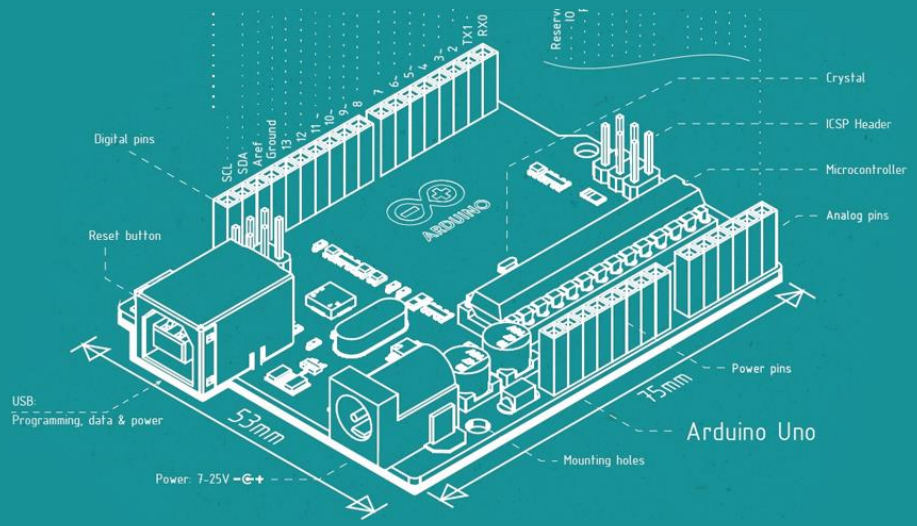


Bibliothèque d|board

Développement Arduino pour objets communiquants

Damien ALBERT | avril 2018

Agenda

Carte Arduino & système Grove
Logiciel Arduino et IDE
Un ordonnanceur pour Arduino
d|board, un nouvel ordonnanceur
d|board, exemple d'utilisation
d|board, pour la création d'objets complexes

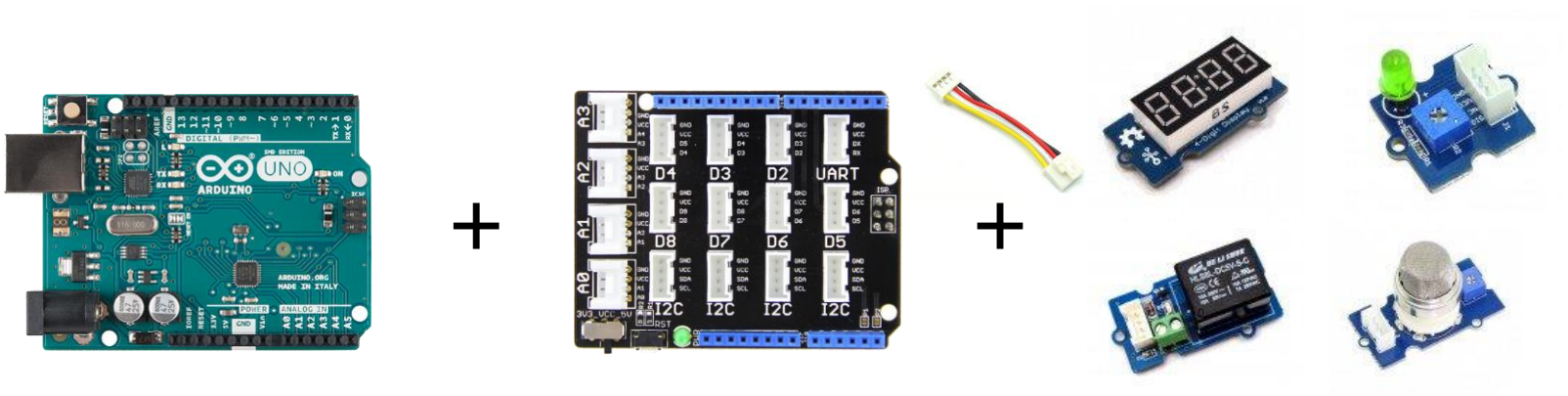
Carte Arduino & système Grove

La carte électronique Arduino

- Embarque un µcontroleur ATMEL (ATMEGA328p)
- Expose des entrées/sorties numériques & analogiques
- Permet d'interagir avec des composants câblés sur celui-ci

