cor azul montado sobre unha plataforma motorizada que permite a rotación e inclinación do móvil sobre o chasis, facendo que o Robobo poida mover a "cabeza".



Figura 3.1: ROB.

- Conexión Bluetooth entre o móvil e o ROB: isto permite, ademais do descrito no punto anterior, o uso de distintos terminais telefónicos; outra opción para conectalos era o USB, pero moitas veces atopamos que os móbiles teñen este porto colocado en diferentes partes.
- LEDs: ten nove en total para interactuar co usuario; cinco deles forman unha matriz na parte dianteira para ofrecer información avanzada mediante cores e formas.
- Motores: conta con dous de corrente continua. Cada motor conta cun codificador magnético que transforma o movemento do motor en pulsos eléctricos interpretables polo software do robot.
- Sensores: infravermellos para a detección de caídas e obstáculos para dar capacidade de movemento autónomo, e que están distribuídos no ROB da seguinte maneira:
  - Dianteiros:
    - Tres sensores; un no centro e dous nos laterais próximos ás rodas, de xeito que se detecten obstáculos cos que o robot puidera chocar.
    - Dous sensores orientados cara o chan e colocados a ambos lados do central para evitar caídas.
  - Traseiros:
    - Un sensor central para detectar colisións.

- Dous sensores situados a ambos lados e orientados cara o chan para evitar caídas.

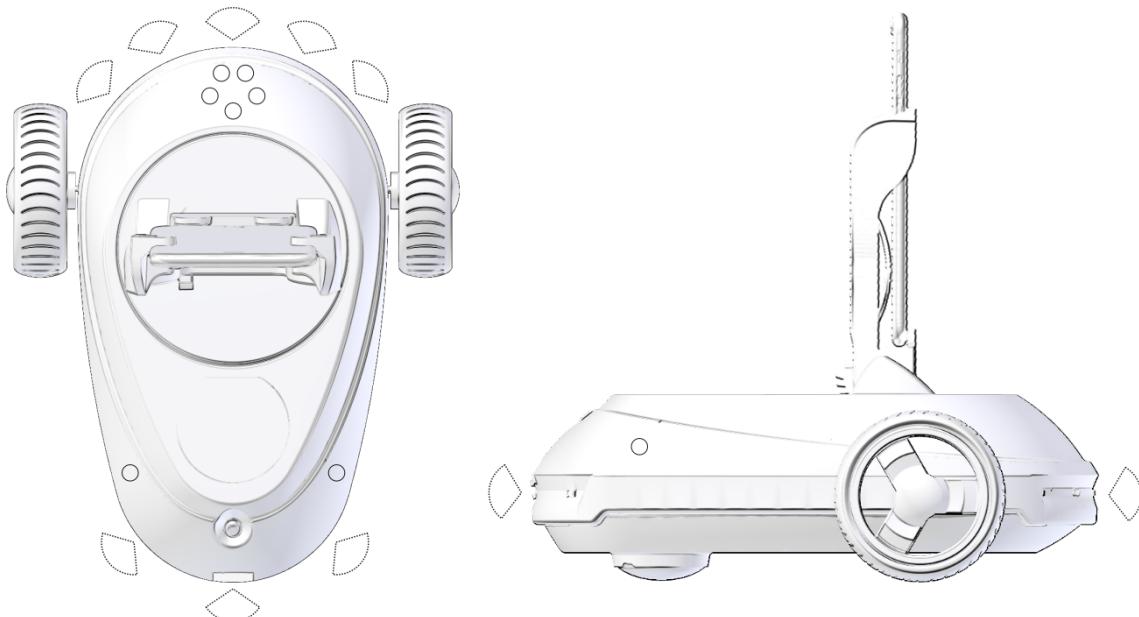


Figura 3.2: Esquema de situación dos actuadores e perceptores.

### 3.1.2. ROBOBO! Framework

O ROBOBO! Framework é unha contorna de traballo de código libre licenciado baixo a licenza LGPL empregado para o desenvolvemento de aplicacións para o robot. Está construído sobre Android, xa que tódolos comportamentos de alto nivel desenvolvidos se executan no móvil que se coloca sobre a base, e de maneira que obriga a seguir unha arquitectura modular; sempre que vaimos desenvolver unha aplicación para o robot deberémoslo facer en forma de módulos que implementen unha interface común chamada *IModule*, cuxa representación en UML podemos ver na Figura 3.3, e que contén as cabeceiras dos métodos que se han chamar para cargar e descargar cada módulo. A idea disto é facer que todo este proceso de carga e descarga de módulos sexa transparente para o programador centralizándoo nun compoñente tamén incluído no framework; a aplicación soamente deberá conectarse a un servizo Android, tamén incluído no framework, para poder xestionar os módulos que vai empregar.

Débese destacar tamén que se trata dun framework orientado a obxectos complexo que ten unha curva de aprendizaxe considerable ata para expertos en programación, xa que non soamente é necesario ter experiencia previa en Java ou con

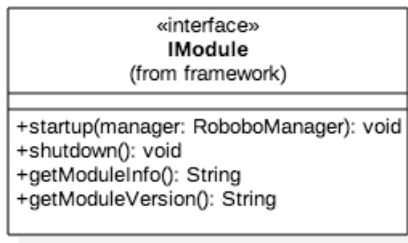


Figura 3.3: Interfaz IModule.

certos mecanismos de Android, senón tamén debido ao gran número de módulos que se deben xestionar e aos diferentes ficheiros de configuración necesarios para a execución dunha aplicación.

### 3.1.3. Módulos do Robobo

Os módulos do Robobo, como xa comentabamos durante a explicación do framework, son onde realmente residen as capacidades que ten o robot. Entre elas atoparemos funcións básicas de movemento, como poder mover o ROB; de interacción con humanos –cousa que ha aparecer indicado no nome da librería que conteña o módulo coas siglas HRI (do inglés Human Robot Interaction)–, como cambiar a cara do robot ou facer que fale; funcións sensitivas como o uso do xiroskopio ou acelerómetro; de control remoto; ou que permiten que o robot sexa compatible con outros frameworks empregados na robótica. A continuación describiremos brevemente cada un dos módulos.

#### 3.1.3.1. Control da base motorizada (ROB)

O módulo *BluetoothRobInterfaceModule* é o que permite o control do ROB, así como o acceso á información dos sensores infravermellos ou do nivel da batería. Tódalas funcións que permite facer a plataforma veñen definidas na interface *IRob*, que podemos ver representada en UML na Figura 3.4. A través dela poderemos recibir as mensaxes de estado do ROB, como os datos dos infravermellos para obter información de caídas e obstáculos, datos do movemento dos motores, ou do nivel da batería. Pero tamén enviar comandos para mover os motores ou o soporte no que vai o móbil, cambiar as cores dos LEDs ou o modo do ROB, e modificar a periodicidade das mensaxes de estado.

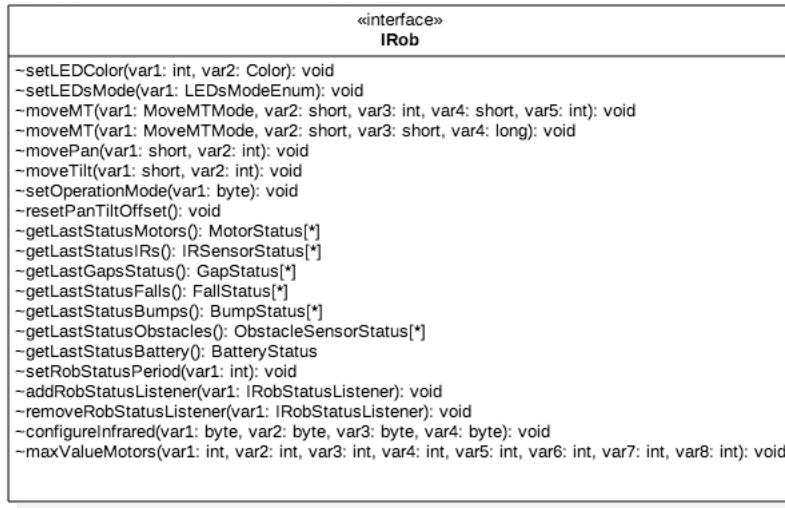


Figura 3.4: IRob interface para o control do ROB.

Non obstante, esta non é a única posibilidade que temos para controlar a base do robot; co módulo *DefaultRobMovementModule* tamén contamos con outra interface para o control da plataforma: *IRobMovementModule*. A diferencia da anterior, esta interface ten un nivel de abstracción superior; facilita funcións de movemento básicas como xirar aos lados, pero ao mesmo tempo fai perder precisión nas accións. Ademais, só permite o movemento do ROB, e non a recepción da información de estado.

### 3.1.3.2. Emocións

O módulo *DefaultEmotionModule* proporciona ao robot a capacidade de expresar emocións faciais a través dunha cara que se amosa na pantalla do móvil. Por defecto, conta cunha pequena colección de caras á disposición do usuario e, aínda que polo momento non é posible, si que se ten intención de proporcionar ao programador a posibilidade de empregar a súa propia colección de caras personalizada.

### 3.1.3.3. Son

Nunha mesma librería (robobo-hri-sound) atoparemos tódolos módulos relacionados tanto coa recepción como coa emisión de son, exceptuando os da fala.

Os que permiten o procesamento do son ambiente dependen doutro chamado *SoundDispatcherModule*, que é o encargado de alimentar aos demais, entre os que están *TarsosDSPClapDetectionModule*, para detectar palmadas ou sons percusores;

*TarsosDSPPitchDetectionModule*, para obter a frecuencia en hertzs do son captado polo micrófono; e *TarsosDSPNoteDetectionModule*, que depende do anterior, e que permite obter a nota musical que estivera soando.

Por outra banda, os módulos que permiten emitir sons polo altofalante son: *AndroidEmotionSoundModule*, para sons de emocións, como unhas risas, un gruñido de enfado, ou un de desaprobación; e *AndroidNoteGenerationModule*, que permite emitir notas musicais.

### 3.1.3.4. Control da pantalla táctil

A detección de xestos cos dedos sobre a pantalla do móvil corre a cargo do módulo *AndroidTouchModule*, que permite detectar catro diferentes, como un pequeno golpe, unha pulsación continuada no tempo, ou o arrastre dun dedo.

### 3.1.3.5. Visión

Para empregar a cámara do móvil, o Robobo conta con tres módulos, un básico que captura fotogramas e alimenta aos demais, chamado *AndroidCameraModule*; *AndroidFaceDetectionModule*, para a detección de caras; e un para o seguimento de obxectos de cores chamado *OpenCVBlobTrackingModule*.

### 3.1.3.6. Producción e recoñecemento da fala

Ademais de ser quen de sentir a contorna e moverse por ela, o robot tamén é quen de interactuar cos humanos dunha maneira cómoda para nós a través da fala, que ven implementada en dous módulos: un para a producción, *AndroidSpeechProductionModule*; e outro para o recoñecemento chamada *PocketSphinxSpeechRecognitionModule*.

### 3.1.3.7. Interacción por mensaxes

Outra das posibilidades coas que conta o robot para interactuar cos humanos ademais da fala, é mediante texto; gracias aos módulos *GMailBackgroundMessagingModule* e *JTwitterModule*, o Robobo ten a capacidade de enviar correos electrónicos ou twits a través da conta de Gmail ou Twitter estipulada polo usuario.

### 3.1.3.8. Compatibilidade con ROS

Ao principio desta subsección comentábase, entre outras cousas, que o Robobo dispoñía dunhas funcións que o fan compatible con outros frameworks empregados

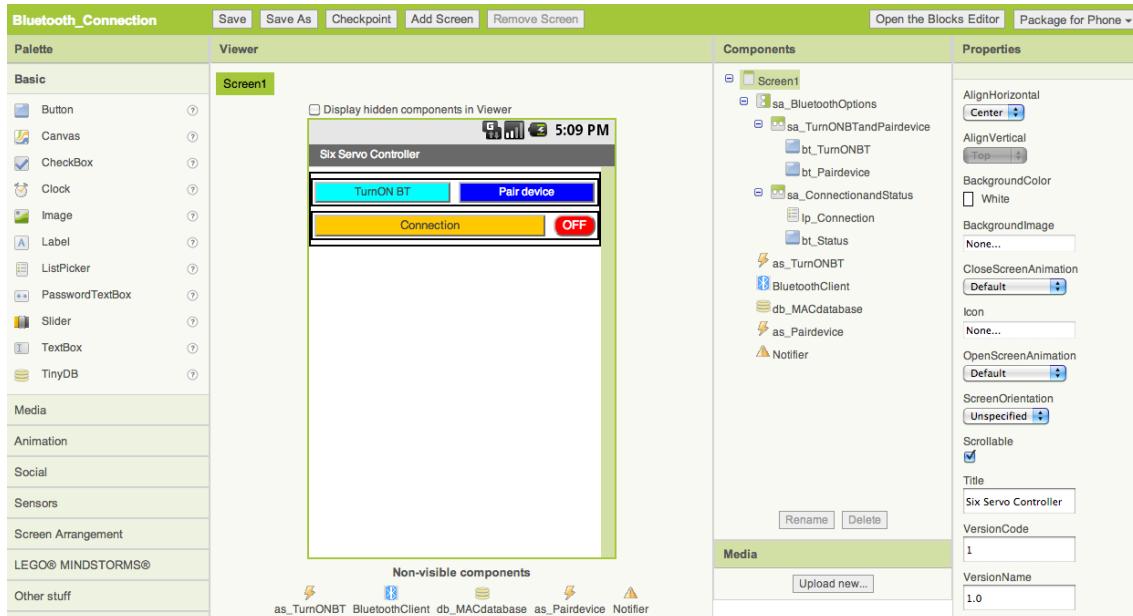
en robótica; esas funcións das que falabamos están contidas no módulo *RoboboRos-Module*, e con outros frameworks estabámonos a referir a ROS (Robot Operating System).

ROS [36] é un framework flexible que permite facer software para robótica. É unha colección de ferramentas, librerías e convencións que procuran simplificar a tarefa de crear complexos e robustos comportamentos para robots a través dunha gran variedade de plataformas. Proporciona os servizos que se poden esperar dun sistema operativo, como a abstracción do hardware, xestión dos dispositivos de baxo nivel, implementación de funcións comunmente empregadas, paso de mensaxes entre procesos, ou xestión de paquetes; pero a pesar de todo isto, non é un sistema operativo propiamente dito, polo que se ten que executar nun entorno Linux. Outra cousa a ter en conta, e que é un proxecto de código libre cunha comunidade ao redor do mundo que contribúe de maneira activa ao proxecto. Ademais, dado que crear robots de propósito xeral é difícil, ROS está construído de xeito que se fomente o desenvolvemento de software colaboracionista.

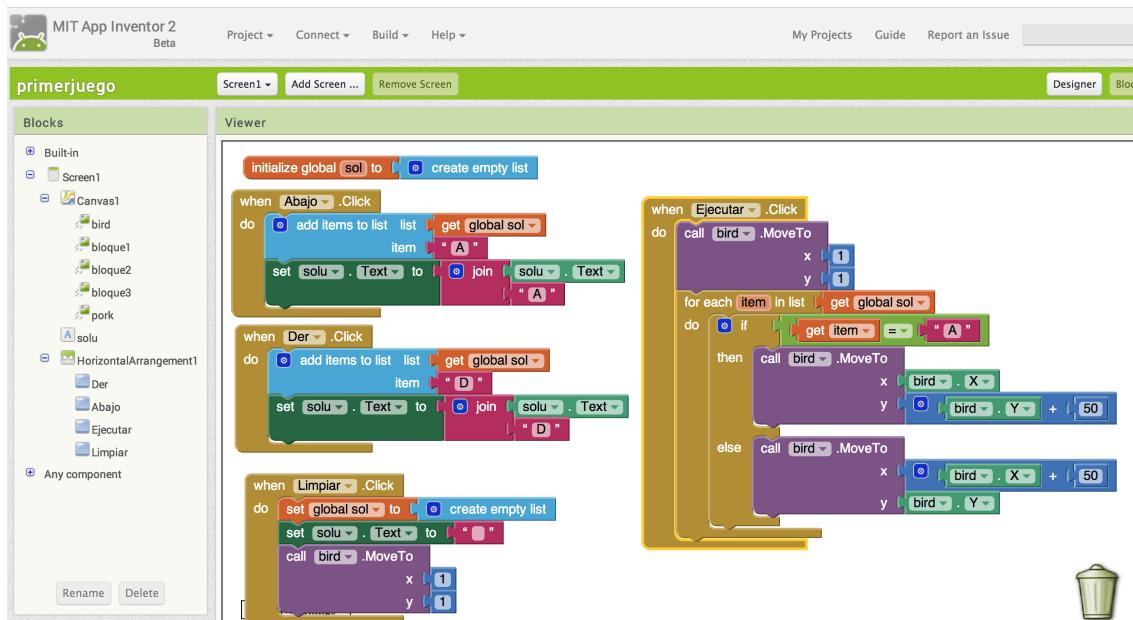
Por todo o exposto no parágrafo anterior, parece moi conveniente que todo robot construído actualmente se poida programar empregando ROS. A librería *robobo-ros-module*, a través do módulo que contén co seu mesmo nome, cumpre tal propósito e exporta a funcionalidade do robot a ROS.

## 3.2. App Inventor

App Inventor [37] é un entorno de programación intuitivo e visual dispoñible a través de Internet que permite que calquera persoa –ata rapaces en idade escolar– poidan crear aplicacións para móveis e tabletas. Emprega unha interface gráfica semellante a Scratch ou StarLogo TNG [38], que permite a construcción de aplicacións mediante bloques visuais que o usuario pode arrastrar e soltar, e que encaixan uns con outros coma as pezas dun crebacabezas; na figura 3.5 podemos ver unha imaxe da contorna de traballo. Inicialmente estaba desenvolvido por Google e actualmente é mantido polo MIT (Massachusetts Institute of Technology) baixo a súa propia licencia. Ademais, conta cunha ampla comunidade de usuarios e desenvolvedores ao longo de todo do mundo que participan activamente no proxecto. Está construído de xeito que calquera pode ampliar as súas funcionalidades por medio de extensíons que pode compartir co resto facilmente, e fomentando que todo o mundo, especialmente rapaces novos, pasen de consumidores a creadores de tecnoloxía.



(a) Vista de deseño.



(b) Vista de edición de bloques.

Figura 3.5: Entorno de traballo de App Inventor.

Na figura 3.5 podemos ver as dúas partes da contorna de desenvolvemento: unha enfocada á parte visual da aplicación, onde escollemos tódolos componentes que se van empregar; e outra para desenvolver as interaccións entre os elementos que forman a aplicación por medio de bloques.

Unha vez desenvolvida a aplicación, App Inventor permite probala a través dun emulador, ou ben instalánda no móvil por medio dunha conexión USB, ou empregando a wifi. Para usar este último sistema será imprescindible instalar a aplicación *MIT AI2 Companion*, e que tanto o ordenador onde se fai o desenvolvemento, e o móvil onde se desexan facer as probas, estean conectados á mesma rede. Ao abrir a aplicación podremos escoller entre dúas opcións; ou ben escribimos manualmente a cadea de caracteres que se nos indica, ou escaneamos un código QR coa cámara do móvil. Feito isto, e tras uns segundos de espera, a nosa aplicación estará instalada e lista para poñela a proba.

Outra opción para instalar é xerar o APK da aplicación, descargala empregando o mesmo sistema do código QR, e facer as probas. Non obstante, se estamos depurando a aplicación non é a opción máis recomendable, pois de haber un erro, a mensaxe que indica a causa amósase no móvil e durante pouco tempo; polo que se o texto é moi longo, é moi probable que non deamos lido a mensaxe. Sen embargo, co método descrito no parágrafo anterior, o erro amósase na pantalla do ordenador que executa App Inventor ata que pechamos a xanela onde aparece.

Por todo o exposto anteriormente, e dado que é unha das ferramentas software más potentes para o desenvolvemento de aplicacións Android por usuarios sen coñecementos de Java; esta ha ser a ferramenta empregada para proporcionar ao proxecto da linguaxe visual por bloques que precisa.

### 3.3. Android

Na introdución deste traballo falábbase que o obxectivo do mesmo era desenvolver unha API simplificada e de máis alto nivel que a actual para que usuarios non expertos puideran crear aplicacións para o Robobo empregando App Inventor ou Java. Ademais, debido a que o robot é a combinación dunha base motorizada (ROB) e un móvil (OBO) que executa os comportamentos, o sistema operativo base sobre o que se desenvolve tanto o framework como os módulos do Robobo, e o presente traballo, é Android.



# **Capítulo 4**

## **DESENVOLVEMENTO**

Neste capítulo hase describir o proceso de desenvolvemento e os detalles de implementación que foron necesarios para alcanzar os obxectivos a cumplir que se propoñían na introdución deste traballo.

### **4.1. Introdución**

Robobo é un robot programable baseado na combinación dun móbil como centro de procesamento, e unha base motorizada con capacidade básica para sentir o entorno a través de sensores incorporados na mesma, cuxo principal obxectivo é converterse nunha ferramenta didáctica que docentes de tódolos niveis poidan empregar nas aulas como catalizador da formación en destrezas como o traballo en equipo, a resolución de problemas, ou a autonomía; e como vehículo para a iniciación de novos programadores.

Este robot comercialízase actualmente cun framework de desenvolvemento en Java que permite a creación de comportamentos en forma dunha arquitectura modular, e cun conxunto de módulos de baixo nivel que proporcionan as súas funcións, que van dende algo básico como o movemento da base, a outras más complexas que involucran moitos dos sensores cos que conta un móbil actualmente, como poden ser a cámara, o micrófono, ou o acelerómetro. Lamentablemente, a complexidade do framework e dos módulos produce que a curva de aprendizaxe ata para expertos en programación sexa algo a considerar, non soamente debido á necesidade de experiencia previa con Java e Android –sistema sobre o que está desenvolvido o framework–, senón tamén debido aos ficheiros de configuración necesarios. Todo isto, ademais das dificultades que poden xurdir da integración nunha aplicación dos módulos requiri-

dos para esta, impiden que un usuario non experto empregue o Robobo coma unha ferramenta de aprendizaxe.

Como se comentaba ao inicio desta sección, o obxectivo deste traballo é transformar o robot nunha ferramenta didáctica que se poida empregar nas aulas independentemente dos coñecementos técnicos en materia de programación que os alumnos posúan, polo que é totalmente necesario atopar unha solución que evite que os estudiantes se teñan que enfrentar con demasiadas dificultades nos seus primeiros pasos como programadores. A proposta deste traballo para solucionar isto é crear un novo framework de desenvolvemento de máis alto nivel que resulte máis intuitivo e evite a necesidade de ficheiros de configuración, e de coñecer conceptos propios de Java e Android para facilitar a primeira toma de contacto entre alumno e robot. Inspirándose en produtos de robótica educativa como os que se poden ver no capítulo de antecedentes, escolleuse a ferramenta de programación gráfica para aplicacóns Android chamada App Inventor, da que aproveitaremos principalmente a súa linguaxe visual por bloques, de xeito que nunha primeira proximación á programación o usuario tampouco precise traballar directamente con código Java.

Outra das cousas a ter en conta ademais, é o seguinte paso a dar polo programador iniciado mediante unha linguaxe visual por bloques; unha vez este domina tódolos conceptos involucrados nunha contorna de programación gráfica coma esta, o lóxico é progresar cara unha linguaxe de programación máis clásica coma pode ser Java. Non obstante, se a intención é que o alumno aprenda a codificar en Java ao mesmo tempo que aprende a traballar co framework de desenvolvemento actual do Robobo, coas dificultades que isto implica, probablemente o número de estudiantes que continúen con éxito os cursos de programación véxase claramente reducido. Xa que a intención é crear unha API de alto nivel que simplifique o traballo do usuario e facilite a súa introdución no mundo da robótica educativa, tamén deberemos reducir o salto en coñecementos técnicos que se precisa para pasar da API de desenvolvemento actual á nova simplificada. Para solucionar este problema, a proposta deste traballo consiste en proporcionar un novo nivel de abstracción entre as dúas APIs, de xeito que ao avanzar cara o uso da actual, o requerimento de coñecementos técnicos aumente gradualmente sen atafegar aos alumnos.

Por todo o exposto anteriormente, e coa intención de acadar os obxectivos deste traballo mediante unha solución o máis completa posible, desenvolverase unha API de máis alto nivel que simplifique a xa existente empregando dous niveis de abstracción: un primeiro nivel en Java con funcións de máis alto nivel e que simplifique

o seu uso, pero mantendo unha estrutura semellante que facilite a transición entre ambos; e outro nivel construído sobre o anterior e que implemente as mesmas funcións ca este empregando a linguaxe de App Inventor. De maneira que os estudiantes que dominen a programación por bloques poidan primeiro afacerse a Java sen tanto esforzo, e despois progresar cara a implementación de aplicacíons co *ROBOBO! Framework* unha vez dominen o nivel anterior.

## 4.2. Aproximación conceptual

Actualmente os módulos que proporcionan as funcións para crear comportamentos para o Robobo están agrupados en librerías dispoñibles a través de Maven Central, e de xeito que aqueles que gardan máis relación entre eles comparten librería: tódolos que empegan a cámara na de visión, os que usan o altofalante ou o micrófono na de son, tódolos relacionados coa interacción a través da fala noutra, etc... Debido a isto, unha das dificultades coas que se pode atopar un usuario non experimentado, como xa se dixo anteriormente, é a maneira de integrar tódolos módulos necesarios nunha aplicación. Co fin de axudar ao novo programador a xestionar as súas necesidades respecto dos módulos que vai empregar, durante o desenvolvemento deste traballo decidiuse seguir un deseño que os agrupara en componentes seguindo unha analogía cos sentidos humanos, de maneira que resulte intuitivo abondo para que rapaces en idade escolar poidan escoller os que precisen pensando que sentido necesitarían eles mesmos para resolver un problema, pero ao mesmo tempo favorecendo unha mellor compresión tanto do problema que se está a solucionar, como doutros semellantes.

Non obstante, non tódolos módulos se poden agrupar mediante esta analogía; nin o movemento do robot, nin as súas capacidades para falar, nin poñer unha cara riseira, son sentidos. É por isto que se decide crear unha nova categoría a maiores dos sentidos: expresión. De xeito que poidamos expandir a analogía que xa existía, e facer unha nova más completa na que se inclúan tamén outras capacidades dos seres humanos como a expresión corporal, facial, escrita e sonora.

A categoría de expresión agrupa tódolos módulos do robot que permiten que este exprese a seu comportamento; da mesma maneira que se temos un sistema que transforma unha entrada nunha saída baleira non poderemos saber se está cumprindo ou non o seu propósito, co robot pasaría igual: se non coñecemos o programa do robot, e ao acendelo non amosa ningún tipo de sinal de que está acendido, como podemos

## 4.2. APROXIMACIÓN CONCEPTUAL CAPÍTULO 4. DESENVOLVEMENTO

saber se está funcionando correctamente? Non se pode. Pero o mesmo ocorre coas persoas, por exemplo o persoal sanitario, cando atopa un paciente completamente inmóbil que non responde a ningún tipo de estímulo, o primeiro que faría sería comprobar se está vivo comprobando se respira ou medíndolle o pulso, xa que doutra maneira non podería concluir nada.

Con isto quérrese ilustrar que toda saída dun sistema, aínda que non sexa o seu obxectivo, ou sexa algo involuntario coma no exemplo do paciente, expresa algo; e por tanto, xustificar o deseño conceptual adoptado.

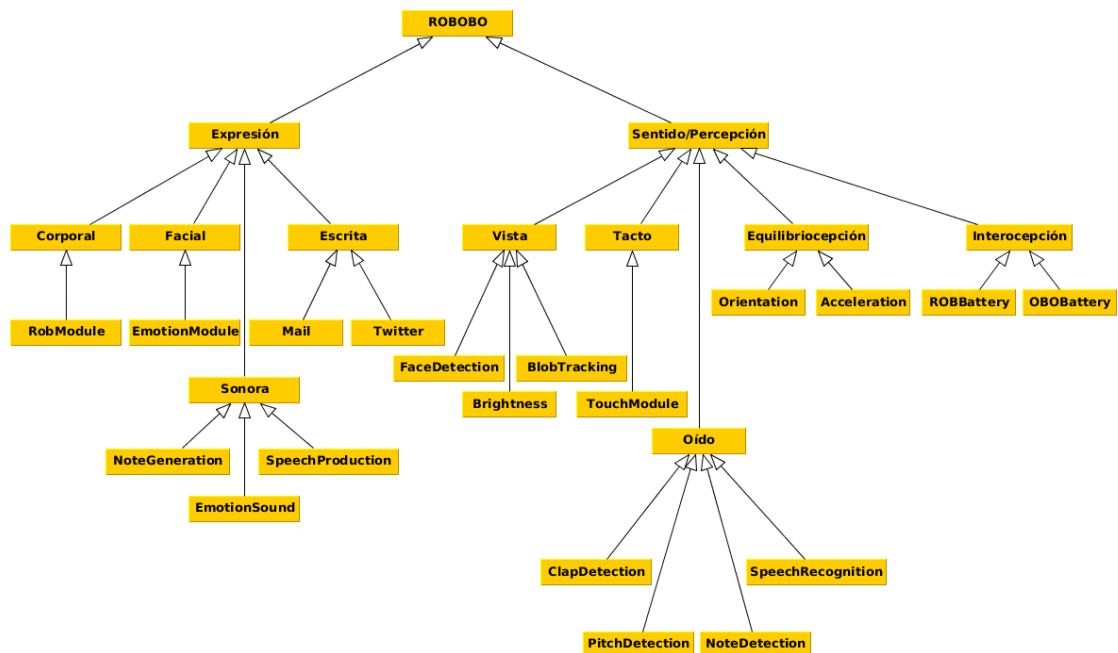


Figura 4.1: Deseño conceptual.

Na Figura 4.1 podemos observar o resultado da proposta que se fixo nos parágrafos anteriores. Á esquerda de todo vemos como o módulo que controla o ROB (RobModule) está dentro da categoría de expresión e baixo o tipo corporal. Normalmente, se falamos de expresión corporal, o primeiro no que pensamos é na danza ou nos xestos que facemos ao falar, pero a cousa non queda aí, xa que a simple posición do corpo pode facer entender moitas cousas sen necesidade de falar; por exemplo os sinais que fan os ciclistas e motoristas co brazo cando circulan sen luces; a man pechada co polgar cara arriba que significa aceptación, algo positivo; ou a man completamente aberta e en posición vertical co brazo estarricado cara adiante que simboliza unha orde de parada.

