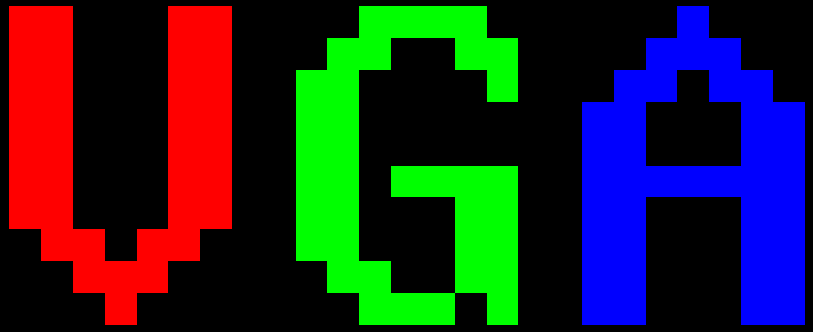



# Návrhové vzory ve hrách

Jan Matoušek

15. 10. 2026

# *Command*

# Kontext

```
void InputHandler::handleInput() {
    if (isPressed(BUTTON_X)) jump();
    else if (isPressed(BUTTON_Y)) fireGun();
    else if (isPressed(BUTTON_A)) swapWeapon();
    else if (isPressed(BUTTON_B)) lurchIneffectively();
}
```

Problémy?

- Chceme oddělit zpracování vstupu (handling) od reakce na vstup (update).
- Někdy nechceme na vstup reagovat hned (zpoždění vyhodnocování).
- Někdy chceme vstup vyhodnocovat opakovaně nebo se vracet.
- Chceme příkazy vykonávat jinde (na jiném stroji, v jiné komponentě atd.).

```
class Command {
public:
    virtual ~Command() {}
    virtual void execute() = 0;
};

class JumpCommand : public Command {
public:
    void execute() override { jump(); }
};

class FireCommand : public Command {
public:
    void execute() override { fireGun(); }
};
```

# Upravené zpracování vstupu

```
class InputHandler {
public:
    void handleInput();
private:
    Command* buttonX_;
    Command* buttonY_;
    Command* buttonA_;
    Command* buttonB_;
};
void InputHandler::handleInput() {
    if (isPressed(BUTTON_X)) buttonX_->execute();
    else if (isPressed(BUTTON_Y)) buttonY_->execute();
    else if (isPressed(BUTTON_A)) buttonA_->execute();
    else if (isPressed(BUTTON_B)) buttonB_->execute();
}
```

# Lépe oddělené zpracování vstupu

```
Command* InputHandler::handleInput() {
    if (isPressed(BUTTON_X)) return buttonX_;
    else if (isPressed(BUTTON_Y)) return buttonY_;
    else if (isPressed(BUTTON_A)) return buttonA_;
    else if (isPressed(BUTTON_B)) return buttonB_;
    return nullptr;
}
void Game::loop() {
    while (true) {
        auto command = inputHandler.handleInput();
        if (command) command->execute();
        // ...
    }
}
```

Problémy? Co když chci skákat a střílet zároveň?

# Rozšíření: Fronta příkazů

```
void InputHandler::handleInput(std::queue<Command*> q) {
    if (isPressed(BUTTON_X)) q.push_back(buttonX_);
    else if (isPressed(BUTTON_Y)) q.push_back(buttonY_);
    else if (isPressed(BUTTON_A)) q.push_back(buttonA_);
    else if (isPressed(BUTTON_B)) q.push_back(buttonB_);
}
void Game::loop() {
    while (true) {
        inputHandler.handleInput(commandQueue);
        while (!commandQueue.empty()) {
            commandQueue.front()->execute();
            commandQueue.pop_front();
        }
        // ...
    }
}
```

# Další možná rozšíření

- Příkazy pro konkrétní/různé objekty:
  - metoda `execute()` může přijímat parametry,
  - instance třídy Command může obsahovat další parametry.
- Undo (a redo):
  - přidání virtuální metody `undo()`,
  - zásobníky na provedené a vrácené příkazy.
- Doplnění o čas / číslo snímku:
  - možnost pamatovat si historii s časovými údaji,
  - vhodné pro schopnost opětovného přehrávání vstupů a událostí.
- Serializace / deserializace:
  - pro ukládání sekvence příkazů na disk k následné rekonstrukci stavu hry,
  - pro přenos příkazů po síti.

# *Observer*

# Kontext

```
void Hero::update() {
    // ...
    if (tookCoin) {
        score += 5;
        gui.scoreDisplay.setScore(score);
        if (score >= 1000000) {
            awardAchievement(A_BIG_MONEY);
        }
    }
    // ...
}
```

Problémy?

