

Un gato blanco para el autonomismo dinámico

Bernat Romagosa i Carrasquer

Experiencia de Edutec (Citilab)
en la construcción de herramientas
para la robótica educativa.

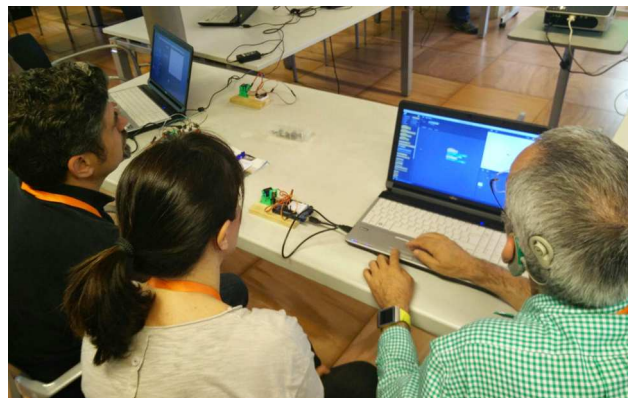
Generalizar está muy mal visto, y clasificar a personas en grupos sin antes realizar una encuesta exhaustiva y científicamente correcta acostumbra a ser una muy mala idea. En cualquier caso, en este artículo vamos a ser valientes (o temerarios) y clasificaremos, generalizaremos, realizaremos asunciones solamente fundadas en nuestra experiencia personal y pondremos nombre a corrientes que quizás ni siquiera existan. Para empezar a generalizar y clasificar, establezcamos que en el mundo de la robótica educativa existen tres corrientes de pensamiento distintas para las que –hasta donde yo sé– todavía no se ha inventado nombre.

La primera, llamémosle **autonomismo estático**, entiende que un robot debe ser un ente autónomo que se programa en un lenguaje compilado y estático desde un segundo dispositivo, como un ordenador personal. Lo que prima en esta corriente es el hecho de que el robot pueda trabajar por sí solo, desconectado de cualquier otro dispositivo, y que toda su lógica esté integrada en su cerebro electrónico. En general, los defensores del autonomismo estático proceden, o bien del mundo de la electrónica, o bien del mundo de los lenguajes textuales clásicos y compilados como C, Java o Processing.

La segunda, que nombraremos **dinamismo dependiente**, defiende que el robot es simplemente un aparato controlado por un dispositivo más potente que alberga un lenguaje de programación dinámico e interpretado en tiempo real. En esta corriente se pone por delante la posibilidad de modificar y construir la inteligencia artificial del robot en vivo y de una forma constructorista, y se entiende como un simple inconveniente el hecho de que el robot y el ordenador necesiten de una interconexión física, ya sea por cable o mediante ondas de cualquier tipo. Los defensores del dinamismo dependiente provienen más bien de lenguajes dinámicos e interpretados como Lisp, Smalltalk o, incluso, Scratch. Los más veteranos aprendieron o enseñaron a programar en LOGO.

La tercera corriente, pongámosle por nombre **autonomismo dinámico**, bebe de las dos corrientes anteriores, tomando de ellas sus puntos fuertes y descartando los débiles. A saber: el robot debe ser programado dinámicamente y en tiempo real, pero al mismo tiempo tiene que poder liberarse de su dispositivo maestro en cualquier momento. Los autonomistas dinámicos creen que debería existir una herramienta que concilie las dos primeras corrientes, y presionan a unos y a otros para que los sistemas autónomo-estáticos o dinámico-dependientes solucionen sus respectivas pegadas y tomen de su corriente opuesta los puntos fuertes. Los integrantes de este grupo acostumbran a ser personas profesionalmente alejadas tanto de la electrónica como de la programación. Aquí caben, entre otros, profesionales de la enseñanza y la pedagogía o autodidactas tecnológicos.