Unha vez máis, podemos comprobar como todo movemento ou posición do corpo

é expresión corporal, polo que tamén se ha considerar todo o que o Robobo pode facer a través do ROB –o seu corpo– como tal, aínda que o significado máis habitual de expresión corporal non sexa exactamente ese.

Ademais de todo exposto anteriormente, outro dos argumentos a favor da analogía da expresión é que nos permite introducir conceptos propios da robótica comparándoos con capacidades propias dos seres humanos; mediante esta analogía e a dos sentidos poderemos empezar a explicar os conceptos de actuador e percepto dun robot dunha maneira más intuitiva, e que sexa fácil de comprender ata polos rapaces máis novos, xa que deixarían de ser algo tan abstracto ao telos representados con esta proposta como expresión e sentidos do robot.

Por outra banda, na introdución deste capítulo expúxose que se ía seguir unha estratexia baseada en dous niveis de abstracción, de xeito que gardaran o maior número de semellanzas posibles entre eles e o framework actual, co fin de facilitar a transición entre tódolos niveis. Segundo con esta idea, e co exposto nesta sección, tanto a implementación da API Java, como a de App Inventor, contarán exactamente co mesmo deseño, e sen variar ningunha das funcións de alto nivel que se definan, para que así os novos programadores poidan transformar os seus programas en bloques a código Java sen moitas dificultades.

## 4.3. Arquitectura

Antes de nada, empezaremos esta sección expoñendo os aspectos técnicos e funcionais que xustifican as decisións adoptadas no deseño da arquitectura proposta neste proxecto.

### 4.3.1. Análise crítico e xustificación

Se acordamos do capítulo anterior, onde se presentaban os diferentes produtos ou tecnoloxías involucradas neste traballo, describíanse, entre outras cousas, as características de App Inventor, e onde tamén se mencionaba que era posible que calquera usuario estendera as súas funcionalidades creando extensións; que non son más ca clases Java para crear novos compoñentes cuxo único requisito é estender unha das dúas clases que se proporcionan para este fin, unha para compoñentes que visualicen algo en pantalla, e outra para os que non. Deste xeito, para cumplir o obxectivo deste traballo, só sería necesario crear unha capa por riba dos módulos actuais do Robobo que ocultara certos detalles complexos da implementación, que

se correspondería co primeiro nivel de abstracción do que se falaba na introdución deste capítulo; e crear compoñentes de App Inventor que chamaran aos métodos desta API simplificada para constituir o que sería o segundo nivel de abstracción.

Ademais de App Inventor, no capítulo anterior describiamos as características do framework e dos módulos do Robobo. Quíxose destacar tamén que se trataba dunha ferramenta complexa e de baixo nivel que requiría certo tempo de adaptación ata para usuarios con experiencia previa en programación. As principais razóns expostas foron a necesidade de ter coñecementos sobre a linguaxe Java e certos mecanismos de Android, pero tamén debido aos ficheiros de configuración necesarios. Todo iso sen esquecer que se trata dun framework orientado a obxectos, que require ademais que o usuario teña un mínimo de experiencia con este estilo de programación.

Primeiro de todo, en calquera proxecto Android é necesario establecer as dependencias mediante un ficheiro de configuración de Gradle –ferramenta libre de compilación para os proxectos baseados neste sistema operativo– no que, ademais das librerías nas que se atopen os módulos que se vaian empregar, tamén se deben establecer os números de versión. É certo que este non parece un proceso especialmente complicado, pero nun ámbito escolar, sobre todo cos máis pequenos, complicaría e retardaría as clases innecesariamente.

Outro dos problemas sería a escolla dos módulos; nun proxecto de aplicación para o Robobo, os módulos que se van empregar débense definir nun ficheiro baixo o directorio *assets* chamado *modules.properties*, no que se debe incluír unha lista numerada co nome da clase que implementa o módulo e o nome completo do paquete no que se atopa. Pero ademais, como existen algunas dependencias entre algúns dos módulos, e estes deben aparecer ordenados de xeito que se declaren primeiro os módulos dos que dependen outros, a tarefa non fai más ca complicarse de maneira absurda.

Segundo coas dificultades que impón o framework actual, outro dos problemas cos que se van atopar os novos usuarios son os ficheiros de configuración dalgúns dos módulos. Por exemplo, o módulo que captura imaxes empregando a cámara precisa un ficheiro no que se estableza a resolución, o de recoñecemento da fala esixe un ficheiro que inclúa a gramática das frases tipo que se van recoñecer, e o que captura son debe coñecer a frecuencia de mostraxe. Pero ademais, o usuario deberá establecer os permisos que precisa a aplicación no *Manifest* cando un módulo empregue algún tipo de función Android que o requira.

Por último, e para rematar a exposicións das complicacións derivadas da com-

plexidade do framework de desenvolvemento actual do robot, o programador tamén debe ter un mínimo de experiencia con servizos Android, posto que a conexión entre a aplicación desenvolvida e o framework realizaase desta maneira.

Como vemos tras a lectura dos últimos parágrafos, o RÓBOBO! Framework ao inicio do desenvolvemento deste traballo é demasiado complexo para ser empregado como ferramenta didáctica en cursos de iniciación á programación, e máis aínda se os alumnos son rapaces de ensinanza primaria ou secundaria.

Con respecto a este problema, xa se propuxeran dúas estratexias para paliar as dificultades coas que se poden atopar os programadores noveis. Unha delas é a creación de compoñentes que agruparan os módulos baixo dúas categorías: expresión e sentido/percepción. E a outra é a creación dunha API en dous niveis de abstracción, tal e como amosa a figura 4.2, que simplifique o framework actual; o primeiro empregando Java, e o segundo mediante App Inventor e a súa linguaxe de bloques. Non obstante, ademais de tódolos problemas expostos ata este momento, tamén aparecen outros de carácter técnico relacionados con App Inventor que se han explicar a continuación.

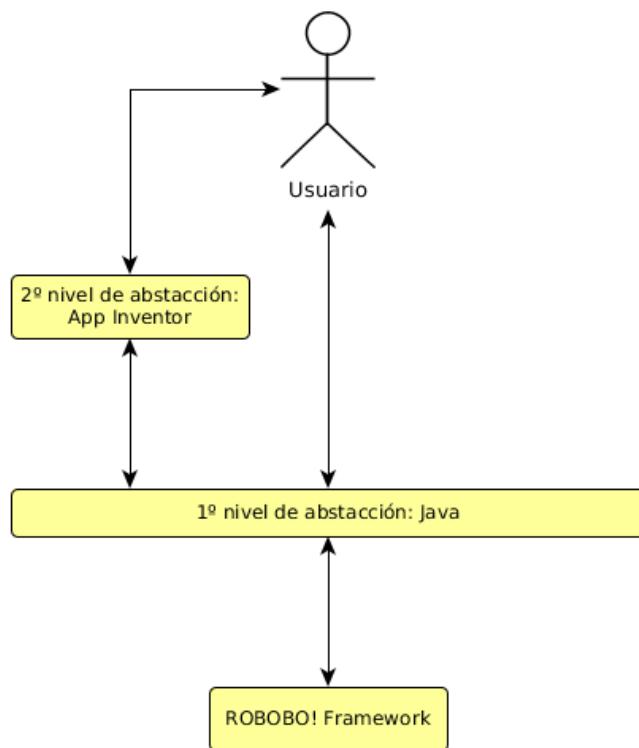


Figura 4.2: Diagrama dos niveis de abstracción.

Para crear unha extensión deste software coa que poder acceder as funcións do primeiro nivel de abstracción da API simplificada a través da linguaxe de bloques – creando o que sería o segundo nivel–, é necesario descargarase unha versión do mesmo a través do seu repositorio público, e seguir o modelo de clases esixido para a creación de novos componentes. Concretamente, todo componente que precise amosar algo en pantalla debe estender unha clase abstracta chamada *AndroidVisibleComponent*, e outra chamada *AndroidNonVisibleComponent* en caso contrario. O problema xorde debido a que se trata dun proxecto pesado con moitos elementos que fai que o seu tempo de compilación aumente ata situarse en torno aos 10 ou 15 minutos, cousa que fai que o desenvolvemento sexa moito máis lento.

Pero ademais do tempo de compilación, outra das dificultades que se atoparon, e que deriva tamén da complexidade do *ROBOBO! Framework*, foi a resolución das dependencias dos componentes de App Inventor creados; para o que se deben editar catro ficheiros de configuración diferentes da ferramenta de compilación empregada por este software: Apache ANT. Ademais, en relación a isto mesmo, todo componente que precise de librerías externas para funcionar debe declarar o seu nome na cabeceira da clase que o implementa para que se inclúan os ficheiros *jar* necesarios. Non obstante, a diferencia da comprobación de dependencias do compilador de Java, esta faise ao final de todo; polo que un erro ortográfico no nome dunha librería provoca un erro de compilación que se notifica 10 ou 15 minutos despois.

Por se isto fora pouco, e a pesar de ser un software que permite desenvolver aplicacións Android mediante a súa linguaxe visual, tódalas librerías externas que sexan necesarias para o desenvolvemento dun componente deben ser *jar*; é dicir, non se permiten librerías Android.

Como xa se dixo, todo isto non fai máis ca provocar que o desenvolvemento das extensións sexa lento e incómodo, pero quizais resulta máis molesto áinda a escasa e as veces incorrecta documentación para desenvolvedores do proxecto, que xunto coa imposibilidade de empregar o depurador dun IDE coa extensión de App Inventor, empeora áinda máis a situación. Por outra banda, tamén é importante recordar que grazas á implementación proposta en dous niveis pódense reducir en gran medida case tódalas consecuencias dos problemas técnicos expostos antes: o primeiro nivel pódese depurar a través do Android Studio sen dificultades, e así reducir as probas necesarias para a extensión de App Inventor. Ademais, a consecuencia disto último, tamén evitamos ter que compilar o proxecto de App Inventor tantas veces; polo que tamén se conseguuen reducir os tempos de espera.

Por último, e para rematar de expoñer tódolos problemas de carácter técnico, debemos recordar que App Inventor é un proxecto Android, e como toda aplicación deste sistema ten un límite máximo de métodos que pode ter. Os ficheiros das aplicacións de Android (APK) conteñen ficheiros de códigos de byte executables en forma de ficheiros *Dalvik Executable* (DEX) que conteñen o código compilado empregado para executar a túa aplicación. A especificación de Dalvik Executable limita a cantidade total de métodos aos que se pode facer referencia nun ficheiro Dex a 65536, incluídos os métodos do framework de Android, de biblioteca, e do teu propio código. Debido a que  $65536 = 64 \times 1024$ , este límite recibe o nome de *límite de referencia 64K*. Non obstante, isto pódese evitar empregando a biblioteca de compatibilidade de MultiDex, e habilitando dita opción no script de Gradle do proxecto. O problema é que App Inventor non é un proxecto Android normal: emprega Apache ANT en vez de Gradle como ferramenta de compilación, e o *Manifest* xérase automaticamente a partir de ficheiros de texto. En calquera caso, é algo que queda fora do noso control, xa que ao facer cambios para habilitar o soporte MultiDex tamén se debería bifurcar App Inventor da rama principal para manter unha nova rama de desenvolvemento únicamente para este propósito, cousa que non interesa.

Debido a todos estes inconvenientes foi estritamente necesario optar por unha solución que dalgunha maneira illara completamente tanto ao programador novel, como á extensión de App Inventor, da complexidade e das complicacións que se describiron anteriormente; polo que a decisión foi implementar a nova API seguindo a arquitectura cliente-servidor que podemos ver na figura 4.3. Deste xeito, temos un servidor nunha aplicación Android con tódolos ficheiros de configuración e dependencias, e un cliente máis lixeiro e sinxelo libre de toda complexidade. A continuación hanse expoñer as ferramentas necesarias para o desenvolvemento desta arquitectura en Android.

#### 4.3.2. Arquitectura de alto nivel

O elemento básico co que conta Android para desenvolver aplicacións cliente-servidor son os servizos; dos que hai dous tipos: iniciados ou de enlace. A diferencia entre eles radica na maneira de inicialos e finalizalos: os do primeiro tipo requiren unha chamada á función *startService()*, e deben rematar por si mesmos; e os do segundo tipo iníicianse e rematan automaticamente dependendo de se hai clientes enlazados. Para este caso, o escollido foi o servizo enlazado, de maneira que cada componente da API simplificada é un cliente do servizo, e non se hai que preocupar

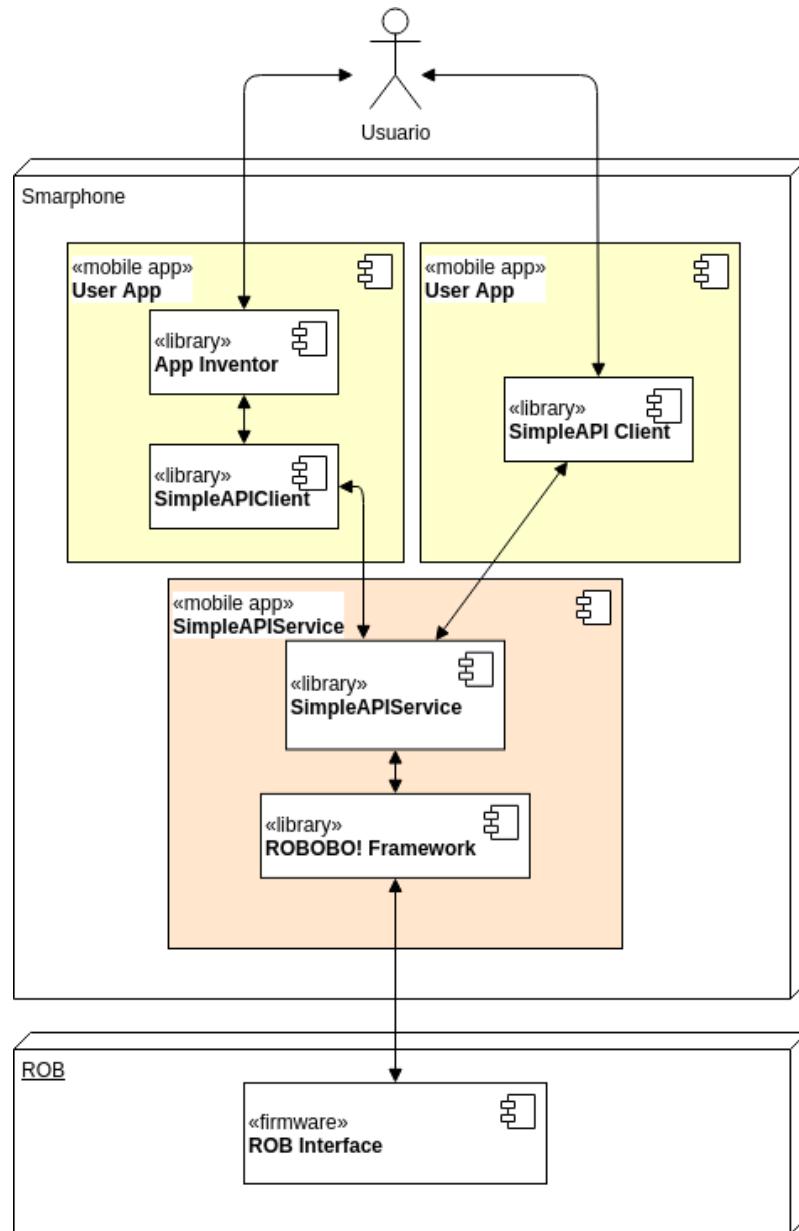


Figura 4.3: Diagrama de disposición física dos elementos da API simple.

por inicialo ou finalizalo.

Outra cousa a ter en conta nunha aplicación cliente-servidor é como se van comunicar as partes, xa que se ha precisar un mecanismo que permita a comunicación entre ambos procesos. Neste aspecto, Android conta con dúas posibilidades: empregar un mensaxeiro, ou AIDL (Android Interface Definition Language).

Un mensaxeiro proporciona unha interface para que os clientes e o servizo se comuniquen a través do paso de mensaxes; é a maneira máis sinxela de establecer

comunicación entre procesos (IPC), xa que o mensaxeiro pon en cola tódalas solicitudes nun só subprocesso de modo que non haxa que deseñar o servizo a fin de que sexa seguro para subprocessos. Ademais, é o xeito recomendado pola documentación de Android para realizar IPC cando non é necesario xestionar múltiples fíos de execución, xa que daría lugar a unha implementación do servizo máis complicada. De feito, o método subxacente baixo este mecanismo é AIDL, que non é máis ca unha linguaxe de definición de interfaces para Android.

Para que o lector se faga unha idea do funcionamento deste mecanismo, engádese a continuación un pequeno resumo:

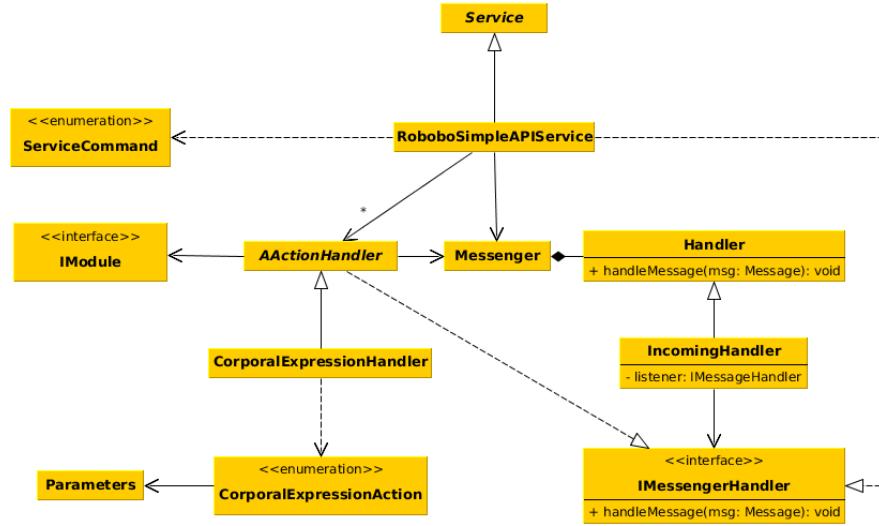
- O servizo implementa un handler que recibe un callback por cada chamada dun cliente.
- Emprégase ese handler para crear un mensaxeiro.
- O mensaxeiro crea unha interface que devolve aos clientes.
- Os clientes empregan a interface para crear as instancias do mensaxeiro que han as funcións para enviar mensaxes ao servizo.
- O servizo recibe cada mensaxe no handler.

Como xa se viña dicindo nos parágrafos anteriores, para evitar toda a cantidade de problemas que se comentaban, decidiuse implementar unha aplicación cliente-servidor que emprega o mensaxeiro de Android para realizar a comunicación entre ambos procesos; na figura 4.4 podemos ver o diagrama de clases simplificado en UML –soamente aparece representado o compoñente da expresión corporal para evitar sobrecargalo–. Por outra banda, débese destacar que a "clase" App Inventor que vemos no diagrama non é unha clase como tal, senón soamente unha maneira de representar como se comunica co resto.

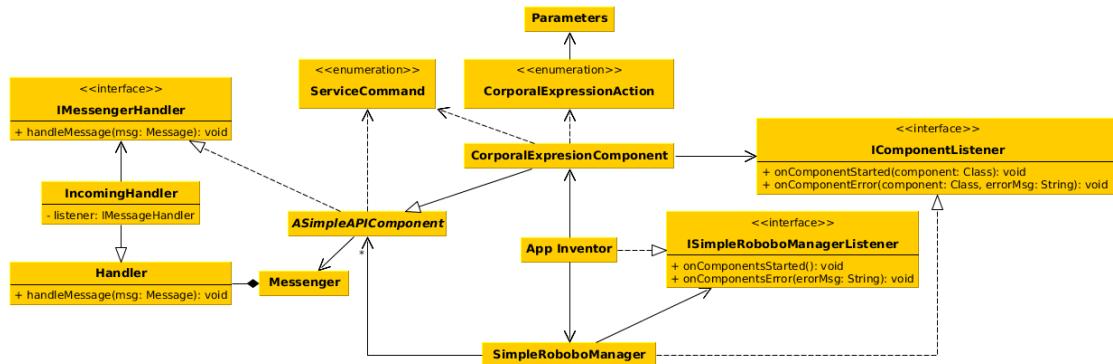
### 4.3.3. Principais compoñentes da arquitectura

#### 4.3.3.1. Compoñentes da API simple

Se acordamos o diagrama da Figura 4.1 do deseño conceptual, agrupabamos tódolos módulos existentes do Robobo baixo as categorías de expresión ou sentido, e baixo un tipo concreto dentro de cada categoría, como expresión corporal ou sentido da vista. Cada combinación dunha categoría e un tipo recibirá o nome de compoñente a partir de agora.



(a) Servidor.



(b) Cliente.

Figura 4.4: Diagrama de clases do cliente e servidor para o componente de expresión corporal.

Igual que os módulos eran os que implementaban os comportamentos do robot, o mesmo ocorre cos componentes na API simple. Cada componente –cliente– ten un análogo no servizo –servidor– que recibe o nome de *Handler*; como vemos na figura 4.4 temos o *CorporalExpressionComponent* e o *CorporalExpressionHandler*. O componente é o que fai de interface do módulo Robobo; posúe os métodos que ha chamar o usuario para programar comportamentos do robot. E o handler é o encargado de recibir as mensaxes do componente e traducilos na execución dos métodos do módulo Robobo correspondente. Para que cada componente e handler se comuniquen débese definir tamén unha *acción* de tipo Enum –*CorporalExpressionAction*– no que se ha definir un valor para cada método que se desexa exportar. Ademais,

cada unha destas *accions* vai conter unhos *parámetros* –Parameters–, no que se estipulan certos detalles internos das mensaxes. Na figura 4.5 podemos ver un pequeno exemplo de como sería a comunicación entre componentes e handlers no suposto de que se execute un método que espera unha resposta.

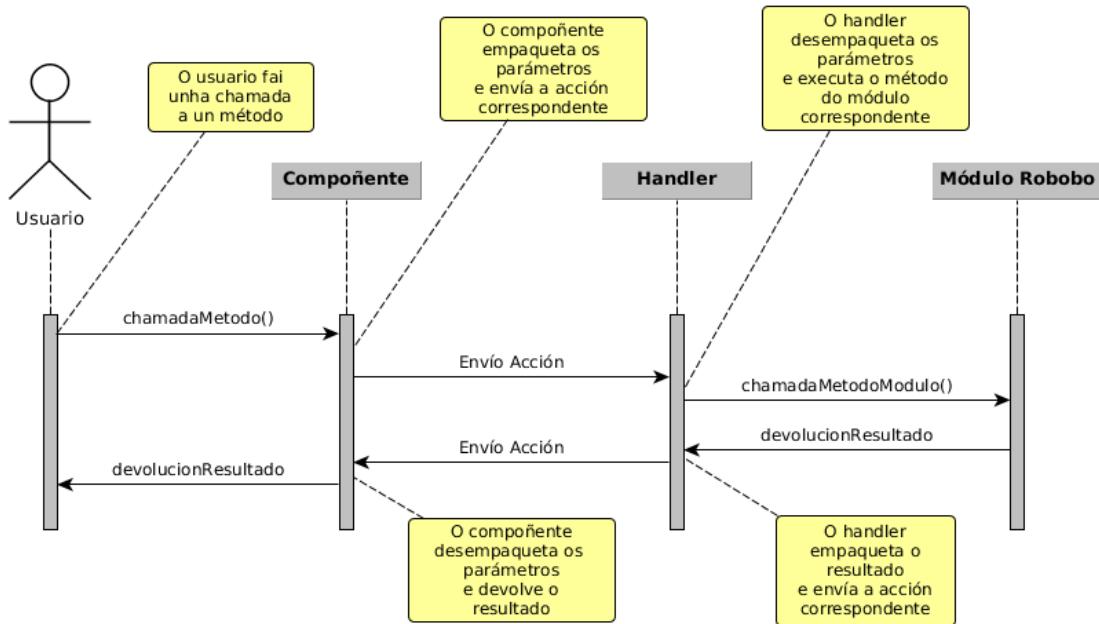


Figura 4.5: Diagrama básico da comunicación entre componentes e handlers.

Tanto os componentes como os handlers están construídos de maneira que todo o proceso de paso de mensaxes sexa transparente para eles, xa que toda esta implementación do protocolo queda encapsulada nas clases abstractas *ASimpleAPIComponent* e *AActionHandler* que deben estender. De feito, a única función dos componentes e handlers, ademais de facer de interface e lanzar chamadas a métodos de módulos concretos, é empaquetar e desempaquetar os parámetros necesarios para a execución dos métodos que se inclúen nas mensaxes, é dicir; procesar as *accions*.

O paso dos parámetros dos métodos realiza-se por medio do *Bundle* que toda mensaxe pode conter, e que permite engadir información por medio de pares chave-valor. Ademais, para que tódalas chaves empregadas sexan coñecidas por ambas partes, estas deben ser establecidas nos *parámetros* que toda *acción* debe conter.

Por outra banda, o proceso a seguir para estender estas dúas clases abstractas é moi sinxelo e semellante para ambos casos. No caso do handler, o encargado de instancialos sería o servizo *RoboboSimpleAPIService*, que se vemos o construtor, soamente debe establecer unha etiqueta de *log*, o cliente do componente co que se debe

```

AActionHandler
- LOG_TAG: String
- mCmdValues: SparseArray<Enum>
- mMessenger: Messenger
- mClient: Messenger
- roboboManager: RoboboManager
- classes: Class[]
- modules: IModule[]
- mStartedUp: boolean
# AActionHandler(logTag: String, client:Messenger, roboboManager: RoboboManager, classes: Class[])
+ handleMessage(msg: Message): void
# handleClientMessage(msg: Message): void
# onStartUp(): void
# onShutDown(): void
# getModuleInstance(): IModule
# getModuleInstance(index: Int): IModule
# send(action: Enum, data: Bundle): void
# send(action: Enum): void
- startUpReceived(): void
- getInstances(): void
- send(cmd: ServiceCommand): void
- send(cmd: ServiceCommand, data: Bundle): void

```

Figura 4.6: Clase AActionHandler.

```

ASimpleAPIComponent
- LOG_TAG: String
- mBound: boolean
- context: Context
- mConnection: ServiceConnection
- mClientType: ServiceCommand
- mCmdValues: SparseArray<Enum>
- mMessenger: Messenger
- mService: Messenger
- mRobName: String
- mRobBound: boolean
- mHandler: Messenger
- mHandlerBound: boolean
- listener: IComponentListener
# ASimpleAPIComponent(context: Context, clientType: ServiceCommand, clientClass: Class, logTag: String, listener: IComponentListener)
+ handleMessage(msg: Message): void
# handleHandlerMessage(msg: Message): void
# onStartUp(): void
+ bind(robnName: String): void
+ unbind(): void
- onConnectedService(service: Messenger): void
- onDisconnectedService(): void
- send(dest: Messenger, msg: Message): void
- sendService(msg: Message): void
- sendService(cmd: ServiceCommand, data: Bundle): void
- sendService(cmd: ServiceCommand): void
# sendHandler(msg: Message): void
# sendHandler(action: Enum, data: Bundle): void
# sendHandler(action: Enum): void

```

Figura 4.7: Clase ASimpleAPIComponent.

comunicar, o *RoboboManager* para que obteña os módulos que precisa para funcionar, e as clases deses módulos. E para os componentes, cuxa instanciação depende xa do xestor que se explica na seguinte subsección, precisase o contexto para poder enlazarse ao servizo, o comando que representa ao componente, a clase do fillo para poder notificar ao observador de que o componente está listo, e o propio observador; que como vemos na figura 4.7 debe implementar a interface *IComponentListener*, que cuxo diagrama UML podemos ver na figura 4.8.

Outra cousa a comentar da clase *AActionHandler* son os dous métodos abstrac-



Figura 4.8: Interface `IComponentListener` para a notificación do estado dun componente.

tos: `onStartUp()` e `onShutDown()`. Estes métodos deben ser implementados polos fillos de `AActionHandler` coa intención de ser notificados cando se solicite o inicio ou finalización dos mesmos por parte do componente cliente, de maneira que poidan configurarse para empezar a recibir accións. Do mesmo xeito, a clase `ASimpleAPI-Component` conta co método `onStartUp()` para o mesmo fin.

Ademais dos métodos para a configuración interna de handlers e componentes, ambas clases abstractas contan con dous métodos análogos: `handleHandlerMessage()` e `handleClientMessage()`. Que como o lector xa se puido imaxinar polo seu nome, teñen como fin procesar as mensaxes do handler e do cliente respectivamente.

Por último, xa que pode desconcertar ao lector, débese comentar que o atributo `mCmdValues` que conteñen ambas clases abstractas ten a misión de almacenar os comandos de tipo `ServiceCommand` que poden recibir do seu análogo. Non obstante, o proceso de comunicación entre eles, e a razón de ser deste atributo, explicaranse na subsección Comunicación cliente-servidor da páxina 45.

#### 4.3.3.2. Xestor de componentes

Outro dos elementos que intervén entre as aplicacións do usuario e os módulos do Robobo e o `SimpleRoboboManager`, que é o encargado de instanciar e iniciar os componentes solicitados. O seu obxectivo consiste en centralizar todo o proceso de inicio dos módulos. O programador podería traballar directamente cos componentes sen necesidade de empregar este xestor, pero tamén debería encargarse de manexar os callbacks dos componentes; cando un componente se inicia, este require un pequeno espazo de tempo entre que recibe a orde de inicio e está preparado para executar métodos. Unha vez remata todo este proceso de configuración, o componente avisa do seu estado a través da interface `IComponentListener` que todo usuario do componente debe implementar. Se o proceso transcorre de maneira normal, execútase o método `onComponentStarted(component: Class)`; e en caso contrario, `onComponentError(component: Class, errorMsg: String)`. Debido a que calquera aplicación

non trivial ha precisar máis dun componente, e iniciar os módulos de maneira secuencial manexando un callback por cada un vólvese unha tarefa molesta e nada recomendable para quen empeza a programar en Java, decídese crear este xestor seguindo o patrón de instancia única, e cuxa representación en UML podemos ver na Figura 4.9.