- Třída `Hero` obsahuje logiku jiných částí programu
- Obtížná údržba kódu

```
class Observer {
    virtual void notifyObserver(EventData event) = 0;
};

class Subject {
    std::vector<Observer*> observers;
public:
    void addObserver(Observer* observer); // obvious
    void removeObserver(Observer* observer); // obvious
protected:
    void notifyObservers(EventData event) {
        for (auto observer: observers) {
            observer->notifyObserver(event);
        }
    }
};
```

# Aplikace v ukázce

```
class GuiScoreObserver : public Observer { ... };
class AchievementObserver : public Observer { ... };

class Hero : public Subject { ... };

void Hero::update() {
    // ...
    if (tookCoin) {
        score += 5;
        notifyObservers(EventData{ ... });
    }
    // ...
}
```

# Registrace observerů

- Zodpovědnost observerů, nikoliv subjektů
  - ideálně ve svém konstruktoru zavolá `addObserver()`
  - ideálně ve svém destruktoru zavolá `removeObserver()`
  - pokud observer pozoruje více subjektů, musí si je pamatovat
- Potenciální problém: zánik subjektu dříve než observeru
  - v C++ může řešit použití `std::weak_ptr`,
  - případně odložit rušení objektů do jiné části herní smyčky, viz `queue_free()` v Godot engine

# Další rozšíření

- Zpracování observerů je synchronnní; pozor na rychlost!
  - lze odložit provádění observerů odkládáním jejich notifikací do fronty,
  - případně lze využít multithreading (pozor na data races).
- Předávání více informací:
  - do notifikace lze přidat bližší informace o změně stavu,
  - lze přidat i odkaz na subjekt, jeden observer potom může pozorovat více subjektů.
- Subjekt nemusí nutně být nejmenší část herní scény:
  - typicky např. fyzikální engine může vyhodnocovat kolize objektů a posílat notifikace o zjištěných kolizích objektů.
- Pro každý druh notifikací lze mít vlastní hierarchii subjektů a observerů
  - např. pomocí šablon `Observer<EventData>` a `Subject<EventData>`,
  - pozor na spotřebu paměti na registraci observerů!

# *Publisher / Subscriber*

# Kontext

Vzor Command:

- Někudy se předá příkaz a vykoná se.
- *Co když potřebuji různé reakce na příkaz na různých místech?*

Vzor Subject / Observer:

- Observer musí vědět, koho sleduje.
- *Co když je množina subjektů nestálá a často se mění?*

# Zprávy a kanál

- Vytvoříme místo, kam se mohou zprávy zadávat a odkud se mohou vyzvedávat.
- Kdo potřebuje informovat, pošle zprávu, kdo potřebuje reagovat, přečte zprávu.
- Nejjednodušší informační kanál je fronta:

```
struct Message { ... };
std::queue<Message> q;
class Producer {
    std::queue<Message> * outgoingMessages;
};
class Consumer {
    std::queue<Message> * ingoingMessages;
};
```

Problémy?

# Více čtenářů

- Zprávu si nemůže přečíst více čtenářů. – *Což se někdy nehodí.*
- Možné řešení: sledovaný kanál.

```
class ObservedChannel : public Subject {
    Message msg; // proč nepotřebuji frontu?
public:
    Message & read() { msg; }
    void send(Message message) {
        msg = message;
        notifyObservers();
    }
}
```

Problémy?

# Více čtenářů, s vyřizováním zpráv najednou (A)

- Nechceme obtěžovat adresáty pokaždé, když dostaneme zprávu.
- Řešení A: Rozdělíme frontu na vstupní a výstupní; každému čtenáři tzv. broker zkopíruje zprávu do jeho fronty.

```
class ObservedChannel : public Subject {
public:
    void send(Message message) { notifyObservers(msg); }
}
class ChannelObserver : public Observer {
    std::queue<Message> q;
public:
    void notifyObserver(Message msg) override { q.push(msg); }
    // top, pop, empty, clear
}
```

Problémy?

# Více čtenářů, s vyřizováním zpráv najednou (B)

- Nechceme obtěžovat adresáty pokaždé, když dostaneme zprávu.
- Řešení B: Každý adresát bude mít svůj stav čtení kanálu.

```
class Channel {
    std::vector<Message> data;
public:
    void push(Message message);
    size_t size();
    Message& at(size_t position);
}
```

```
class ChannelReader {
    Channel& ch;
    size_t cursor = 0;
public:
    ChannelReader(channel& ch) : ch(ch);
    bool empty() {
        return cursor == ch.size()
    }
    Message& get() {
        return ch.at(cursor++);
    }
}
```

Problémy?