Los integrantes de las dos primeras corrientes son, a menudo, constructores de herramientas basadas en sus principios, mientras que los autonomistas dinámicos son, también a menudo, los usuarios finales de estas herramientas. Así pues, parece que algo horrible está sucediendo en el panorama de la robótica educativa, y es que, aparentemente, los constructores de herramientas están ignorando por completo las necesidades y peticiones de sus usuarios.





Què és Edutec

El projecte Edutec és una iniciativa de caire essencialment docent. Es caracteritza per la seva premissa de partida: l'aprenentatge de la programació d'ordinadors és el que fa possible que el ciutadà passi de ser consumidor passiu de tecnologia a ser participant actiu del món tecnològic. A més, les conseqüències derivades d'aquest aprenentatge tenen com a resultat canvis en la metodologia de resolució de problemes (generals, no només informàtics) molt adequats al nostre entorn social actual.

Objectius

L'objectiu principal d'Edutec és la divulgació i difusió de la programació d'ordinadors.

A Edutec treballam sota la hipòtesi que tothom ha de poder aprendre a programar. En aquest sentit, estem en constant recerca i desenvolupament d'eines i mètodes que ens permetin apropar conceptes de diferents àmbits de la computació a la població.

Context

La realidad es que, afortunadamente, no todos los constructores de herramientas somos integristas de nuestras respectivas corrientes. El motivo por el que parece que no escuchemos a nuestros usuarios es que la conciliación de ambas corrientes es un problema de difícil solución técnica, y las aproximaciones que podemos realizar con poco esfuerzo son siempre absolutamente insuficientes.

En el grupo Edutec del Citilab (Cornellà de Llobregat, Barcelona) somos constructores de herramientas dinámico-dependientes. Recordemos: manipulación directa, dinamismo, y cables. Como dinamistas dependientes, recibimos a menudo peticiones y críticas de usuarios que quieren deshacerse, no solamente del cable, sino de cualquier tipo de atadura entre el ordenador y el robot, aunque ésta sea una conexión inalámbrica vía Bluetooth.

Esto nos sucedió primero con S4A, una modificación de Scratch 1.4 desarrollada en el Citilab el año 2009 que permite controlar la placa Arduino. Más recientemente, estas peticiones las estamos recibiendo con cierta regularidad respecto a nuestra nueva herramienta, Snap4Arduino, una modificación de Berkeley Snap! que también permite controlar la placa Arduino.

Examinemos primero los motivos de nuestro posicionamiento que, recordemos, sitúa delante de todo la posibilidad de construir sistemas en tiempo real y de forma constructorista, sacrificando por el camino la capacidad de autonomía de estos sistemas.

En la teoría constructorista, que deriva del constructivismo de Piaget, se trabaja bajo la hipótesis de que el estudiante avanza en su camino de aprendizaje mediante la experimentación, el uso de información ya conocida y la existencia de un proyecto en el horizonte. En otras palabras, el constructorismo defiende que el aprendizaje se desarrolla mejor cuando vislumbramos un objetivo final y vamos adquiriendo los conceptos y métodos en el momento en que los necesitamos para la consecución de los distintos pasos hacia dicho objetivo.

Alejémonos por un instante del caso que nos ocupa e imaginemos que las herramientas de las que hablamos no tienen nada que ver con la programación de computadoras. Pongamos por caso que nuestros alumnos tienen entre manos un proyecto de construcción de castillos de arena. Idealmente, queremos que nuestros aprendices asimilen conceptos sobre materiales, estructura y diseño. Veamos las distintas posturas que, en un ejercicio de reducción al absurdo, tomarían un constructor de castillos tradicional y un constructorista.

El primero empezaría enseñando a los proyectistas todos los conceptos involucrados en la construcción de castillos. Primero hay que conocer las propiedades de la arena, aprender cómo el material cambia de densidad y adquiere adherencia al mezclarse con el agua, memorizar unas cuantas fórmulas sobre las relaciones entre peso y volumen de sólidos, aprender a clasificar los castillos de arena en distintas categorías de tamaño, forma y tipos de materiales utilizados y, por supuesto, dedicar un tiempo al estudio de los distintos tipos de palas, rastrillos y cubos existentes, así como las formas que podemos conseguir con cada una de estas herramientas.