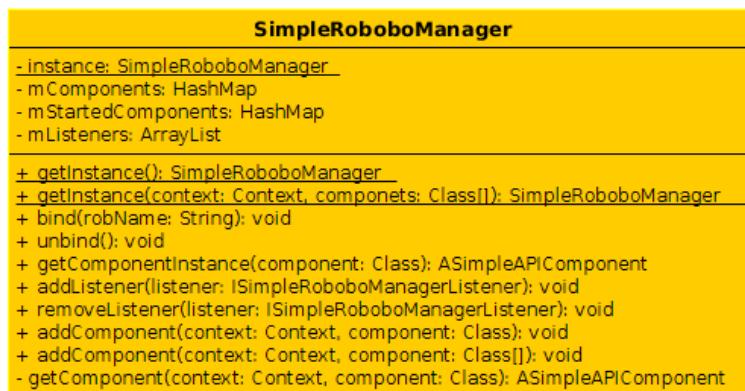


Figura 4.9: Diagrama UML do xestor de componentes.

O usuario deberá obter o xestor, establecer os módulos que vai empregar, e executar o método *bind*. Ademais, para ser informado do final da fase de inicio de tódolos componentes, unha clase deberá implementar a interface *ISimpleRoboboManagerListener*, e engadirse para escoitar o resultado. Se todo ocorre correctamente durante o inicio de tódolos componentes solicitados, execútase o método *onComponentsStarted()*; e en caso contrario, *onComponentsError()*.



Figura 4.10: Interface *ISimpleRoboboManagerListener* para a notificación do estado dos componentes.

Débese destacar tamén que este xestor está inspirado no que existe no framework actual do Robobo, *RoboboManager*, que funciona dun xeito moi semellante; de feito, a intención de crear este xestor, ademais do exposto anteriormente, radica en ir afacendo ao novo usuario ao sistema empregado no framework actual para así facilitar o seu progreso dunha API á outra.

#### 4.3.3.3. Comunicación cliente-servidor

Anteriormente, cando se falaba dos componentes e os handlers, comentabamos que estes eran unicamente os encargados de procesar os parámetros dos métodos que se desexaban executar, e que todo o proceso de intercambio de mensaxes quedaba baixo a responsabilidade das súas súper-clases: *ASimpleAPIComponent* e *AActionHandler*. Cuxa representación UML podemos ver nas figuras 4.6 e 4.7 na páxina 42.

Dende a clase *ASimpleAPIComponent* cada componente xestiona a súa conexión co servizo sendo cada un deles un cliente, de maneira que é o propio servizo o que se inicia ou detén automaticamente cando hai clientes enlazados; o primeiro que deben facer ambas partes é intercambiar o seus mensaxeiros a fin de ter unha interface para o paso de mensaxes.

Cada mensaxe, entre outros, contén un campo *what* de tipo enteiro que se emprega para enviar un comando ou acción, un de tipo *Bundle* onde engadir pares chave-valor que se vai empregar para pasar os parámetros das acciones, e un de tipo *Messenger* onde establecer o remitente da mensaxe. Os comandos veñen definidos na clase *ServiceCommand* de tipo enumerado, onde ademais dos comandos básicos de conexión como *OK*, *ERROR*, *CLIENT* e *ROB\_NAME*; temos comandos para a solicitude dos componentes: *CORPORAL\_EXPRESSION*, *FACIAL\_EXPRESSION*, etc; e para o inicio e detención de cada un deses componentes: *STARTUP*, *SHUTDOWN*. Pero debido a que o campo *what* das mensaxes é un enteiro, o que empregaremos será o código hash de cada un dos comandos.

Se acordamos do último que se expoña na subsección Comunicación cliente-servidor sobre o atributo *mCmdValues* que tiñan as clases abstractas *AActionHandler* e *ASimpleAPIComponent*, onde se dixo que tiña como misión almacenar os comandos de tipo *ServiceCommand* que podían recibir, poderemos entender mellor agora a razón de ser de esta variable. Xa que imos empregar o código hash de cada un dos valores dun tipo enumerado, e non sería viable calculalo no momento para tódalas posibilidades, o que se vai facer é obter tódolos valores do tipo en cuestión, e calcular os seus códigos para almacenalos neste atributo en forma de pares chave-valor coa chave igual ao hash calculado; de xeito que cando se reciba unha nova mensaxe so sería necesario comprobar nesta estrutura de datos cal é o valor correspondente a ese código. Este proceso ademais vai ser o mesmo para tódalas clases que precisen comunicarse a través deste protocolo empregando comandos ou acciones definidas nun tipo enumerado.

Continuando coa explicación sobre a recepción de mensaxes, o mensaxeiro esixe

no momento da instanciación que se estableza un *Handler* –non confundir cos handlers do protocolo– para procesar as mensaxes entrantes. Esta clase define o método *handleMessage(msg: Message)* que se ha chamar dende o mensaxeiro cada vez que chegue unha mensaxe. Xa que todo elemento que empregue este mecanismo debe implementar este método a través da clase *Handler*, e dado que non sería posible facer que o servizo estendera dita clase porque estaría traballando con herdanza múltiple, decidiuse crear a clase *IncomingHandler* para realizar esta acción independente de quen sexa o receptor. Cada *IncomingHandler* ten asociado un observador de tipo *IMessengerHandler* ao que se lle van retransmitir as mensaxes recibidas para procesalas, de maneira que o receptor só precisa implementar dita interface, e evitando o problema da heranza múltiple.

O proceso de inicio sería o seguinte:

- O compoñente enlázase co servizo que proporciona *RoboboSimpleAPIService*.
- Se todo ocorre correctamente, o servizo envía o seu mensaxeiro.
- Empregando o mensaxeiro recibido, o compoñente envía unha petición *CLIENT* que inclúe o seu mensaxeiro, de xeito que o servizo poida responder.

Rematado este proceso, xa existe comunicación entre ambas partes, polo que o seguinte paso consiste en enlazar a base do robot. Para isto, o servizo envía o comando *ROB\_NAME* solicitando o nome do ROB que se desexa empregar, ao que o cliente responde co mesmo comando e indicando o nome dentro do *Bundle* da mensaxe con chave *ROB\_NAME.name()*. Se non hai ningún erro, *RoboboSimpleAPIService* responde co comando *OK*, indicando que o framework do Robobo se iniciou correctamente, e que o robot xa ten base enlazada. Non obstante, se nalgún destes pasos xorde algún problema, o servizo responderá cunha mensaxe co comando *ERROR*, indicando o motivo do mesmo nun campo do *Bundle* con chave *ERROR.name()*, de xeito que o compoñente poida notificar a causa do erro ao usuario a través do método *onComponentError()* da interface *IComponentListener*.

Unha vez o compoñente ten comunicación co servizo e se iniciou correctamente o framework do robot, este debe solicitar o módulo ao que representa por medio doutro comando. O servizo pola súa parte, ha ser o encargado de seleccionar o *AActionHandler* concreto que debe instanciar, e de poñelo en contacto co cliente que o solicita; todo handler recibe como parámetro no construtor o mensaxeiro do compoñente co que se debe comunicar, de xeito que toda a comunicación entre eles

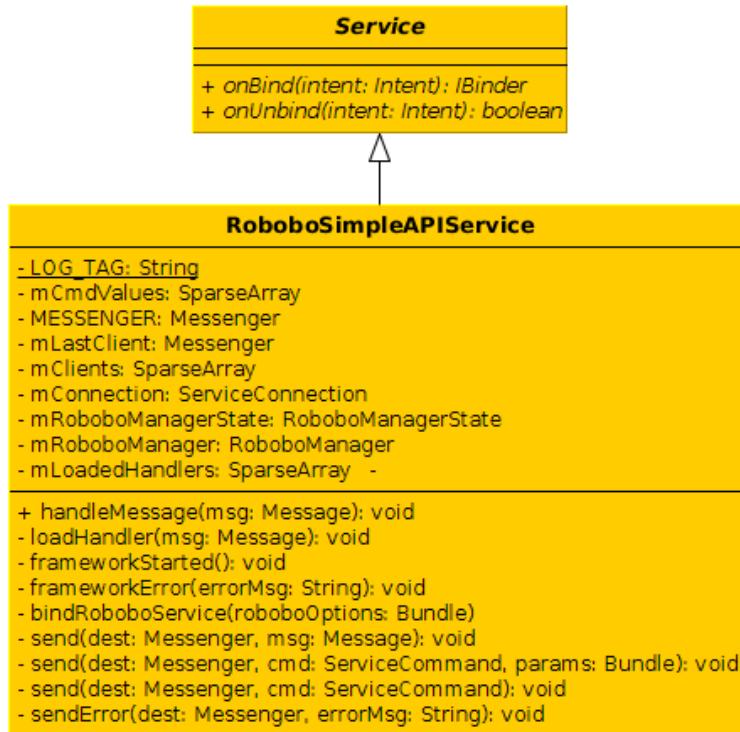


Figura 4.11: Servizo: RoboboSimpleAPIService.

vaia por outra canle, e o servizo sexa soamente un intermediario que xestioná o inicio de tódalas conexións.

Por último, e para ter o compoñente listo para executar métodos, unha vez remata o proceso de instanciación do handler, e como último paso do mesmo, este ha enviar unha petición *CLIENT*, que inclúe o seu mensaxeiro, ao compoñente co que se vai comunicar. Debido a que todo o protocolo de paso de mensaxes está encapsulado en *AActionHandler* e *ASimpleAPIComponent*, este último deberá enviar unha petición de *STARTUP* ao seu análogo para poder executar accións, de xeito que o handler prepare os módulos (*IModule*) Robobo necesarios, e retransmita as mensaxes recibidas que non sexan do tipo *ServiceCommand* á súa subclase. Se todo ocorre correctamente, o handler responde cun *OK*, e o procesamento de mensaxes no compoñente pasa da súper-clase á subclase. A secuencia de inicio para o compoñente de expresión corporal que se pode ver na figura 4.12, daría como resultado o seguinte *log*:

```
I/CorporalComponent(25471): Connected to Simple API Service
D/RoboboSimpleAPIService(25513): handleMessage(): CLIENT
```

D/CorporalComponent(25471): handleMessage(): ROB\_NAME  
 D/RoboboSimpleAPIService(25513): handleMessage(): ROB\_NAME  
 D/RoboboSimpleAPIService(25513): Starting Face  
 I/RoboboSimpleAPIService(25513): frameworkStarted  
 D/CorporalComponent(25471): handleMessage(): OK  
 D/RoboboSimpleAPIService(25513): handleMessage(): CORPORAL\_EXPRESSION  
 D/RoboboSimpleAPIService(25513): loadHandler(): CORPORAL\_EXPRESSION  
 D/CorporalComponent(25471): handleMessage(): CLIENT  
 D/CorporalHandler(25513): handleMessage(): STARTUP  
 D/CorporalComponent(25471): handleMessage(): OK  
 D/AICorporalExpression(25471): Robobo started

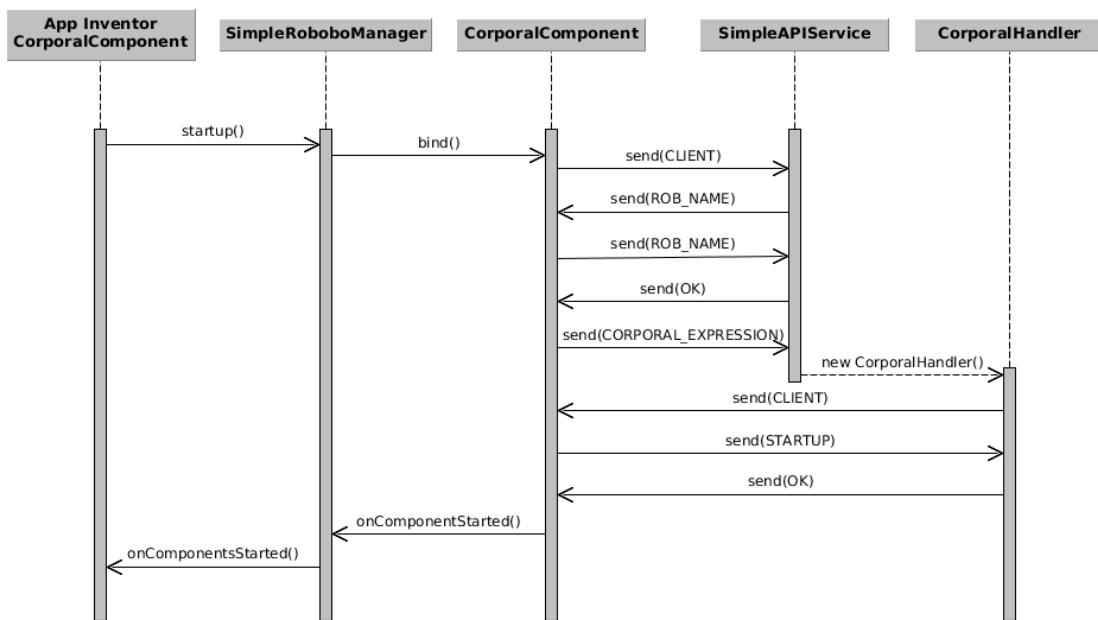


Figura 4.12: Secuencia de inicio dun compoñente.

Ademais do comando *STARTUP*, tamén existe *SHUTDOWN*, que vai facer todo o contrario que o primeiro, e pechar a comunicación que permite a execución de comportamentos. Ao recibir esta petición, o handler deixará de procesar as mensaxes recibidas no método *handleClientMessage(msg: Message)* da subclase, e volverá a procesalas todas no método *handleMessage(msg: Message)* da clase abstracta. Non obstante, o envío da petición de *shutdown* non é estritamente necesario, pois nin se descargan os módulos do *RoboboManager*, nin se deteñen, pero sí hai compoñentes

que quedarían facendo algunha tarefa en segundo plano de non facelo; como o de visión, que quedaría capturando imaxes ata que o servizo se detivese. Tanto o *STAR-TUP* como o *SHUTDOWN* son enviados por *ASimpleAPIComponent* ao executar os métodos *bind()* e *unbind()* respectivamente.

Outra cousa que destacar da comunicación entre handlers e compoñentes é o paso de parámetros ou devolución de resultados tras a execución de métodos. Xa se comentara anteriormente que o paso desta información facíase a través do *Bundle* que conteñen as mensaxes, que permitía establecer pares chave-valor para diferentes tipos de datos. Tamén se comentou que cada acción tiña asociada uns parámetros que establecían certos detalles internos das mensaxes; ao que se referían eses detalles internos son ás chaves que se deben empregar para o paso de información a través do *Bundle*, que se deben definir nun tipo enumerado xunto coas accións de cada compoñente.

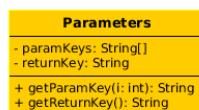


Figura 4.13: Clase Parameters.

#### 4.3.3.4. Compoñentes de App Inventor

Como xa se comentou anteriormente, App Inventor conta cun modelo que se esixe seguir para desenvolver novos compoñentes; a través das clases abstractas *AndroidVisibleComponent* e *AndroidNonVisibleComponent*, o desenvolvedor pode implementar os seus compoñentes dependendo de se precisan amosar algo por pantalla ou non. Neste traballo, ademais, decidiuse implementar unha clase abstracta chamada *RoboboComponent* que todo compoñente do segundo nivel de abstracción da API simplificada debe estender, de xeito que se integran nela algúns fragmentos de código que doutra maneira habería que repetir en tódolos compoñentes. Ademais, é requisito imprescindible para o correcto funcionamento dos fillos desta clase que implementen tódolos métodos públicos definidos nesta, e engadan unha chamada ao mesmo método da súper-clase como última instrución. Podemos ver un exemplo de como quedarían algúns dos bloques na figura 4.14.

Como vemos na representación UML da clase *RoboboComponent* da figura 4.15, a clase encapsula a interacción co *SimpleRoboboManager* e o inicio e finalización dos compoñentes. Ademais, algo que pode sorprender ao lector, os métodos da interface



Figura 4.14: Bloques dalgúns dos compoñentes de App Inventor para Robobo.

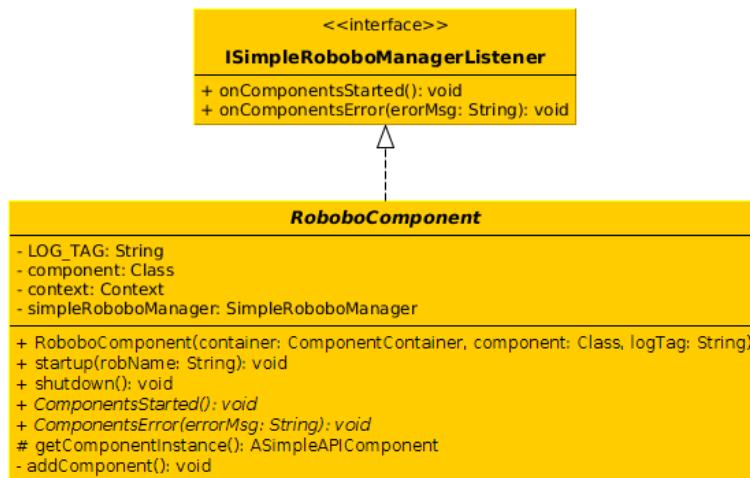


Figura 4.15: Clase de App Inventor RoboboComponent.

*ISimpleRoboboManagerListener* parecen duplicados con outro nome. A razón disto é que de non facelo así, se o programador non emprega o bloque que representa a eses eventos, áinda que sexa baleiro, o código Java desta función non se executaría; polo que se decidiu crear os métodos *ComponentsStarted* e *ComponentsError*, de xeito que independentemente de se o programador precisa executar algo cando se produzcan eses eventos, o código dos métodos da interface *ISimpleRoboboManagerListener* que implementa *RoboboComponent* se execute igualmente. O obxectivo é forzar que todo compoñente teña un método para recibir estes eventos, que han ser disparados dende *onComponentsStarted* e *onComponentsError* como vemos no seguinte fragmento de

código:

```

@Override
public void onComponentsStarted() {
    Log.d(LOG_TAG, "Robobo started");
    EventDispatcher.dispatchEvent(this, "ComponentsStarted");
}

@Override
public void onComponentsError(String errorMsg) {
    Log.e(LOG_TAG, "Robobo error: " + errorMsg);
    EventDispatcher.dispatchEvent(this, "ComponentsError", errorMsg);
}

```

Respecto deste último fragmento de código, a función *EventDispatcher.dispatchEvent()* é a que nos permite producir os eventos, aos que tamén podemos engadir unha lista de parámetros de tamaño variable despois de indicar quen dispara o evento e cal –*errorMsg* por exemplo–. Polo demais, os compoñentes de App Inventor son clases Java normais; as únicas características a maiores son que toda función, evento ou propiedade –atributos ou variables dos obxectos Java– debe estar indicado coa anotación *@SimpleFunction*, *@SimpleEvent* ou *@SimpleProperty*; e que só se permiten tipos primitivos e cadeas para os parámetros dos métodos da clase –nin sequera arrays–. Ademais, todos estes elementos, e o compoñente en si, deben declarar o seu nome no ficheiro *OdeMessages* da seguinte maneira:

- Compoñentes:

```

@DefaultMessage("RoboboCorporalExpression")
@Description("")
String roboboCorporalExpressionComponentPallette();

@DefaultMessage("")
@Description("")
String RoboboCorporalExpressionHelpStringComponentPallette();

```

- Métodos:

```

@DefaultMessage("startup")
@Description("")
String startupMethods();

```

- Eventos:

```

@DefaultMessage("ComponentsStarted")
@Description("")
String ComponentsStartedEvents();

```

- Propiedades:

```
@DefaultMessage("Happy")
@Description("")
String HappyProperties();
```

Ademais, de querer notificar un erro mediante unha pequena mensaxe empregaremos a función:

```
form.dispatchErrorOccurredEvent(this, functionName,
    ErrorMessages.ERROR_ROBOBO_COMPONENT_NOT_FOUND,
    e.getMessage());
```

Onde deberemos indicar como parámetros o obxecto que fai a chamada, o nome da función, o erro, e unha mensaxe opcional. Ademais, os erros débense definir coma constantes da clase *ErrorMessages*, onde deberemos asignar un número enteiro que o defina, e engadir esa constante máis unha cadea de texto que explique o erro ao mapa *errorMessages*. A continuación amosamos un exemplo no que a cadea que describe o erro recíbese por parámetros:

```
public static final int ERROR_ROBOBO_COMPONENT_NOT_FOUND = 3300;

errorMessages.put(ERROR_ROBOBO_COMPONENT_NOT_FOUND, "%s");
```

Por último débese destacar a diferencia de uso con respecto á implementación en Java, e é que en esta o usuario non interactúa directamente co *SimpleRoboboManager*, senón que isto se fai dende a clase *RoboboComponent*, de xeito que resulta transparente para o programador. Botemos un ollo ao código das funcións *startup* e *shutdown* de *RoboboComponent*:

```
public void startup(String robName) {
    String functionName = "startup";
    simpleRoboboManager.addListener(this);
    simpleRoboboManager.bind(robName);
}

public void shutdown() {
    simpleRoboboManager.removeListener(this);
    simpleRoboboManager.removeComponent(component);
}
```

Como podemos ver, baixo estas funcións queda oculta a interacción co *SimpleRoboboManager* de xeito que unha única chamada ao *startup* dun compoñente iniciará automaticamente tódolos demais. Con esta solución procurouse evitar que o programador tivera que iniciar cada un dos compoñentes por dous motivos: para un usuario sen experiencia é máis fácil acordar de facelo unha vez para todos, e

ademas así evítase que o espazo de traballo cos bloques se vexa reducido debido aos múltiples *startup*.

## 4.4. Deseño e implementación

### 4.4.1. Metodoloxía

Para a implementación deste proxecto optouse por unha metodoloxía iterativa cuxos pasos consisten en implementar e probar un ou varios compoñentes do primeiro nivel de abstracción, e unha vez comprobado que todo o funcionamento é correcto implementar o segundo nivel conectando App Inventor cos compoñentes do primeiro nivel desenvolvidos anteriormente. Concretamente, hase seguir a seguinte orde:

- Primeira iteración: daremos soporte ao control do ROB, xa que realmente é este compoñente o que converte a combinación base motorizada e móbil nun robot.
- Segunda iteración: implementaranse os compoñentes necesarios para dotar ao Robobo do sentido da vista, xa que debido as dependencias do módulo básico para o control da cámara, que debe capturar imaxes en segundo plano mentres outro módulo as procesa, tamén é un dos que máis incerteza crea respecto de posibles problemas que poidan aparecer; polo que neste paso hanse desenvolver visión, expresión facial, e tacto, xa que tanto a visión como o tacto dependen directamente da actividade que amosa as caras do robot.
- Terceira iteración: implementaranse tódolos compoñentes relacionados tanto coa percepción, como coa expresión de sons. Igual que pasaba na iteración anterior, esta implementación é a que máis incerteza crea, de entre os módulos restantes, debido a súa complexidade.
- Cuarta iteración: realizarase o desenvolvemento dos compoñentes restantes, dando soporte ao acelerómetro, xiroscopio, niveis de batería, e nivel de luz ambiente. A razón de agrupar todos estes módulos nunha soa iteración é debido que son os más sinxelos e con menor número de funcións de todos, polo que é innecesario facelo en máis dunha iteración.

Debemos destacar tamén que non se ha dar soporte nin ao módulo de recoñecemento da fala, nin ao de mensaxes; en ambos casos debido a que dada a súa implementación actual, e á estrutura da nova API, non sería posible. No caso do módulo

de mensaxes, a razón é que precisa un ficheiro coas credenciais do lado do servidor, de xeito que o programador non tería control sobre el; e no caso do recoñecemento da fala, debido a que non pode haber dous módulos empregando o micrófono, xa que entraría en conflito co resto dos módulos do oído, e non funcionaría. Ademais, en ambientes ruidosos coma pode ser unha aula, o módulo de recoñecemento non aporta os resultados desexados.

Respecto das probas, primeiro desenvolverase unha proba básica para cada compoñente implementado na iteración para comprobar mediante mensaxes de log que todo funciona correctamente, e despois unha proba completa da iteración a través dunha aplicación que empregue tódolos compoñentes desenvolvidos. Ademais, para facilitar a execución das probas, todas elas han ser incluídas nunha aplicación que permita inicialas a través de botóns.

Por último, antes de expoñer o proceso seguido durante o desenvolvemento das iteracións, é importante destacar a importancia do deseño arquitectónico adoptado, xa que facilita moito a creación de handlers e compoñentes, e é realmente a peza chave que fai que este proceso sexa tan sinxelo; pois evita que durante o seu desenvolvemento haxa que inverter tempo na implementación da comunicación entre as partes, e soamente sexa necesario centrarse en dar acceso as súas funcionalidades.

## **4.4.2. Primeira iteración**

Nesta primeira iteración, o obxectivo ha ser implementar o protocolo exposto na sección anterior, e o compoñente que permite controlar o corpo do robot.

Na sección anterior vimos que todo compoñente da API simple que estamos a desenvolver neste traballo debe contar con tres piares fundamentais: un *handler*, o compoñente, e as accións. Como xa se comentou anteriormente, o compoñente envía accións a través do servizo para que o handler as procese e execute os métodos do módulo de Robobo correspondente. Na figura 4.16 podemos ver a representación UML desta primeira iteración.

### **4.4.2.1. Expresión corporal**

#### **4.4.2.1.1 Deseño funcional**

O primeiro a escoller son os métodos da interface IRob aos que daremos acceso dende esta API, e crear a clase enumerada con tódalas accións permitidas para proporcionar ditas funcionalidades.

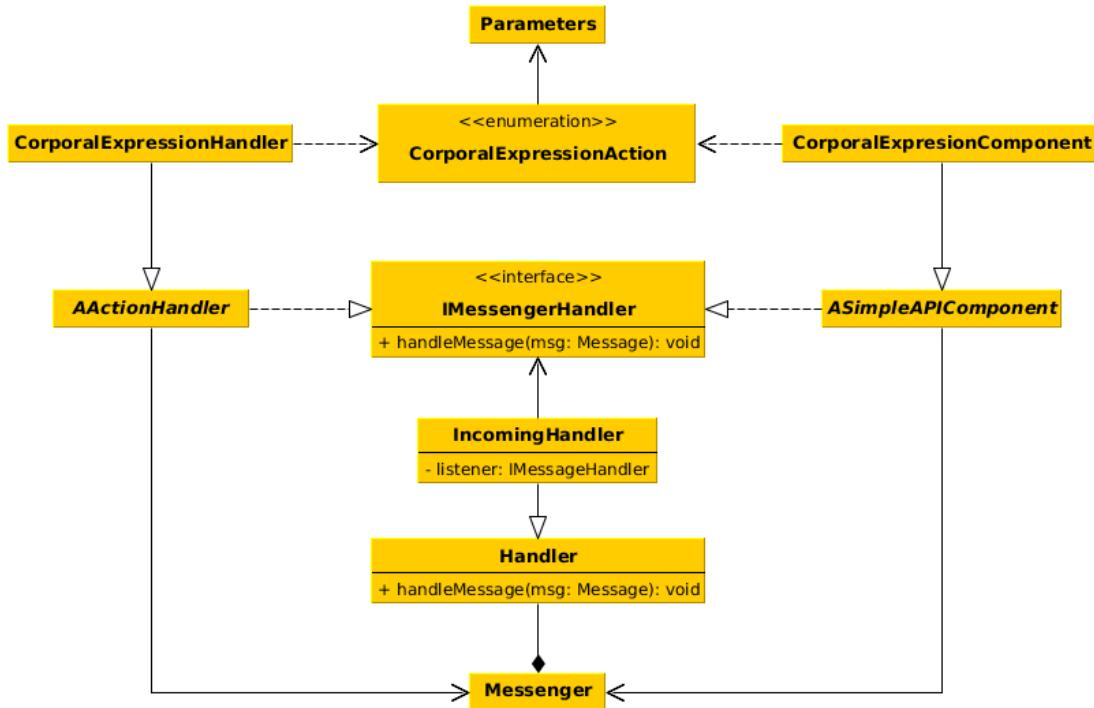


Figura 4.16: Comunicación handler-compoñente na primeira iteración.

Se acordamos do descrito no capítulo de fundamentos tecnolóxicos, o módulo que permite o control do ROB, ademais de recibir comandos de movemento, para o control dos LEDs, ou para certo tipo de configuracións; tamén permitía coñecer a información dos sensores infravermellos. Ademais, durante a execución de calquera dos métodos do módulo de movemento podería producirse algúun erro que sería necesario notificar, polo que no conxunto de funcións que se han definir para este compoñente, imos ter accións tanto nun sentido –compoñente-handler–, coma no outro –handler-compoñente–.

A continuación describiremos cada unha das accións definidas que podemos ver na figura 4.17 indicando tamén o nome real que reciben:

- *SET\_LED\_COLOR*: para cambiar de cor os LEDs do ROB.
- *MOVE\_MOTOR\_DEGREES*: para mover os motores un número de grados determinado.
- *MOVE\_MOTOR\_TIME*: para mover os motores durante un período de tempo determinado.
- *MOVE\_PAN*: para rotar o soporte do móbil.

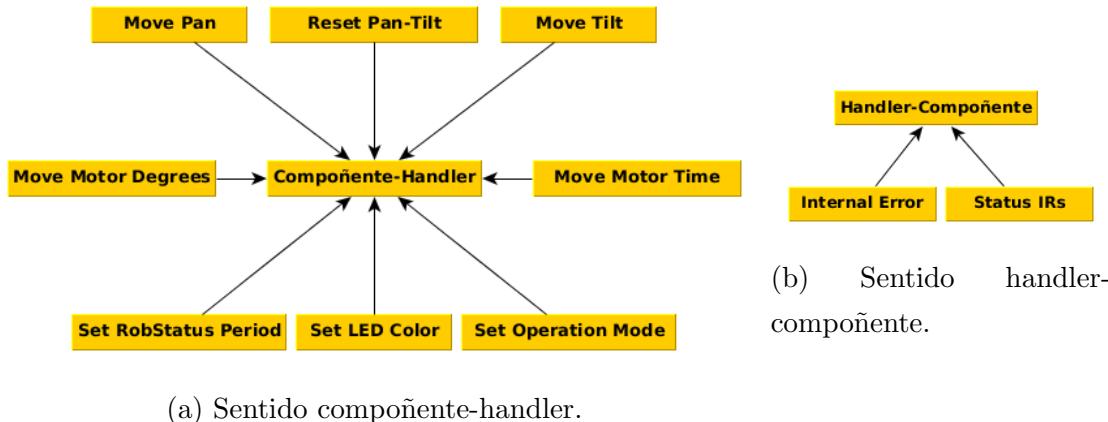


Figura 4.17: Acciones de expresión corporal.

