Návrhové vzory

Design Patterns

Stanislav Vítek · Katedra radioelektroniky · ČVUT FEL

Proč návrhové vzory?

OOP nabízí velkou paletu nástrojů — dědičnost, polymorfismus, šablony, přetěžování… Ale jak je správně kombinovat v rozsáhlejším systému?

Bez sdíleného slovníku řeší každý programátor stejný problém jinak. Výsledek je kód, který sice funguje, ale je těžko čitelný a ještě hůř rozšiřitelný.

> ***Návrhový vzor*** *= pojmenované, osvědčené řešení typického problému v konkrétním kontextu*

- Vzory nejsou kusy kódu, které zkopírujeme — jsou to **principy** a **struktury**
- Poskytují **společný jazyk**: místo zdlouhavého popisu stačí říct „použij Singleton" nebo „to je Facade nad HAL"
- Předpokládají, že systém bude **žít**: bude se udržovat, rozšiřovat, předávat dalším

Původ: Christopher Alexander — architektura budov (1977). Formalizace pro SW: Gang of Four, *Design Patterns* (1994).

Tři kategorie vzorů

Tvořivé (Creational)

- Factory Method
- Factory
- Abstract Factory
- Singleton
- Builder
- Prototype

Abstrakce procesu vytváření objektů

Strukturální (Structural)

- Adapter
- Bridge
- Facade
- Proxy
- Decorator
- Composite
- Flyweight

Skládání tříd a objektů

Chování (Behavioral)

- Observer ★
- State / FSM ★
- Strategy ★
- Command
- Iterator
- Template Method
- … (GoF celkem: 23)

Komunikace objektů

★ = klíčové pro embedded

Tvořivé vzory

Creational Patterns

Co, kdo, jak a kdy se vytváří

Factory — motivace

Mějme jednoduchou třídu `Auto` s veřejným konstruktorem. Samozřejmě můžeme kdykoliv vytvořit libovolnou instanci:

```
Auto fabia   = Auto("Škoda", "Fabia");
Auto octavia = Auto("Škoda", "Octavia");
```

Jenže v naší aplikaci pracujeme velmi často se Škodou Felicií — a ta má automaticky nastavených třeba 30 atributů (typ motoru, počet sedadel, výbava…). Psát `Auto("Škoda", "Felicia", 1.3, "GLX", 5, ...)` na každém místě je zdlouhavé a náchylné k chybám.

Potřebujeme způsob, jak tuto složitou inicializaci **zapouzdřit a pojmenovat**.

Řešení: tovární vzory — inicializaci přesuneme do dedikované metody nebo třídy. Vzniknou nám dvě varianty:

- **Factory Method** — virtuální metoda v hierarchii tříd
- **Factory** — (statická) metoda přímo ve třídě produktu

Factory Method

Návrhový vzor Factory Method využívá virtuální metodu, která volá konstruktor. Klíčové je, že **každá podtřída může metodu překrýt** a vrátit jiný typ objektu — aniž by se změnil kód, který továrnu volá.

```
// Abstraktní produkt — společné rozhraní všech aut
class Auto {
public:
    virtual ~Auto() = default;
    virtual void zobraz() const = 0;
};
```

Definujeme abstraktní rozhraní produktu. Konkrétní třídy ho implementují pro každý model zvlášť.

```
// Konkrétní produkt
class Felicia : public Auto {
    std::string motor = "1.3 MPI";
    int         sedadla = 5;
public:
    void zobraz() const override {
        std::cout << "Felicia, " << motor
                  << ", " << sedadla << " sedadel\n";
    }
};
```

Factory Method — továrna

Nyní vytvoříme továrnu. Oddělíme tím konstrukční kód od zbytku aplikace — třída `Auto` zůstane čistá.

```
// Abstraktní továrna — definuje rozhraní
class AutoTovarna {
public:
    virtual ~AutoTovarna() = default;

    // Tovární metoda — podtřídy ji překryjí
    virtual std::unique_ptr<Auto> vytvorAuto() = 0;

    // Metody továrny mohou tovární metodu libovolně využívat
    void predstavAuto() {
        auto a = vytvorAuto();   // 'auto' = odvození typu kompilátorem
        a->zobraz();
    }
};

// Konkrétní továrna pro Felicii
class FeliciaTovarna : public AutoTovarna {
public:
    std::unique_ptr<Auto> vytvorAuto() override {
        return std::make_unique<Felicia>();
    }
};
```

Nyní stačí zavolat jedinou metodu továrny. Kdybychom chtěli místo Felicie vyrábět Fabii, změníme jen továrnu — zbytek kódu zůstane beze změny.

```
FeliciaTovarna tovarna;
tovarna.predstavAuto();    // Felicia, 1.3 MPI, 5 sedadel
```

Factory — statická varianta

Druhá varianta umístí tovární metodu přímo do třídy produktu jako **statickou metodu**. Konstruktor je privátní, takže objekt nelze vytvořit jinak než přes továrnu.

```
class Auto {
    std::string znacka, model;

    // Privátní konstruktor — žádný přímý new zvenku
    Auto(std::string z, std::string m)
        : znacka(std::move(z)), model(std::move(m)) {}

public:
    // Pojmenovaná statická tovární metoda
    static Auto Felicia() {
        return Auto("Škoda", "Felicia");
    }

    // Varianta vracející pointer na haldu
    static std::unique_ptr<Auto> FeliciaPtr() {
        return std::make_unique<Auto>("Škoda", "Felicia");
    }

    void zobraz() const {
        std::cout << znacka << " " << model << "\n";
    }
};
```

Pojmenovaná tovární metoda je výhodná zejména tehdy, kdy jeden typ objektu má více různých způsobů inicializace — konstruktory by se nelišily názvem, ale tovární metody ano.

```
auto f  = Auto::Felicia();      // vytvoření hodnotou na zásobníku
auto fp = Auto::FeliciaPtr();   // vytvoření na haldě přes unique_ptr
// Auto x("Škoda","Octavia");   // ✗ chyba překladu — konstruktor privátní
```

Abstract Factory — záměr

Situace se zkomplikuje, když potřebujeme vytvářet celou **rodinu vzájemně závislých objektů**. Klasický příklad: GUI prvky pro Windows a macOS musí vypadat konzistentně — Windows tlačítko s Windows checkboxem, macOS tlačítko s macOS checkboxem. Nikdy nesmíme namíchat obě rodiny dohromady.

Abstract Factory definuje rozhraní pro tvorbu celé takové rodiny, aniž by klientský kód věděl, pro jakou platformu pracuje.