Solamente cuando el alumno conozca todos estos conceptos pasaremos a la fase de trazar un plano del castillo sobre el papel, siempre de acuerdo con unas convenciones simbólicas que deberemos aprender y respetar. Finalmente, llegará el ansiado momento de construir el castillo, tarea que no realizará el alumno, sino que será delegada a una tercera persona (el compilador de castillos) que sabrá interpretar el plano hasta el más mínimo detalle. Si algo falla en la construcción, será consecuencia de que el aprendiz no ha estado escuchando cuando se le explicaban los distintos símbolos a usar en el plano o las propiedades de la arena y el agua salada, así que habrá que destruir el castillo por completo, modificar los planos y mandarlo a reconstruir de acuerdo con las nuevas especificaciones.

El resultado del uso de este método será, probablemente, que pocos estudiantes habrán conseguido construir un castillo en condiciones. Por supuesto, algunos habrán aprendido todos los conceptos y los habrán utilizado para diseñar el castillo. Otros habrán hecho trampas, memorizando mecánicamente los conceptos para poder pasar a la fase de diseño, donde habrán copiado de los planos del castillo del estudiante de al lado, habrán necesitado la ayuda del maestro constantemente, habrán sido ayudados por alumnos brillantes que ya han entregado sus planos, o habrán encontrado algún plano básico en Internet y lo habrán copiado, añadiéndole quizás una ventana aquí o una tira de almenas allá.

Por contra, el constructorista no cree que levantar castillos de arena sea una tarea tan complicada. Desde luego, si en su infancia él consiguió levantar unos cuantos de forma autodidacta, cualquier persona puede hacerlo. Lejos de preparar un temario subdividido en lecciones y subapartados, él comenzará el curso llevando a los alumnos a la playa y abriendo una caja de palas, cubos y rastrillos.

A medida que los aprendices vayan construyendo sus estructuras, el constructorista empleará diversas estrategias para que, por el camino, éstos adquieran los conceptos deseados sobre diseño, materiales y estructura. ¿Se ha fijado este alumno en el castillo de su compañero? Quizás debería preguntarle cómo lo ha hecho para que el puente del foso no se desmorone. La muralla almenada de esta otra alumna es preciosa, ¡vamos a pedirle que se la enseñe a toda la clase! Hay una pequeña arquitecta que, desde muy pequeña, construye castillos cada verano con su familia, y le comenta al constructorista que nunca han conseguido levantar una torre de más de un metro y medio de altura. ¿Es posible que sea ésta una limitación física del sistema, y no un problema en su diseño? ¿Podríamos dotar a la torre de alguna suerte de armadura que nos permita salvar el obstáculo? Seguro que alguien ya se ha encontrado con un problema similar anteriormente, investiguémolo.

Cuando todo parece ir bien, el constructorista descubre a un par de aprendices que, en lugar de un castillo, han cavado juntos un pozo hasta encontrar agua. En lugar de regañarles por no completar la tarea asignada, podemos pararnos a observar si el nivel del agua de dentro del pozo es el mismo que el del mar. ¿Cuál es el diámetro más estrecho posible para que el pozo no termine sepultándose solo? Sería fabuloso conectar este pozo con el foso del resto de castillos para proporcionarles agua, pero ¿cómo hacer que el agua venza la ley de la gravedad?

Siguiendo con la metáfora de los castillos, llegará un momento en que esa alumna brillante y ávida constructora de castillos de arena quiera convertirse en una constructora de casas de verdad, y quizás entonces deberá aprender a dibujar planos concisos. El resto de alumnos que sólo querían construir un castillo de arena no tienen por qué ser torturados con convenciones sobre simbología y nomenclatura antes de poder empezar a disfrutar de la actividad, cuando un simple castillo de arena bastaba.