- *MOVE\_TILT*: para inclinar o soporte do móvil.
- *RESET\_PAN\_TILT*: para accionar a secuencia de calibración do soporte do móvil.
- *SET\_ROB\_STATUS\_PERIOD*: para establecer o período de refresco da información de sensores.
- *INTERNAL\_ERROR*: que permite ao handler informar de posibles errores internos do robot.
- *STATUS\_IRS*: para que o compoñente reciba a información dos infravermellos.

#### 4.4.2.1.2 Primeiro nivel de abstracción

Unha vez definidas as funcións do módulo de control da base, o seguinte paso debe ser a implementación do handler e o compoñente de expresión corporal, que como xa se dixo durante a descripción do deseño conceptual, estes soamente cumpren a función de empaquetar os parámetros necesarios para a execución dos métodos. Vexamos un exemplo cunha das funcións dos motores:

```
public void moveMTDegrees(MoveMTMode mode, int angVelR, int angleR, int angVelL, int angleL)
    throws RemoteException {
    Bundle params = new Bundle();
    params.putByte(MOVE_MOTOR_DEGREES.getParameters().getParamKey(0), mode.getMode());
    params.putInt(MOVE_MOTOR_DEGREES.getParameters().getParamKey(1), angVelR);
    params.putInt(MOVE_MOTOR_DEGREES.getParameters().getParamKey(2), angleR);
    params.putInt(MOVE_MOTOR_DEGREES.getParameters().getParamKey(3), angVelL);
    params.putInt(MOVE_MOTOR_DEGREES.getParameters().getParamKey(4), angleL);
    sendHandler(MOVE_MOTOR_DEGREES, params);
```

}

Este é o método que o programador deberá executar para mover os motores do robot indicando os graos de xiro, e que está dispoñible no *CorporalExpressionComponent*. Como podemos ver neste fragmento de código, o método unicamente crea un *Bundle* no que vai inserindo tódolos parámetros que recibe coas chaves establecidas na clase *Parameters* que toda acción posúe, e despois envía a mensaxe ao handler mediante o método *sendHandler()* da súa superclase. Analogamente, *CorporalExpressionHandler* vai facer o seguinte ao recibir a mensaxe:

```
private void moveMTDegrees(Bundle params) throws InternalErrorException {
    Parameters moveMTDegreesParameters = MOVE_MOTOR_DEGREES.getParameters();
    MoveMTMode mode =
        MoveMTMode.toMoveMTMode(params.getByte(moveMTDegreesParameters.getParamKey(0)));
    Integer angVel1 = params.getInt(moveMTDegreesParameters.getParamKey(1));
    Integer angle1 = params.getInt(moveMTDegreesParameters.getParamKey(2));
    Integer angVel2 = params.getInt(moveMTDegreesParameters.getParamKey(3));
    Integer angle2 = params.getInt(moveMTDegreesParameters.getParamKey(4));
    rob.moveMT(mode, angVel1, angle1, angVel2, angle2);
}
```

Da mesma maneira que no compoñente, o único que se fai é procesar os parámetros e executar o método correspondente da interface *IRob* do robot.

Outra cousa a ter en conta é que todo método da interface para controlar o ROB pode producir a excepción *InternalErrorException*, que indicaría calquera erro relacionado con algo interno da base ou coa conexión bluetooth entre o móvil e o ROB. De se producir, o *CorporalExpressionHandler* ha notificar dito fallo ao compoñente a través da acción do mesmo nome: *INTERNAL\_ERROR*. A través dela tamén se notificará ao compoñente o motivo da excepción enviando a mensaxe que esta devolve; débese recordar que os erros non teñen un obxecto parámetros asociado, posto que se emprega o propio nome da acción como chave.

Ademais de informar sobre os erros ocorridos durante a execución, é responsabilidade do handler de expresión corporal notificar sobre o estado dos sensores de xeito que o compoñente poida comunicar dita información ao programa do usuario, e este teña capacidade para reaccionar ante obstáculos e outras dificultades do terreo. Para isto, o handler debe implementar a interface *IRobStatusListener* e engadirse como observador. Unha vez recibidos no handler, todos estes datos son transmitidos periodicamente a través da acción *STATUS\_IRS* que se expoñía anteriormente segundo o intervalo de tempo establecido polo usuario a través da acción *SET\_ROB\_STATUS\_PERIOD*.

Debido a que o programa do usuario debe ser informado tanto de posibles erros

## 4.4. DESEÑO E IMPLEMENTACIÓN CAPÍTULO 4. DESENVOLVEMENTO

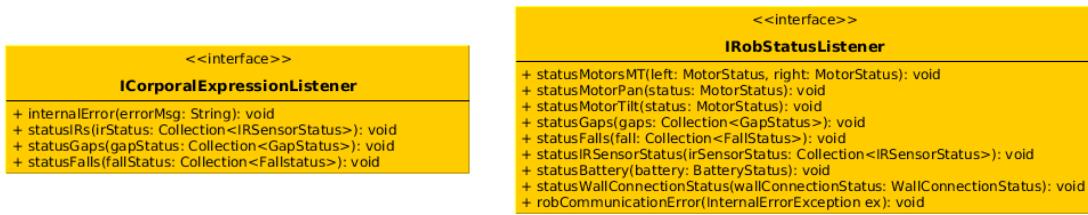


Figura 4.18: Interfaces para a recepción da información de sensores.

que se produzan durante a execución, como do estado dos sensores do ROB, calquera elemento que desexe empregar o compoñente de expresión corporal, e recibir notificacións de sensores ou erros, deberá estipular mediante o método *addListener()* que clase vai ser a encargada de procesar toda esa información. Isto faise por medio da interface *ICorporalExpressionListener*, cuxa representación UML podemos ver na figura 4.18.

Tralo desenvolvemento deste compoñente, a fachada coa que traballaría o usuario quedaría como se pode ver na figura 4.19, do que debemos destacar as funcións para cambiar as cores dos LEDs, e as que permiten o movemento da base.

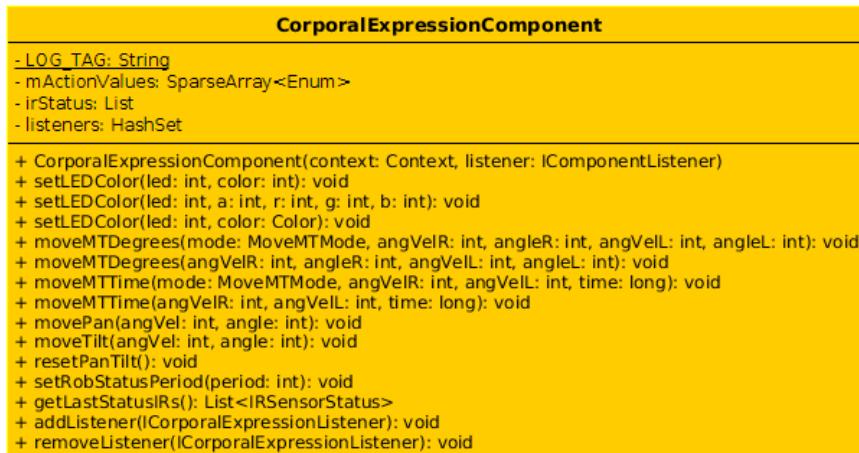


Figura 4.19: Expresión corporal no primeiro nivel de abstracción.

Para acceder á matriz de LEDs contamos con tres funcións. A primeira que vemos na figura 4.19 está deseñada para ser empregada con App Inventor, xa que para poder empregar as cores que define é necesario poder facelo a través dun enteiro que represente a cor como ARGB. A segunda require os valores ARGB por separado, de xeito que o usuario poida experimentar facilmente con diferentes compoñentes. E a terceira ten a mesma interface que o método que realiza esta función no módulo

do Robobo, que emprega o tipo *Color* propio do framework.

Por outra banda, para mover o ROB contamos con catro funcións. Actualmente a versión do módulo do Robobo que se está a emplegar para o desenvolvemento conta cos métodos *MoveMTDegrees* e *MoveMTTime*, que permiten o movemento en base aos graos de xiro ou ao tempo, e cuxa interface coincide coas que podemos ver na figura 4.19 co parámetro *mode*: *MoveMTMode*, que permite indicar o sentido do movemento. Adicionalmente, nesta implementación decidiuse engadir outros dous métodos que non requeriran deste parámetro debido a consideralo innecesario, xa que o sentido da marcha pódese establecer mediante velocidades de xiro negativas. Ademais, está previsto que posteriores versións do módulo de control do ROB cambien de interface e prescindan do campo *mode*.

Finalmente, tralo desenvolvemento do compoñente de expresión corporal deste primeiro nivel de abstracción, o diagrama de clases e relacións quedaría como podemos ver na figura 4.20; onde debemos destacar que a clase *RoboboCorporalExpression* representa ao programa do usuario ou ao compoñente de App Inventor que permite controlar o ROB.

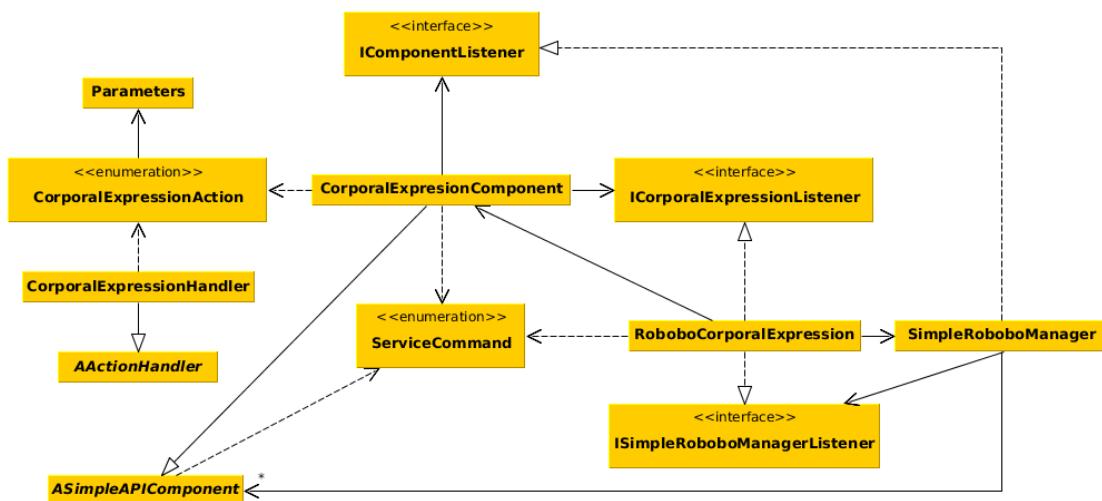


Figura 4.20: Primeira iteración.

#### 4.4.2.1.3 Segundo nivel de abstracción

Como comentabamos na subsección Compoñentes de App Inventor da páxina 49, todo compoñente de App Inventor debe estender a clase abstracta *RoboboComponent*, que define unha serie de métodos comúns que todo compoñente precisa: *startup*,

*shutdown*, *onComponentsStarted* e *onComponentsError*, que se deben sobrescribir; e *ComponentsStarted* e *ComponentsError*, que se deben implementar. Recordemos que a razón pola que hai dous pares de componentes con nomes parecidos é debido a que en App Inventor, se non se emprega o bloque dun evento, o seu código non se ha executar; polo que se optou por duplicalos e así evitar ao programador novel a responsabilidade de empregalos áinda que o seu programa non os precise. Desta maneira aproveítase o métodos *onComponentsStarted* para configurar o componente de App Inventor e chamar ao evento que si empregará o usuario no seu programa: *ComponentsStarted*. A continuación amosamos o seu código:

```

@Override
public void onComponentsStarted() {
    corporalComponent = (CorporalExpressionComponent)
        getComponentInstance();
    corporalComponent.addListener(this);
    super.onComponentsStarted();
}

```

Outra das cousas que se comentaban durante a descripción dos componentes de App Inventor é que tódolos métodos públicos da clase *RoboboComponent* deben ser sobrescritos, e ademais chamar ao método da súperclase como última instrución, tal e como podemos ver neste último fragmento de código.

<b>RoboboCorporalExpression</b>
- <b>LOG_TAG</b> : String - corporalComponent: CorporalExpressionComponent - irDistance: float[]
+ RoboboCorporalExpression(container: ComponentContainer) + startup(robName: String): void + shutdown(): void + onComponentsStarted(): void + ComponentsStarted(): void + onComponentsError(errorMsg: String): void + ComponentsError(errorMsg: String): void + setLEDColor(led: int, color: int): void + moveMTDegrees(angVelR: int, angleR: int, angVelL: int, angleL: int): void + moveMTTime(angVelR: int, angVelL: int, time: long): void + movePan(angVel: int, angle: int): void + moveTilt(angVel: int, angle: int): void + resetPanTilt(): void + setRobStatusPeriod(period: int): void + getDistance(sensor: int): float + internalError(errorMsg: string): void + statusUpdated(): void + statusIRs(coll: Collection<IRStatus>): void

Figura 4.21: Expresión corporal no segundo nivel de abstracción.

O resto das funcións da expresión corporal no segundo nivel de abstracción son simples chamadas ao método do *CorporalExpressionComponent* do nivel anterior,

excepto *getDistance*; que é unha función para facilitar o acceso aos datos dos infravermellos do atributo *irDistance*, cuxos valores son actualizados cada vez que se recibe o evento *statusIRs*. A razón de ser desta función, así como do evento *statusUpdated*, é que o método *statusIRs* recibe como parámetro un tipo de dato non permitido para os bloques de App Inventor, polo que ademais de crear o evento *statusUpdated* para que o usuario conte cun bloque que notifique a actualización do estado dos infravermellos, tamén é necesario establecer un mecanismo para acceder dun en un a eses valores.

```

@SimpleEvent
public void statusUpdated() {
    Log.d(LOG_TAG, "statusUpdated()");
    EventDispatcher.dispatchEvent(this, "statusUpdated");
}

@Override
public void statusIRs(Collection<IRSensorStatus> coll) {
    IRSensorStatus[] irStatus;
    irStatus = coll.toArray(new IRSensorStatus[coll.size()]);
    irDistance = new float[coll.size()];
    for (int i = 0; i < irDistance.length; i++)
        irDistance[i] = irStatus[i].getDistance();
    statusUpdated();
}

```

No fragmento anterior de código podemos comprobar o funcionamento dos eventos descritos no parágrafo anterior: *statusIRs* recibe e actualiza o contido do atributo que garda o estado dos sensores, e *statusUpdated* dispara o evento para o bloque correspondente.

Poñamos antes de continuar outro exemplo de código, pero desta volta dunha das funcións que xa atopabamos no primeiro nivel de abstracción do compoñente, concretamente a que permite mover os motores por graos:

```

@SimpleFunction
public void moveMTDegrees(int angVelR, int angleR, int angVelL, int
    angleL) {
    String functionName = "moveMTDegrees";
    try {
        if (corporalComponent != null) {
            corporalComponent.moveMTDegrees(angVelR, angleR, angVelL,
                angleL);
        }
    } catch (RemoteException e) {
        form.dispatchErrorOccurredEvent(this, functionName,
            ErrorMessages.ERROR_SIMPLE_API_SERVICE_NOT_AVAILIABLE);
    }
}

```

Para finalizar a descripción deste compoñente de App Inventor, amosaremos na figura 4.22 os bloques cos que pode contar o usuario no segundo nivel de abstracción.



Figura 4.22: Bloques de expresión corporal.

#### 4.4.2.2. Demostración

A proba que se propón para a expresión corporal consiste no desenvolvemento dunha aplicación que faga que o robot sexa capaz de recorrer un labirinto; é dicir, debe ser capaz de avanzar cara adiante entre dúas paredes evitando chocar contra elas. Dado que este é o único compoñente desenvolvido na primeira iteración, o robot empezará o seu movemento xusto en canto se dispare o evento que indica que está preparado: *onComponentsStarted*.

Para cumplir este obxectivo o primeiro paso debe ser determinar uns limiares para cada infravermello, de xeito que nos permita ter unha medida da distancia que hai aos obstáculos, e así poder calcular un coeficiente de axuste da velocidade baseado na información que se recibe de cada infravermello. Xa que só se pretende que o robot avance cara adiante, soamente se han calcular cinco coeficientes de axuste; un para cada sensor dianteiro.

O método empregado para establecer os limiares foi en base a diferentes probas feitas colocando o robot en orientacións diferentes respecto dunha parede que servía de obstáculo, e escollendo unha cota superior despois de observar como variaba a información de sensores recibida durante uns minutos. Segundo este método, o sensor máis fácil de calibrar é o central, cuxo valor é igual ao que devolve ese infravermello co robot colocado mirando á parede deixando a distancia mínima imprescindible para poder xirar sen chocar. Tomando esa posición como referencia, e simulando ese xiro manualmente, buscáronse os ángulos nos que os outros catro sensores ofrecen valores máis altos, e asignouse a cota superior como limiar.

Unha vez calibrados os sensores, definíronse as funcións que van servir de coeficientes de axuste da velocidade como a seguinte fórmula:

$$sensorAdjust = \max(0, \min(1, \alpha \cdot threshold - irDistance))$$

Onde  $\alpha$  é un parámetro que permite establecer unha marxe de erro dependente da velocidade. Non obstante, xa que este é soamente unha demostración do funcionamento da API, decidiuse empregar unha constante.

Tras calcular estes coeficientes de axuste da velocidade que proporciona cada sensor, o seguinte paso foi determinar de que maneira van afectar estes valores á velocidad dos motores; polo que se decide un novo coeficiente para cada motor:  $rcAdj$  para o motor derecho, e  $lcAdj$  para o esquerdo.

Os sensores están numerados empezando polo frontal do lado esquierdo –mirando o ROB dende a parte de atrás–, que está ao lado do motor esquierdo, e seguindo en sentido horario: o 3 sería o sensor central dianteiro, e o 7 sería o central traseiro. Segundo esta numeración, os sensores a ter en conta para a velocidad do motor derecho son o 1, 2 e 3; e para o motor esquierdo o 3, 4 e 5. Debido á sinxeleza e ao seu funcionamiento aceptable, decidiuse que os coeficientes de axuste da velocidad para cada motor tomaran o valor da media aritmética dos coeficientes de axuste que proporciona cada un dos sensores involucrados; tal e como vemos nas seguintes fórmulas:

$$rcAdj = \frac{s1Adj + s2Adj + s3Adj}{3}$$

$$lcAdj = \frac{s4Adj + s5Adj + s3Adj}{3}$$

Con todo isto, xa temos todo o necesario para que o robot se manteña entre dúas paredes sen chocar con elas, pero sen embargo, se dera contra unha esquina en vez de cunha curva más suave, ou contra un obstáculo puntual que soamente fora detectado

polo infravermello central, o robot continuaría a súa marcha ata chocar. Para evitar isto, estableceuse unha condición de xeito que se  $s3Adj$  fora cero, débense axustar novamente os coeficientes  $rcAdj$  e  $lcAdj$  tendo en conta soamente o sensor número un: se este detecta un obstáculo,  $rcAdj = 0$ , e o robot xiraría cara á dereita; e en caso contrario,  $lcAdj = 0$ , e xiraría á esquerda. Ademais, do mesmo xeito que para o calculo dos *sensorAdjust*, para determinar se hai obstáculo engadiuse un parámetro  $\beta$  que permite establecer unha marxe de erro ao limiar do sensor; a continuación amosamos a condición que determina se o sensor 1 detecta ou non un obstáculo:

$$\frac{irDistance}{S1\_THRESHOLD} > \beta$$

Por último, agora que tódolos coeficientes de axuste están calculados, a velocidade resultante para cada motor sería igual ao coeficiente correspondente multiplicado por unha constante que establece a velocidade máxima:

$$angVelR = MAX\_ANG\_VEL \times rcAdj$$

$$angVelL = MAX\_ANG\_VEL \times lcAdj$$

Este algoritmo foi implementado empregando os dous niveis de abstracción desenvolvidos neste traballo. A continuación, coa intención de comparar as virtudes e defectos de ambos niveis, amosaremos ambas implementacións empezando polo primeiro nivel de abstracción:

```

private void adjustSpeed() {
    float alpha = (float) 1.1, // [1,2]
          beta = (float) 0.2;
    float s1Adj = max(0, min(1, (alpha * S1_THRESHOLD - irDistance[0]))),
          s2Adj = max(0, min(1, (alpha * S2_THRESHOLD - irDistance[1]))),
          s3Adj = max(0, min(1, (alpha * S3_THRESHOLD - irDistance[2]))),
          s4Adj = max(0, min(1, (alpha * S4_THRESHOLD - irDistance[3]))),
          s5Adj = max(0, min(1, (alpha * S5_THRESHOLD - irDistance[4]))),
          rcAdj = (s1Adj + s2Adj + s3Adj) / 3,
          lcAdj = (s4Adj + s5Adj + s3Adj) / 3;

    if (s3Adj <= 0) {
        Log.d(LOG_TAG, "Front Obstacle");
        if (irDistance[0] / S1_THRESHOLD > beta) {
            rcAdj = 0;
        } else {
            lcAdj = 0;
        }
    }

    angVelR = round(MAX_ANG_VEL * rcAdj);
    angVelL = round(MAX_ANG_VEL * lcAdj);
}

```

Como podemos comprobar, nunha pequena función de menos de 25 liñas podemos condensar todo o algoritmo explicado anteriormente. Sen embargo, en App Inventor, o cálculo de  $s3Adj$  quedaría como se pode ver na figura 4.23, e o do axuste da velocidade de ambos motores como na figura 4.24.



Figura 4.23: Axuste da velocidade do sensor central dianteiro en App Inventor.

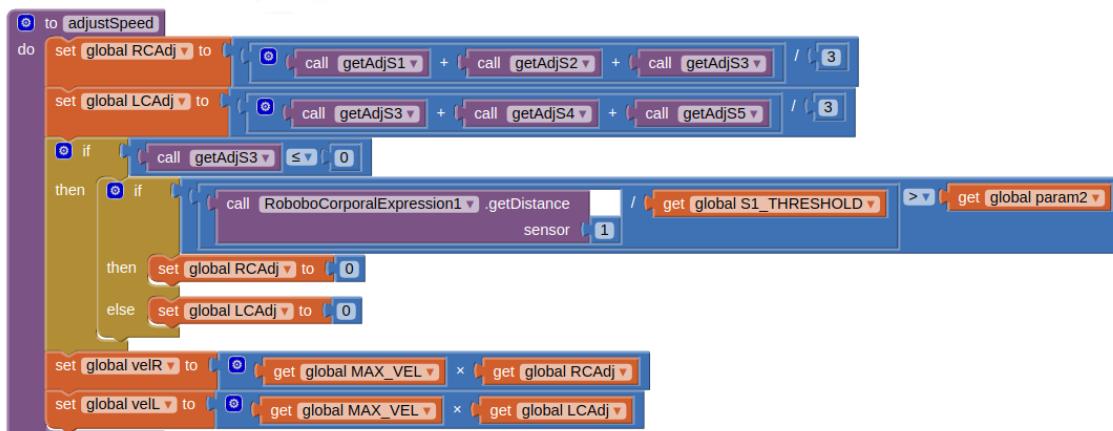


Figura 4.24: Axuste da velocidade dos motores en App Inventor.

Ata o momento estivemos a describir o proceso seguido para o axuste das velocidades dos motores para conseguir que o robot sexa capaz de manterse entre dúas paredes sen bater contra elas, pero sen embargo aínda non se comentou nada de como se facía para enviar as accións de movemento ao robot.

Realmente o desenvolvemento é moi semellante en ambos casos, pois se programa unha tarefa que executa a función *moveMTTime* coa velocidade calculada anteriormente, e unha duración de dous veces o tempo de refresco da información de sensores para tentar que o movemento sexa o máis fluído posible. Podemos ver a función de movemento en Java no seguinte código, e os bloques correspondentes de App Inventor na figura 4.25.

```
protected void startMovement() throws RemoteException {
    final long actionPeriod = 2 * statusPeriod;
    corporal.setRobStatusPeriod(statusPeriod);
    movementStarted = true;
    TimerTask movementTask = new TimerTask() {
```

```

@Override
public void run() {
    if (movementStarted) {
        try {
            corporal.moveMTTime(moveMTMode, angVelR, angVelL, actionPeriod);
        } catch (RemoteException e) {
            Log.e(LOG_TAG, "Error moving ROB: " + e.getMessage());
            timer.cancel();
            stop();
        }
    }
};

timer.schedule(movementTask, 0, statusPeriod);
}
    
```

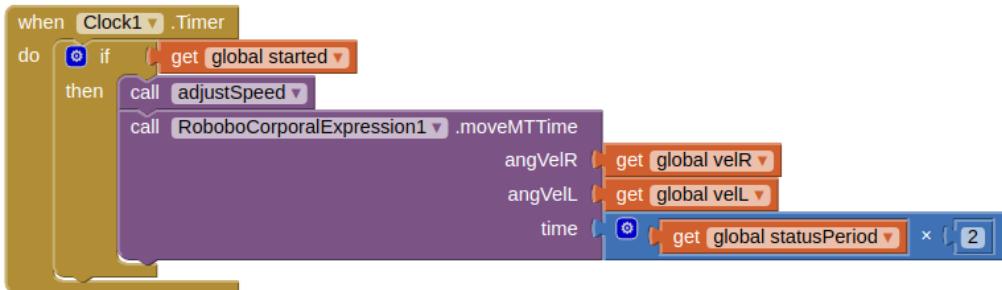


Figura 4.25: Acción periódica do movemento dos motores en App Inventor.

Tamén podemos ver unhas imaxes amosando como sería o comportamento de aproximarse a unha parede na figura 4.25.

Se comparamos o fragmento anterior de código coa figura 4.25 podemos ver coma neste caso App Inventor ofrece unha vantaxe sobre Java para accionar o movemento dos motores, pois non soamente é máis intuitivo, senón tamén máis sinxelo; cousa que non pasaba co algoritmo de axuste de velocidades, no que ocorría todo o contrario. Ademais, se pensamos nunha aplicación máis complexa, e vendo o tamaño dos bloques necesarios para calcular a velocidad, xa nos podemos imaxinar o rápido que se complica seguir a programación con bloques a medida que aumenta o número de elementos; polo que podemos concluír que a programación con bloques é un fantástico método para a iniciación de novos programadores, pero sen embargo, así que unha aplicación require a interacción de moitos elementos, aínda que non se trate de algo complexo, rapidamente fai que o desenvolvemento se complique.

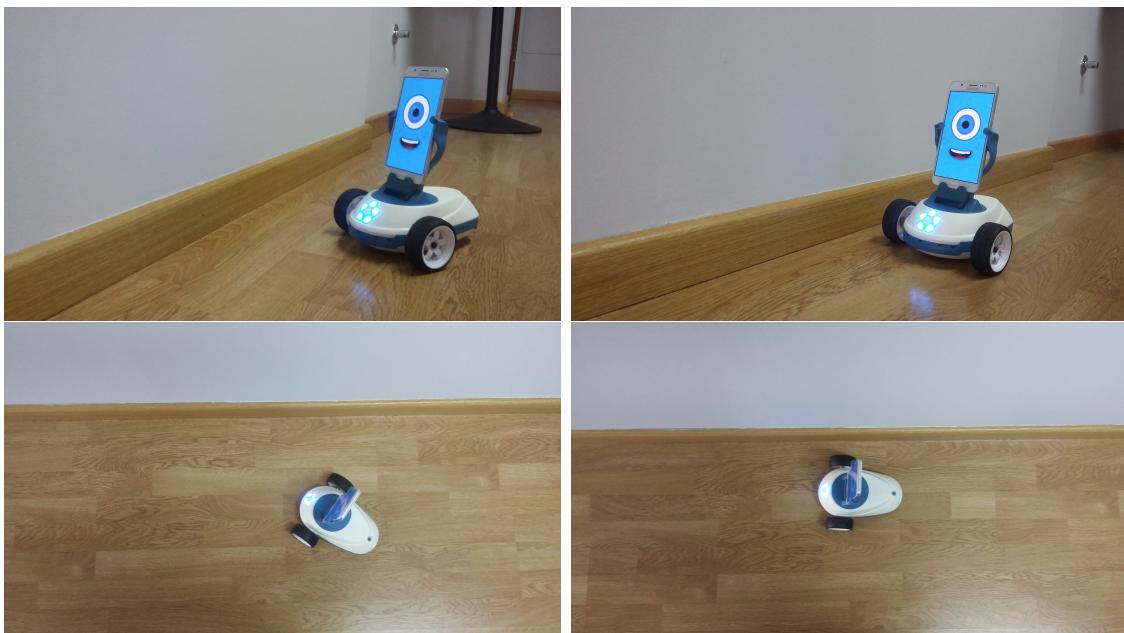


Figura 4.26: Comportamento do robot ao achegarse a unha parede.

### 4.4.3. Segunda iteración

O obxectivo desta segunda iteración ha ser dotar á API que se está a desenvolver dos compoñentes que permitan empregar as capacidades de visión do robot. Non obstante, debido á dependencia que teñen os módulos que controlan a cámara e o do tacto, coa actividade Android que amosa a cara do robot, tamén se han implementar nesta iteración o compoñente de expresión facial e o do sentido do tacto.

#### 4.4.3.1. Sentido da vista

##### 4.4.3.1.1 Deseño funcional

Como na anterior iteración, primeiro empezaremos definindo as accións que imos precisar para implementar o sentido da vista, que igual que pasaba coa expresión corporal, vai ter accións en ambos sentidos; pero a diferencia desta, neste sentido hai tres módulos: *FaceDetection*, *BlobTracking* e *Brightness*.