Le système Grove de Seeedstudio

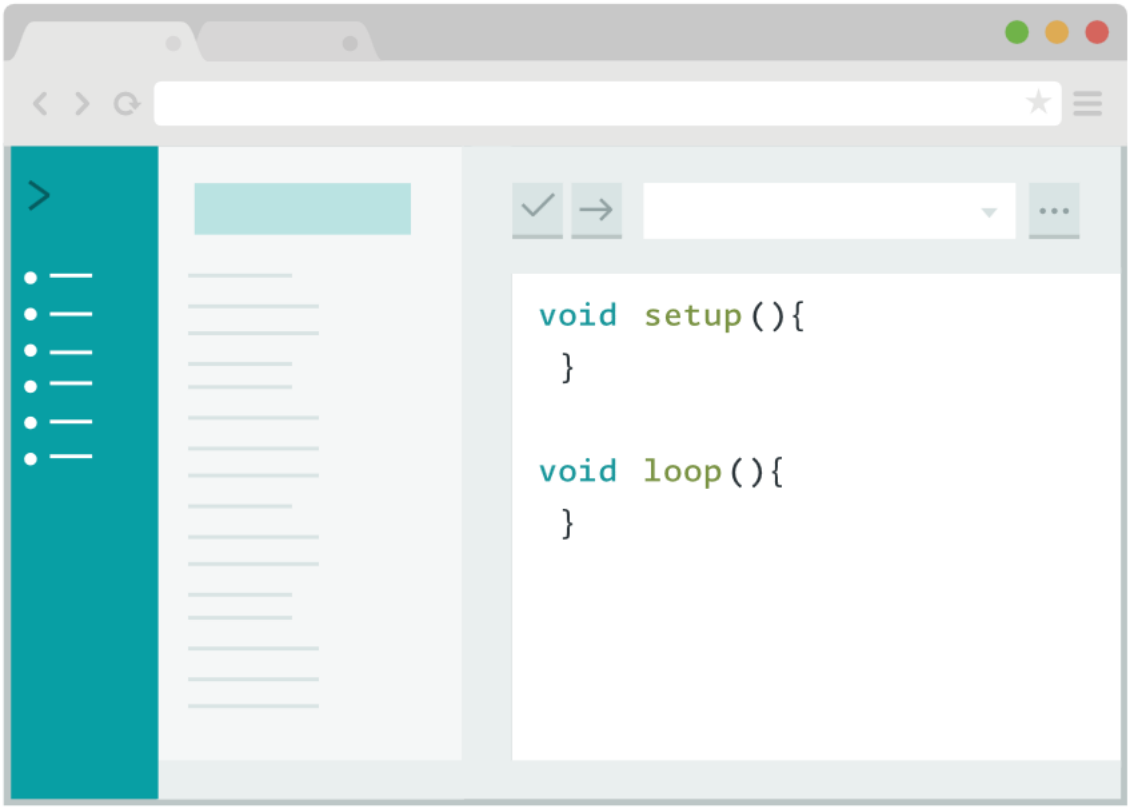
- Est un ensemble de composants plug-and-play open-source
- Permet de les brancher sans soudure ni carte de prototypage
- Repose sur un "shield" spécifique muni de connecteurs
- Assure un prototypage rapide et solide (et transportable)



Logiciel Arduino & IDE

L'environnement intégré de développement IDE

Propose un éditeur de programme simple d'utilisation
et une chaîne de compilation et de transfert des programmes
dont le corps principal s'articule autour de deux fonctions



Logiciel Arduino & IDE

Arduino & la programmation procédurale

L'écriture d'un programme Arduino consiste à remplir la boucle principale (i.e. **loop()**) après avoir fait les initialisations nécessaires (i.e. **setup()**) des différents composants câblés sur la carte électronique

Arduino et la gestion du temps

La gestion du temps se fait avec la fonction **delay()** qui introduit un délai d'attente entre deux lignes de code
Par exemple ci-dessous le code pour faire clignoter une LED :