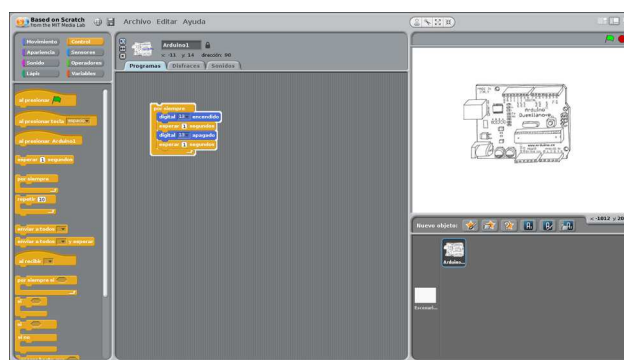
Volvamos al tema principal del artículo y, a riesgo de parecer demagógicos, planteemos la siguiente pregunta: ¿es mejor sacrificar la autonomía de un robot para poderlo programar de forma dinámica y mediante manipulación directa, o es tan importante que el robot se valga por sí solo que podemos sacrificar la interactividad y la programación en tiempo real en su favor?

En el equipo Edutec del Citilab creímos en su momento que lo más importante era conservar la manipulación directa y la posibilidad de construir castillos (o robots, qué más da) de una forma iterativa y divertida. Cualquier proceso extraordinario que el aprendiz tenga que realizar entre el momento en que tiene la idea de añadirle una ventana al castillo y el momento de ver cómo el interior de su castillo gana en luminosidad, es una barrera arbitraria para su aprendizaje.

En ese sentido, Scratch es un lenguaje-entorno de programación que encaja perfectamente con la idea del constructorismo, porque en sus principios está la eliminación del máximo de barreras de entrada posibles. En el artículo "Scratch: Programming for All", Mitchel Resnik y su equipo del Life Long Kindergarten decían sobre este lenguaje:

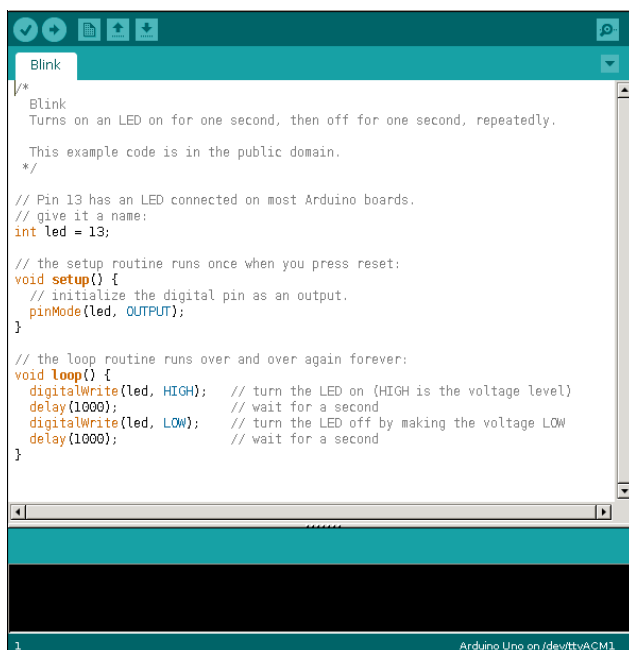
"Papert defendió que los lenguajes de programación deberían tener un "suelo bajo" (facilidad para empezar) y un "techo alto" (oportunidades para crear proyectos de complejidad creciente con el tiempo). Adicionalmente, los lenguajes necesitan "paredes anchas" (soporte para diversos tipos de proyectos, de forma que gente con distintos intereses y estilos de aprendizaje puedan motivarse por igual)."

Cumplir con estos requisitos es uno de los objetivos principales de Scratch. En 2009, en el Citilab queríamos construir una herramienta para enseñar robótica y electrónica digital, puesto que lo mejor que había en ese momento, educativamente hablando, era un lenguaje con el que solamente los alumnos con inclinaciones claramente tecnológicas y científicas podían disfrutar de la construcción de proyectos. Como Scratch era el lenguaje que cumplía con todos (o casi todos, hablaremos de ello más tarde) los requisitos constructoristas que tanto nos gustaban, decidimos simplemente adaptarlo para también dotarlo de conceptos de electrónica.