A continuación expónense cada unha das accións da figura 4.27 co nome que reciben no tipo enumerado:

- *CHANGE\_CAMERA*: para cambiar entre a cámara dianteira e traseira do móvil.

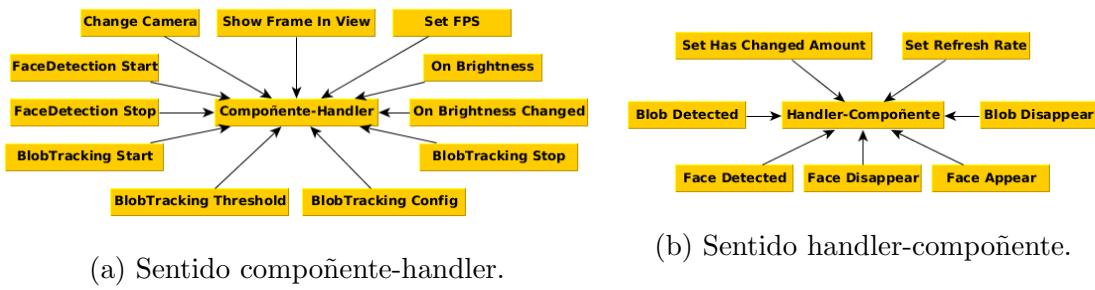


Figura 4.27: Accións do sentido da vista.

- *SHOW\_FRAME\_IN\_VIEW*: que permite amosar por pantalla o que pode ver a cámara.
- *SET\_FPS*: para cambiar o número de fotogramas por segundo que se deben capturar.
- *BLOB\_TRACKING\_START*: para activar o módulo de detección de obxectos de cores.
- *BLOB\_TRACKING\_STOP*
- *BLOB\_TRACKING\_CONFIG*: para configurar as cores dos obxectos que se deben detectar.
- *BLOB\_TRACKING\_THRESHOLD*: que permite establecer un límite para a detección dos obxectos.
- *BLOB\_DETECTED*: para notificar ao compoñente de que se detectou un obxecto das características desexadas.
- *BLOB\_DISAPPEAR*: para notificar ao compoñente de que o obxecto que se detectara xa non está á vista.
- *FACE\_DETECTION\_START*: para iniciar a detección de caras.
- *FACE\_DETECTION\_STOP*: para deter a detección de caras.
- *FACE\_DETECTED*: para informar ao compoñente de que se detectou unha cara nun fotograma.
- *FACE\_APPEAR*: para informar ao compoñente de cando o robot pode ver unha cara; non se volvería recibir mentres a cara se manteña no seu campo de visión, e ata que non volva a aparecer nel.

- *FACE\_DISAPPEAR*: para informar ao compoñente de que desapareceu a cara que se detectaba no último fotograma.
- *SET\_REFRESH\_RATE*: permite establecer período de actualización da luminosidade.
- *SET\_HAS\_CHANGED\_AMOUNT*: permite establecer un límiar para recibir o evento *onBrightnessChanged()*.
- *ON\_BRIGHTNESS*: para informar ao compoñente do nivel de luminosidade detectada.
- *ON\_BRIGHTNESS\_CHANGED*: para notificar ao compoñente que se superou o límiar establecido.

#### 4.4.3.1.2 Primeiro nivel de abstracción

Definidas as accións, o seguinte paso é a implementación do handler e o compoñente do mesmo xeito ca na iteración anterior, pero debido a que os parámetros que debemos empaquetar nesta iteración non son tipos básicos, a maneira de empaquetalos no *Bundle* ha ser algo diferente.

O problema neste paso é debido ao módulo de detección de obxectos coloreados, xa que se empregan dúas clases definidas na librería de visión do robot: *Blob*, e *Blobcolor*. Que ao non ser tipos propios de Android ou Java non se poden incluír no *Bundle* das mensaxes tan facilmente.

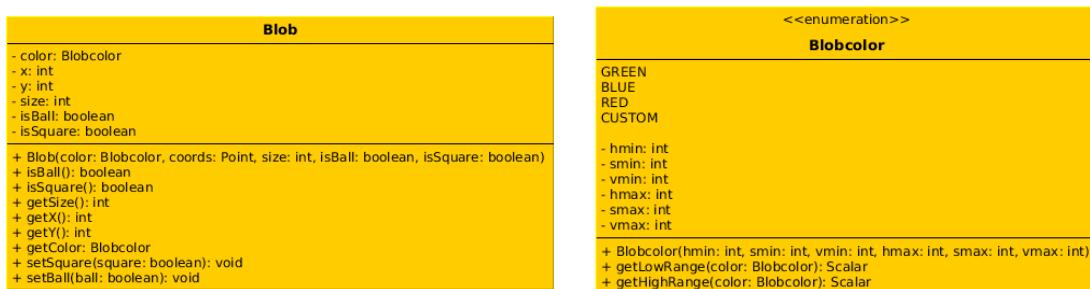


Figura 4.28: UML de Blob e Blobcolor.

Para solucionar este problema pódese optar por tres opcións. As dúas primeiras son moi semellantes e consisten en facer que ambos tipos de datos implementen as interfaces *Parcelable* ou *Serializable*, xa que deste xeito a súa inclusión no *Bundle*

resulta trivial, pois é tan simple coma facelo para tipos de datos primitivos. Non obstante, ao formar parte da librería de visión do Robobo, e por tanto, quedar fora do ámbito deste traballo, optouse pola terceira opción, que sería descompoñer ditos obxectos nos atributos que os compoñen, e envialos como se viña facendo na iteración anterior. Segundo este procedemento, o proceso para empaquetar esta información no handler quedaría como segue:

```

public void onTrackingBlob(Blob blob) {
    Parameters blobDetectedParameters = BLOB_DETECTED.getParameters();
    Bundle params = getBundleBlobColorParams(blobDetectedParameters, blob.getColor());
    params.putInt(blobDetectedParameters.getParamKey(6), blob.getX());
    params.putInt(blobDetectedParameters.getParamKey(7), blob.getY());
    params.putInt(blobDetectedParameters.getParamKey(8), blob.getSize());
    params.putBoolean(blobDetectedParameters.getParamKey(9), blob.isBall());
    params.putBoolean(blobDetectedParameters.getParamKey(10), blob.isSquare());
    try {
        send(BLOB_DETECTED, params);
    } catch (RemoteException e) {
        Log.e(LOG_TAG, e.getMessage());
        onShutDown();
    }
}

private Bundle getBundleBlobColorParams(Parameters parameters, Blobcolor c) {
    Bundle params = new Bundle();
    params.putInt(parameters.getParamKey(0), c.hmin);
    params.putInt(parameters.getParamKey(1), c.smin);
    params.putInt(parameters.getParamKey(2), c.vmin);
    params.putInt(parameters.getParamKey(3), c.hmax);
    params.putInt(parameters.getParamKey(4), c.smax);
    params.putInt(parameters.getParamKey(5), c.vmax);
    return params;
}

```

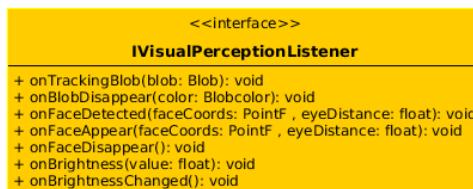


Figura 4.29: Interface para recibir eventos da vista.

Outra cousa que se debe ter en conta para esta iteración, igual que se facía na anterior, é que o programa que empregue o compoñente de percepción visual debe ser quen de recibir os eventos de detección de caras e obxectos de cores. Segundo a mesma estratexia que para o ROB e a recepción da información de sensores, decidíuse

definir a interface *IVisualPerceptionListener*, cuxo diagrama UML podemos ver na figura 4.29.

Ademais, para que o handler da vista reciba a información dos tres módulos do Robobo deste sentido: *BlobTracking*, *FaceDetection* e *Brightness*. Este debe implementar as interfaces *IBlobListener*, *IFaceListener* e *IBrightnessListener*, que conteñen tódolos métodos incluídos na interface *IVisualPerceptionListener* da figura 4.29.

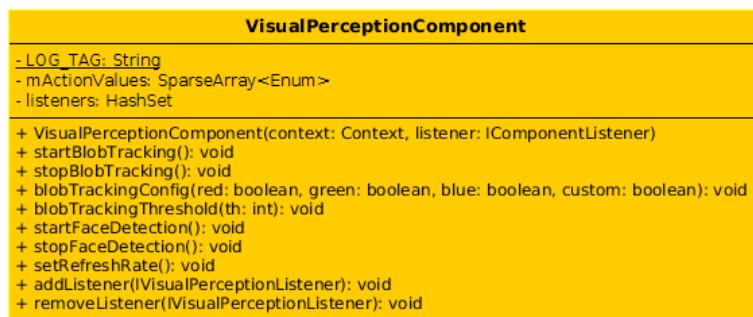


Figura 4.30: Sentido da vista no primeiro nivel de abstracción.

#### 4.4.3.1.3 Segundo nivel de abstracción

Como xa se dixo anteriormente, todo compoñente deste nivel de abstracción debe estender a clase creada para tal propósito: *RoboboComponent*. Nela están definidas as funcións básicas que todo compoñente debe ter, e que deberán ser sempre sobreescritas e chamadas dende esta nova implementación como última instrución. Por exemplo, o código baixo o evento *onComponentsStarted* e *onComponentsError* quedaría como segue:

```

@Override
public void onComponentsStarted() {
    visualComponent = (VisualPerceptionComponent) getComponentInstance();
    visualComponent.addListener(this);
    super.onComponentsStarted();
}

@Override
public void onComponentsError(String errorMsg) {
    if (visualComponent != null) {
        visualComponent.removeListener(this);
        visualComponent = null;
    }
    super.onComponentsError(errorMsg);
}

```

<b>RoboboVisualPerception</b>
- <code>LOG_TAG: String</code>
- <code>visualComponent: VisualPerceptionComponent</code>
+ <code>RoboboVisualPerception(container: ComponentContainer)</code>
+ <code>startup(robName: String): void</code>
+ <code>shutdown(): void</code>
+ <code>onComponentsStarted(): void</code>
+ <code>ComponentsStarted(): void</code>
+ <code>onComponentsError(errorMsg: String): void</code>
+ <code>ComponentsError(errorMsg: String): void</code>
+ <code>startBlobTracking(): void</code>
+ <code>stopBlobTracking(): void</code>
+ <code>blobTrackingConfig(red: boolean, green: boolean, blue: boolean, custom: boolean): void</code>
+ <code>blobTrackingThreshold(threshold: int): void</code>
+ <code>startFaceDetection(): void</code>
+ <code>stopFaceDetection(): void</code>
+ <code>setHasChangedAmount(amount: int): void</code>
+ <code>trackingBlob(color: int, isBall: boolean, isSquare: boolean; size: int, x: int, y: int): void</code>
+ <code>onTrackingBlob(blob: Blob): void</code>
+ <code>blobDisappear(color: int): void</code>
+ <code>onBlobDisappear(color: Blobcolor): void</code>
+ <code>faceDetected(x: float, y: float, eyeDistance: float): void</code>
+ <code>onFaceDetected(pointF: PointF, eyeDistance: float): void</code>
+ <code>faceAppear(x: float, y: float, eyeDistance: float): void</code>
+ <code>onFaceAppear(pointF: PointF, eyeDistance: float): void</code>
+ <code>onFaceDisappear(): void</code>
+ <code>onBrightness(value: float): void</code>
+ <code>onBrightnessChanged(): void</code>

Figura 4.31: Sentido da vista no segundo nivel de abstracción.

Outra cousa que tamén se comentou cando se describiu como ían ser os compoñentes de App Inventor na subsección 4.3.3.4, foi que os métodos que foran ser empregados para implementar un bloque da linguaxe visual só podían conter tipos primitivos; esta razón é a que obrigou a duplicar algúns dos eventos que podemos ver no diagrama UML deste compoñente na figura 4.31. De feito, o que se está a facer dentro do corpo das funcións que proveñen da interface *IVisualPerceptionListener* é descompoñer os parámetros en tipos primitivos, tal e como podemos ver no seguinte código:

```

@SimpleEvent
public void trackingBlob(int color, boolean isBall, boolean isSquare,
                         int size, int x, int y) {
    EventDispatcher.dispatchEvent(this, "trackingBlob",
                                  color, isBall, isSquare, size, x ,y);
}

@Override
public void onTrackingBlob(Blob blob) {
    Log.d(LOG_TAG, "trackingBlob()");
    Blob lastTrackedBlob = blob;
    trackingBlob(blob.getColor().hashCode(), blob.isBall(), blob.isSquare(),
                blob.getSize(), blob.getX(), blob.getY());
}

```

Se nos fixamos, o primeiro parámetro do evento *trackingBlob* obtense a partires

do código hash do obxecto que devolve `blob.getColor()`, que é un tipo enumerado chamado `Blobcolor` do framework do Robobo actual, e que define valores para as cores: vermello, verde, azul, e un personalizado. Como non se aceptan tipos enumerados como parámetros dos bloques, decidiuse empregar o truco do hash coma na transmisión de mensaxes entre handlers e compoñentes; cousa que tamén obriga a ter unha lista nalgúnha parte con tódolos códigos das cores dispoñibles.

Para solucionar isto, optouse por desenvolver un compoñente non visible de App Inventor –non estende `RoboboComponent`– chamado `RoboboBlobcolor`, que permitira ao usuario obter os hash das cores a través de bloques, e cuxo fragmento de código podemos ver a continuación:

```
@SimpleProperty
public int Red() {
    return RED.hashCode();
}

@SimpleProperty
public int Green() {
    return GREEN.hashCode();
}
```

Por outra banda, o resto dos métodos non son máis ca interfaces para chamar ao compoñente de visión do primeiro nivel de abstracción da API. A continuación amosamos o código dunha destas funcións:

```
@SimpleFunction
public void startFaceDetection() {
    String functionName = "startFaceDetection";
    try {
        visualComponent.startFaceDetection();
    } catch (RemoteException e) {
        form.dispatchErrorOccurredEvent(this, functionName,
            ErrorMessages.ERROR_SIMPLE_API_SERVICE_NOT_AVAILABLE);
    }
}
```

Por último, e para rematar a descripción do compoñente de App Inventor da visión, amosaranse nas figuras 4.32 e 4.33 unhas imaxes cos bloques dos que dispón.

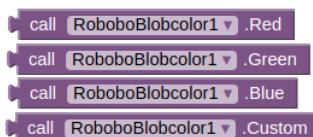


Figura 4.32: Bloques das cores para o sentido da vista.

## 4.4. DESEÑO E IMPLEMENTACIÓN CAPÍTULO 4. DESENVOLVEMENTO

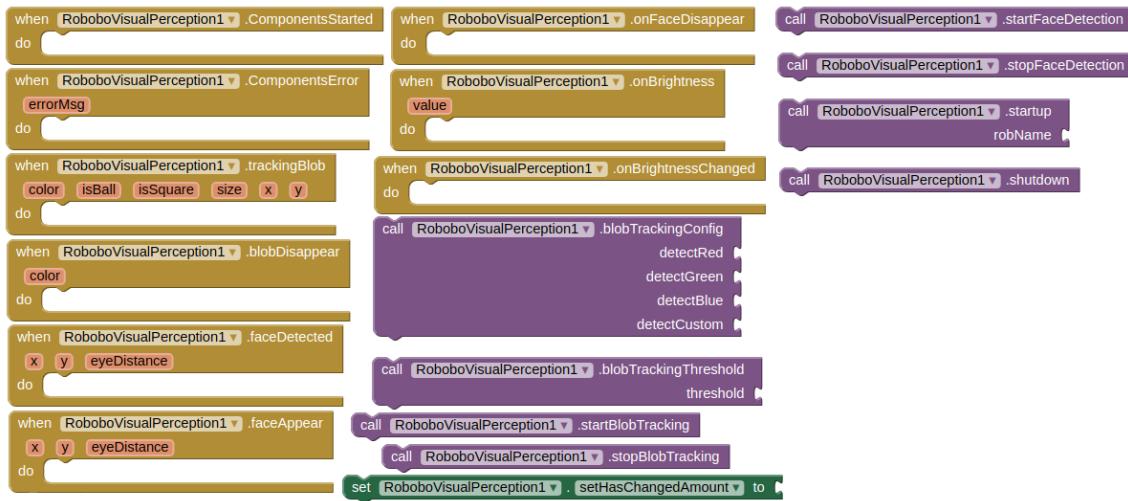


Figura 4.33: Bloques do sentido da vista.

### 4.4.3.2. Expresión facial

#### 4.4.3.2.1 Deseño funcional

Segundo co mesmo proceso, primeiro realizaremos a definición das accións necesarias para implementar a expresión facial, que coma pasaba nos dous últimos casos, tamén ha contar con accións en ambos sentidos.

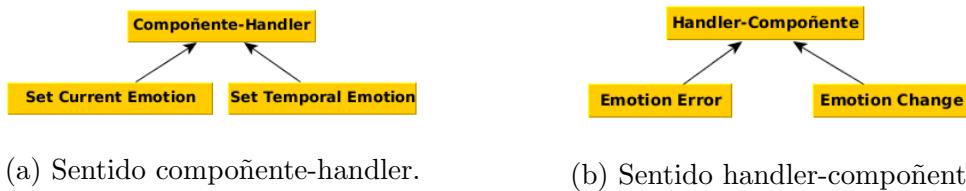


Figura 4.34: Accións de expresión facial.

A continuación hanse explicar cada unha das accións definidas indicando o nome co que figuran no tipo enumerado:

- *SET\_CURRENT\_EMOTION*: que permite establecer unha nova emoción.
- *SET\_TEMPORAL\_EMOTION*: para establecer unha emoción temporal.
- *EMOTION\_ERROR*: para notificar ao compoñente de que non se atopou a cara solicitada.
- *EMOTION\_CHANGE*: que permite notificar o evento de cando se produce un cambio de emoción.

#### 4.4.3.2.2 Primeiro nivel de abstracción

Ata o momento tódolos compoñentes desenvolvidos eran soamente clases que procesaban os parámetros de execución dos métodos e os enviaban a través do servizo de mensaxería; todo o que se solicitaba ao compoñente era retransmitido ao módulo correspondente. Neste caso, atopámonos cunha dificultade a maiores, pois é recomendable que o programador teña acceso á emoción que esta amosando actualmente o robot; debe haber un *getter* para obter a emoción actual igual que existe no módulo de emocións do Robobo. Pero xa que non sería posible implementar isto sen empregar un callback de telo que solicitar ao handler, cousa que non sería nada recomendable para a función de *getter*, o compoñente deberá manter unha copia da emoción actual. Debido a isto, a implementación do método *setTemporalEmotion()* non pode soamente enviar a petición ao handler, senón que tamén debe encargarse de actualizar dita copia.

Para solucionar este problema decidiuse actualizar a copia da emoción actual do robot a través do evento que notifica un cambio de emoción. A outra posibilidade era programar unha tarefa a través dun *Timer* e un *TimerTask* para facer os cambios de emoción nos tempos asignados, pero dado que se fallase algo no protocolo que impedira a execución da acción, o compoñente faría ese cambio pola súa conta, acabaría tendo información incorrecta do que realmente pasou; así que se decidiu optar pola anterior solución.

Polo que respecta ao proceso de empaquetado dos parámetros, o único que se precisa comentar é que as emocións son valores dun tipo enumerado que non podemos incluír directamente no *Bundle* das mensaxes, e unha vez más, parte dunha librería actual do robot que non podemos modificar; polo que se empregará o hashcode de cada un deses valores coma faciamos para transmitir as accións entre handlers e compoñentes. A continuación amósase o empaquetado dos parámetros no compoñente:

```
public void setTemporalEmotion(Emotion emotion, Long duration, final Emotion nextEmotion)
    throws RemoteException {
    Bundle params = new Bundle();
    params.putInt(SET_TEMPORAL_EMOTION.getParameters().getParamKey(0), emotion.hashCode());
    params.putLong(SET_TEMPORAL_EMOTION.getParameters().getParamKey(1), duration);
    params.putInt(SET_TEMPORAL_EMOTION.getParameters().getParamKey(2), nextEmotion.hashCode());
    sendHandler(SET_TEMPORAL_EMOTION, params);
}
```

Igual que ocorría co resto dos compoñentes desenvolvidos ata o momento, o de expresión facial tamén debe ser quen de recibir os eventos de cambio de emoción

que se producen no módulo, polo que o handler deberá implementar a interface do módulo Robobo destinada a tal propósito: *IEmotionListener*. Que unicamente conta co método *newEmotion(emotion: Emotion)*.



Figura 4.35: Interface para eventos da expresión facial.

Ademais, se o usuario do compoñente precisa ser notificado destes cambios, existe unha interface semellante á descrita no parágrafo anterior –*IFacialExpressionListener*–, e que podemos ver na figura 4.35.

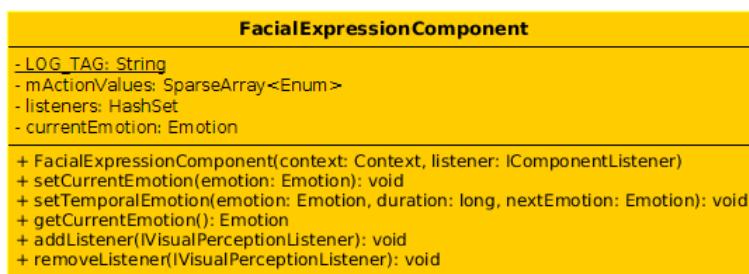


Figura 4.36: Expresión facial no primeiro nivel de abstracción.

#### 4.4.3.2.3 Segundo nivel de abstracción

Igual que todo compoñente do segundo nivel de abstracción desta API simplificada, a expresión facial en App Inventor vai ser unha clase que estenda *Robobo-Component* e sobrescriba tódolos métodos que se definen nesta, engadindo ademais unha chamada ao da súperclase como última instrución. Amósase a continuación un exemplo de código:

```

@Override
public void onComponentsError(String errorMsg) {
    if (facialComponent != null) {
        facialComponent.removeListener(this);
        facialComponent = null;
    }
    super.onComponentsError(errorMsg);
}
    
```

<b>RoboboFacialExpression</b>
<pre> -LOG_TAG: String -facialComponent: FacialExpressionComponent -emotionValues: SparseArray&lt;Enum&gt;  + RoboboFacialExpression(container: ComponentContainer) + startup(robName: String): void + shutdown(): void + onComponentsStarted(): void + ComponentsStarted(): void + onComponentsError(errorMsg: String): void + ComponentsError(errorMsg: String): void + setCurrentEmotion(emotion: int): void + setTemporalEmotion(emotion: int, duration: long, nextEmotion: int): void + getCurrentEmotion(): int + EmotionChange(emotion: int): void + emotionChange(emotion: Emotion): void </pre>

Figura 4.37: Expresión facial no segundo nivel de abstracción.

Outro dato importante dos componentes de App Inventor que xa se comentou antes, é que non se permiten bloques con parámetros de tipos complexos. Neste caso, o problema xorde debido as emocións, que ao ser valores dun tipo enumerado non se poden empregar nos parámetros. Así que dunha maneira semellante a como se fixo co sentido da vista, empregarase o código hash das emocións para os parámetros, e mediante unha clase chamada *RoboboEmotion* tipo a que se empregaba co componente anterior –*RoboboBlobcolor*–, definiranse un conxunto de funcións que permitan obter o código hash de tódalas emocións a través de bloques.

A continuación dous métodos de exemplo da clase *RoboboEmotion*:

```

@SimpleProperty
public int Happy() {
    return HAPPY.hashCode();
}

@SimpleProperty
public int Sad() {
    return SAD.hashCode();
}

```

E a implementación da función *setTemporalEmotion*:

```

@SimpleFunction
public void setTemporalEmotion(int emotion, long duration, int nextEmotion){
    String functionName = "setTemporalEmotion";
    Emotion e = (Emotion) emotionValues.get(emotion);
    Emotion next = (Emotion) emotionValues.get(nextEmotion);
    try {
        facialComponent.setTemporalEmotion(e, duration, next);
    } catch (RemoteException e1) {
        form.dispatchErrorOccurredEvent(this, functionName,

```

```

        ErrorMessages.ERROR_SIMPLE_API_SERVICE_NOT_AVAILIABLE);
    }
}

```

Por último, para finalizar esta descripción amosaremos os bloques que resultan do desenvolvemento deste compoñente como se describiu anteriormente.

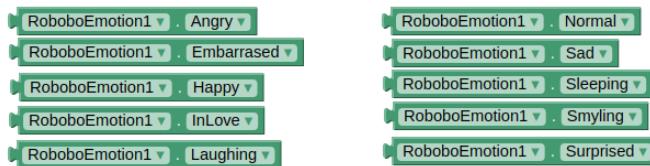


Figura 4.38: Bloques das emocións para a expresión facial.

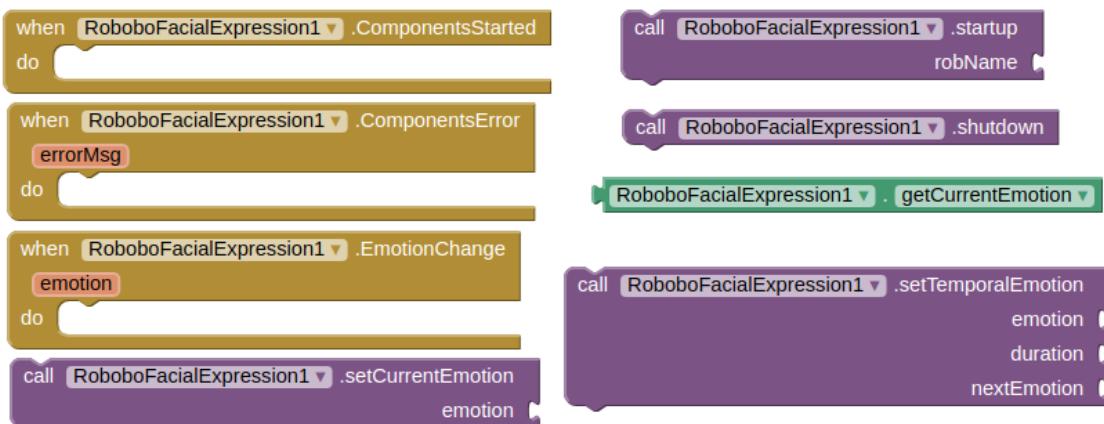


Figura 4.39: Bloques de expresión facial.

#### 4.4.3.3. Sentido do tacto

##### 4.4.3.3.1 Deseño funcional

Este é o compoñente máis sinxelo dos desenvolvidos nesta segunda iteración, xa que soamente traballa con comunicación nun único sentido: dende o handler ao compoñente. A súa misión consiste en notificar os eventos disparados por toques na pantalla do móvil, correspondendo cada un deles a unha acción no compoñente do tacto.

A continuación expónense e explícanse tódalas accións co nome que reciben no tipo enumerado que as define:

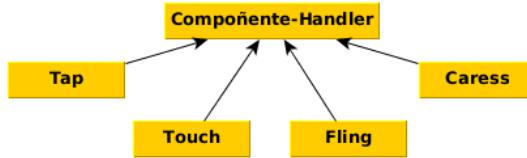


Figura 4.40: Accións do sentido do tacto.

- *TAP*: cando se produce un toque simple e rápido.
- *TOUCH*: para un toque mantido.
- *CARESS*: cando detecta unha escorregada sobre a pantalla.
- *FLING*: ao rematar a escorregada.

#### 4.4.3.3.2 Primeiro nivel de abstracción

Como en casos anteriores, agora que xa se definiron as accións coas que vai contar este compoñente, pasarase a comentar a súa implementación no primeiro nivel de abstracción.

Igual que xa ocorreu durante o desenvolvemento dalgúns dos compoñentes anteriores, atopámonos agora cun tipo dun parámetro que non se pode incluír directamente nas mensaxes do protocolo: *TouchGestureDirection*. Este parámetro é dun tipo enumerado definido na librería do Robobo na que se atopan os módulos relacionados co control da pantalla táctil do móvil, e cuxo propósito é indicar a dirección da escorregada detectada. A solución escollida, igual que para os comandos do protocolo, radica en empregar o código hash dos valores a transmitir. Podemos ver este proceso no seguinte fragmento de código do handler:

```

public void fling(TouchGestureDirection dir, double angle, long time, double distance) {
    Bundle params = new Bundle();
    params.putInt(FLING.getParameters().getParamKey(0), dir.hashCode());
    params.putDouble(FLING.getParameters().getParamKey(1), angle);
    params.putLong(FLING.getParameters().getParamKey(2), time);
    params.putDouble(FLING.getParameters().getParamKey(3), distance);
}
  
```

Unha vez máis, para que o handler poida notificar ao compoñente cando estes eventos suceden, este debe implementar a interface *ITouchListener* que proporciona a librería do tacto do Robobo; pero ademais, tamén debe implementar a interface *ITouchEventListener* que inclúe a librería de emocións. A primeira é a que permite

recibir os catro eventos que expoñiamos anteriormente, e a segunda recibe calquera toque sobre a pantalla.



Figura 4.41: Interfaces do framework actual para a detección de accións sobre a pantalla táctil.

Comentábase ao inicio desta segunda iteración que existía unha dependencia entre o módulo do tacto e a actividade que amosa a cara do robot. En Android, as actividades son as únicas responsables do control da pantalla, polo que para detectar accións sobre ela é imprescindible que sexa a actividade da cara a que notifique cando se produce un. Mediante o método *onScreenTouch(motion: MotionEvent)* da interface *ITouchEventListerner*, o handler poderá alimentar ao módulo do tacto para que este procese o evento, e informe de volta ao handler co tipo de toque producido a través dos métodos definidos na interface *ITouchListener*, que serán retransmitidos ao compoñente empregando as accións que se expuxeron ao principio.

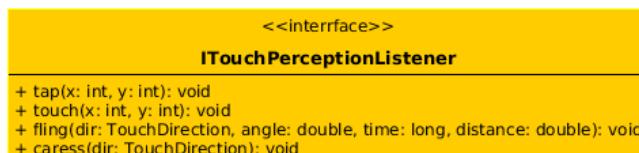


Figura 4.42: Interface para eventos do sentido do tacto.