# Uklízení kanálu na konci snímku

- Někdy bychom měli kanál uklidit. Např. (včetně použití):

```
// na straně každého čtenáře
while(!reader.empty()) {
    auto msg = reader.get();
    // ...
}
reader.reset();

// jednou, na konci snímku:
channel.reset();
```

*Teď už můžeme realizovar vzor Publisher / Subscriber, ale…*

Problém? Co když je subscriber ve snímku na řadě před publisherem?

# Rozšíření: Double buffering

- Kanál bude mít ve skutečnosti dvě pole zpráv – hlavní a z předchozího snímku.
- Na konci snímku se pole prohodí a přední se vyprázdní.

```
class Channel {
    std::vector<Message> a, b;
    bool current_is_a = false; // proč nepřehazovat pole přímo?
};
```

- Čtenář si rozšíří cursor o informaci, zda čte z `a` nebo z `b`.

```
class ChannelReader {
    Channel& ch;
    size_t cursor = 0;
    bool reads_a = false;
}
```

*Využívá např. Bevy framework.*

# *State*

# Kontext

```
void Heroine::handleInput(Input input) {
    if (input == PRESS_B) {
        yVelocity_ = JUMP_VELOCITY;
        setGraphics(IMAGE_JUMP);
    }
}
```

Je tam někde chyba? Skákání ve vzduchu.

# Kontext

```
void Heroine::handleInput(Input input) {
    if (input == PRESS_B) {
        if (!isJumping_) {
            isJumping_ = true;
            // Jump...
        }
    }
}
```

Přidejme další druh pohybu…

# Kontext

```
void Heroine::handleInput(Input input) {
    if (input == PRESS_B) {
        // Jump if not jumping...
    }
    else if (input == PRESS_DOWN) {
        if (!isJumping_) {
            setGraphics(IMAGE_DUCK);
        }
    }
    else if (input == RELEASE_DOWN) {
        setGraphics(IMAGE_STAND);
    }
}
```

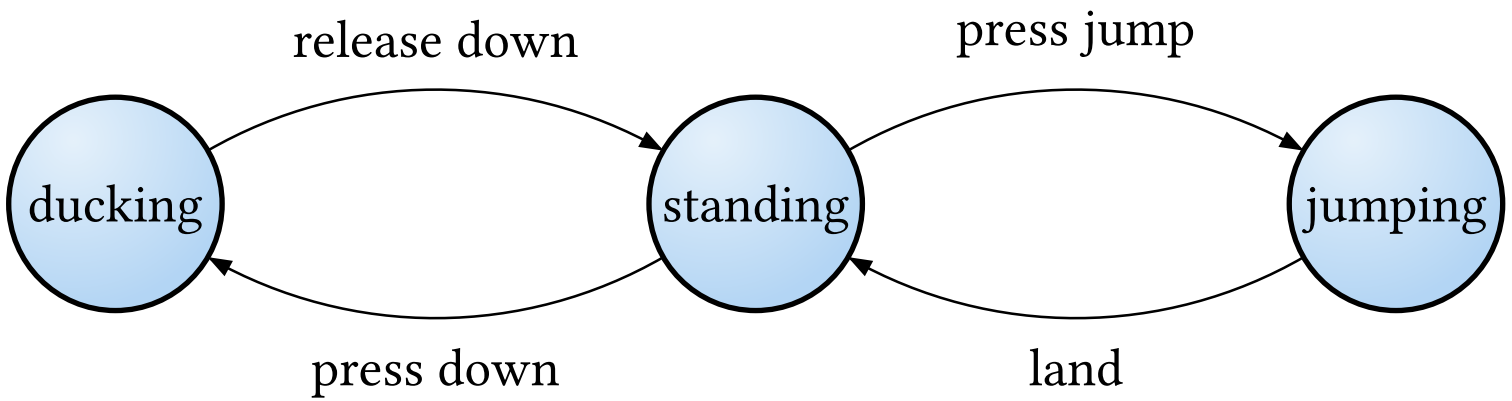
Je tam někde chyba? Skrčím se, vyskočím, odkrčím se → nastaví se špatná grafika.

# Kontext

```
void Heroine::handleInput(Input input) {
    if (input == PRESS_B) {
        if (!isJumping_ && !isDucking_) { /* Jump... */ }
    } else if (input == PRESS_DOWN) {
        if (!isJumping_) {
            isDucking_ = true; setGraphics(IMAGE_DUCK);
        }
    } else if (input == RELEASE_DOWN) {
        if (isDucking_) {
            isDucking_ = false; setGraphics(IMAGE_STAND);
        }
    }
}
```

A dál? S dalšími flagy bude kód stále méně přehledný.