En Processing, el lenguaje oficial con que se programa Arduino, el alumno debe aprender primero una sintaxis, debe tener mucho cuidado de no equivocarse ni siquiera en un solo carácter, debe recordar convenciones de código que no entiende, pero que son imprescindibles para que su programa funcione, debe, además, memorizar toda una serie de palabras clave que esconden estructuras computacionales detrás de crípticos nombres. Además de todo esto, este lenguaje que tan complicado resulta para quien no tenga ya de por sí una inclinación tecnológica, ralentiza nuestro proceso de aprendizaje obligándonos, cada vez que queramos modificar cualquier detalle de nuestro programa, a pasar por un proceso de traducción de todo el programa entero a algo que el procesador de la placa sepa entender.

En definitiva, el suelo de Processing es demasiado alto, y sus paredes extremadamente estrechas. Solamente los alumnos que igualmente hubieran aprendido a programar aprenderán robótica o electrónica digital con un lenguaje así. Lo cierto es que el techo de Processing es mucho más alto que el de S4A, puesto que en S4A siempre necesitaremos una conexión física entre el ordenador y la placa, pero esperamos que se entienda el motivo de este sacrificio en pro de una mayor inclusividad.



```

Blink
/*
  Blink
  Turns on an LED on for one second, then off for one second, repeatedly.

  This example code is in the public domain.
  */

// Pin 13 has an LED connected on most Arduino boards.
// give it a name:
int led = 13;

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
  pinMode(led, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {
  digitalWrite(led, HIGH); // turn the LED on (HIGH is the voltage level)
  delay(1000);             // wait for a second
  digitalWrite(led, LOW);  // turn the LED off by making the voltage LOW
  delay(1000);             // wait for a second
}

```

En otras palabras, en lenguajes como Processing hay que enseñar a programar, mientras que lenguajes como Scratch facilitan que los alumnos aprendan a programar mientras juegan.

Seymour Papert, padre del construccionismo, hablaba de enseñar y aprender en una charla por videoconferencia que realizó en un congreso de educadores de Japón en los años 80:

"Mi trabajo se centra en ayudar a los niños a aprender, no sólo en enseñar. Ahora he inventado una expresión para ilustrarlo: construccionismo e instruccionalismo son denominaciones para dos aproximaciones a la innovación educativa. El instruccionalismo es la teoría que reza, "Para conseguir una mejor educación, es necesario mejorar la instrucción. Si vamos a usar computadoras, haremos que sean las computadoras quienes mejoren la instrucción". Esto da lugar a la idea de la enseñanza asistida por ordenador.

Bien, enseñar es importante, pero aprender lo es mucho más. El construccionismo significa "Ofrecer a los niños cosas buenas que hacer de forma que, durante ese proceso, puedan aprender mucho mejor de lo que aprendían antes". Es por eso que considero que las nuevas tecnologías son muy, muy ricas en tanto que proporcionan a los niños nuevas cosas para hacer de forma que puedan aprender las matemáticas como parte de algo real"

Hablando de alturas de techo, fue justamente la sensación de que Scratch no contaba con un techo lo suficientemente alto lo que llevó a Jens Möning y a Brian Harvey a desarrollar un nuevo lenguaje inspirado en Scratch pero que contaba con capacidades computacionales mucho más avanzadas. Con él, podían incluso aprenderse conceptos que la mayoría de recién (y no tan recién) licenciados en ingeniería informática ni siquiera han oído mencionar.