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

Logiciel Arduino & IDE

Attente active *vs* partage du temps avec Arduino

L'appel à **delay()** empêche d'autres opérations (une autre LED à faire clignoter pour l'exemple le plus simple) de s'executer
Pour éviter cela, on utilise la fonction **millis()** qui retourne le nombre de millisecondes depuis le démarrage de la carte :

```
// Fonction loop(), appelée continuellement en boucle tant que la carte Arduino est alimentée
void loop() {

  // Récupére la valeur actuelle de millis()
  unsigned long currentMillis = millis();

  // Si BLINK_INTERVAL ou plus millisecondes se sont écoulés
  if(currentMillis - previousMillis >= BLINK_INTERVAL) {

    // Garde en mémoire la valeur actuelle de millis()
    previousMillis = currentMillis;

    // Inverse l'état de la LED
    etatBrocheLed = !etatBrocheLed;
    digitalWrite(BROCHE_LED, etatBrocheLed);
  }
}
```

Logiciel Arduino & IDE

Attente active *vs* partage du temps avec Arduino

- Dans le cas de plusieurs LED ou autres composants, il suffit d'écrire à la suite les différentes portions de code
- Les composants effectuent des opérations si leur intervalle respectif est écoulé et passe la main au composant suivant et ainsi de suite jusqu'au tour de **loop()** suivant

Inconvénient de la proposition

- Fastidieux si le nombre de composants augmentent
- Le code devient illisible avec une fonction **loop()** très longue
- On se perd facilement entre le code de composants totalement différents - il n'y a plus de séparation
- On donne la main aux différents composants sans tenir compte du moment ou il devront effectuer une opération

Un ordonnanceur pour Arduino

Une solution élégante : un **ordonnanceur** multitâches

La solution typique de ce genre de situation repose sur la création d'un ordonnanceur, un bout de code qui sera dédié à l'attribution du processeur aux différents composants retrouvant ainsi la clarté du programme principale

Programmation parallèle

Les composants s'exécutent ainsi presque en « parallèle » dans chacune de leur classe respective, de façon isolée
Les dialogues entre eux sont dès lors rendu compliqué à gérer
Cela abouti à une nouvelle approche de la programmation :

Programmation événementielle

En programmation événementielle, le déroulement n'est plus linéaire mais est contrôlé par la survenue d'événements, les actions de l'utilisateur mais également des capteurs

d|board, un nouvel ordonnanceur

Caractéristiques principales et atouts

#Efficacité

Ordonnanceur multitâches coopératif à priorité statique
Séquencement des tâches par échéancier

#Souplesse

Support de tâches multiples pour un même composant
Possibilité d'utiliser **delay()** n'importe où dans le code
Programmation et arrêt des tâches simplifiée
Support d'un nombre non limité de tâches

#Sécurité

Surveillance de la mémoire disponible
Gestion du débordement du compteur **millis()**
Support du watchdog-timer matériel du µcontrôleur
Avertissement sur dépassement de la tâche en cours
Avertissement des retards de lancement des tâches

d|board, un nouvel ordonnanceur

Des limitations

- Pas de gestion de la consommation électrique du µcontrôleur
- Pas de statistiques d'utilisation des tâches
- Pas de préemption/reprise des tâches
- Une empreinte mémoire de 500 octets (SRAM 2k disponible)

En comparaison

- Il existent des dizaines d'ordonnanceurs pour Arduino ...
- Certains sont très complexes, d'autres limités à un usage
- Pour d|board, le choix s'est porté sur la simplicité d'écriture du programme principal et la facilité d'extension de la bibliothèque de composants par inspiration, héritage et composition d'objets.
- Le principe est d'isoler et segmenter la complexité dans chacun des composants - principe général qui régit la bibliothèque standard d'Arduino.