Abstract Factory — produkty

- Nejprve definujeme abstraktní rozhraní pro oba typy produktů — tlačítko a checkbox.
- Každá platforma pak tato rozhraní implementuje po svém.

```cpp
// Abstraktní produkty — rozhraní pro každou rodinu
class Button {
public:
    virtual ~Button() = default;
    virtual void render() = 0;
};

class Checkbox {
public:
    virtual ~Checkbox() = default;
    virtual void render() = 0;
};
```

Abstract Factory — továrna

- Abstraktní továrna sdružuje metody pro tvorbu celé rodiny.
- Všimněte si, že továrna vrací **abstraktní typy** — klient nikdy nevidí konkrétní třídy produktů.

```
class GUIFactory {
public:
    virtual ~GUIFactory() = default;
    virtual std::unique_ptr<Button>   createButton()   = 0;
    virtual std::unique_ptr<Checkbox> createCheckbox() = 0;
};
```

Abstract Factory — konkrétní implementace

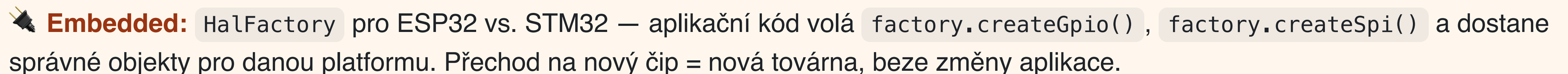
- Konkrétní produkty a továrna pro Windows jsou od sebe odděleny.
- Přidání nové platformy (Linux, Android) znamená pouze přidat novou skupinu tříd — stávajícího kódu se nedotýkáme.

```cpp
// Konkrétní produkty pro Windows
struct WinButton   : Button   { void render() override { puts("WinButton");   } };
struct WinCheckbox : Checkbox { void render() override { puts("WinCheckbox"); } };

// Konkrétní továrna pro Windows — vytváří výhradně Win produkty
class WindowsFactory : public GUIFactory {
public:
    std::unique_ptr<Button>   createButton()   override {
        return std::make_unique<WinButton>();
    }
    std::unique_ptr<Checkbox> createCheckbox() override {
        return std::make_unique<WinCheckbox>();
    }
};
```

Abstract Factory — klientský kód

- Klientský kód dostane továrnu jako parametr a pracuje výhradně přes abstraktní rozhraní.
- Celé UI pak lze přepnout na jinou platformu záměnou jediného argumentu.

```cpp
void renderUI(GUIFactory& factory) {
    auto btn = factory.createButton();    // Button*, ne WinButton*
    auto chk = factory.createCheckbox();
    btn->render();
    chk->render();
}
int main() {
    WindowsFactory wf;
    renderUI(wf);    // snadno zaměnit za MacOSFactory
}
```



🔌 **Embedded:** `HalFactory` pro ESP32 vs. STM32 — aplikační kód volá `factory.createGpio()`, `factory.createSpi()` a dostane správné objekty pro danou platformu. Přechod na nový čip = nová továrna, beze změny aplikace.

Singleton — záměr

Singleton zajistí, aby třída měla **právě jednu instanci** v celé aplikaci, a poskytne k ní globální přístup.

Základní postup:

1. Privátní konstruktor — nikdo nesmí vytvořit instanci přímo
2. Zakázat kopírování a přiřazování — aby nevznikly duplikáty
3. Statická metoda `instance()`, která při prvním volání objekt vytvoří, při dalších vrátí ten samý

```
// Krok 1–3: základní kostra Singletonu
class Logger {
    Logger() { Serial.begin(115200); }   // 1. privátní konstruktor

    Logger(const Logger&)            = delete;   // 2. zákaz kopírování
    Logger& operator=(const Logger&) = delete;   //    zákaz přiřazování
```

```cpp
public:
    // 3. Přístupová metoda — Meyer's Singleton (thread-safe od C++11)
    static Logger& instance() {
        static Logger inst;   // vznikne jednou, zanikne se static storage
        return inst;
    }

    void log(const std::string& zprava) {
        Serial.println(zprava.c_str());
    }
};
```

Statická lokální proměnná `inst` je v C++11 garantovaně inicializována **právě jednou**, i při souběžném přístupu z více vláken. To je důvod, proč preferujeme tuto variantu oproti staršímu přístupu s explicitním pointerem.

Singleton — použití a kdy ho nepoužívat

Použití je pak přímočaré — kdekoliv v kódu, vždy tatáž instance:

```
Logger::instance().log("Systém spuštěn");
Logger::instance().log("Senzor inicializován");

// Nelze napsat:
// Logger l;                         // ✗ privátní konstruktor
// Logger l2 = Logger::instance();  // ✗ zákaz kopírování
```

⚠️ **Singleton je globální stav v OOP hávu.** Skrývá závislosti, ztěžuje testování a může způsobit problémy ve vícevláknových aplikacích (Qt, FreeRTOS). Používejte ho *pouze* pro skutečně unikátní, globálně sdílené zdroje — HW periferie, logger. Pokud si nejste jisti, zda Singleton potřebujete, pravděpodobně ho nepotřebujete.

🔌 **Embedded — sdílená SPI sběrnice:** Na mikrokontroléru existuje jeden fyzický SPI bus. Singleton přirozeně modeluje tento HW fakt — různé části firmware (display driver, SD karta, RF modul) sdílí jednu instanci bez rizika konfliktu.

```
class SPISbernice {
    SPISbernice() { SPI.begin(); }
    SPISbernice(const SPISbernice&) = delete;
    SPISbernice& operator=(const SPISbernice&) = delete;
public:
    static SPISbernice& instance() { static SPISbernice s; return s; }
    void posli(uint8_t* data, size_t n) { SPI.transfer(data, (int)n); }
};
// Displej i SD karta sdílí jednu instanci — žádný konflikt:
SPISbernice::instance().posli(framebuf, sizeof(framebuf));
```

Builder — záměr

Představte si, že chceme vytvořit objekt s mnoha volitelnými parametry. Například WiFi připojení může mít SSID, heslo, MQTT broker, port, TLS příznak, QoS úroveň, a ještě dalších deset položek. Jak to řešit?

Problém s konstruktorem: přidáváme parametry, až konstruktor vypadá takto:

```
// tzv. "telescoping constructor" — anti-pattern
WiFiConn c("ssid", "pass", "broker.io", 8883, true, 2, 30, false, ...);
//                                               ↑ Co to je? true??
```

Na volací straně není zřejmé, co který argument znamená.