De xeito semellante aos casos anteriores, defínese tamén a interface *ITouchPerceptionListener*, que permitirá informar ao usuario deste compoñente dos eventos producidos na pantalla do móvil. Como podemos comprobar na figura 4.42, esta interface contén exactamente os mesmos métodos ca tiña *ITouchListener*, polo que un podería pensar: para que ter dúas interfaces con diferente nome e os mesmos métodos? A razón é evitar completamente as dependencias dos módulos do Robobo do lado dos compoñentes.

#### 4.4.3.3.3 Segundo nivel de abstracción

Xa sabemos de anteriores ocasións que para desenvolver componentes para o segundo nivel de abstracción desta API estes deben estender a clase *RoboboComponent* e sobreescibir tódolos seus métodos. Ademais, a última instrución debe chamar á implementación do método da súperclase tal e como se pode ver no seguinte fragmento de código:

```
@Override
public void onComponentsStarted() {
    touchComponent = (TouchPerceptionComponent) getComponentInstance();
    touchComponent.addListener(this);
    super.onComponentsStarted();
}
```

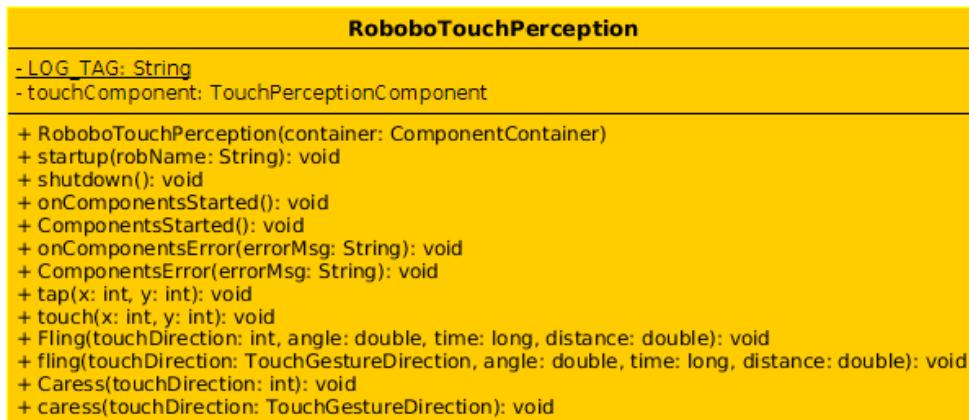


Figura 4.43: Sentido do tacto no segundo nivel de abstracción.

Unha vez máis, como podemos observar no diagrama UML da figura 4.43, foi necesario duplicar dous métodos debido ao tipo dos parámetros que reciben; recordemos que os bloques de App Inventor só permiten emplegar tipos primitivos e cadeas. Concretamente, neste caso, o tipo conflitivo foi *TouchGestureDirection*, que é un tipo enumerado definido co módulo do Robobo que detecta eventos táctiles da pantalla, e que permite indicar a dirección dos arrastres.

Do mesmo xeito que no caso da expresión facial e as emocións, a estratexia escollida consiste en crear un componente que devolva os códigos hash de cada unha das direccións definidas en *TouchGestureDirection*, e empregar ese valor como parámetros dos eventos duplicados. Amosamos a continuación un exemplo:

```
@SimpleProperty
public int Left() {
```

```

        return LEFT.hashCode();
    }

    @SimpleProperty
    public int Right() {
        return RIGHT.hashCode();
    }
}

```

Por outra banda, a implementación dos eventos *fling* e *Fling* quedaría como segue:

```

@SimpleEvent
public void Fling(int touchDirection, double angle, long time,
                  double distance) {
    EventDispatcher.dispatchEvent(this, "Fling",
        touchDirection, angle, time, distance);
}

@Override
public void fling(TouchGestureDirection touchDirection,
                  double angle, long time, double distance) {
    Fling(touchDirection.hashCode(), angle, time, distance);
}

```

Con todo o desenvolvemento rematado, os elementos que interveñen no sentido do tacto no segundo nivel de abstracción producen os bloques que podemos ver na



Figura 4.44: Bloques das direccións para o sentido do tacto.

#### 4.4.3.4. Demostración

Para a demostración desta segunda iteración o que se vai facer é construír unha nova aplicación de proba empregando o desenvolvido na anterior; o obxectivo é engadir unha función de cada un dos compoñentes desta iteración é integrala coa demostración da anterior.

Se acordamos, a aplicación de proba da expresión corporal consistía nun comportamento que permitiría ao robot recorrer un labirinto avanzando soamente cara adiante e sen bater contra ningún obstáculo. Ademais, ese movemento iniciábase automaticamente nada máis se recibía o evento que informa que os compoñentes están iniciados e preparados para o seu uso, polo que a primeira función que se vai

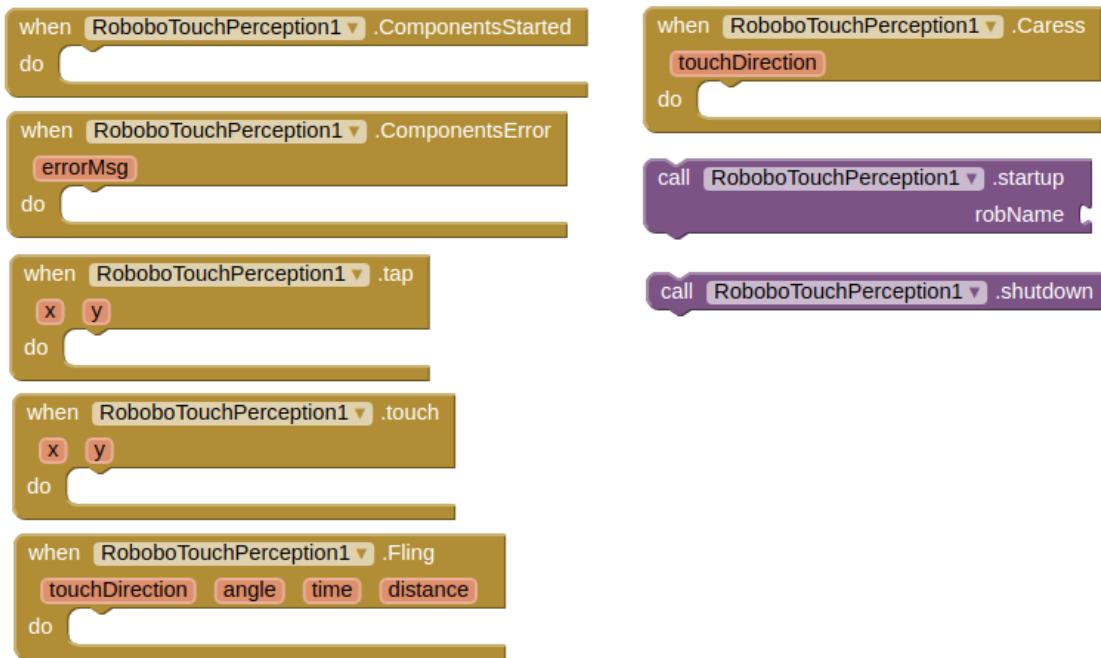


Figura 4.45: Bloques do sentido do tacto.

engadir sobre o desenvolvido como proba da primeira iteración ha ser unha maneira de iniciar e deter o movemento; para o que se decidiu empregar o sentido do tacto a modo de interruptor, de xeito que un toque na pantalla permita activar e desactivar o proceso de movemento. Podemos observar esta implementación empregando os bloques de App Inventor na figura 4.46, e a do primeiro nivel de abstracción no seguinte fragmento de código; ánda que dado que só se trata dun interruptor, non resulta de gran interese:

```

@Override
public void tap(int x, int y) {
    if (started()) {
        stopMovement();
    } else {
        try {
            startMovement();
        } catch (RemoteException e) {
            Log.d(LOG_TAG, "Error starting movement: " + e.getMessage() + "\n\nStopping test");
            stop();
        }
    }
}

```

O seguinte paso, unha vez que xa podemos activar e desactivar o movemento por medio do sentido do tacto, é empregar algunha das funcións da vista e da expresión



Figura 4.46: Interruptor do movemento en App Inventor empregando o tacto.

facial. Dado que o noso obxectivo é soamente demostrar o funcionamento dos dous niveis de abstracción desenvolvidos neste traballo, optouse por un comportamento sinxelo que fixera que o robot cambiara de cara dependendo de se detecta un obxecto vermello a través da cámara. Concretamente, de se detectar algo con esa cor, o robot ha poñer cara de namorado; e nada máis desapareza do seu campo de visión, cabréase. No seguinte fragmento de código e na figura 4.47 podemos ver este comportamento empregando ambos niveis de abstracción.

```

@Override
public void onTrackingBlob(Blob blob) {
    Log.d(LOG_TAG, "onTrackingBlob()");
    if (blob.getColor() == RED)
        try {
            facial.setCurrentEmotion(IN_LOVE);
        } catch (RemoteException e) {
            Log.e(LOG_TAG, e.getMessage());
        }
}

@Override
public void onBlobDisappear(Blobcolor c) {
    if (c == RED)
        try {
            facial.setTemporalEmotion(ANGRY, (long) 3000, NORMAL);
        } catch (RemoteException e) {
            Log.e(LOG_TAG, e.getMessage());
        }
}

```

Tamén podemos ver os resultados deste comportamento no robot na figura ??.

Como uidemos observar nos exemplos de código e bloques desta demostración, para o desenvolvemento de comportamentos sinxelos App Inventor, mediante o segundo nivel de abstracción da API simplificada, resulta unha ferramenta moi sinxela e intuitiva de usar, moi recomendable para aqueles sen experiencia previa en programación. É certo que na aplicación de proba da primeira iteración os bloques necesarios son moitos e ocupan unha gran parte da contorna de traballo de App Inventor, pero dado que ese non é un exemplo de aplicación apropiado para un novato, tampouco debe ser considerado como unha gran desvantaxe.

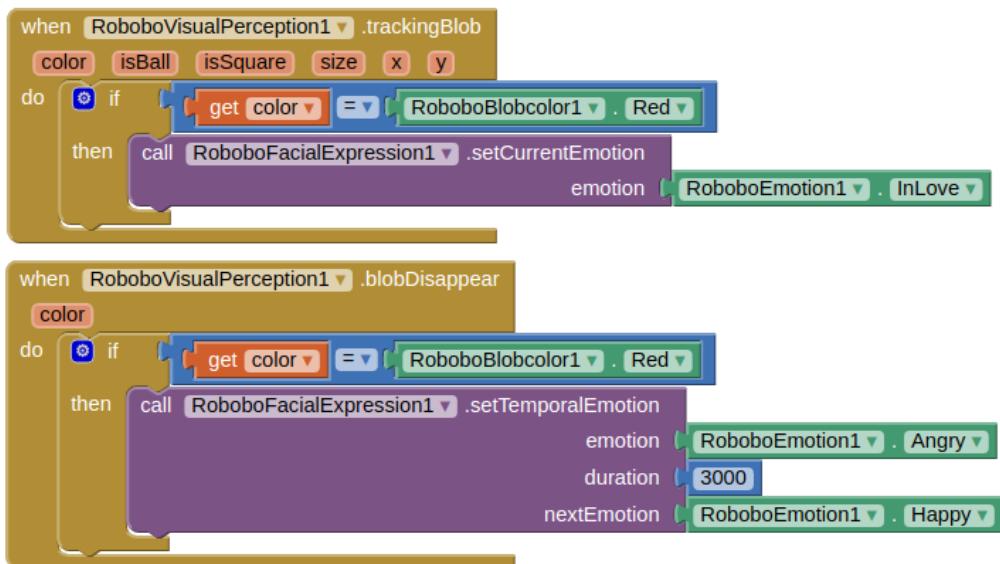


Figura 4.47: Cambio de caras por obxecto vermello en App Inventor.



(a) Robobo enamorado por ver unha pelota vermella.



(b) Robobo porque xa non ten unha pelota vermella.

Outro asunto a ter en conta é o da reutilización do código. Empregando o primeiro nivel de abstracción temos mecanismos para aproveitar todo comportamento implementado de xeito que se cambiamos algo da primeira iteración, ese cambio tamén se ve reflectido nas seguintes, xa que empegan o código das anteriores. No caso de App Inventor isto non é posible, e débese facer unha copia completa de tódolos bloques da iteración anterior; polo que de detectar un erro na primeira, sería necesario amañalo en tódalas seguintes.

#### 4.4.4. Terceira iteración

Nesta iteración, o obxectivo ha ser dotar á nova API do sentido do oído e das capacidades de expresión sonora. Recordamos que debido a que en Android non pode haber dous elementos empregando o micrófono ao mesmo tempo, non se vai dar soporte ao módulo de recoñecemento da fala do Robobo.

##### 4.4.4.1. Expresión sonora

###### 4.4.4.1.1 Deseño funcional

O compoñente de expresión sonora vai ser o que agrupe as funcionalidades de tódolos módulos do robot que emiten algúns tipo de son: dará soporte ao módulo *emotionSoundModule*, para emitir sons de emocións; ao de xeración de notas, chamado *noteGenerationModule*; e ao *speechProductionModule*, que permite que o robot se comunique falando.

Coa descripción do parágrafo anterior pode parecer que só ha ser necesario implementar as comunicacións no sentido compoñente-handler, pero ademais de executar os comandos para a emisión de sons, estes módulos tamén contan cunha serie de eventos que permiten notificar o final da execución destes comandos. Podemos ver un gráfico coas accións definidas na figura 4.49.

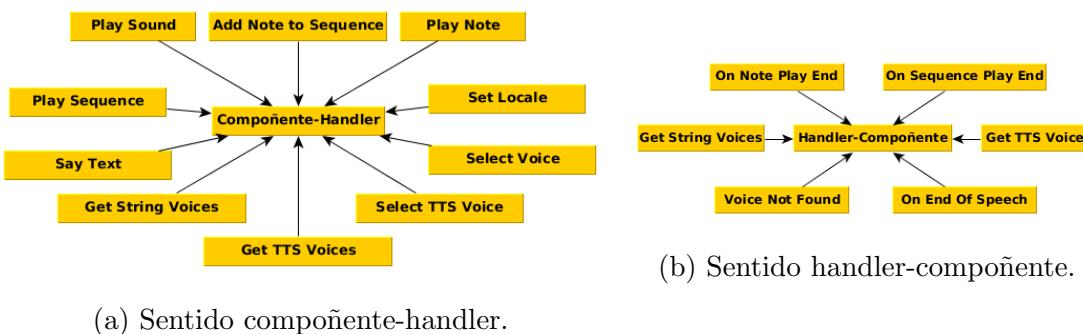


Figura 4.49: Accións de expresión sonora.

Algo que pode chamar a atención dos gráficos anteriores son as accións *GetString Voices* e *Get TTS Voices*, xa que aparecen en ambos. A razón disto é porque ambas accións son empregadas polo compoñente para solicitar o conxunto de voces disponible, e tamén polo handler para responder a esa petición.

A continuación describiremos cada unha das accións asociada ao seu nome real como valor dun tipo enumerado:

- *PLAY\_SOUND*: permite emitir sons de emocións.
- *PLAY\_NOTE*: para emitir notas musicais.
- *ADD\_NOTE\_TO\_SEQUENCE*: engade notas a unha secuencia.
- *PLAY\_SEQUENCE*: toca a secuencia de notas engadidas mediante a acción anterior.
- *SAY\_TEXT*: di o texto que se lle pasa por parámetros.
- *SET\_LOCALE*: permite cambiar a configuración de idioma da producción da fala.
- *SELECT\_VOICE*: para escoller unha voz de entre as dispoñibles polo nome.
- *SELECT\_TTS\_VOICE*: para escoller unha voz empregando un obxecto *Voice*.
- *GET\_STRING\_VOICE*: permite obter os nomes das voces que están dispoñibles.
- *GET\_TTS\_VOICE*: permite obter os obxectos *Voice* das voces dispoñibles.
- *ON\_NOTE\_PLAY\_END*: notifica aos observadores cando se rematou de emitir unha nota.
- *ON\_SEQUENCE\_PLAY\_END*: notifica cando se rematou de emitir a secuencia.
- *ON\_END\_OF\_SPEECH*: avisa cando o robot remata de falar.
- *VOICE\_NOT\_FOUND*: indica que o nome de voz seleccionado non existe.

#### **4.4.4.1.2 Primeiro nivel de abstracción**

Se acordamos, co compoñente da expresión facial fora necesario crear un atributo onde manter unha copia da cara actual do robot debido a que era necesario permitir que o programador a puidese obter mediante un mecanismo tipo *getter*, xa que de facelo a través dunha acción implicaría un *callback* para notificar a recepción, e complicaría innecesariamente o proceso. Neste caso, ocorre algo semellante coas funcións que permiten obter as voces dispoñibles, pero a diferencia do caso anterior, no deseño funcional do compoñente de expresión sonora si que se incluíron accións para solicitar esta información ao handler, xa que no proceso de inicio –dentro da

función *onStartUp()*, o compoñente deberá solicitar dita información para despois podela ofrecer ao usuario a través dun *getter* corrente.

O paso de parámetros neste caso tamén é sinxelo, xa que o único caso no que non é posible pasar esa información directamente debido a que se trata dun tipo de dato que non se pode incluír nas mensaxes, é o das notas musicais, que veñen definidas como valores dun tipo enumerado. Chegados a este punto, e debido á solución escollida en anteriores ocasións, o lector pode pensar que se vai optar unha vez máis por empregar o código hash das notas musicais; non obstante, xa que existen dous tipos de datos diferentes para a representación das notas musicais –un para o módulo de xeración e outro para o de detección–, nos que por suposto están definidas da mesma maneira, decidiuse empregar o nome que devolve a función *name()* dos tipos enumerados. O obxectivo é facer que as notas, independentemente do tipo que as define, sexan intercambiáveis facilmente, de xeito que o programador novel que empregue este compoñente poida usar ambos tipos sen ter que se preocupar por cal importa, e así poñer toda a súa atención na aprendizaxe da sintaxe de Java; posteriormente, o profesor tamén podería esixir o uso concreto dalgún dos dous tipos para destacar a importancia dos *imports* e os nomes completos dos tipos e paquetes, e facer que a aprendizaxe de Java se desenvolva dun xeito máis gradual se fora necesario. Pero ademais, existe outro motivo de carácter técnico relacionado co compoñente de percepción sonora que se ha explicar no punto 4.4.4.2.2 desta subsección na páxina 93. A continuación podemos ver a implementación dos métodos *playNote* empregando os dous tipos existentes para notas musicais.

```
public void playNote(Note note, int timems) throws RemoteException {
    playNote(note.name(), timems);
}

public void playNote(com.mytechia.robobo.framework.hri.sound.noteDetection.Note note,
                     int timems) throws RemoteException {
    playNote(note.name(), timems);
}

private void playNote(String note, int timems) throws RemoteException {
    Bundle params = new Bundle();
    params.putString(PLAY_NOTE.getParameters().getParamKey(0), note);
    params.putInt(PLAY_NOTE.getParameters().getParamKey(1), timems);
    sendHandler(PLAY_NOTE, params);
}
```

Non obstante, non debemos esquecer que estes non son os únicos sons que este compoñente vai empregar, xa que tamén nos permite emitir sons de emocións. Sen embargo, con estes non é necesario buscar unha solución para transmitilos nas

mensaxes, pois a diferencia do caso anterior, estes veñen definidos coma constantes de tipo enteiro na interface que implementa o módulo de emocións; polo que a súa inclusión é directa.

Outro caso a destacar no paso de parámetros é o da acción *Set Locale*, cuxo parámetro é de tipo *Locale*. Sen embargo, ao ser un tipo propio de Java, e que implementa a interface *Serializable*, a súa inclusión é directa, coma coas variables e arrays de tipos primitivos.

Respecto dos eventos definidos nas accións, para que o usuario poida recibir as notificacións do handler, este deberá implementar a interface destinada a tal propósito: *ISoundExpressionListener*. Que podemos ver na figura 4.50.

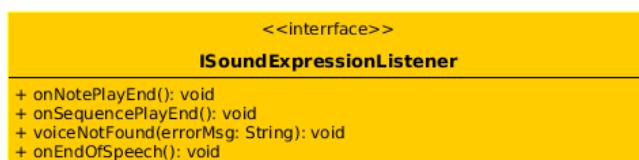


Figura 4.50: Interface para eventos da expresión sonora.

Por suposto, o handler debe implementar tamén as interfaces do módulos do Robobo necesarias para recibir a información que despois ha enviar ao compoñente: *ISpeechProductionListener* e *INotePlayListener*. Das que obtén os catro eventos definidos na interface *ISoundExpressionListener* da figura 4.50.

Como podemos ver no diagrama UML do compoñente de expresión sonora da figura 4.51, existen dous métodos *playNote* e dous *addNoteToSequence* cuxa única diferencia é o tipo do primeiro parámetro de ambos, e que se corresponden co exposto no fragmento de código anterior.

#### 4.4.4.1.3 Segundo nivel de abstracción

Ao inicio da descripción do desenvolvemento do segundo nivel de abstracción de tódolos compoñentes anteriores díxose que todos eles debían estender a clase *RoboboComponent*, pois é a que facilita a interacción destes co *SimpleRoboboManager*, que é o encargado de iniciar e finalizar cada un deles. *RoboboComponent* define tamén tódolos métodos básicos que un compoñente de App Inventor da API simplificada debe ter, e dos que dixemos ademais que debían ser sobreescritos e chamados dende a última instrución da nova implementación de cada método, tal e como podemos comprobar no seguinte fragmento de código correspondente á expresión sonora:

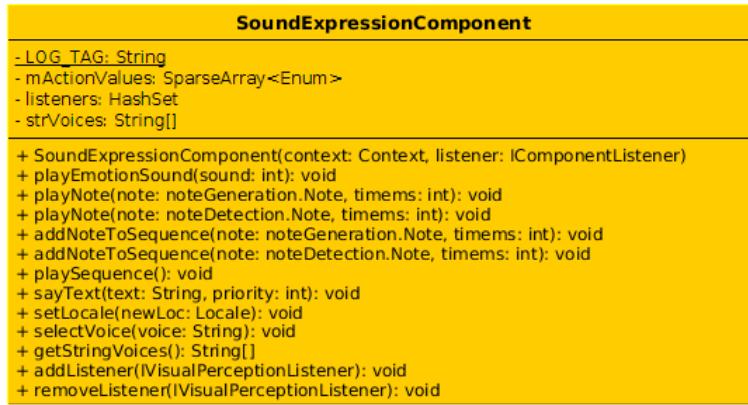


Figura 4.51: Expresión sonora no primerio nivel de abstracción.

```

@Override
public void onComponentsError(String errorMsg) {
    if (soundComponent != null) {
        soundComponent.removeListener(this);
        soundComponent = null;
    }
    super.onComponentsError(errorMsg);
}
    
```

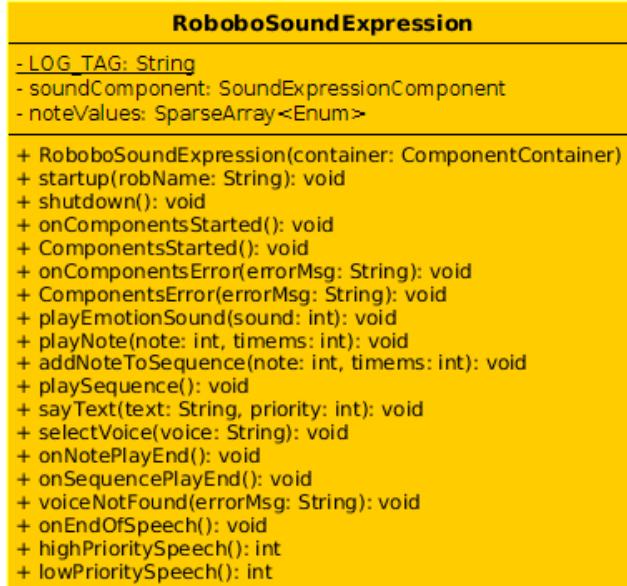


Figura 4.52: Expresión sonora no segundo nivel de abstracción.

Unha vez máis, o desenvolvemento deste compoñente require tipos de dato complexos que non se permiten como parámetros dunha función dun bloque; concreta-

mente, o tipo *Note* do módulo do Robobo *noteDetection*.

Pode sorprender que para a expresión sonora se empregue o tipo relacionado co módulo de detección de notas en vez de o de xeración de notas, xa que ademais o desenvolvemento do primeiro nivel de abstracción deste compoñente fíxose pensando en que fora compatible con ambos. Non obstante, xa que o sentido do oído ten que traballar con este tipo, pois é o que emprega o módulo *noteDetection*, e dar soporte ao outro tamén implicaría xestionar máis eventos, decidiuse que a expresión sonora no segundo nivel de abstracción empregara este tipo en vez do outro; de xeito que o único tipo de nota permitida sexa a que se define xunto ao módulo *noteDetection*.

Para o desenvolvemento desta estratexia, como xa pasou con algúns dos compoñentes anteriores, foi imprescindible definir a clase *RoboboNote*, que da xeito semellante ao que pasaba coas emocións, implementa unha función que devolve o código hash de cada nota:

```
@SimpleProperty(description = "C4")
public int C4() { return C4.hashCode(); }

@SimpleProperty(description = "E4")
public int E4() { return E4.hashCode(); }

@SimpleProperty(description = "G4")
public int G4() { return G4.hashCode(); }
```

A razón de escoller o código hash en vez do nome da nota, como se facía para a transmisión entre handler e compoñente de expresión sonora, é debido a que así aseguramos que o programador empregue os bloques de *RoboboNote* en vez de introducir os nomes das notas mediante un bloque para inserción de texto.

Segundo esta estratexia, por exemplo o método para que o robot toque unha nota determinada quedaría da seguinte maneira:

```
@SimpleFunction
public void playNote(int note, int timems) {
    String functionName = "playNote";
    try {
        soundComponent.playNote((Note) noteValues.get(note), timems);
    } catch (RemoteException e) {
        form.dispatchErrorOccurredEvent(this, functionName,
            ErrorMessages.ERROR_SIMPLE_API_SERVICE_NOT_AVALIABLE);
    }
}
```

Débese comentar tamén, que para realizar o desenvolvemento deste compoñente foi necesario crear un atributo onde almacenar tódolos códigos hash das notas disponibles, dun xeito semellante a como se fixo coas accións entre handlers e com-

poñentes. De feito, no fragmento de código anterior podemos ver como se accede a este atributo para obter o obxecto *Note* que require o método *playNote* no primeiro nivel de abstracción.

Para finalizar, amosamos nas figuras 4.53 e 4.54 unhas imaxes onde se poden ver os bloques que resultan do desenvolvemento da expresión sonora en App Inventor.

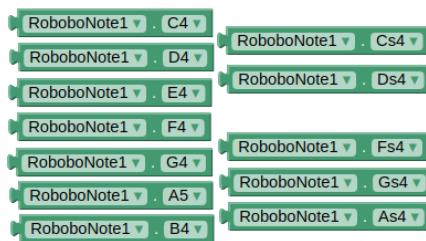


Figura 4.53: Octava das notas disponíveis para a expresión sonora.

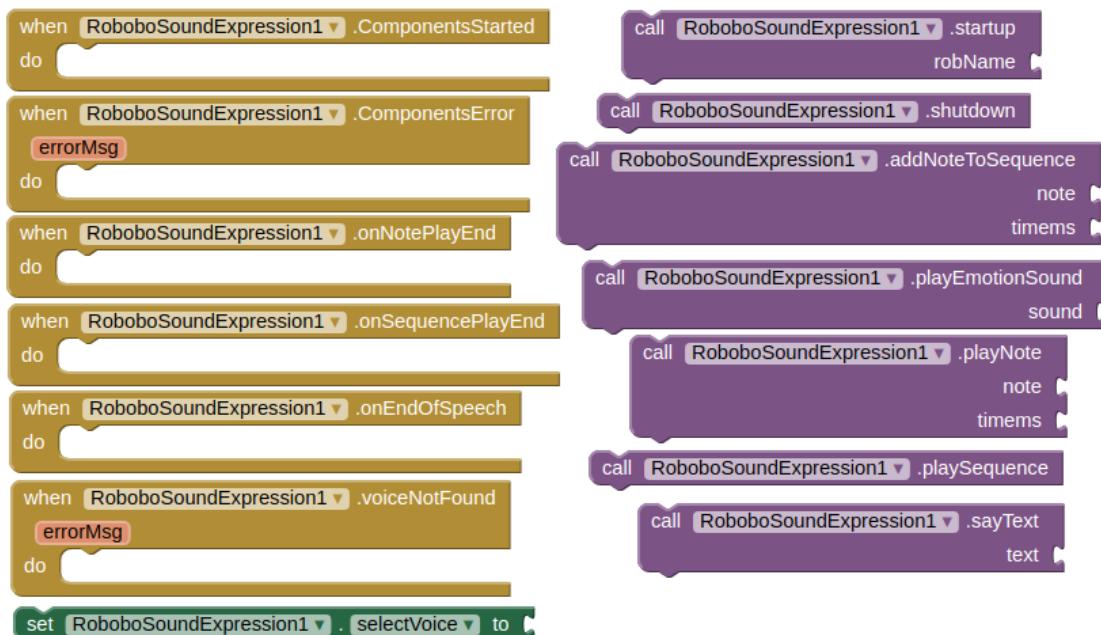


Figura 4.54: Bloques da expresión sonora.

#### 4.4.4.2. Percepción sonora

##### 4.4.4.2.1 Deseño funcional

Este compoñente é o que vai agrupar tódolos módulos relacionados co ”oído” do robot, que son: *soundDispatcherModule*, *pitchDetectionModule*, *clapDetectionModule*

e *noteDetectionModule*.

Debido a que se trata dun sentido para o que non é necesario establecer ningún tipo de configuración, a comunicación ha ser soamente no sentido handler-compoñente. Podemos ver un diagrama coas accións escollidas na figura 4.55.

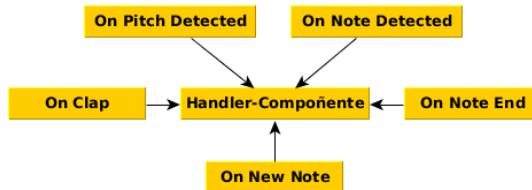


Figura 4.55: Accións de percepción sonora.