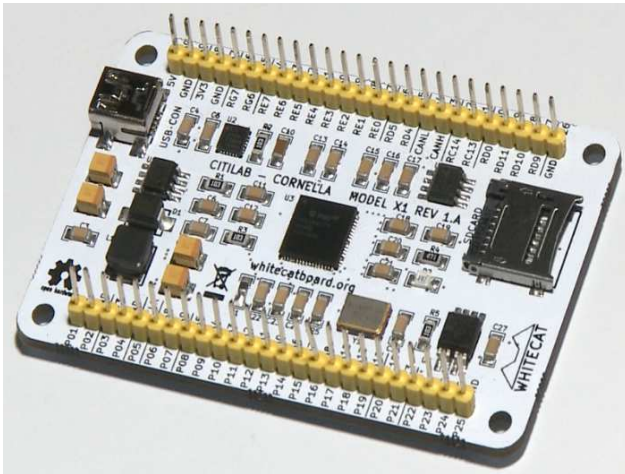
Este nuevo sistema, llamado Snap!, nos cautivó enseguida, y no pasó demasiado tiempo hasta que desarrollamos una modificación de este lenguaje para la plataforma Arduino. Con nuestra evidente falta de imaginación para bautizar herramientas, le pusimos a esta modificación el nombre de Snap4Arduino. Este nuevo sistema también contaba con la misma limitación de dependencia entre la placa y el ordenador, lo cual seguía siendo para nosotros una pequeña desventaja que se traducía en una gran limitación para el resto del mundo.

Todavía no hemos hablado demasiado de nuestra postura frente a la tercera corriente de pensamiento para la cual nos hemos inventado un nombre al principio de este artículo, el autonomismo dinámico. La verdad sea dicha, estoy convencido de que éste es el grupo que lleva la razón. Los motivos técnicos son problema de los técnicos y, por muchas ventajas que la aproximación constructorista de Snap4Arduino ofrezca respecto a otras alternativas, he enumerado todos esos argumentos a su favor con la boca pequeña, porque todos los dinamistas dependientes somos, en realidad, autonomistas dinámicos frustrados por la inexistencia de una herramienta que satisfaga nuestras necesidades. Somos dependientes por circunstancias técnicas, no por placer.

A principios de 2015, en el Citilab comenzamos a pensar en cómo debería ser un sistema que diera respuesta a esta tercera corriente huérfana de herramientas. Recapitulemos: la herramienta ideal nos permitiría programar el robot en tiempo real, mediante manipulación directa, sin procesos intermedios, en un lenguaje que no tuviera barreras de entrada mnemotécnicas y evitase las frustraciones derivadas de los errores de sintaxis o las convenciones de código. Además, y aquí reside la parte complicada, una vez tuviéramos el robot comportándose a nuestro antojo, tendríamos que poder desconectar el cable y liberarlo por la habitación sin ningún tipo de proceso de transferencia de archivos o traducción de código.

Un entorno así cubriría, también, un segundo propósito: difuminar tanto la línea entre prototipo y producto final como la línea entre el mundo educativo y el mundo semiprofesional. Para retomar la metáfora de los castillos, si encontrásemos arena lo suficientemente compacta como para construir casitas estables, quizás no haría falta que quien simplemente quiera hacerse su propio humilde hogar tenga que aprender las técnicas que los arquitectos de rascacielos y catedrales emplean. Lo fabuloso sería que pudiéramos saltar sin darnos cuenta de un juego de playa a la construcción de nuestra vivienda.

En medio de estas reflexiones, descubrimos que en CSSIberica e IberoXarxa, un par de empresas del entorno del Citilab, se estaba desarrollando una nueva placa de hardware de propósito general. Las características de esta placa la hacían atractiva para ser el punto de partida para la construcción de esta herramienta, ya que la placa contaría con un lenguaje de programación dinámico llamado Lua, a modo de sistema operativo. Gracias a algunas semejanzas técnicas entre Snap! y Lua, y gracias a la inmensa versatilidad de este lenguaje, debería ser posible implementar un entorno que nos permitiera programar esta placa mediante bloques y desconectarla en cualquier momento preservando su comportamiento.