# Co potřebujeme reprezentovat?




Můžeme si na stavy zavést `enum` a větvit kód podle jeho hodnoty.

- To má smysl, dokud je různých stavů málo, nebo když se vyhýbám OOP.

# Pattern

```
class HeroineState {
public:
    virtual ~HeroineState() {}
    virtual void handleInput(Heroine& heroine, Input input) {}
    virtual void update(Heroine& heroine) {}
};
class DuckState : public HeroineState;
class StandState : public HeroineState;
class JumpState : public HeroineState;
class Heroine {
    HeroineState* currentState;
public:
    void handleInput(Input input) { currentState->handleInput(*this, input); }
    void update() { currentState->update(*this); }
};
```

# Kde alokovat stavy?

- Stavy se mohou měnit často, dynamická alokace nemusí být žádoucí;
  - statická alokace často stačí.
- Někdy však stav může zahrnovat další informaci, např. snímek animace;
  - dynamická alokace je možnost (pozor na správu paměti),
  - některé stavové proměnné mohou být uložené v herním objektu vedle stavu:

```
class Heroine {
    HeroineState* currentState;
    char currentStateData[STATE_DATA_SIZE]; // each state casts it
    // ...
};
```

# Přechody mezi stavy

```
class HeroineState { // ...
    virtual void onEnter(Heroine& heroine) {}
    virtual void onExit(Heroine& heroine) {}
    virtual HeroineState* handleInput(Heroine& heroine, Input input) {}
};
class Heroine { // ...
    void handleInput(Input input) {
        auto nextState = currentState->handleInput(*this, input);
        if (nextState) {
            currentState->onExit(*this);
            currentState = nextState;
            currentState->onEnter(*this);
        }
    }
};
```

# Nedostatky

- I konečný stavový automat může být zbytečně komplexní
  - příklad: hráč může chodit ve stoje i v kleče, může přitom mít nasazený noktovizor, a ještě je potřeba řešit efekt zranění…
  - všech možných kombinací je 16 (pokud jsou z každého stavu 2 varianty), přechodů mezi nimi až 240.
  - některé události způsobí přechod do jednoho stavu, ať už byl předchozí stav jakýkoliv

Jaká jsou řešení?

- Souběžné stavové automaty (stavy jsou vzájemně nezávislé)
  - ukládám vzájemně nezávislé stavy pohybu, krčení, noktovizoru, zranění…
- Hierarchické stavové automaty (stavy mají zčásti společné chování)
  - využijeme dědičnost stavů; co neumí potomek, předá rodiči
  - nebo lze využít seznam souběžných stavů; co nezpracuje jeden stav, zpracuje další v seznamu

# Další rozšíření

- Zásobník stavů
  - vhodné pro historii a vracení se k předchozímu stavu
  - příklad: rozehraná hra, pause menu, zpátky k rozehrané hře

# *Sequence*

# Kontext

- Chování entity je často rozloženo do více snímků hry

```
class Building {
    int foundations_pct, building_pct, flag_waving;
public:
    void update() {
        if (foundations_pct < 100) foundations_pct++;
        else if (building_pct < 100) building_pct++;
        else flag_waving++;
    }
};
```

# Možná řešení

- Vícevláknové programování – kolik vláken?
- `async/await` – ne všude máme k dispozici

# Pattern

```
class Step {
public:
    virtual bool update() = 0;
};
class FoundationsStep : public Step;
class BuildingStep : public Step;
class BuiltStep : public Step;
class Sequence {
    std::queue<Step*> steps;
public:
    void update() {
        if (!steps.empty())
            if (steps.front()->update())
                steps.pop_front();
    }
};
```

# Poznámky

- Velmi blízké stavům, lze implementovat podobným způsobem a kombinovat
- Vhodné i pro případné skriptování chování entit
- Lze upravit pro odpočítávání snímků, časů atd.

# *Prototype*

# Kontext

```
class Monster;
class Vampire : public Monster;
class Skeleton : public Monster;

class MonsterSpawner {
public:
    virtual Monster* spawn() = 0;
};
class VampireSpawner : public MonsterSpawner;
class SkeletonSpawner : public MonsterSpawner;
```

Problémy?