A continuación, acompañadas do seu nome real como valor dun tipo enumerado, explicaranse tódalas accións que vemos no gráfico:

- *ON\_PITCH\_DETECTED*: notifica a detección dun son cunha frecuencia determinada.
- *ON\_CLAP*: avisa da detección dun son percutor e indica a súa duración.
- *ON\_NOTE\_DETECTED*: informa da nota que se está a detectar.
- *ON\_NOTE\_END*: notifica o fin dunha nota e indica a súa duración.
- *ON\_NEW\_NOTE*: cando se detecta unha nova nota, diferente da que estaba a soar anteriormente, o handler informa disto a través desta acción indicando de que nota se trata.

#### 4.4.4.2.2 Primeiro nivel de abstracción

Como xa vimos no deseño funcional, este compoñente só conta con accións no sentido handler-compoñente, xa que vai ser responsable únicamente de notificar os eventos recibidos dos módulos do Robobo para a percepción de sons.

Ademais, igual ca pasaba coa expresión sonora, o único problema que imos atopar co paso de parámetros ha ser debido ás notas musicais, xa que van ser outra vez valores dun tipo enumerado, aínda que desta volta, non do mesmo tipo que antes; é dicir, existe un tipo enumerado para as notas do módulo de detección, e outro tipo coas mesmas notas para o módulo de xeración.

Para a implementación deste nivel de abstracción, xa que unha nota é a igual independentemente de se a estamos escoitando ou emitindo, e dado que sería unha dificultade innecesaria para quen está a aprender a programar en Java, unificaremos estes dous tipos.

Para isto, e unha vez máis, xa que non podemos incluír un tipo enumerado nas mensaxes, empregarase algún valor que poidamos extraer dese valor. En anteriores ocasións empregouse o código hash, pero coma neste caso queremos unificar dous tipos de datos diferentes, e o hash non sería o mesmo por este motivo, empregarase o nome do valor, xa que en ambos casos están definidas as mesmas notas e co mesmo nome.

A diferencia do que pasaba coa expresión sonora, onde o usuario podía escoller entre dous métodos diferentes para cada función de emisión de notas que empregaban os dous tipos disponibles na API actual do robot, neste caso só se desenvolveu pensando en empregar o tipo *Note* definido para o módulo de detección de notas, xa que isto obligaría a definir dúas interfaces, unha para cada tipo; ou unha interface co dobre de métodos. En ambos casos, algo que complica a API sen aportar novas funcionalidades.

Non obstante, debido á implementación do protocolo empregando o nome do valor en vez do código hash, deixamos a porta aberta para aqueles usuarios que se queiran aventurar a modificar ou estender o actual compoñente para facelo compatible coas notas do módulo de xeración, pero sen ter que desenvolver unha función que realice a transformación de tipos.

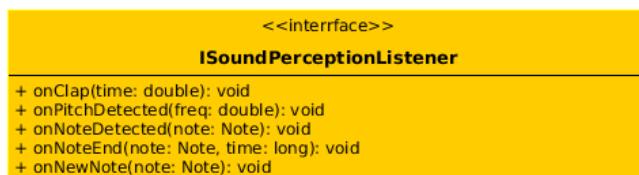


Figura 4.56: Interface para eventos do sentido do oído.

Por outra banda, respecto da notificación dos eventos correspondentes ás accións definidas anteriormente, o usuario deste compoñente conta coa interface *ISoundPerceptionListener*, que permite recibir os eventos relacionados con este sentido.

#### 4.4.4.2.3 Segundo nivel de abstracción

Igual que para tódolos demais compoñentes de App Inventor, o primeiro que se debe facer é estender a clase deseñada para servir de base para crear compoñentes para o segundo nivel de abstracción da nova API, e despois sobreescibir tódolos seus métodos engadindo unha chamada aos da súperclase como última instrución:

```
@Override
public void onComponentsStarted() {
    soundComponent = (SoundPerceptionComponent) getComponentInstance();
    soundComponent.addListener(this);
    super.onComponentsStarted();
}
```

<b>RoboboSoundPerception</b>	
-	LOG_TAG: String
-	soundComponent: SoundPerceptionComponent
-	noteValues: SparseArray<Enum>
+	RoboboSoundPerception(container: ComponentContainer)
+	startup(robName: String): void
+	shutdown(): void
+	onComponentsStarted(): void
+	ComponentsStarted(): void
+	onComponentsError(errorMsg: String): void
+	ComponentsError(errorMsg: String): void
+	onClap(duration: double): void
+	onPitchDetected(freq: double): void
+	onNoteDetected(note: Note): void
+	NoteDetected(note: int): void
+	onNoteEnd(note: Note, duration: double): void
+	NoteEnd(note: int, duration: double): void
+	onNewNote(note: Note): void
+	NewNote(note: int): void

Figura 4.57: Sentido do oído no segundo nivel de abstracción.

Durante a descripción do desenvolvemento do compoñente anterior comentouse que no segundo nivel de abstracción íase empregar o tipo *Note* do módulo de detección de notas en vez do de xeración; débese recordar que de querer ofrecer ambas posibilidades para a percepción tamén sería necesario duplicar os eventos recibidos, cousa que complicaría innecesariamente o desenvolvemento neste nivel de abstracción sen aportar nada a maiores.

Nesa descripción, tamén se expuxo que debido a que en App Inventor non se permiten bloques con parámetros de tipos complexos, o que han empregar estas funcións en vez do tipo enumero son os códigos hash das notas de *noteDetection*. Desta maneira definíase outro compoñente chamado *RoboboNote* cuxas funcións devolvían este código.

Ademais, a diferenza da expresión sonora, na percepción necesitamos implementar obligatoriamente os eventos da interface *ISoundPerceptionListener*, que contén métodos con parámetros de tipo *Note*. Por este motivo, e igual que ocorreu anteriormente con outros compoñentes, os eventos que reciban parámetros complexos deben ser duplicados de xeito que se cambie ese tipo por un primitivo; como pasa por exemplo con *onNoteDetected* ou *onNoteEnd*.

```
@Override
public void onNoteDetected(Note note) {
    Log.d(LOG_TAG, "onNoteDetected(): " + note.name());
    EventDispatcher.dispatchEvent(this, "NoteDetected", note.hashCode());
}
```

Rematado o desenvolvemento do compoñente de percepción sonora do segundo nivel de abstracción, os bloques resultantes quedan como podemos ver nas figuras 4.58 e 4.59.

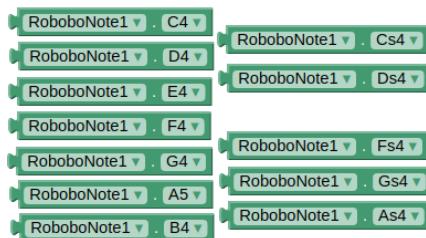


Figura 4.58: Octava das notas disponíveis para a percepción sonora.

#### 4.4.4.3. Demostración

Nesta terceira iteración, igual ca faciamos na segunda, colleremos a aplicación de proba da subsección anterior e engadiremos algúns comportamentos relacionados con tódolos compoñentes desenvolvidos nesta; é dicir, faremos que a aplicación empriegue tanto a expresión como a percepción sonora. Como xa dixemos anteriormente, o obxectivo soamente é facer algo sinxelo que nos sirva para ilustrar o seu funcionamento, polo que de entrada faremos que o robot avise de que está listo para funcionar emitindo unha pequena melodía e saudando.

Se acordamos, cando tódolos compoñentes involucrados na aplicación están listos para funcionar, o *SimpleRoboboManager* informa aos seus observadores a través do método *onComponentsStarted* – *ComponentsStarted* en App Inventor –, así que cando se reciba este evento, o robot debe saudar.

```
@Override
```

**CAPÍTULO 4. DESENVOLVEMENTO 4.4. DESEÑO E IMPLEMENTACIÓN**

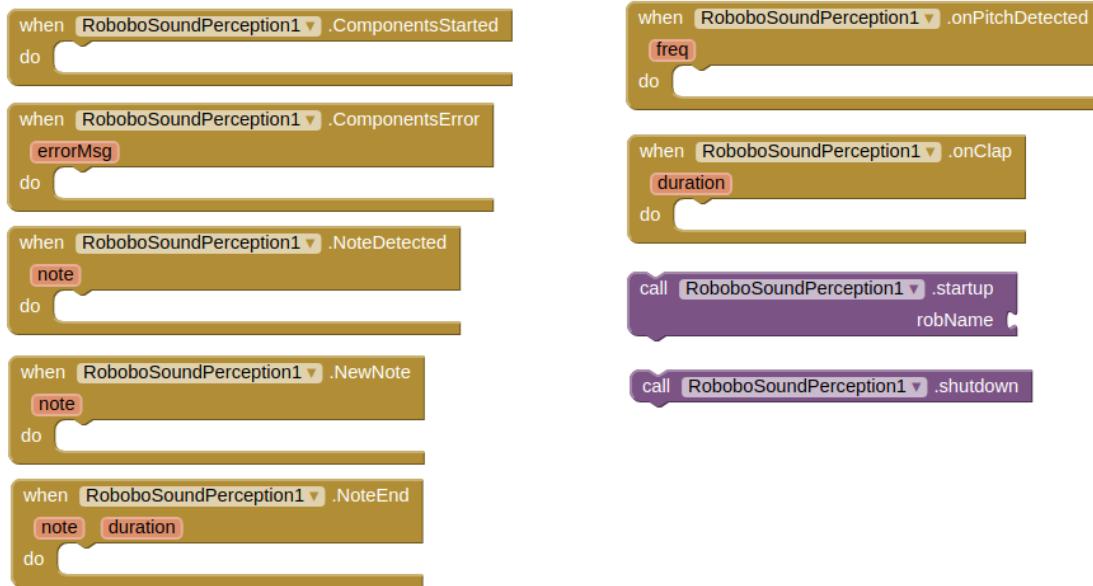


Figura 4.59: Bloques da percepción sonora.

```

public void onComponentsStarted() {
    try {
        int noteDur = 250;
        soundExpression.addNoteToSequence(G5, noteDur);
        soundExpression.addNoteToSequence(B5, noteDur);
        soundExpression.addNoteToSequence(D6, noteDur);
        soundExpression.playSequence();
        new Timer().schedule(new TimerTask() {
            @Override
            public void run() {
                try {
                    sayText("Hola! Soy Robobo");
                } catch (RemoteException e) {
                    Log.d(LOG_TAG, "Error saying hello: " + e.getMessage() +
                            "\n\nStopping test");
                    stop();
                }
            }
        }, 3 * noteDur + 2000);
        super.onComponentsStarted();
    } catch (RemoteException e) {
        Log.d(LOG_TAG, "Error starting blob tracking: " + e.getMessage() + "\n\nStopping test");
        stop();
    }
}

```

Na figura 4.60 e no fragmento de código anterior podemos ver que para facer que o saúdo soe despois da melodía foi necesario empregar un temporizador que executara o método *sayText*, pero tamén podemos comprobar como neste caso o

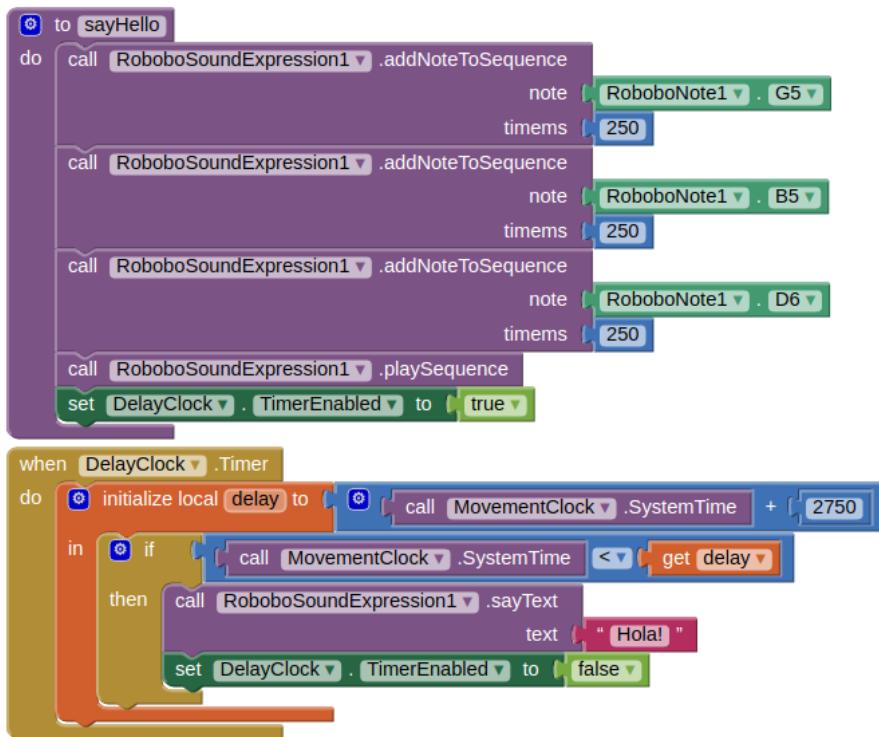


Figura 4.60: Saúdo do robot en App Inventor cando está listo para funcionar.

desenvolvemento en App Inventor parece un pouco más enrevesado ca en Java. O mecanismo é o mesmo, pero sen embargo nos bloques debemos ser nos os que implementemos o retardo comprobando o tempo restante en cada execución do evento do temporizador, mentres en Java xa contamos cun método que permite facer isto.

Agora que xa temos un exemplo coa expresión sonora, pasaremos a facer un relacionado co sentido do oído que permita activar o movemento a través de notas musicais. A idea é que se inicie cando o robot detecte un C4 –Do central dun piano de 88 teclas– e se deteña cun C5. A continuación nun fragmento de código, e na figura 4.61, podemos ver as dúas implementacións deste comportamento.

```

@Override
public void onNoteEnd(Note note, long time) {
    try {
        if (note == Note.C4) {
            movementVision.startMovement();
        } else if (note == Note.C5){
            movementVision.stopMovement();
        }
    } catch (ComponentNotFoundException e) {
        Log.e(LOG_TAG, "Error starting movement");
    }
}

```

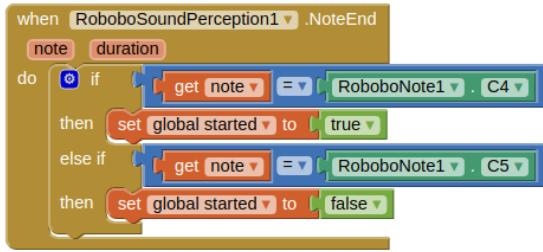


Figura 4.61: Interruptor do movemento co son en App Inventor.

Cos exemplos dos dous niveis de abstracción desta aplicación de proba confírmase o que se dicía ao final da demostración da iteración anterior, App Inventor resulta moi intuitivo e fácil para facer cousas sinxelas, pero así que o desenvolvemento implique comportamentos más elaborados, ou que involucren un número máis grande de bloques, a implementación vólvese complicada de manexar.

#### 4.4.5. Cuarta iteración

Como última iteración no desenvolvemento deste traballo implementaránse os últimos módulos do Robobo aos que se dará soporte a través desta nova API, e que son: o acelerómetro, xiroscopio, e nivel das baterías da base e do móvil.

Para isto hanse crear dous novos compoñentes: equilibriocepción, que inclúe ao acelerómetro e ao xiroscopio; e interocepción para recibir o estado das baterías.

##### 4.4.5.1. Equilibriocepción

A equilibriocepción é o sentido do equilibrio que permite aos animais e aos seres humanos camiñar sen caer [39]. A través das canles circulares situadas no oído interno nos tres planos do espazo, e cheos dun líquido chamado endolinfa, son capaces de detectar movementos do corpo grazas aos cambios de presión do líquido que conteñen.

Mediante este sentido pretendemos facer unha analogía entre como os seres humanos percibimos a posición do noso corpo e os sensores de "equilibrio" do robot, é dicir; o acelerómetro e o xiroscopio.

###### 4.4.5.1.1 Deseño funcional

Debido a que as funcións propias do sentido do equilibrio consisten nada máis que en obter a información dos dous sensores que levan os móbiles para detectar cambios

na súa posición ou orientación, este compoñente ha realizar as súas comunicacións sobre todo no sentido compoñente-handler; non obstante, tamén conta cunha acción que permite establecer un límiar de detección para ser notificados soamente de se superar ese valor.

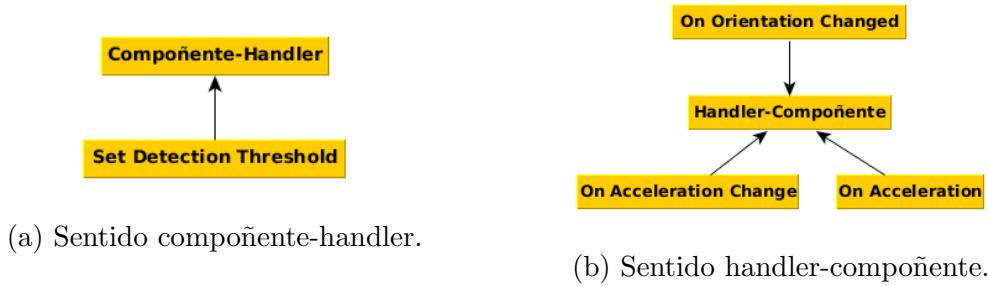


Figura 4.62: Accións de equilibriocepción.

A continuación hanse explicar as accións da figura 4.62 cos nomes reais que levan no tipo enumerado que as define:

- *SET\_DETECTION\_THRESHOLD*: para establecer un límiar para a detección de aceleracións.
- *ON\_ACCELERATION*: permite informar da aceleración detectada nas tres dimensións.
- *ON\_ACCELERATION\_CHANGE*: notifica que houbo unha aceleración superior ao límiar establecido mediante a primeira acción.
- *ON\_ORIENTATION\_CHANGED*: permite informar do cambio en posición do móvil mediante tres parámetros: *yaw*, *pitch* e *roll*.

#### 4.4.5.1.2 Primeiro nivel de abstracción

Para o desenvolvemento deste compoñente no primeiro nivel de abstracción só foi necesario implementar os métodos que empaquetan e desempaquetan a información transmitida a través das mensaxes, xa que tódolos parámetros consisten en tipos primitivos. Podemos ver a continuación un exemplo co código das funcións que extraen e inclúen esta información no *Bundle* para acción *On Orientation Changed*, primeiro no handler, e despois no compoñente.

```
public void onOrientationChanged(float yaw, float pitch, float roll) {
    Log.d(LOG_TAG, "onOrientationChanged(): (" + Float.toString(yaw) + ", " +
        Float.toString(pitch) + ", " + Float.toString(roll));
}
```

```

        Float.toString(pitch) + ", " + Float.toString(roll) + ")");
    Bundle data = new Bundle();
    data.putFloat(ON_ORIENTATION_CHANGED.getParameters().getParamKey(0), yaw);
    data.putFloat(ON_ORIENTATION_CHANGED.getParameters().getParamKey(1), pitch);
    data.putFloat(ON_ORIENTATION_CHANGED.getParameters().getParamKey(2), roll);
    send(ON_ORIENTATION_CHANGED, data);
}

private void onOrientationChanged(Bundle data) {
    Float yaw = data.getFloat(ON_ORIENTATION_CHANGED.getParameters().getParamKey(0));
    Float pitch = data.getFloat(ON_ORIENTATION_CHANGED.getParameters().getParamKey(1));
    Float roll = data.getFloat(ON_ORIENTATION_CHANGED.getParameters().getParamKey(2));
    for (IBalancePerceptionListener listener: listeners)
        listener.onOrientationChanged(yaw, pitch, roll);
}

```

#### 4.4.5.1.3 Segundo nivel de abstracción

Do mesmo xeito que cos casos anteriores, empezaremos este apartado recordando que todo componente do segundo nivel de abstracción debe ser fillo da clase *RoboboComponent*, que declara os métodos que deben implementar os componentes, e oculta a interacción co *SimpleRoboboManager*, polo que tódalas implementacións dos métodos deberán chamar ao da súperclase como última instrución. Poñamos un exemplo:

```

@SimpleFunction
@Override
public void shutdown() {
    if (balanceComponent != null)
        balanceComponent.unbind();
    super.shutdown();
}

```

RoboboBalancePerception
-LOG_TAG: String -balanceComponent: BalancePerceptionComponent
+ RoboboBalancePerception(container: ComponentContainer) + startup(robName: String): void + shutdown(): void + onComponentsStarted(): void + ComponentsStarted(): void + onComponentsError(errorMsg: String): void + ComponentsError(errorMsg: String): void + setDetectionThreshold(threshold: int): void + onAccelerationChange(): void + onAcceleration(x: int, y: int, z: int): void + onOrientationChange(yaw: float, pitch: float, roll: float): void

Figura 4.63: Equilibriocepción no segundo nivel de abstracción.

Como podemos ver diagrama UML deste compoñente na figura 4.63, esta é a primeira ocasión na que non foi necesario ningún artificio para evitar tipos complexos nos parámetros dos eventos deste sentido. Non obstante, engadimos un fragmento de código como exemplo:

```

@SimpleFunction
public void setDetectionThreshold(int threshold) {
    String functionName = "setDetectionThreshold";
    try {
        balanceComponent.setDetectionThreshold(threshold);
    } catch (RemoteException e) {
        form.dispatchErrorOccurredEvent(this, functionName,
            ErrorMessages.ERROR_SIMPLE_API_SERVICE_NOT_AVAILABLE);
    }
}

@SimpleEvent
@Override
public void onAccelerationChange() {
    EventDispatcher.dispatchEvent(this, "onAccelerationChange");
}

```

Para rematar, amosamos os bloques de App Inventor resultado do desenvolvemento deste compoñente na figura 4.64.

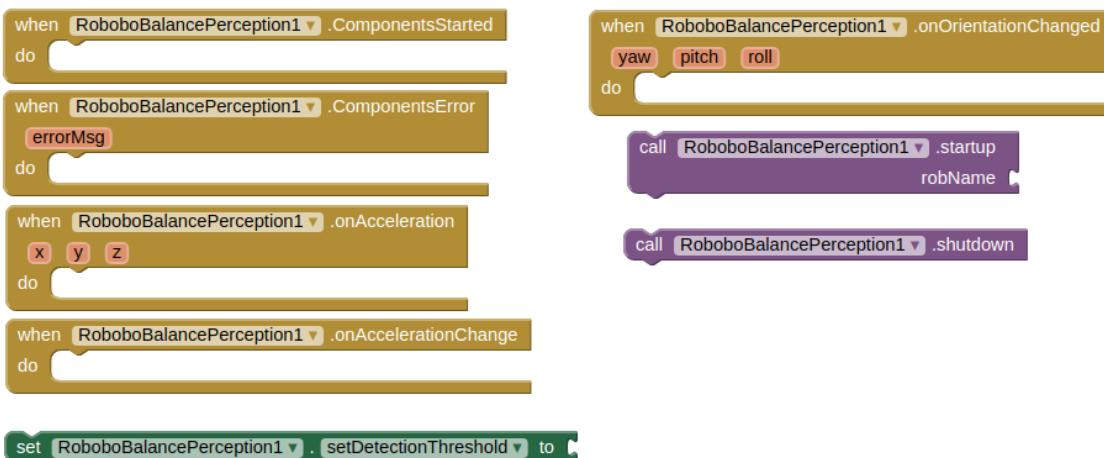


Figura 4.64: Bloques de equilibriocepción.

#### 4.4.5.2. Interocepción

As funcións interoceptivas ou interocepción é un sentido exclusivamente interno que fai referencia a aqueles estímulos ou sensacións que proveñen dos órganos internos do corpo humano; son a representación do estado dos órganos do noso corpo

[40], entre as que podemos atopar sensacions como por exemplo sede, fame, náuseas, ou as chamadas "taquicardias".

Como o lector xa se pode imaxinar, a intención de crear este sentido baséase en modelar a batería dos compoñentes do robot facendo unha analogía coa fame ou a sede.

#### 4.4.5.2.1 Diseño funcional

Igual ca pasaba coa equilibriocepción, a interocepción vai ser un compoñente que cuxo principal obxectivo e informar sobre o estado do robot; concretamente dos niveis de batería dos dous compoñentes dos que está constituído. Pero ademais, tamén permite establecer a frecuencia coa que desexamos recibir esas actualizacións, quedando o diagrama das accións deste compoñente como podemos observar na figura 4.65.

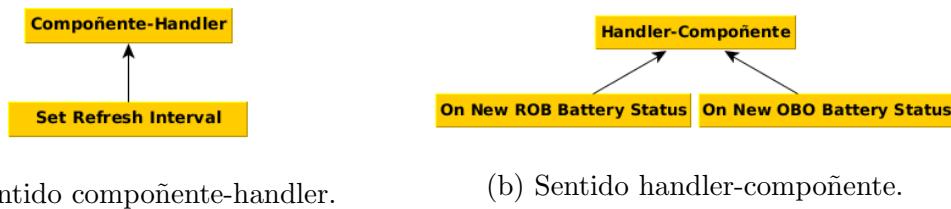


Figura 4.65: Accins de interocepcn.

A continuación explicaranse cada unha das accións deste compoñente empregando o nome real que levan no tipo enumerado no que foron definidas:

- *SET\_REFRESH\_INTERVAL*: para establecer un intervalo de actualización da batería do móvil (OBO).
  - *ON\_NEW\_ROB\_BATTERY\_STATUS*: permite recibir o nivel de batería do ROB.
  - *ON\_NEW\_OBO\_BATTERY\_STATUS*: para obter o estado da batería do móvil.

Débese destacar que non foi posible definir unha acción semellante a *Set Refresh Interval* para o nivel de batería do ROB, nin facer que esta acción cambiara o intervalo de actualización de ambos niveis, debido a que cambiaría tamén os intervalos de actualización da información dos sensores do ROB; polo que o nivel de batería da base do robot ha chegar segundo o intervalo de tempo definido no compoñente de expresión corporal.

#### 4.4.5.2.2 Primeiro nivel de abstracción

Como se comentaba, este compoñente é o que permite agrupar os niveis de batería da base e do móbil coma un único sentido chamado interocepción de implementación moi sinxela, xa que tódolos parámetros que se deben transmitir son todos tipos básicos fáciles de empaquetar nas mensaxes. Sen embargo, xa que o módulo do Robobo que permite obter o nivel da batería do ROB é o mesmo que permite o seu control, o handler deste compoñente deberá contar coa mesma instancia que o de expresión corporal pero sen permitir o acceso ao resto das funcións. Da mesma maneira, o nivel de batería non é accesible dende o compoñente para controlar o ROB, pois non é o seu propósito.

#### 4.4.5.2.3 Segundo nivel de abstracción

Unha vez máis, para realizar o desenvolvemento dos compoñentes de App Inventor para este segundo nivel de abstracción débese estender a clase *RoboboComponent*, e implementar novamente os seus métodos no fillo pero engadindo unha chamada aos da súperclase como última instrución, xa que encapsulan procesos imprescindibles e comúns a tódolos compoñentes. O seguinte fragmento de código ilustrará esta explicación:

```
@Override
public void onComponentsError(String errorMsg) {
    if (internalComponent != null) {
        internalComponent.removeListener(this);
        internalComponent = null;
    }
    super.onComponentsError(errorMsg);
}
```

RoboboInternalPerception
-LOG_TAG: String
-internalComponent: InternalPerceptionComponent
+ RoboboInternalPerception(container: ComponentContainer)
+ startup(robName: String): void
+ shutdown(): void
+ onComponentsStarted(): void
+ ComponentsStarted(): void
+ onComponentsError(errorMsg: String): void
+ ComponentsError(errorMsg: String): void
+ setRefreshInterval(millis: int): void
+ onNewOboBatteryStatus(battery: int): void
+ onNewRobBatteryStatus(battlevel: int, charging: boolean): void

Figura 4.66: Interocepción no segundo nivel de abstracción.

Igual que pasou coa equilibriocepción, neste sentido non son necesarios tipos complexos para ningún parámetro, polo que tampouco se tiveron que duplicar métodos para adaptar a recepción de parámetros as limitacións de App Inventor.

```
@SimpleEvent
@Override
public void onNewOboBatteryStatus(int battery) {
    EventDispatcher.dispatchEvent(this, "onNewOboBatteryStatus", battery);
}
```

```
@SimpleEvent
@Override
public void onNewRobBatteryStatus(int battlevel, boolean charging) {
    EventDispatcher.dispatchEvent(this, "onNewRobBatteryStatus", battlevel,
        charging);
}
```

Para finalizar, do mesmo xeito que cos anteriores componentes, amosaremos o conxunto de bloques resultante deste desenvolvemento; que podemos ver na figura 4.67.

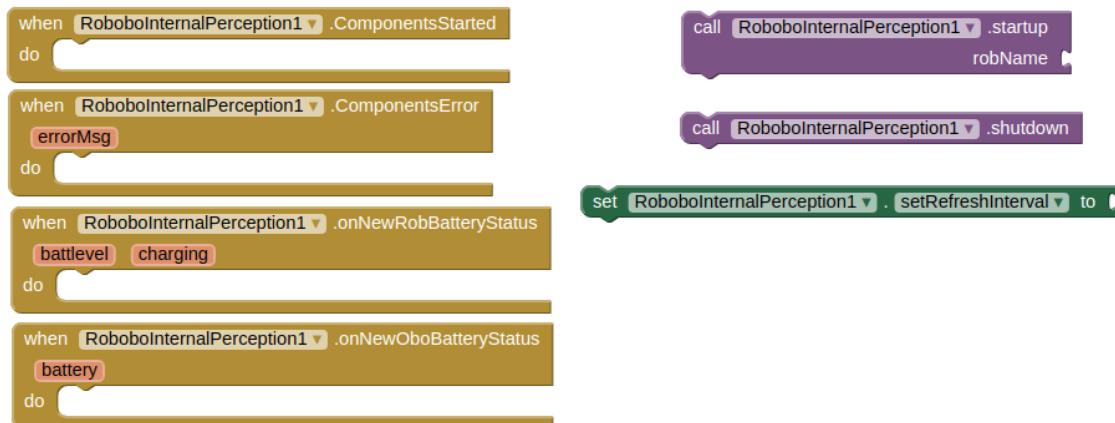


Figura 4.67: Bloques de interocepción.