Convencidos, nos liamos la manta a la cabeza y comenzamos a picar código a tres bandas para este nuevo sistema. Como guiño a la cantidad de animales que pueblan el mundo de la programación educativa, pusimos al sistema el nombre de WhiteCat, que se traduce como gato blanco. Gato, como evidente homenaje a la mascota de Scratch, y blanco por su pretensión de ser un sistema de propósito general, como un lienzo a estrenar en el que podemos pintar lo que se nos ocurra, sean cuales sean nuestras habilidades artísticas y nuestro estilo pictórico.

Después de muchos prototipos caseros, a día de hoy tenemos ya la placa en nuestras manos y, aunque el sistema de bloques no está completamente desarrollado, parece que la hipótesis inicial de que Lua sería un buen compañero de viaje se ha cumplido en gran parte. Digo en gran parte porque Lua, como muchos lenguajes del mundo profesional, no soporta paralelismo de por sí, pero los ingenieros de la WhiteCat han dotado a su versión particular de este lenguaje de esta característica, lo cual nos ha permitido no tener que renunciar a aquello que creemos que Scratch y Snap! hacen tan bien: camuflar la inmensa complejidad interna de la programación en paralelo.

En Scratch, desde el primer día estamos programando en paralelo. Siempre estamos construyendo distintos trozos de código que se ejecutan a la vez en varios objetos diferentes, y utilizamos mensajes para comunicar estos trozos de código entre sí de forma natural. En general, la falta de paralelismo es una carencia importante en los sistemas de bloques autónomo-estáticos, donde los bloques no son más que representaciones gráficas de un lenguaje textual estático y compilado, y en el mundo de la electrónica son muy pocos los lenguajes que soportan paralelismo de forma sencilla.

Como novedad, la WhiteCat cuenta con abstracciones para comunicación entre placas y otros dispositivos, ya sea mediante protocolos estándar de la Internet de las cosas o mediante el protocolo industrial CAN, lo cual añade a su programación en paralelo y basada en eventos una capa de computación distribuida, permitiéndonos dividir nuestro sistema en diversos agentes que realizan tareas especializadas y se comunican entre sí.

Entre diversos problemas por solventar y bastantes limitaciones que todavía quedan por pulir, el sistema WhiteCat ofrece la sensación de programación en tiempo real, la manipulación directa y el automatismo automático que buscábamos. Podemos construir un programa de forma iterativa, probando sus partes paso a paso y experimentando en tiempo real. Cuando nos gusta el resultado, basta con desconectar la placa del ordenador y alimentarla con una batería externa para ver cómo el sistema funciona de forma autónoma. Uno de los secretos para que esto funcione reside en que todo sucede dentro la placa. El ordenador no está realizando prácticamente ningún trabajo, a parte de servir de puente entre la placa y nosotros, y todo lo que hacemos se va acumulando constantemente en la memoria de la WhiteCat.

En definitiva, como autonomistas dinámicos frustrados, hemos comenzado a andar un camino para salvar la falta de herramientas de este sector, y aunque nuestro gato blanco es todavía un pequeño cachorro inocente, a día de hoy ya puede levantar castillos de arena e incluso algún hogar.

Referencias

CITILAB (citilab.eu)

EDUTEC (edutec.citilab.eu)

PAPERT, S. "Constructionism vs. Instructionism". Disponible en: www.papert.org/articles/const_inst/const_inst1.html.

RESNICK, M. et altri. "Scratch: Programming for All". Disponible en: web.media.mit.edu/~mres/papers/Scratch-CACM-final.pdf.

S4A (s4a.cat)

SNAP4ARDUINO (s4a.cat/snap)

WHITECAT (whitecatboard.org)

Bernat Romagosa i Carrasquer



Ingeniero Técnico en Informática de Gestión por la Universitat Oberta de Catalunya (UOC). Forma parte del equipo Edutec del Citilab. Actualmente es el desarrollador principal de Snap4-Arduino, Beetle Blocks y el entorno de programación por bloques de la WhiteCat.

Twitter: [@bromagosa](https://twitter.com/bromagosa)