Builder tento problém řeší oddělením sestavení objektu do série pojmenovaných kroků. Výsledkem je tzv. **fluent interface** — řetězení volání:

```
// Čitelné, samodokumentující, bezpečné — vynechání položek nevadí
WiFiConfig cfg = WiFiConfigBuilder()
    .setSsid("FEL_Lab_IoT")
    .setHeslo("tajne123")
    .setMqtt("broker.hivemq.com", 8883)
    .enableTls()
    .build();
```

Builder — implementace

```
class WiFiConfig {
public:
    std::string ssid, heslo, mqttBroker;
    uint16_t    mqttPort  = 1883;
    bool        tlsEnable = false;
    uint8_t     qos       = 1;
};
```

Builder drží rozestavěný objekt a každá metoda vrátí `*this` — díky tomu lze volání řetězit.

```
class WiFiConfigBuilder {
    WiFiConfig cfg;   // rozestavěný objekt
public:
    // Každá metoda nastaví jednu vlastnost a vrátí builder zpět
    WiFiConfigBuilder& setSsid(std::string s)  { cfg.ssid = std::move(s); return *this; }
    WiFiConfigBuilder& setHeslo(std::string h) { cfg.heslo = std::move(h); return *this; }
    WiFiConfigBuilder& setMqtt(std::string b, uint16_t p = 1883) {
        cfg.mqttBroker = std::move(b);
        cfg.mqttPort   = p;
        return *this;
    }
```

```
    WiFiConfigBuilder& enableTls()             { cfg.tlsEnable = true; return *this; }
    WiFiConfigBuilder& setQos(uint8_t q)       { cfg.qos = q; return *this; }

    // Finalizace — vrátí hotový objekt
    WiFiConfig build() { return cfg; }
};
```

Builder lze dále rozšířit o **validaci** v metodě `build()` — například kontrolu, zda je SSID neprázdné, nebo zda při `enableTls()` není port 1883 (nešifrovaný).

Prototype — klonování objektů

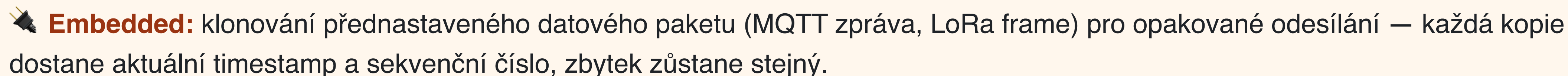
Prototype řeší situaci, kdy potřebujeme vytvořit kopii **již inicializovaného objektu**, aniž bychom záviseli na jeho konkrétní třídě.

Místo `new KonkretniTrida(...)` požádáme existující objekt, aby se sám naklonoval.

```
// Rozhraní prototypu — každý objekt umí sám sebe zkopírovat
class Tvar {
public:
    virtual ~Tvar() = default;
    virtual void vykresli() const = 0;
    virtual std::unique_ptr<Tvar> klonuj() const = 0;  // klíčová metoda
};

class Kruh : public Tvar {
    int polomer;
public:
    explicit Kruh(int r) : polomer(r) {}
    void vykresli() const override {
        std::cout << "Kruh r=" << polomer << "\n";
    }
    // Klonování přes copy constructor — zkopíruje všechny členy
    std::unique_ptr<Tvar> klonuj() const override {
        return std::make_unique<Kruh>(*this);
    }
};
```

Klientský kód pak nepotřebuje vědět, s jakým typem tvaru pracuje:

```cpp
std::unique_ptr<Tvar> original = std::make_unique<Kruh>(10);
auto kopie1 = original->klonuj();   // nová instance, stejné parametry
auto kopie2 = original->klonuj();   // a ještě jedna
kopie1->vykresli();   // Kruh r=10
```



🔌 **Embedded:** klonování přednastaveného datového paketu (MQTT zpráva, LoRa frame) pro opakované odesílání — každá kopie dostane aktuální timestamp a sekvenční číslo, zbytek zůstane stejný.

Strukturální vzory

Structural Patterns

Jak skládat třídy a objekty do větších a přehledných celků

Adapter — záměr

Adapter (adaptér) řeší jeden z nejčastějších praktických problémů: máme existující třídu nebo knihovnu, která dělá přesně to, co potřebujeme — ale poskytuje jiné rozhraní, než jaké náš kód očekává.

Typické situace:

- Integrace **C knihovny výrobce čipu** do C++ projektu
- **Migrace** z jedné senzorové knihovny na jinou
- Použití **legacy kódu** v novém systému

Adapter funguje jako redukce v elektrotechnice — na jedné straně zástrčka pro starý kód, na druhé zásuvka pro nový systém.

```
Klient → [ Nové rozhraní ] ←→ [ Adapter ] ←→ [ Staré/cizí rozhraní ]
```

Klíčová vlastnost: **klient vůbec neví, že za adapterem je jiné rozhraní**. Pracuje s ním stejně jako s jakýmkoliv jiným objektem splňujícím dané rozhraní.

> Adapter nemění chování adaptované třídy — pouze překládá volání mezi rozhraními. Nepřidává logiku, jen adaptuje.

Adapter — kód

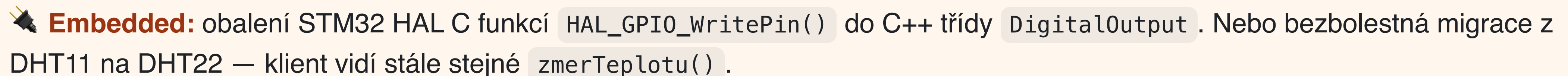
Uvažujme situaci: náš systém očekává senzor s metodou `zmerTeplotu()`. Ale ovladač konkrétního čipu je napsaný v C a vrací teplotu jako celé číslo v setinách stupně.

```cpp
// Nové rozhraní — co klientský kód očekává
class Senzor {
public:
    virtual ~Senzor() = default;
    virtual float zmerTeplotu() = 0;
};
```

```cpp
// Stará C-style funkce výrobce čipu — nelze měnit
// Vrací hodnotu jako int v jednotkách 1/100 °C
extern "C" int chip_read_temp(int channel);
```

Adapter tyto dvě strany propojí. Dědí z nového rozhraní, interně zavolá starou funkci a provede konverzi jednotek.

```cpp
class LegacyChipAdapter : public Senzor {
    int kanal;
public:
    explicit LegacyChipAdapter(int ch) : kanal(ch) {}

    float zmerTeplotu() override {
        // Konverze: celé číslo v 1/100 °C → float v °C
        return chip_read_temp(kanal) / 100.0f;
    }
};

// Klientský kód pracuje výhradně s rozhraním Senzor:
LegacyChipAdapter sensor(0);
float t = sensor.zmerTeplotu();    // 23.45 °C
```



🔌 **Embedded:** obalení STM32 HAL C funkcí `HAL_GPIO_WritePin()` do C++ třídy `DigitalOutput`. Nebo bezbolestná migrace z DHT11 na DHT22 — klient vidí stále stejné `zmerTeplotu()`.