d|board, un nouvel ordonnanceur

Architecture générale

Le code principal est porté par deux classes d'objet

La classe DBoard

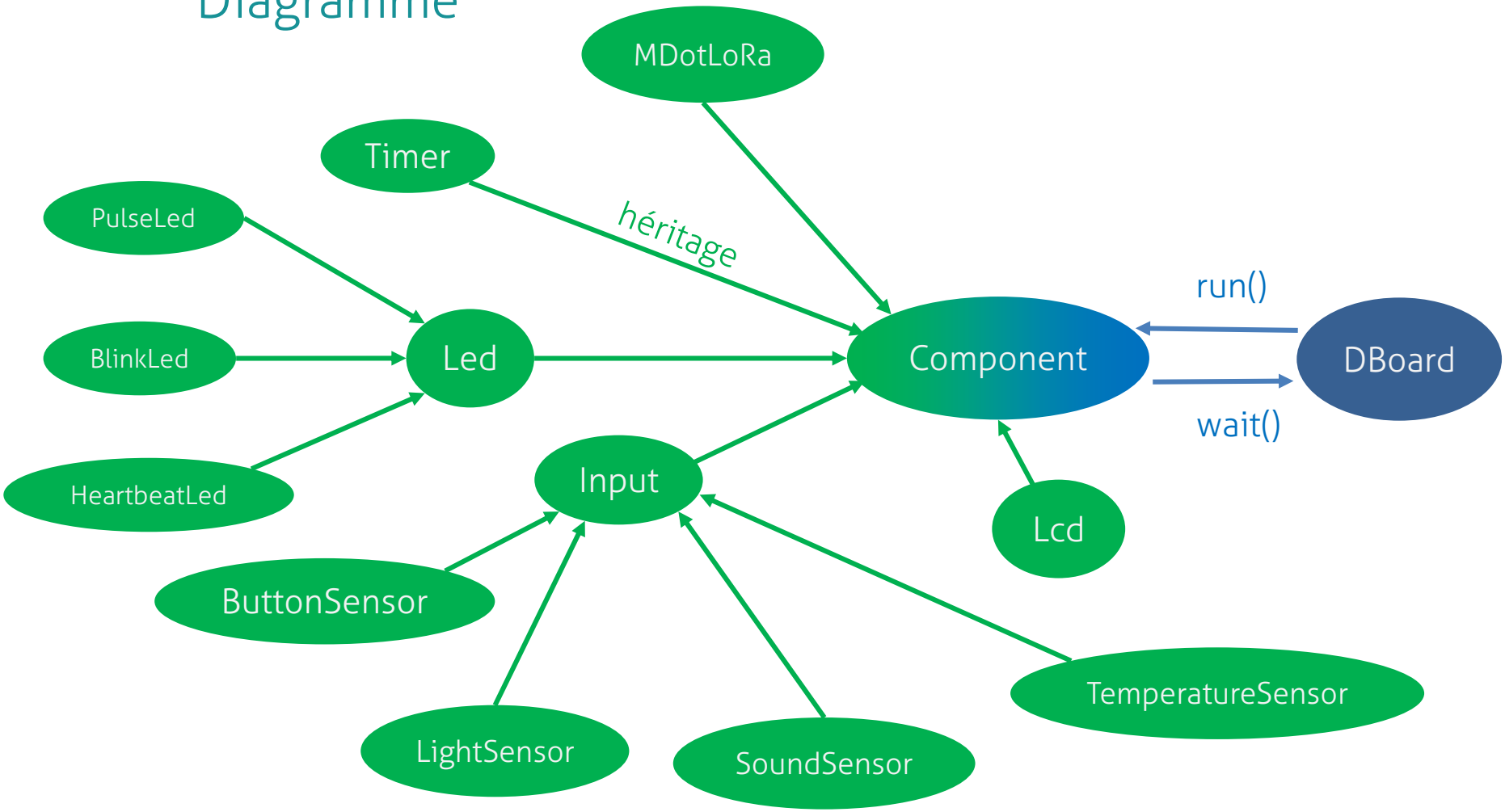
Cette classe initialise un objet unique masqué du programme principal et de l'utilisateur final (celui qui code le programme)
Elle encapsule toute la complexité de l'ordonnanceur

