🧠 TP 3 — Module 5

Machines à États (FSM)

Switch-Case · State Pattern · Lambdas C++11

📝 Changelog — V0.1.0

- Deck 5 : Théorie FSM, 3 niveaux d'abstraction, scénario lampe intelligente, comparatif.

❌ Le problème du code séquentiel

```
// ❌ Code bloquant avec delay()
void loop() {
    digitalWrite(MOTOR_UP, HIGH);
    delay(5000);                    // Figé 5 secondes !
    digitalWrite(MOTOR_UP, LOW);
    delay(1000);
    // Impossible de détecter un obstacle pendant delay() !
}
```

Conséquence : si un obstacle surgit pendant `delay(5000)`, le moteur continue.
Objectif : un programme **réactif** qui peut répondre à n'importe quel événement **à tout moment**.

> *La solution : les **Machines à États Finis** (FSM).*

📐 Théorie des Automates Finis (FSM)

Trois composants d'un automate :

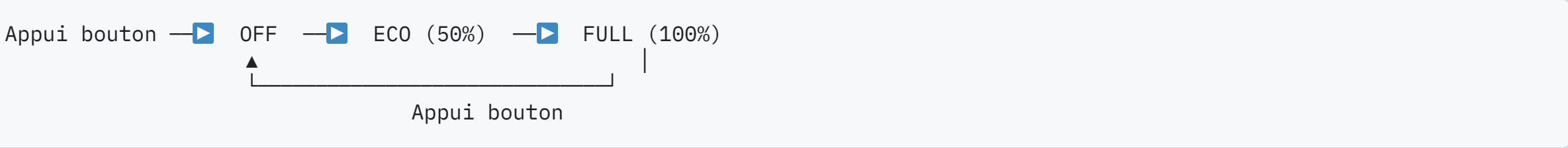
Composant	Définition	Exemple (lampe)
États	Condition actuelle du système	OFF, ECO, FULL
Événements	Déclencheurs de changement	bouton_cliqué
Transitions	Règle A → B sur événement E	OFF → ECO si clic

Propriété fondamentale : le système n'est dans **qu'un seul état à la fois**.

🔦 Scénario : la Lampe Intelligente

Un bouton, trois états, trois comportements distincts :

🔍 Décomposition de l'automate



Règles métier :

- En état `OFF` → LED éteinte, aucun courant
- En état `ECO` → PWM 128/255 (~50% luminosité, économie d'énergie)
- En état `FULL` → PWM 255/255 (luminosité maximale)
- La transition se fait toujours dans le même sens (cycle)

⭐ Niveau 1 : Switch-Case Classique

```
enum State { OFF, ECO, FULL };
State currentState = OFF;

Led lampe(26);
Button btn(12);

void loop() {
    if (btn.wasClicked()) {          // Un seul clic = un seul événement
        switch (currentState) {
            case OFF:
                currentState = ECO;
                lampe.setBrightness(128);
                break;
            case ECO:
                currentState = FULL;
                lampe.setBrightness(255);
                break;
            case FULL:
                currentState = OFF;
                lampe.off();
                break;
        }
    }
}
```

✅ Analyse Niveau 1 : Switch-Case

Forces :

- Très simple à lire et comprendre
- Performance maximale (proche du langage machine)
- Aucune allocation mémoire dynamique
- Idéal pour ≤ 4-5 états simples

Faiblesses :

- Devient un **"code spaghetti"** dès 8-10 états
- Logique métier mélangée avec la gestion matérielle
- Les `onEnter` / `onExit` doivent être gérés manuellement
- Difficile de tester unitairement

> ***Recommandation :*** *Pour les automates simples ou les ressources très contraintes (Uno).*

⭐⭐ Niveau 2 : State Pattern (POO)

```
class LampFSM {
  private:
    Led&    led;
    Button& btn;

    // Pointeur vers la méthode d'état courante
    void (LampFSM::*currentTick)();

    void stateOff()  { if (btn.wasClicked()) transitionTo(&LampFSM::stateEco);  }
    void stateEco()  {
        led.setBrightness(128);
        if (btn.wasClicked()) transitionTo(&LampFSM::stateFull);
    }
    void stateFull() {
        led.setBrightness(255);
        if (btn.wasClicked()) transitionTo(&LampFSM::stateOff);
    }

    void transitionTo(void (LampFSM::*next)()) { currentTick = next; }

  public:
    LampFSM(Led& l, Button& b) : led(l), btn(b), currentTick(&LampFSM::stateOff) {}
    void update() { (this->*currentTick)(); } // Appel de la méthode courante
};
```