Bridge — záměr

Bridge (most) řeší problém, který nastane, když potřebujeme rozšiřovat systém **ve dvou nezávislých dimenzích**. Klasický příklad: různé typy tvarů (kruh, čtverec) × různé způsoby kreslení (obrazovka, tiskárna).

Bez Bridge bychom museli mít `KruhNaObrazovce`, `KruhNaTiskarne`, `CtverecNaObrazovce`, `CtverecNaTiskarne`… Při 3 tvarech a 3 médiích = **9 tříd**. Přidáme tiskárnu PDF? Další 3 třídy. Přidáme trojúhelník? Další 3 třídy. Kód exploduje.

Bridge odděluje abstrakci od implementace tak, aby se obě mohly vyvíjet nezávisle. Místo dědičnosti použije **kompozici** — abstrakce (tvar) drží referenci na implementaci (způsob kreslení).

```
// Implementace — rozhraní pro přenosové médium
class Komunikace {
public:
    virtual ~Komunikace() = default;
    virtual bool odeslat(const uint8_t* data, size_t n) = 0;
};

// Konkrétní implementace — tři různá média
class WiFiKom : public Komunikace { /* ... */ };
class LoRaKom : public Komunikace { /* ... */ };
class BLEKom  : public Komunikace { /* ... */ };
```

Bridge — abstrakce a klientský kód

Abstrakce (IoT zařízení) drží odkaz na implementaci (komunikační kanál). Obě strany jsou rozšiřitelné nezávisle.

```
// Abstrakce — IoT zařízení nezávisí na konkrétním médiu
class IoTZarizeni {
protected:
    std::shared_ptr<Komunikace> kanal;  // odkaz na implementaci
public:
    explicit IoTZarizeni(std::shared_ptr<Komunikace> k) : kanal(k) {}
    virtual ~IoTZarizeni() = default;
    virtual void odesliData() = 0;
};

// Konkrétní abstrakce — meteorologická stanice
class MeteoStanice : public IoTZarizeni {
    using IoTZarizeni::IoTZarizeni;
public:
    void odesliData() override {
        uint8_t paket[] = { readTemp(), readHum() };
        kanal->odeslat(paket, sizeof(paket));  // WiFi, LoRa nebo BLE
    }
};
```

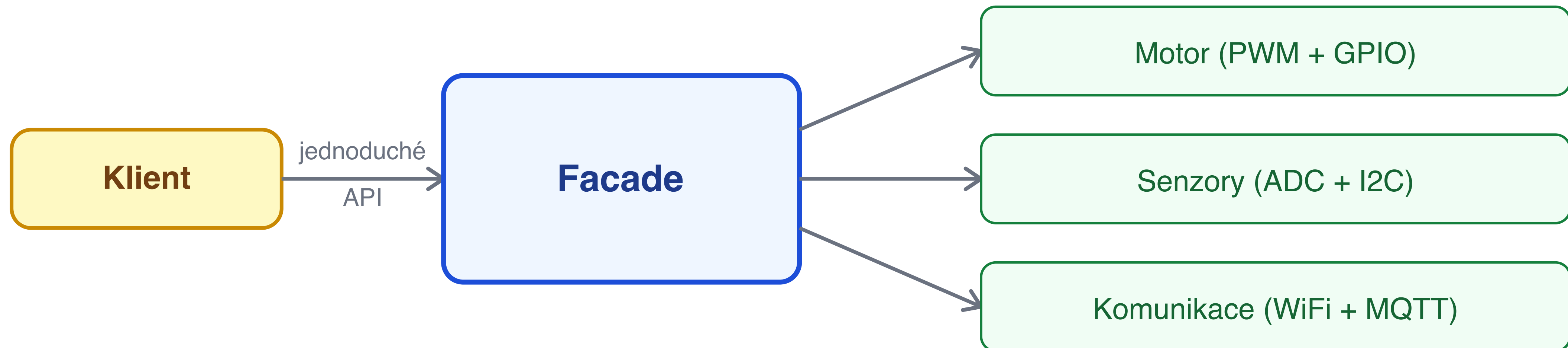
Komunikační médium se vkládá při vytvoření a lze ho kdykoli zaměnit — bez úpravy logiky stanice.

```
// 3 typy stanice × 3 média = stačí 6 tříd (ne 9!):
MeteoStanice ms1(std::make_shared<WiFiKom>("api.example.com"));
MeteoStanice ms2(std::make_shared<LoRaKom>(868e6));
// Přidáme NB-IoT médium → jen 1 nová třída, ne 3
```

> Bridge je cenný zejména tehdy, kdy stejná aplikační logika musí běžet s různými komunikačními moduly — WiFi při vývoji a ladění, LoRa v poli s nízkým pokrytím.

Facade — záměr

Facade (fasáda) poskytuje **zjednodušené rozhraní ke složitému subsystému**. Klient nemusí vědět, z kolika tříd se subsystém skládá, ani v jakém pořadí je volat.



Klient neví o interní složitosti — volá jen startAuto() nebo zastavAuto()

Facade — příklad

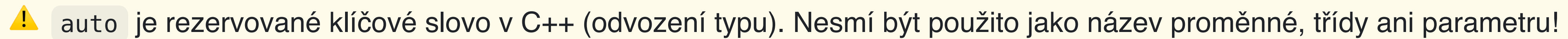
Mějme automobil, jehož start a zastavení zahrnuje koordinaci tří subsystémů. Bez Facade by klient musel znát správné pořadí volání každého z nich.

```
// Subsystémy — každý se stará o svou část
struct Motor {
    void start();
    void stop();
};

struct SystemRizeni {
    void nastavRychlost();
    void zastav();
};

struct SystemPaliva {
    void doplnPalivo();
    void uzavriNadrz();
};
```

Facade sjednotí přístup pod dvě jednoduché metody:

```cpp
class AutoFacade {
    Motor           motor;
    SystemRizeni    rizeni;
    SystemPaliva    palivo;
public:
    void startAuto() {
        motor.start();                  // 1. nastartuj motor
        rizeni.nastavRychlost();        // 2. nastav řízení
        palivo.doplnPalivo();           // 3. doplň palivo
    }
    void zastavAuto() {
        rizeni.zastav();                // 1. zastav kola
        motor.stop();                   // 2. vypni motor
        palivo.uzavriNadrz();           // 3. uzavři nádrž
    }
};
```

```cpp
AutoFacade facade;          // ✓ správný název proměnné
facade.startAuto();
```