La classe Component

C'est une classe abstraite (C++) qui nécessite d'être héritée pour être instancié en composant réel
C'est la mère virtuelle de tous les composants
Elle propose une API simple (façade) à ceux-ci en masquant les appels systèmes réalisés vers l'instance de DBoard

d|board, un nouvel ordonnanceur

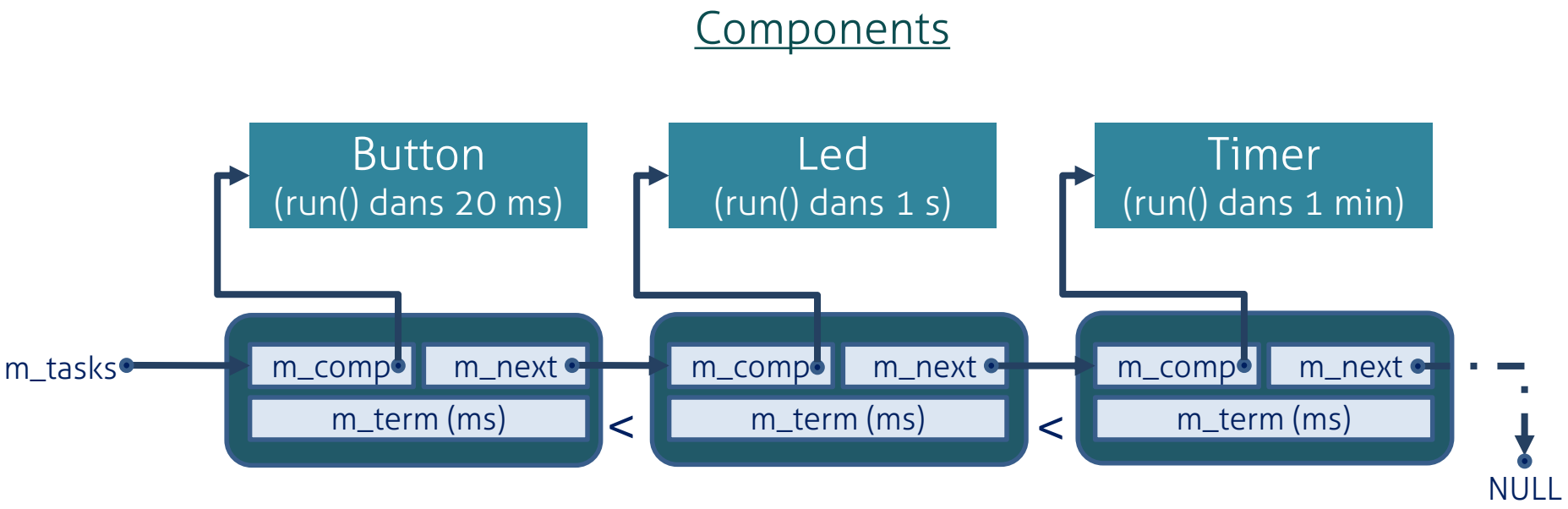
Diagramme



d|board, un nouvel ordonnanceur

Architecture de l'ordonnanceur

Enregistrement des tâches en cours dans une structure dynamique - une liste chaînée - triée en fonction de l'échéance (priorité) de chacune des tâches



d|board, exemple d'utilisation

Faire clignoter une LED – programme principal

Inclusion des fichiers d'entête (.h) des composants utilisés
Déclaration des composants (variables statiques)

Par sa déclaration, l'objet est instancié avec ses arguments
Le paramètre classique est le numéro de la broche (pin) par laquelle le composant physique est câblé sur l'Arduino

```
#include <ButtonSensor.h>
#include <BlinkLed.h>

ButtonSensor    button(4); // pin D4
BlinkLed        led(5);    // pin D5
```

d|board, exemple d'utilisation

Faire clignoter une LED – programme principal

- Ecriture des gestionnaires d'événements (ici le bouton)
- Le prototype du gestionnaire est toujours le même
- Il prend en argument un numéro d'événement transmis par le composant au programme principal

```
14  void onEventButton(eventType e) {
15      switch (e) {
16          case EVENT_BTN_SHORT_PRESS:
17              led.blink(50, 100);
18              break;
19          case EVENT_BTN_LONG_PRESS:
20              led.off();
21              break;
22      }
23  }
24
```

d|board, exemple d'utilisation

Faire clignoter une LED – programme principal

Dans la fonction **setup()** d'Arduino on branche le composant en y ajoutant son gestionnaire d'événement si besoin