✅ Analyse Niveau 2 : State Pattern

Forces :

- **Code très modulaire** : ajouter un état = ajouter une méthode
- Encapsulation parfaite de l'automate
- Bon compromis RAM / lisibilité
- Testable (on peut injecter des mocks de `Led` et `Button`)

Faiblesses :

- Syntaxe des pointeurs de méthodes membres complexe : `void (LampFSM::*ptr)()` → difficile au premier abord
- Les `onEnter` / `onExit` restent verbeux à coder
- Structure de classe plutôt rigide

> ***Recommandation :*** *Pour les projets professionnels structurés sur ESP32.*

★★★ Niveau 3 : Lambdas et `std::function`

```cpp
#include <functional>
#include <map>

enum State { OFF, ECO, FULL };

std::map<State, std::function<void()>> states;
State current = OFF;
Led lampe(26);
Button btn(12);

void setup() {
    states[OFF] = [&]() {
        lampe.off();
        if (btn.wasClicked()) { current = ECO; }
    };
    states[ECO] = [&]() {
        lampe.setBrightness(128);
        if (btn.wasClicked()) { current = FULL; }
    };
    states[FULL] = [&]() {
        lampe.setBrightness(255);
        if (btn.wasClicked()) { current = OFF; }
    };
}

void loop() {
    states[current]();  // Exécute la lambda de l'état actuel
}
```

🧩 Anatomie d'une Lambda C++11

```
//   Capture     Paramètres     Corps
//   ↓           ↓              ↓
    [&]         ()           { lampe.off(); if (btn.wasClicked()) current = ECO; }
```

Syntaxe	Signification
`[]`	Capture rien
`[&]`	Capture toutes les variables par **référence**
`[=]`	Capture toutes les variables par **valeur**
`[&btn, &current]`	Capture seulement `btn` et `current` par référence

✅ Analyse Niveau 3 : Lambdas

Forces :

- **Flexibilité maximale** : états définis à l'exécution
- Lisibilité "Fluent API" : les règles métier sont isolées
- Comportement modifiable dynamiquement (reconfiguration à chaud)
- Capturer des variables locales dans les lambdas

Faiblesses :

- Consommation RAM plus élevée (`std::map` + `std::function` ≈ 2-5 KB)
- Syntaxe `[&](){}` déroutante au premier abord
- Risque de fuite mémoire si les lambdas capturent des références invalides

> ***Recommandation :*** *ESP32, projets dynamiques.* ❌ *Déconseillé sur Arduino Uno.*

📊 Comparatif des 3 approches

📋 Tableau de synthèse

Critère	Switch-Case	State Pattern	Lambdas
Complexité	⭐ Faible	⭐⭐ Moyenne	⭐⭐⭐ Haute
Maintenance	Difficile	Facile	Très facile
RAM consommée	Négligeable	Faible	Significative
Testabilité	Faible	Bonne	Bonne
Usage idéal	≤ 4 états	Projets structurés	Systèmes dynamiques
Compatibilité	Tous MCU	ESP32, STM32	ESP32 uniquement

🏋️ Exercice FSM : Lampe Intelligente

Implémentez les 3 approches pour la lampe à 3 états.

Tâches :

1. **Niveau 1** : implémenter avec `enum` + `switch/case` → anti-rebond 300ms
2. **Niveau 2** : encapsuler dans `LampFSM` avec pointeur de méthode
3. **Niveau 3** : réécrire avec `std::map<State, std::function<void()>>`

Bonus :

- Ajouter un 4e état `BLINK` (clignotement) entre `FULL` et `OFF`
- Afficher l'état courant sur le moniteur série à chaque transition

🎯 Synthèse du TP 3

Concept	Retenir
`enum State`	Nommer les états avec des constantes lisibles
`switch/case`	Approche simple, efficace, limitée en scalabilité
Pointeur méthode	`void (Class::*ptr)()` — approche modulaire
Lambda `[&](){}`	Fonction anonyme capturant l'environnement
`std::function`	Type pour stocker n'importe quelle fonction/lambda
`std::map`	Table de correspondance état → lambda

Prochain module → Capteurs, Signaux et Réseau

👨‍💻 Questions ?

TP 3 — Machines à États (FSM)
Réda BOUREBABA & Sébastien Antonico