Facade — embedded: HAL jako Facade

HAL (Hardware Abstraction Layer) je Facade vzor v praxi. Motor driver skrývá složitost PWM, GPIO a ADC za tří jednoduché metody.

```
#include <Arduino.h>

class MotorDriver {    // Facade nad PWM (ledcWrite) + GPIO + ADC
    uint8_t pinPwm, pinDir, pinFb;

public:
    MotorDriver(uint8_t pwm, uint8_t dir, uint8_t fb)
        : pinPwm(pwm), pinDir(dir), pinFb(fb) {
        // Interní inicializace třech subsystémů — klient o tom neví
        pinMode(pinDir, OUTPUT);
        ledcSetup(0, 20'000, 8);    // PWM kanál 0, 20 kHz, 8bitové rozlišení
        ledcAttachPin(pinPwm, 0);
    }

    // Jednoduché API pro klienta:
    void pohybuj(int rychlost) {     // rychlost: -255 .. +255
        digitalWrite(pinDir, rychlost >= 0 ? HIGH : LOW);
        ledcWrite(0, static_cast<uint8_t>(abs(rychlost)));
    }
    void zastav()       { ledcWrite(0, 0); }
    float zmerProud()   { return analogRead(pinFb) * 3.3f / 4095.0f / 0.1f; }
};
```

Klient neví nic o `ledcSetup`, `ledcWrite`, `analogRead` ani o tom, na jakém pinu je zpětná vazba proudu:

```
MotorDriver motor(18, 19, 34);
motor.pohybuj(180);
if (motor.zmerProud() > 2.5f) motor.zastav();   // ochrana proti přetížení
```

Proxy — záměr

Proxy (zástupce) slouží jako prostředník mezi klientem a skutečným objektem. Klient si myslí, že pracuje přímo s objektem — ale ve skutečnosti volá proxy, která přidává vlastní logiku.

Typické využití proxy:

- **Lazy loading** — skutečný objekt (např. velký soubor) se vytvoří až při prvním skutečném použití
- **Caching** — výsledky drahých operací se uloží a znovu vrátí bez nového výpočtu
- **Přístupová kontrola** — proxy ověří oprávnění před delegováním na skutečný objekt

```cpp
// Společné rozhraní — klient pracuje vždy s tímto typem
class Dokument {
public:
    virtual ~Dokument() = default;
    virtual void zobraz() = 0;
};
```

```cpp
// Skutečný objekt — jeho vytvoření je drahé (čte soubor z disku)
class SkutecnyDokument : public Dokument {
    std::string obsah;
public:
    explicit SkutecnyDokument(const std::string& soubor) {
        std::cout << "Načítám z disku: " << soubor << "\n";
        obsah = "Obsah: " + soubor;
    }
    void zobraz() override { std::cout << obsah << "\n"; }
};
```

Proxy — lazy loading

Proxy implementuje stejné rozhraní jako skutečný objekt. Vytvoří ho teprve při prvním volání `zobraz()`.

```
class LazyProxy : public Dokument {
    std::shared_ptr<SkutecnyDokument> doc;   // zatím nullptr
    std::string soubor;
public:
    explicit LazyProxy(std::string s) : soubor(std::move(s)) {}

    void zobraz() override {
        // Lazy loading — skutečný objekt vznikne až tady, poprvé
        if (!doc)
            doc = std::make_shared<SkutecnyDokument>(soubor);
        doc->zobraz();
    }
};
```

Při vytvoření proxy se na disk nesahá. Načtení proběhne pouze tehdy, když klient dokument skutečně potřebuje zobrazit.

```
LazyProxy proxy("manual.pdf");    // disk se nečte

std::cout << "Až teď:\n";
proxy.zobraz();   // teprve zde se soubor načte
proxy.zobraz();   // druhé volání — načtení již neprobíhá
```

🔌 **Embedded — I2C senzor s cache:** Proxy čte fyzický senzor jen jednou za 500 ms; mezi tím vrací uloženou hodnotu. Šetří I2C sběrnici i energii — senzory jako BME280 nebo SHT31 mají omezenou frekvenci měření.

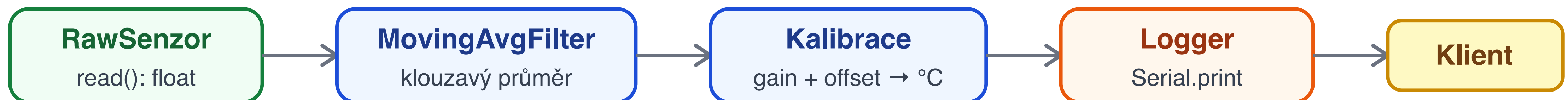
```
float zmerTeplotu() override {
    if (!nacteno || (millis() - cacheTime) > 500) {
        cachedVal = i2c.read(BME280_ADDR); cacheTime = millis(); nacteno = true;
    }
    return cachedVal;
}
```

Decorator — záměr

Decorator (dekorátor) přidává objektu nové chování **dynamicky za běhu** — bez dědičnosti a bez zásahu do původní třídy.

Situace: máme senzor vracející surovou hodnotu ADC. Chceme postupně přidávat zpracování — filtr šumu, kalibraci na fyzikální jednotky, logování. Každý krok je volitelný a záměnný.

Dědičností bychom skončili u kombinatorické exploze: `FiltrovanyKalibrovanyLogovanyNtcSenzor`...



read() prochází celým řetězcem — každý dekorátor obohatí nebo transformuje výsledek

Decorator — základní třídy

Rozhraní `Senzor` je sdílené pro původní objekt i všechny dekorátory. Díky tomu jsou zcela zaměnitelné.

```
// Společné rozhraní
class Senzor {
public:
    virtual ~Senzor() = default;
    virtual float read() = 0;
};

// Základní senzor — vrací surové napětí z ADC pinu
class AnalogSenzor : public Senzor {
    uint8_t pin;
public:
    explicit AnalogSenzor(uint8_t p) : pin(p) {}
    float read() override {
        return analogRead(pin) * 3.3f / 4095.0f;
    }
};
```

Abstraktní dekorátor drží odkaz na obalovaný senzor. Podtřídy ho mohou před nebo po volání `src->read()` libovolně rozšířit.

```
// Abstraktní dekorátor — základ pro všechny konkrétní dekorátory
class SenzorDecorator : public Senzor {
protected:
    std::shared_ptr<Senzor> src;    // obalovaný senzor
public:
    explicit SenzorDecorator(std::shared_ptr<Senzor> s) : src(s) {}
};
```