La fonction **loop()** ne contient qu'une seule ligne correspondant à l'appel de l'ordonnanceur (un appel à une méthode de classe en C++)

```
25  void setup() {
26      button.plug(onEventButton);
27      led.plug();
28  }
29
30  void loop() {
31      Component::loop();
32  }
```

d|board, exemple d'utilisation

Faire clignoter une LED – le composant BlinkLed

- Hérite d'un composant plus générique, la Led
- Doit exposer les méthodes **setup()** et **run()**
- La méthode **setup()** est déclarée dans le composant Led
- La méthode **run()** est surchargée dans le composant BlinkLed
- Expose d'autres méthodes spécifiques à ce composant: **blink()**

```
#include <Arduino.h>
#include <Led.h>

class BlinkLed : public Led {
    public:
        BlinkLed(int pin);
        void run(serviceType);
        void blink(long on, long off);

    private:
        unsigned long m_onTime;     // milliseconds of on-time
        unsigned long m_offTime;    // milliseconds of off-time
};
```

d|board, exemple d'utilisation

Faire clignoter une LED – le composant BlinkLed

- Constructeur
- Déclaration de **run()**
- Déclaration de **blink()**

```
/**

 Blinking model

  __
 |  |
_|  |______
  1    2

  1 : onTime
  2 : offTime


**/
```

```
#include <BlinkLed.h>

BlinkLed::BlinkLed(int _pin) :
    Led(_pin) {
    // nothing more
} // constructor

void BlinkLed::run(serviceType _serv) {

    if(m_ledState == HIGH)
    {
        setOff();
        wait(m_offTime);
    }
    else if (m_ledState == LOW)
    {
        setOn();
        wait(m_onTime);
    }
} // run

void BlinkLed::blink(long _on, long _off) {
    m_onTime = _on;
    m_offTime = _off;
    leave();
    wait(0);
} // blink
```

d|board, exemple d'utilisation

Faire clignoter une LED – le composant ButtonSensor

- Hérite d'un composant plus générique, Input
- Doit exposer les méthodes **setup()** et **run()**
- La méthode **setup()** est déclarée dans le composant Input
- La méthode **run()** est écrite dans le composant ButtonSensor
- Déclare deux événements pouvant être lancés par l'objet

```
#include <Arduino.h>
#include <Input.h>

#define EVENT_BTN_SHORT_PRESS 0
#define EVENT_BTN_LONG_PRESS  1

class ButtonSensor : public Input {
    public:
        ButtonSensor(int pin);
        void run(serviceType);

    private:
        bool            m_state;            // current state
        unsigned long   m_pressedMillis;    // time of the touch
};
```

d|board, exemple d'utilisation

Faire clignoter une LED – le composant ButtonSensor

```
#include <ButtonSensor.h>

ButtonSensor::ButtonSensor(int _pin) :
    Input(_pin, true),
    m_state(0) {
    // nothing more
} // constructor

void ButtonSensor::run(serviceType _serv) {
    int prevState = m_state;
    m_state = read();
    if (prevState == LOW && m_state == HIGH) {
        m_pressedMillis = millis();
    }
    else if (prevState == HIGH && m_state == LOW) {
        if (millis() - m_pressedMillis < 50) {
            // ignore this for debounce
        } else if (millis() - m_pressedMillis < 300) {
            event(EVENT_BTN_SHORT_PRESS);
        } else {
            event(EVENT_BTN_LONG_PRESS);
        }
    }
    wait(10);
} // run
```