#### 4.4.5.3. Demostración

Como nos dous casos anteriores, a demostración desta iteración ha consistir en desenvolver unha aplicación de proba que empregue todo o feito nas anteriores e ademais engada algún comportamento empregando os componentes desenvolvidos nesta iteración. Non obstante, debido a que para facer a proba cos niveis de batería habería que esperar á descarga da base ou do móvil, a aplicación de proba desta

iteración non ha facer uso do compoñente de interocepción. O obxectivo desta proba, por tanto, vai ser facer que o robot se queixe cando estea parado se alguén o move.

Para cumplir este obxectivo, a aplicación vai empregar o acelerómetro e o xiroscoPIO, de xeito que se detecta unha aceleración superior a un límítar establecido, ou calquera cambio de orientación, o robot di que pares. É certo que comprobando soamente a orientación basta para conseguir este efecto, pero dado que non tódolos móbiles teñen xiroscoPIO, decidiuse empregar tamén a aceleración.

```

private void complain() {
    try {
        if (!movementVisionSound.started())
            movementVisionSound.sayText("Eh! Para!");
    } catch (RemoteException e) {
        String errorStr = "Handler is no longer available";
        Log.e(LOG_TAG, errorStr);
        onComponentsError(errorStr);
        stop();
    }
}

@Override
public void onAccelerationChange() {
    complain();
}

@Override
public void onOrientationChanged(float yaw, float pitch, float roll) {
    complain();
}

```

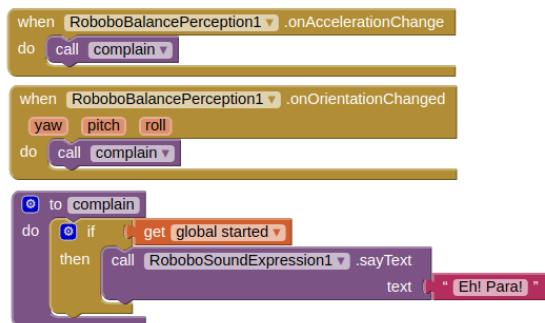


Figura 4.68: Queixas do robor por movelo.

Unha vez máis, volvemos comprobar como o uso dos bloques do segundo nivel de abstracción da nova API facilita o traballo en moitos aspectos. Neste caso, por exemplo, a gran diferencia é que non resulta necesario manexar as excepcións que poden lanzar os métodos do primeiro nivel de abstracción, tal e como podemos

comprobar no último fragmento de código e na figura 4.68; pero aínda así non se debe esquecer o caso da primeira iteración, xa que algo que se podía facer cunha pequena e sinxela función, no segundo nivel de abstracción convertíase nun conxunto de bloques considerable que complicaría a depuración de calquera tipo de erro.



# Capítulo 5

## CONCLUSIÓNS

Dende a aparición dos primeiros ordenadores ata agora, case 80 anos despois, a programación leva sendo unha actividade practicada soamente por persoas cunha formación específica; científicos e enxeñeiros son case os únicos que contan cun mínimo de experiencia neste campo. Tamén dende os anos 60 se leva intentando cambiar esta situación procurando achegar a programación a outro público por medio de linguaxes como BASIC ou Logo, pero quizais pola falta de dispoñibilidade de ordenadores fora do ámbito laboral, non se deu transformado en realidade. Hoxe, que praticamente todo o mundo ten acceso a un ordenador gracias á gran difusión dos smartphones, pode que esa realidade estea máis preto de cambiar.

Cada vez máis, a tecnoloxía acompañaos no noso día a día axudándonos a realizar más cousas, e proximamente, gracias á chegada da internet das cousas, estaremos rodeados por máis dispositivos interconectados; dende electrodomésticos, ata a nosa propia roupa, levarán conexión a internet.

Tal crecemento no número de dispositivos ha disparar a demanda de profesionais capaces non só de empregalos e desenvolverlos, senón tamén de realizar un mantenemento apropiado dos mesmos, polo que parece case imprescindible que as escolas do futuro contén con materias e ferramentas coas que os alumnos se poidan iniciar no mundo da programación dunha maneira divertida e atractiva para eles. Do mesmo xeito que as matemáticas nos seus inicios eran unha disciplina practicada unicamente por estudiosos e intelectuais, e agora é materia obligatoria en tódalas escolas dende unha idade moi temperá, o mesmo pode ocorrer nos próximos anos coa programación e a robótica educativa. De feito, o gran número de produtos desta clase que podemos atopar actualmente no mercado, como algúns dos que se presentaron no capítulo de ANTECEDENTES, xa fai ver o crecente interese neste campo que

houbo nos últimos anos. Non obstante, a importancia da robótica educativa non radica soamente na demanda de profesionais da tecnoloxía que vai traer consigo a internet das cousas, senón tamén no que pode aportar ao mundo da docencia como ferramenta didáctica, onde realmente ten un gran potencial.

Actualmente as clases na gran maioría dos centros, dende a ensinanza primaria ata a universidade, a docencia baséase principalmente na sesión maxistral combinada con algúns espazos destinados ao traballo persoal do alumno de maneira individual e, soamente nalgunhas ocasións, en grupo. Débese destacar ademais que os traballos en grupo son poucas veces en equipo, xa que habitualmente divídense as tarefas entre os membros, e cada un fai a súa parte por separado, polo que ao final non hai case interacción, nin un proceso de toma de decisións entre os compoñentes. Neste sentido, a robótica educativa presenta unhas características favorables para mellorar o traballo en equipo e a comunicación dos seus integrantes, xa que áinda dividindo o traballo desta maneira, cada parte deberá estar ben conectada co resto para poder funcionar en conxunto, e polo tanto faise imprescindible un proceso previo de deseño e toma de decisións onde cada integrante aporte algo. Por exemplo, mediante a proposición de exercicios abertos onde sexan eles os encargados de escoller e desenvolver unha estratexia de resolución, non soamente estamos creando unha contorna na que desenvolver as súas capacidades de comunicación e traballo en equipo, senón tamén as de resolución de problemas e xeración de novas ideas.

Outro aspecto a ter en conta sobre os beneficios da robótica como ferramenta didáctica é que permite a aprendizaxe simultánea de capacidades relacionadas coas diferentes materias. Dende disciplinas STEM, ata as artes e as letras pódense beneficiar do emprego desta ferramenta. Por exemplo, mediante a redacción do proceso de desenvolvemento do proxecto dun xeito semellante a esta memoria, o alumno pode mellorar a súa capacidade de expresión oral e escrita; nunha clase de debuxo artístico poderíase propoñer o deseño de novas caras, ou a posta en escena dun comportamento que se debe desenvolver co robot; ou nunha clase de música poderían programar un ou varios exemplares de xeito que reproduzan unha melodía e un acompañamiento creados por eles mesmos. Ademais, tamén poden ser os alumnos os que propoñan novos exercicios para estudantes de cursos anteriores baseándose na súa propia experiencia, e nas dificultades que foron atopando; de xeito que o catálogo de exercicios disponible sexa máis completo, e fomentando unha maior interacción entre profesores e alumnos.

Relacionado con isto último, outra actividade que podería formar parte do catálo-

## CAPÍTULO 5. CONCLUSIÓNS

---

go de exercicios para os alumnos máis avanzados, sería a expansión da API simplificada; cousa que pode parecer complexa de máis para programadores con pouca experiencia. Sen embargo, o sistema desenvolvido neste traballo permite a súa expansión moi facilmente; sexan novos compoñentes, ou unha extensión dos xa existentes, o desenvolvemento dos mesmos é tan sinxelo que pode ser unha das prácticas propostas para quen desexe estudar programación orientada a obxectos. De feito, dende o punto de vista da implementación deste traballo, o deseño do protocolo foi realmente o punto chave, pois grazas a súa estrutura, o desenvolvemento dos compoñentes resulta moi fácil e rápido, de xeito que o pode facer calquera cuns mínimos coñecementos de Java e programación orientada a obxectos.

Por último, outro elemento crucial para o éxito do robot como ferramenta didáctica, é a posibilidade de realizar unha aprendizaxe gradual grazas aos dous niveis de abstracción, de xeito que os alumnos poidan progresar sen atopar grandes dificultades en ningún dos pasos do proceso, e favorecendo que no futuro a programación sexa unha actividade practicada por máis xente fora dos ámbitos das ciencias e da enxeñería.

---

*CAPÍTULO 5. CONCLUSIONES*

# Apéndice A

## Manual de usuario

Neste apéndice faremos unha pequena guía de uso da nova API simplificada dende os dous niveis de abstracción existentes, describindo os pasos a seguir para empezar o desenvolvemento de aplicacións para o robot.

### A.1. Instalación das aplicacións

Antes de nada, para permitir que a API simplificada funcione, deberemos instalar a aplicación que contén o servizo e os handlers cos que se comunican os compoñentes, e que está dispoñible a través de internet na páxina [www.theroboboproject.com/simpleapi](http://www.theroboboproject.com/simpleapi).

### A.2. Primerio nivel de abstracción: Java

#### A.2.1. Resolución de dependencias

O primeiro que se debe facer é configurar a contorna de desenvolvemento – Android Studio– para que obteña as dependencias de Maven Central, para o que deberemos engadir unha liña no ficheiro de configuración de Gradle que corresponda. Esa liña tería un aspecto como a seguinte:

```
compile 'com.theroboboproject:robobo-simpleapi:x.x.x'
```

Onde indicamos primeiro o dominio do paquete, *com.theroboboproject*; o nome da librería, *robobo-simpleapi*; e por último, a cadea *x.x.x*, que representa o número de versión que corresponda.

Unha vez feito isto, o IDE descarga tódalas dependencias necesarias e podemos empezar a programar sen necesidade de configurar nada máis.

### A.2.2. Obtención dos componentes

Primeiro de todo deberemos crear unha clase que debe implementar a interface *ISimpleRoboboManager*, e que ha ser a encargada de obter unha instancia do xestor de componentes –*SimpleRoboboManager*–, e de recibir as notificacións sobre o estado dos módulos para saber se o seu inicio foi correcto ou non. Os dous métodos que se deben impelmentar para tal propósito son *onComponentsStarted* e *onComponentsError*. A continuación amosamos un pequeno exemplo empregando unha clase Android:

```
public class ManualActivity extends AppCompatActivity implements ISimpleRoboboManagerListener {

    private SimpleRoboboManager simpleRoboboManager;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_manual);
        simpleRoboboManager = SimpleRoboboManager.getInstance();
    }

    @Override
    public void onComponentsStarted() {

    }

    @Override
    public void onComponentsError(String errorMsg) {

    }
}
```

Como vemos no fragmento anterior de código, a clase ten un atributo que contén unha referencia ao *SimpleRoboboManager*, e no evento *onCreate*, que indicaría a creación da actividade, obtemos unha instancia do xestor.

O seguinte paso sería subscribirse como observador do *SimpleRoboboManager*, e engadir os componentes que se vaian empregar tal e como amosamos a continuación:

```
Context context = getApplicationContext();
simpleRoboboManager.addListener(this);
try {
    simpleRoboboManager.addComponent(context, CorporalExpressionComponent.class);
    simpleRoboboManager.addComponent(context, VisualPerceptionComponent.class);
}
```

```

    simpleRoboboManager.addComponent(context, SoundPerceptionComponent.class);
    simpleRoboboManager.addComponent(context, SoundExpressionComponent.class);
} catch (ComponentNotFoundException e) {
    Log.e(LOG_TAG, "Component not found: " + e.getMessage());
}

```

Para engadir os compoñentes desexados debemos destacar que é imprescindible obter un obxecto *Context*, xa que o método require un parámetro deste tipo. Ademais, tamén deberemos ter en conta que este método pode lanzar a excepción *ComponentNotFound* no caso de que ese compoñente non estea dispoñible.

Agora que o xestor xa coñece os compoñentes que se desexan utilizar, o seguinte paso ha ser enlazalos e obter unha instancia de cada un para poder empregalos. Amosamos a continuación un exemplo:

```

simpleRoboboManager.bind(robName);
corporalExpressionComponent =
    simpleRoboboManager.getComponent(CorporalExpressionComponent.class);
visualPerceptionComponent = simpleRoboboManager.getComponent(VisualPerceptionComponent.class);
soundPerceptionComponent = simpleRoboboManager.getComponent(SoundPerceptionComponent.class);
soundExpressionComponent = simpleRoboboManager.getComponent(SoundExpressionComponent.class);

```

No fragmento de código anterior, o proceso de enlace do compoñente e o handler faise antes de obter unha instancia. Realmente isto non é estritamente necesario, xa que de calquera maneira deberemos esperar a recibir o evento *onComponentsStarted* antes de executar calquera dos métodos dos compoñentes. Non obstante, de facelo así, a única diferencia sería que recibiremos antes a notificación.

## A.3. Segundo nivel de abstracción: App Inventor

### A.3.1. Importar a extensión

O primeiro que debemos facer é importar a extensión de App Inventor que nos da acceso á API simplificada que podemos atopar na URL: [www.theroboboproject.com/simpleapi](http://www.theroboboproject.com/simpleapi). Para isto temos que abrir a contorna de desenvolvemento de App Inventor, crear un novo proxecto, e escoller a opción *Import extension* que está na vista de deseño e baixo a categoría *Extension* do menú da esquerda; onde aparecen tódolos compoñentes que podemos empregar na nosa aplicación.

Para seleccionar a extensión temos dous posibilidades: podemos subir o ficheiro *aix* dende o noso equipo, ou ben establecer unha URL para que App Inventor a obteña automaticamente. Na figura A.1 podemos ver unha imaxe de exemplo deste proceso.

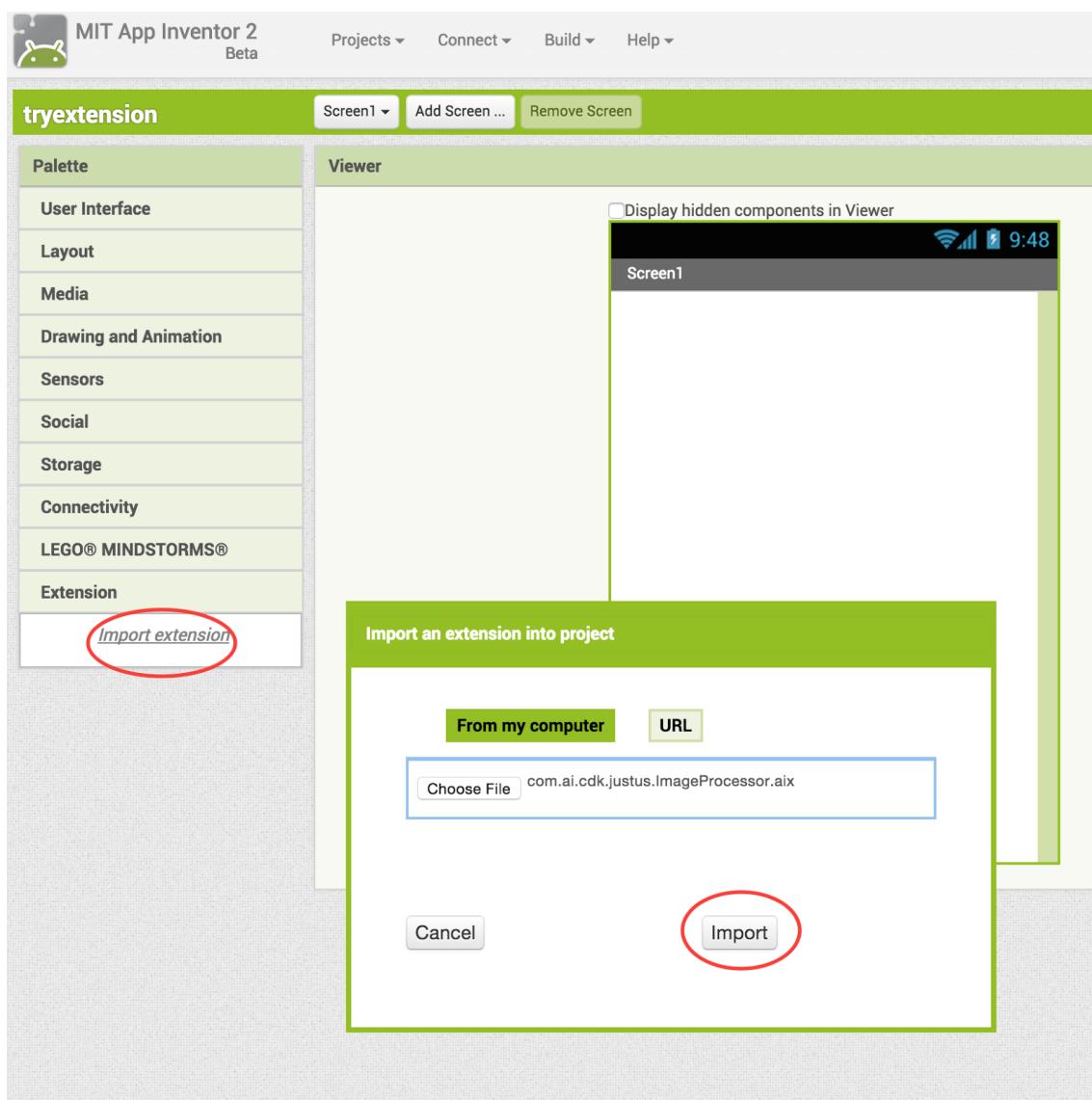


Figura A.1: Como importar unha extensión de App Inventor.

### A.3.2. Engadir os compoñentes

Unha vez temos dispoñibles os compoñentes da API simplificada, só deberemos escoller os que vaimos empregar, e arrastralos co rato enriba da pantalla do móvil; automaticamente apareceran nunha lista a dereita da pantalla, e como iconas na parte de abaixo xunto aos compoñentes non visibles. Podemos ver unha imaxe desta situación na figura A.2.

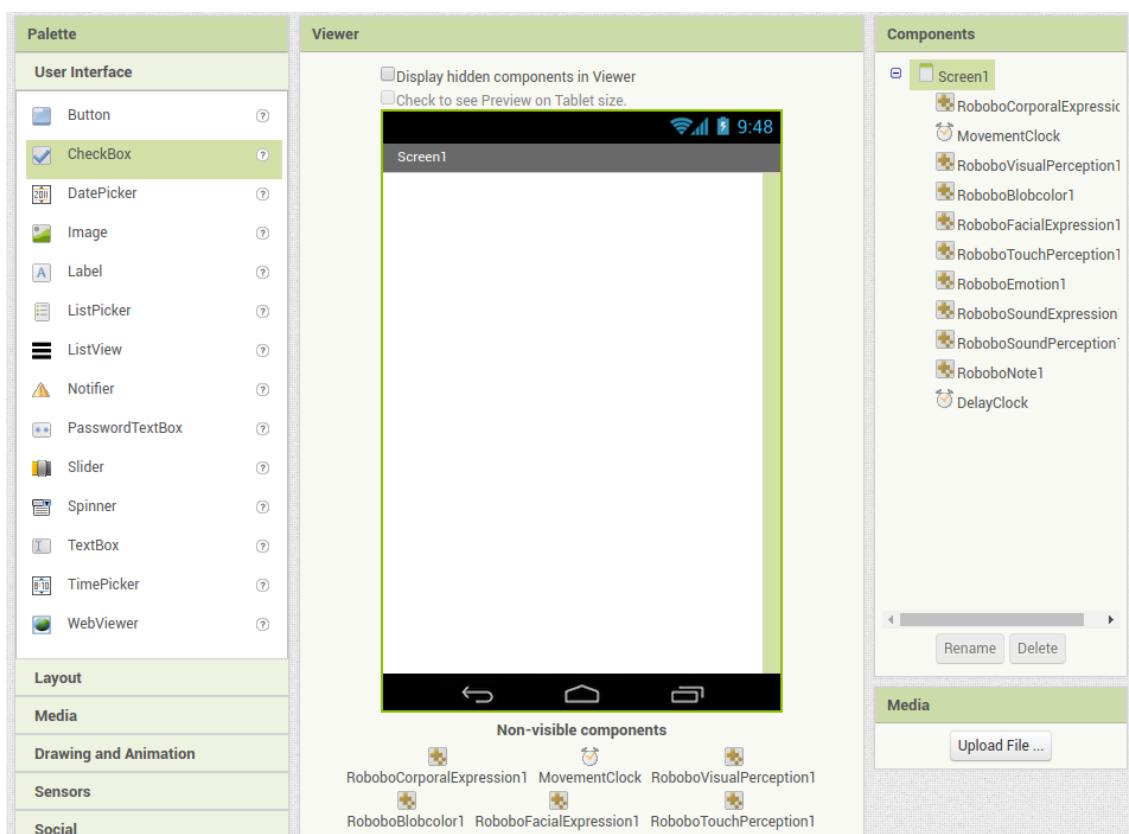


Figura A.2: Vista dos componentes engadidos na vista de deseño de App Inventor.

### A.3.3. Uso dos bloques

Para empezar a programar comportamentos empregando os bloques definidos para cada componente primeiro deberemos pasar da vista de deseño á de bloques a través do botón da esquina superior dereita. Nesta nova vista atoparemos á esquerda tódalas categorías que temos dispoñibles, onde poderemos escoller os bloques que desexemos empregar.

O primeiro que debe facer toda aplicación para o Robobo é enlazar o ROB que corresponda; para o que deberemos clicar no obxecto que representa a pantalla –por defecto recibe o nome de *Screen1*–, e arrastrar o bloque do evento que notifica o seu inicio. Dentro deste, deberemos colocar o bloque que permite iniciar *–startup*– o componente en cuestión, e establecer o nome do ROB por medio dun campo de texto. Por último, para empezar a executar comportamentos, temos que escoller o evento de notificación de inicio do componente que desexemos empregar, e arrastrar ao seu interior os bloques co comportamento desexado. Na figura A.3 podemos ver mediante imaxes os catro pasos descritos anteriormente.

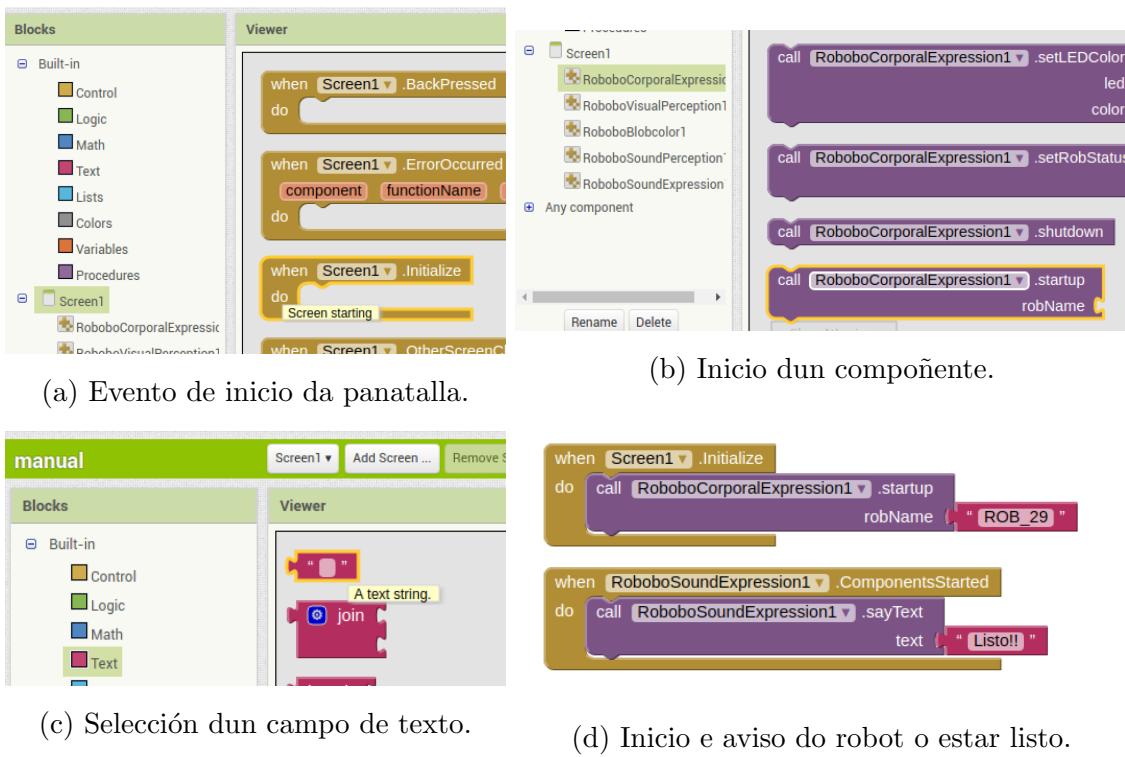


Figura A.3: Pasos para empezar a emplegar os componentes.

Débese destacar que para iniciar tódolos componentes de App Inventor que se vayan emplegar non é necesario executar os métodos *startup* de cada un deles, xa que coa execución dun deles xa se inicien tódolos componentes que inclúan na aplicación. Sen embargo, cos bloques de *shutdown* non ocorre o mesmo; neste caso, para única e exclusivamente o componente do bloque executado, áinda que realmente non é necesario facelo.

Outra cousa a ter en consideración é que, para empezar a execución de comportamentos dun componente determinado, resulta recomendable emplegar os seus eventos de notificación de inicio ou erro. É certo que en moitos casos, sobre todo cando hai poucos componentes involucrados, non existe diferencia entre usar o seu evento propio ou o de outro componente. Non obstante, xa que pode dar lugar a erros, é preferible evitalo.

# Bibliografía

- [1] *D.R.E.A.M. Project.* <http://www.robotsthatdream.eu/>.
- [2] Ullrich Wagner, Steffen Gais, Hilde Haider, Rolf Verleger, and Jan Born. “Sleep inspires insight”. In: *Nature* 427.6972 (2004), pp. 352–355.
- [3] *Robobo Project.* <http://www.theroboboproject.com/>.
- [4] *History of Computers.* <https://en.wikipedia.org/wiki/Computer>.
- [5] *ENIAC.* <https://en.wikipedia.org/wiki/ENIAC>.
- [6] Raúl Rojas, Cüneyt Göktokin, Gerald Friedland, Mike Krüger, Olaf Langmack, and Denis Kuniß. “Plankalkül: The first high-level programming language and its implementation”. In: *Freie Universität Berlin* (2000).
- [7] *Fortran.* <https://en.wikipedia.org/wiki/Fortran>.
- [8] *LISP.* [https://en.wikipedia.org/wiki/Lisp\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Lisp_(programming_language)).
- [9] *BASIC.* <https://en.wikipedia.org/wiki/BASIC>.
- [10] *Logo.* [https://en.wikipedia.org/wiki/Logo\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Logo_(programming_language)).
- [11] *AgentSheets.* <https://en.wikipedia.org/wiki/AgentSheets>.
- [12] *AgentSheets.* <http://www.agentsheets.com/products/index.html>.
- [13] *Alexander Repenning.* [https://en.wikipedia.org/wiki/Alexander\\_Repenning](https://en.wikipedia.org/wiki/Alexander_Repenning).
- [14] *Squeak.* <https://en.wikipedia.org/wiki/Squeak>.
- [15] *Alice.* <http://www.alice.org/>.
- [16] *Alice.* [https://en.wikipedia.org/wiki/Alice\\_\(software\)](https://en.wikipedia.org/wiki/Alice_(software)).
- [17] *NetLogo.* <https://en.wikipedia.org/wiki/NetLogo>.
- [18] *NetLogo.* <https://ccl.northwestern.edu/netlogo/>.

- [19] *Scratch*. [https://en.wikipedia.org/wiki/Scratch\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Scratch_(programming_language)).
- [20] *Scratch*. <https://scratch.mit.edu/about>.
- [21] *La robótica educativa ayuda a los alumnos a razonar; eso vale para Informática y para Filosofía*. [http://www.eldiario.es/norte/navarra/ultima\\_hora/robotica-educativa-alumnos-Informatica-Filosofia\\_0\\_293621134.html](http://www.eldiario.es/norte/navarra/ultima_hora/robotica-educativa-alumnos-Informatica-Filosofia_0_293621134.html).
- [22] *Qué es la robótica educativa*. <https://www.edukative.es/que-es-la-robotica-educativa/>.
- [23] *BeeBot*. <https://www.bee-bot.us/>.
- [24] *BQ Zoqi*. <https://www.xataka.com/analisis/probamos-zowi-un-robot-con-cerebro-arduino-que-puede-dar-mas-de-lo-que-aparenta>.
- [25] *LEGO WeDo*. <http://ro-botica.com/tienda/LEGO-Education/LEGO-Education-WeDo-2/>.
- [26] *LEGO Mindstorms*. <https://www.ro-botica.com/es/tienda/LEGO-Education/LEGO-MINDSTORMS-Education-EV3/>.
- [27] *LEGO Mindstorms*. [https://es.wikipedia.org/wiki/Lego\\_Mindstorms](https://es.wikipedia.org/wiki/Lego_Mindstorms).
- [28] *FischerTechnik*. <https://www.ro-botica.com/es/tienda/Fischertechnik/>.
- [29] *FischerTechnik*. <http://www.fischertechnik.de/en/Home/products/computing.aspx>.
- [30] *MakeBlock*. [https://makeblock.es/que\\_es\\_makeblock/](https://makeblock.es/que_es_makeblock/).
- [31] *WowWee COJI*. <http://blog.bestbuy.ca/toys/toys-games-hobbies/review-wowwee-coji-the-coding-robot>.
- [32] *WowWee COJI*. <http://wowwee.com/coji>.
- [33] *WowWee Coder MIP*. <http://www.techagekids.com/2016/09/wowwee-coder-mip-review.html>.
- [34] *NAO*. <http://aliverobots.com/nao/>.
- [35] *NAO*. [https://en.wikipedia.org/wiki/Nao\\_\(robot\)](https://en.wikipedia.org/wiki/Nao_(robot)).
- [36] *ROS*. <http://www.ros.org/>.
- [37] *App Inventor*. <http://appinventor.mit.edu/>.

- [38] *App Inventor*. [https://en.wikipedia.org/wiki/App\\_Inventor\\_for\\_Android](https://en.wikipedia.org/wiki/App_Inventor_for_Android).
- [39] *Equilibriocepción*. <https://es.wikipedia.org/wiki/Equilibriocepcin>.
- [40] *Interocepción*. [https://es.wikipedia.org/wiki/Funcin\\_interoceptiva](https://es.wikipedia.org/wiki/Funcin_interoceptiva).