Decorator — konkrétní dekorátory

```cpp
// Klouzavý průměr přes posuvné okno 8 vzorků
class MovingAvgFilter : public SenzorDecorator {
    float   buf[8] = {};
    uint8_t idx    = 0;
public:
    using SenzorDecorator::SenzorDecorator;
    float read() override {
        buf[idx++ % 8] = src->read();   // ulož nový vzorek
        return std::accumulate(std::begin(buf), std::end(buf), 0.0f) / 8.0f;
    }
};
```

```cpp
// Kalibrace: napětí → fyzikální jednotka (°C) pomocí gain a offset
class Kalibrace : public SenzorDecorator {
    float gain, offset;
public:
    Kalibrace(std::shared_ptr<Senzor> s, float g, float o)
        : SenzorDecorator(s), gain(g), offset(o) {}
    float read() override {
        return src->read() * gain + offset;   // lineární kalibrace
    }
};
```

Dekorátory skládáme do řetězce — pořadí závisí na naší potřebě:

```cpp
auto raw   = std::make_shared<AnalogSenzor>(34);
auto filtr = std::make_shared<MovingAvgFilter>(raw);       // 1. filtruj šum
auto kalib = std::make_shared<Kalibrace>(filtr, 100.f, -50.f);  // 2. kalibruj

float teplota = kalib->read();  // prochází celým řetězcem: ADC → avg → °C
```

> Každý dekorátor je samostatná třída s jednou odpovědností. Řetězec lze za běhu sestavit jinak — např. vynechat filtr při ladění, přidat logování při diagnostice.

Composite — záměr

Composite umožňuje skládat objekty do stromových struktur a pracovat s jednotlivými objekty i jejich skupinami **jednotným způsobem**. Klientský kód nerozlišuje, zda volá metodu na listu (jednoduchý objekt) nebo na skupině (složený objekt).

```cpp
// Společné rozhraní pro listy i skupiny
class Komponenta {
public:
    virtual ~Komponenta() = default;
    virtual void vykonaj() const = 0;
};

// List — jednoduchý objekt bez podkomponent
class List : public Komponenta {
    std::string nazev;
public:
    explicit List(std::string n) : nazev(std::move(n)) {}
    void vykonaj() const override {
        std::cout << "Provádím: " << nazev << "\n";
    }
};
```

```cpp
// Composite — složený objekt, rekurzivně volá podkomponenty
class Skupina : public Komponenta {
    std::vector<std::shared_ptr<Komponenta>> deti;
public:
    void pridej(std::shared_ptr<Komponenta> k) { deti.push_back(std::move(k)); }
    void vykonaj() const override {
        for (const auto& d : deti) d->vykonaj();   // rekurzivní průchod
    }
};
```

🔌 **Embedded:** strom diagnostických testů při startu — skupiny (I2C, WiFi) sdružují jednotlivé testy; spuštění skupiny automaticky spustí všechny podtesty.

Flyweight — záměr

Flyweight (muška) sdílí **neměnná data** mezi velkým počtem objektů, aby ušetřila paměť.

Rozdělíme stav objektu na dvě části:

- **Vnitřní stav** (intrinsic) — sdílený, neměnný, uložen jednou v továrně
- **Vnější stav** (extrinsic) — unikátní pro každý objekt, předán při volání

```
// Flyweight — sdílená bitmapa jednoho znaku (8 bytů)
class ZnakFont {
    uint8_t bitmapa[8];
public:
    explicit ZnakFont(char c) { /* načti z flash/ROM */ }
    void kresli(int x, int y) const { /* vykresli na LCD */ }
};
```

```cpp
// Flyweight Factory — zajistí, aby každý znak existoval jen jednou
class FontTovarna {
    std::unordered_map<char, std::shared_ptr<ZnakFont>> cache;
public:
    std::shared_ptr<ZnakFont> ziskejZnak(char c) {
        if (!cache.count(c))
            cache[c] = std::make_shared<ZnakFont>(c);  // vytvoř jednou
        return cache[c];   // pak vracíme vždy stejný objekt
    }
};

// Každý znak v textu sdílí bitmapu; unikátní je jen pozice (x, y)
FontTovarna fonty;
for (size_t i = 0; i < text.size(); i++)
    fonty.ziskejZnak(text[i])->kresli(i * 8, 0);
```

🔌 **Embedded:** font renderer pro malý LCD (SSD1306, ILI9341) — bitmapa každého znaku jednou v RAM, texty libovolné délky bez duplicit.

Vzory chování

Behavioral Patterns

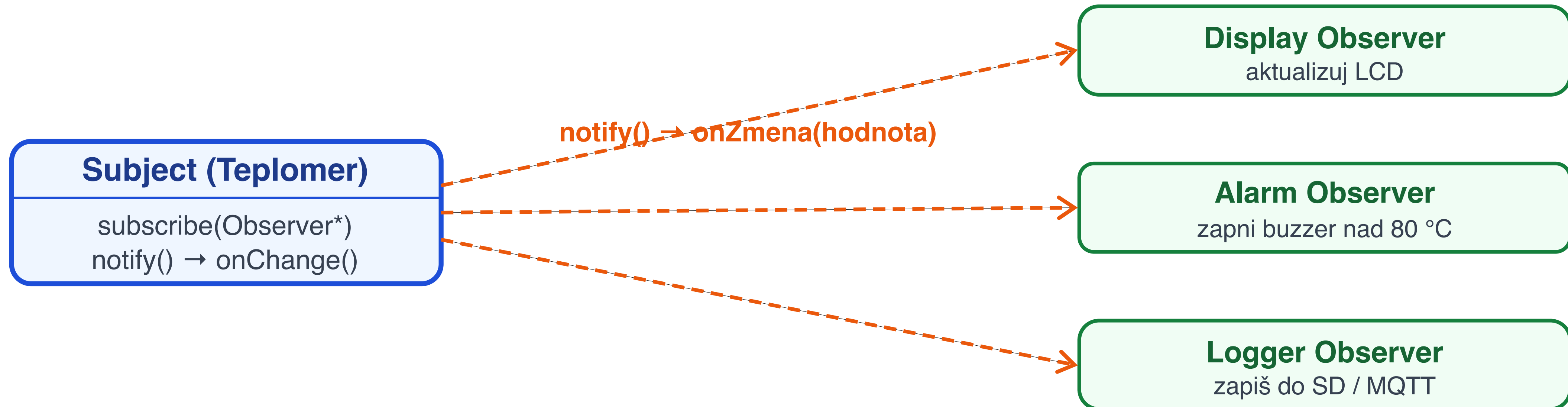
Jak objekty spolupracují a komunikují

★ Klíčové pro event-driven a embedded programování

Observer — záměr

Observer (pozorovatel) definuje závislost 1:N — jeden objekt (Subject) upozorní všechny zaregistrované odběratele (Observers), kdykoli se změní jeho stav. Odběratelé přitom nemusí vědět nic o sobě navzájem ani o tom, kdo jiný je registrován.