d|board, exemple d'utilisation

Autre exemple avec un capteur de CO_2

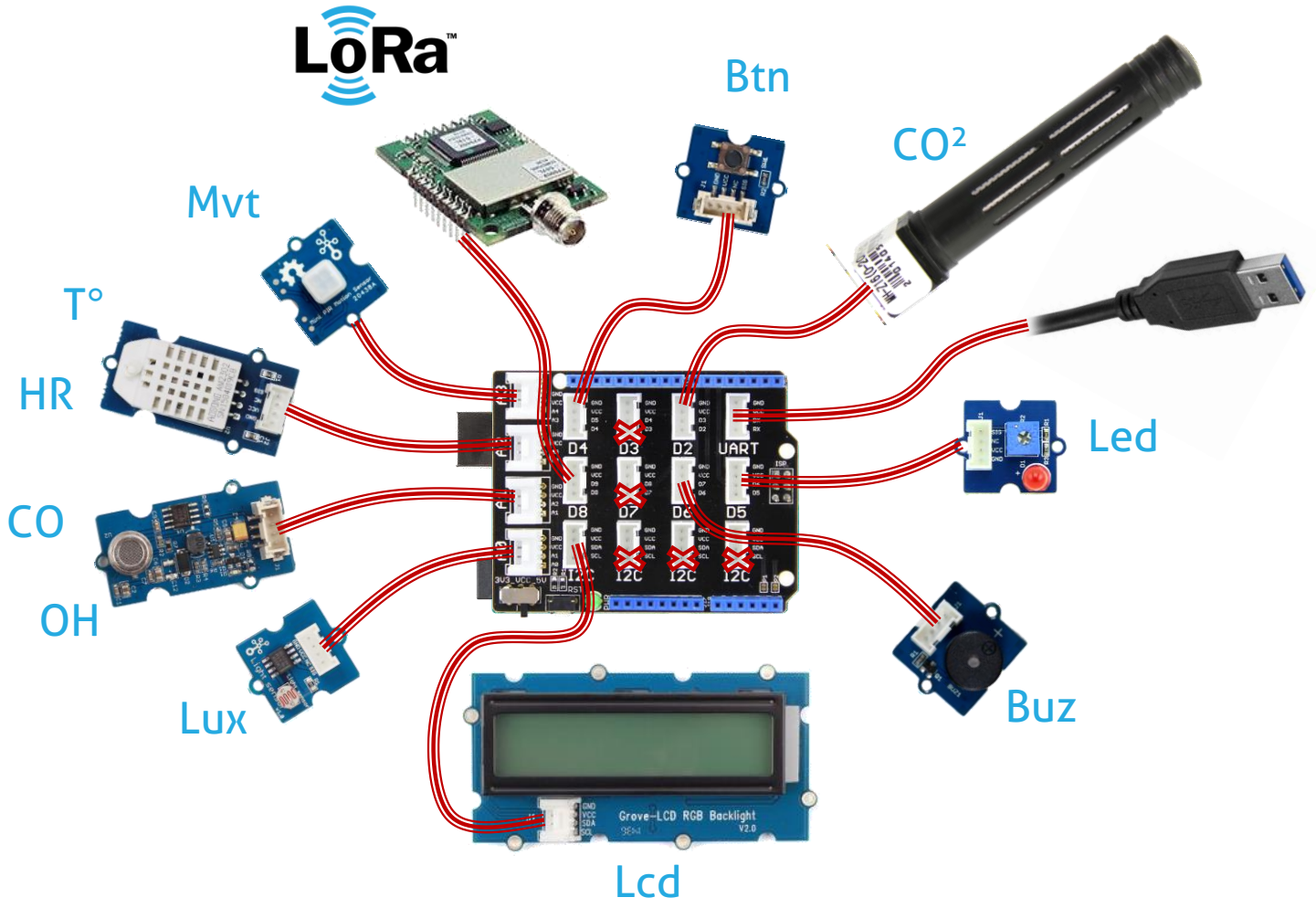
- Programme principal avec gestionnaire d'événements
- Fonctions **setup()** et **loop()** réduite au strict minimum

```
#include <CO2Sensor.h>

CO2Sensor co2(2, 3); // pin D2 & D3

void onEventCo2(eventType e) {
    switch (e) {
        case EVENT_WARMING:
            Serial.print("Warming up (");
            Serial.print(co2.temperature());
            Serial.println(" °C)");
            break;
        case EVENT_READY:
            Serial.println("Sensor is ready !");
            break;
        case EVENT_DATA:
            Serial.print("Co2 concentration : ");
            Serial.print(co2.concentration());
            Serial.println(" ppm");
            break;
    }
}
```

```
void setup() {
    co2.plug(onEventCo2);
}

void loop() {
    Component::loop();
}
```

d|board, pour la création d'objet complexe

Schéma de câblage de d|bonair, un objet LoRaWAN



Retrouvez d|board sur GitHub
https://github.com/albertdamien/DBoard

Merci