Vzor je základem **event-driven programování** — místo neustálého dotazování (`polling`) dostane každý odběratel notifikaci přesně tehdy, kdy nastane událost.



Observer — kód

```cpp
// Rozhraní odběratele — každý Observer implementuje tuto metodu
class SenzorObserver {
public:
    virtual ~SenzorObserver() = default;
    virtual void onZmena(float hodnota) = 0;
};
```

Subject spravuje seznam odběratelů a oznamuje jim změny. Všimněte si podmínky: notifikujeme jen při **skutečné změně** o více než 0,5 °C. Bez toho bychom zbytečně přetěžovali odběratele.

```cpp
class Teplomer {
    std::vector<SenzorObserver*> odberatele;
    float posledni = 0.0f;
public:
    void subscribe(SenzorObserver* o) { odberatele.push_back(o); }

    void zmer() {
        float t = readADC() * 100.0f / 4095.0f;
        if (std::abs(t - posledni) > 0.5f) {   // notifikuj jen při změně
            posledni = t;
            for (auto* o : odberatele) o->onZmena(t);
        }
    }
};

// Konkrétní odběratelé — každý reaguje po svém
class AlarmObserver : public SenzorObserver {
public:
    void onZmena(float h) override {
        digitalWrite(BUZZER_PIN, h > 80.0f ? HIGH : LOW);
    }
};
```

Observer — embedded: ISR jako Subject (1/2)

Event-driven architektura s Observerem je v embedded základní technikou pro práci s přerušeními. ISR by měla být co nejkratší — nastaví jen příznak, vlastní zpracování provede `loop()`.

```
class ButtonSubject {
    uint8_t pin;
    std::vector<std::function<void()>> handlery;
    volatile bool stisknuto = false;   // volatile: přístup z ISR i z loop()

public:
    explicit ButtonSubject(uint8_t p) : pin(p) { pinMode(p, INPUT_PULLUP); }

    void addHandler(std::function<void()> h) { handlery.push_back(std::move(h)); }

    // ISR — voláno z přerušení, musí být co nejkratší!
    void IRAM_ATTR handleISR() { stisknuto = true; }

    // Voláme z loop() — zde bezpečně zpracujeme příznak
    void tick() {
        if (stisknuto) {
            stisknuto = false;
            for (auto& h : handlery) h();   // notify všech odběratelů
        }
    }
};
```


Proč `volatile bool`?

Observer — embedded: ISR jako Subject (2/2)

Registrace handlerů a napojení na přerušení:

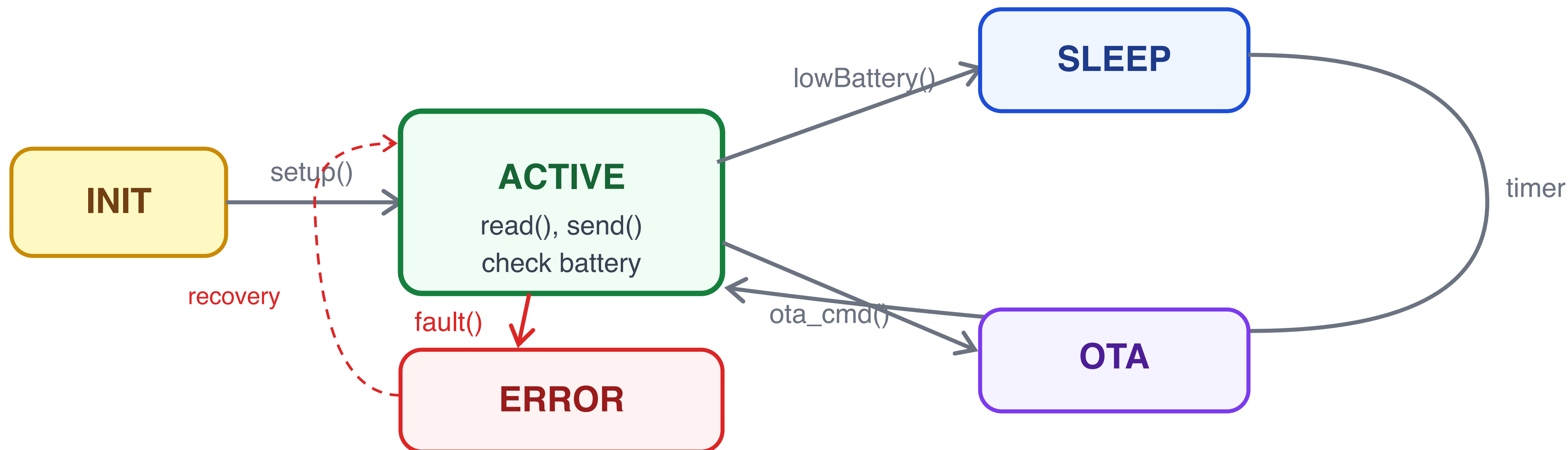
```
ButtonSubject btn(BOOT_PIN);

// Handlery jako lambdy — Observer pattern bez explicitních tříd
btn.addHandler([]() { toggleLED();                          });
btn.addHandler([]() { Logger::instance().log("click!");   });

attachInterrupt(digitalPinToInterrupt(BOOT_PIN),
    []() { btn.handleISR(); }, FALLING);
```

Vzor ISR → volatile flag → loop() zpracování je standard pro embedded C++. Samotná ISR obsahuje jen nastavení příznaku — veškerá logika patří do `tick()`.

State / FSM — záměr

State vzor (stavový automat, FSM — Finite State Machine) umožňuje objektu **měnit své chování spolu se změnou vnitřního stavu**. Namísto rozsáhlých `if`-`else` větvení kód explicitně modeluje, ve kterém stavu zařízení je a co každý stav dělá.

FSM je v embedded programování **naprosto klíčový** — každé IoT zařízení prochází definovaným životním cyklem.



State — implementace (1/2)

Začneme definicí stavů a privátní metodou `vstupDoStavu()`. Ta je **centrálním místem** všech přechodů — každý přechod prochází přes ni, takže vstupní akci stavu definujeme jen jednou.

```
enum class Stav { INIT, ACTIVE, SLEEP, OTA, ERROR };

class IoTZarizeni {
    Stav stav = Stav::INIT;

    // Centrální bod přechodů — vstupní akce každého stavu
    void vstupDoStavu(Stav novy) {
        stav = novy;
        switch (novy) {
            case Stav::SLEEP: esp_deep_sleep(30'000'000ULL); break;  // 30 s
            case Stav::OTA:   ArduinoOTA.begin();            break;
            case Stav::ERROR: Serial.println("FAULT");       break;
            default: break;
        }
    }
```

`esp_deep_sleep()` zařízení skutečně uspí — po probuzení časovačem se ESP32 restartuje a FSM začíná znovu od `INIT`. Vstupní akce stavu `OTA` spustí OTA server; pokud bychom to dělali uvnitř `update()`, spustilo by se to opakovaně při každém průchodu smyčkou.

State — implementace (2/2)

Metoda `update()` se volá z `loop()` a provádí logiku **aktuálního stavu**. Přechod nastane voláním `vstupDoStavu()`.

```
public:
    // Voláme opakovaně z loop()
    void update() {
        switch (stav) {
            case Stav::INIT:
                setupHardware();
                vstupDoStavu(Stav::ACTIVE);   // přechod ihned po inicializaci
                break;
            case Stav::ACTIVE:
                readSensors();
                sendMQTT();
                if (battery() < 10) vstupDoStavu(Stav::SLEEP);
                break;
            case Stav::ERROR:
                blinkLED(3);
                delay(5000);
                vstupDoStavu(Stav::ACTIVE);   // pokus o zotavení
                break;
            default: break;
        }
    }

    // Externí triggery — volané z ISR nebo MQTT handleru
    void onFault() { vstupDoStavu(Stav::ERROR); }
```

Strategy — záměr

Strategy (strategie) definuje **rodinu zaměnitelných algoritmů** a umožňuje je vyměňovat za běhu — bez změny klientského kódu.

Typická situace v DSP: jeden senzorový kanál chceme někdy filtrovat klouzavým průměrem, jindy mediánem (odolnějším vůči výpadkům), a při ladění třeba vůbec. Strategy nám umožní přepínat filtr za běhu — třeba přes MQTT příkaz.

```
// Společné rozhraní pro všechny filtrovací strategie
class FiltrStrategie {
public:
    virtual ~FiltrStrategie() = default;
    virtual float filtruj(float vstup) = 0;
};
```

```cpp
// Strategie A: klouzavý průměr přes posuvné okno 8 vzorků
class KlouzavyPrumer : public FiltrStrategie {
    float buf[8] = {}; int idx = 0;
public:
    float filtruj(float v) override {
        buf[idx++ % 8] = v;
        return std::accumulate(std::begin(buf), std::end(buf), 0.0f) / 8.0f;
    }
};
```

```cpp
// Strategie B: medián přes okno 5 vzorků — odolný vůči spike šumu
class MedianFiltr : public FiltrStrategie {
    float buf[5] = {}; int idx = 0;
public:
    float filtruj(float v) override {
        buf[idx++ % 5] = v;
        float s[5]; std::copy(std::begin(buf), std::end(buf), s);
        std::sort(std::begin(s), std::end(s));
        return s[2];    // prostřední prvek = medián
    }
};
```

Strategy — záměna za běhu

Senzor drží ukazatel na aktivní strategii. Lze ji kdykoli zaměnit — třeba po přijetí MQTT příkazu nebo po BLE konfiguraci.

```
class MericiSenzor {
    std::unique_ptr<FiltrStrategie> filtr;  // aktuálně aktivní strategie
    uint8_t pin;
public:
    explicit MericiSenzor(uint8_t p) : pin(p) {}

    // Záměna strategie za běhu — starý filtr se automaticky uvolní
    void setFiltr(std::unique_ptr<FiltrStrategie> f) {
        filtr = std::move(f);
    }

    float read() {
        float raw = analogRead(pin) * 3.3f / 4095.0f;
        return filtr ? filtr->filtruj(raw) : raw;   // bez filtru = surová hodnota
    }
};
```

Přepínání strategie přes MQTT:

```
MericiSenzor senzor(34);
senzor.setFiltr(std::make_unique<KlouzavyPrumer>());  // výchozí filtr

// Přijetí příkazu z MQTT brokeru → záměna filtru bez restartu zařízení:
void onMqttMsg(const std::string& cmd) {
    if      (cmd == "filtr:median") senzor.setFiltr(std::make_unique<MedianFiltr>());
    else if (cmd == "filtr:avg")    senzor.setFiltr(std::make_unique<KlouzavyPrumer>());
    else if (cmd == "filtr:off")    senzor.setFiltr(nullptr);  // zakáže filtr
}

float t = senzor.read();   // vždy přes aktuálně nastavenou strategii
```

🔌 **Embedded DSP:** Strategy se skvěle hodí pro záměnné PID regulátory (P / PI / PID jako strategie), kompresní algoritmy nebo komunikační protokoly — výběr závisí na kontextu nebo konfiguraci přijaté přes síť.

Vzor	Řeší	Embedded příklad
Singleton	Jeden sdílený zdroj	SPI/I2C sběrnice, Logger
Factory Method	Skrýt konkrétní typ objektu	Senzor dle HW detekce
Abstract Factory	Rodina objektů pro platformu	HAL pro ESP32 vs STM32
Builder	Složitá volitelná konfigurace	WiFi / MQTT nastavení
Adapter	Integrace cizí / C knihovny	STM32 HAL → C++
Bridge	Abstrakce × implementace	WiFi / LoRa / BLE
Facade	Zjednodušit komplexní API	Motor driver HAL
Proxy	Řízený přístup, caching	Lazy I2C, read cache
Decorator	Přidat chování bez dědičnosti	Sensor pipeline
Composite	Strom objektů, jednotné API	Diagnostické testy
Flyweight	Sdílení dat, úspora paměti	Font renderer LCD
Observer ★	Notifikace N příjemců	ISR → event handlery
State / FSM ★	Chování dle stavu zařízení	Init → Active → Sleep
Strategy ★	Záměnné algoritmy	DSP filtry, PID

Shrnutí

Návrhové vzory jsou **společný jazyk** — pojmenují řešení, které by jinak každý implementoval jinak, a tím usnadňují komunikaci i čtení cizího kódu.

V embedded programování jsou nejcennější:
Observer (event-driven ISR), **State/FSM** (device lifecycle),
Facade (HAL), **Strategy** (záměnné algoritmy)

> *"Programs must be written for people to read,*
> *and only incidentally for machines to execute."*
> *— H. Abelson & G. Sussman, SICP*

```
// Dobře pojmenovaný kód říká, co dělá — ne jak to dělá:
motor.pohybuj(200);                              // Facade
senzor.setFiltr(make_unique<MedianFiltr>());     // Strategy
Logger::instance().log("Ready");                // Singleton
if (stav == Stav::ACTIVE) readSensors();        // State FSM
btn.addHandler([]() { toggleLED(); });          // Observer
```