- Množství opakujícího se / boilerplate kódu
- Duplikace hierarchie komplikuje údržbu

```
class Monster {
public:
    virtual Monster* clone() = 0;
};
class Vampire : public Monster;
class Skeleton : public Monster;

class MonsterSpawner {
    Monster* prototype;
public:
    MonsterSpawner(Monster* prototype) : prototype(prototype) {}
    Monster* spawn() {
        return prototype->clone();
    }
};
```

# Výhody a další rozšíření

- Pouze jedna univerzální třída pro spawner.
- Lze mít spawnery pro různě nastavené objekty stejného typu.
- Přidání dodatečných parametrů do metod `spawn()` a `clone()`:
  - vhodné např. pro (deterministickou) randomizaci vytvořených kopií,
  - ale není to nutné.
- Možnost rozšíření spawnerů např. o počítání vytvořených instancí a limity…

# *Builder*

## Kontext

```
class Computer {
public:
    Computer(
        std::optional<RAM> ram,
        std::optional<GPU> gpu,
        std::optional<WiFi> wifi,
        std::optional<CD> cd, ...
    );
};
void somewhere() {
    Computer c1({...}, std::nullopt, std::nullopt, std::nullopt, ...);
    Computer c2({...}, {...}, std::nullopt, std::nullopt, ...);
    // ...
}
```

Problémy? Velké množství nevyužitých parametrů kostruktoru, dlouhý kód

# Pattern

```
class ComputerBuilder {
    std::optional<RAM> ram, std::optional<GPU> gpu, ...;
public:
    ComputerBuilder& withRAM(RAM r) { ram = r; return *this; }
    ComputerBuilder& withGPU(GPU g) { gpu = g; return *this; }
    Computer build() {
        return Computer(ram, gpu, ...);
    }
};
void somewhere() {
    Computer c1 = ComputerBuilder().withRAM({...}).build();
    Computer c2 = ComputerBuilder().withRAM({...}).withGPU({...}).build();
    // ...
}
```

# *Service Locator*

# Kontext

- Některé části programu (služby) nemusí být v různých instancích, ale existují jen jednou
  - např. komunikace s grafickou kartou, práce se souborovým systémem, práce se sítí, integrace se Steam atd.
  - chceme k nim mít přístup odevšud
- Máme OOP, nechceme mít volně poházené funkce.
- Někdy v budoucnu bychom chtěli jednu část programu vyměnit za jinou se stejným rozhraním.

# Jiný (anti)pattern – Singleton

```
class FileSystem {
public:
    static FileSystem& instance() {
        if (!instance_) instance_ = new FileSystem();
        return *instance_;
    }
private:
    inline static FileSystem* instance_;
};
// FileSystem::instance(). ...
```

Problémy?

- Globální proměnná.
- Komplikované (pomalé) použití ve vícevláknovém kódu.
- Nelze jednoduše dle potřeby vyměnit jednu službu za jinou.

# Pattern

```
class FileSystem;
class VirtualFileSystem : public FileSystem;
class DriveFileSystem : public FileSystem;
class ServiceLocator {
    inline static FileSystem* fileSystem;
public:
    static FileSystem* getFileSystem();
    static void registerFileSystem(FileSystem* fileSystem);
};
int main() {
    DriveFileSystem fs;
    ServiceLocator::registerFileSystem(&fs);
    Game.run();
}
```

Co když nikdy nezavolám `registerFileSystem`, ale použiju `getFileSystem`?

# Pattern

```
class FileSystem;
class NullFileSystem : public FileSystem;
class VirtualFileSystem : public FileSystem;
class DriveFileSystem : public FileSystem;
class ServiceLocator {
    inline static NullFileSystem nullFileSystem;
    inline static FileSystem* fileSystem = &nullFileSystem;
public:
    static FileSystem* getFileSystem();
    static void registerFileSystem(FileSystem* fileSystem);
};
int main() {
    DriveFileSystem fs;
    ServiceLocator::registerFileSystem(&fs);
    Game.run();
}
```

# Možné rozšíření pro více druhů služeb (1/2)

```
template<typename Service> class SingleServiceLocator {
    inline static typename Service::NullService nullService;
    inline static Service* service = &nullService;
public:
    static Service* getService();
    static void registerService(Service* fileSystem);
};
class ServiceLocator : public SingleServiceLocator<FileSystem>, ... {
public:
    template<typename Service> static Service* getService() {
        return SingleServiceLocator<Service>::getService();
    }
    template<typename Service> static void registerService(Service* s) {
        SingleServiceLocator<Service>::registerService(s);
    }
};
```

# Možné rozšíření pro více druhů služeb (2/2)

```
class FileSystem {
public:
    virtual AssetData loadAsset(std::string_view path) = 0;
    using NullService = class NullFileSystem;
};
class NullFileSystem : public FileSystem {
    AssetData loadAsset(std::string_view path) override;
};
class VirtualFileSystem : public FileSystem;
class DriveFileSystem : public FileSystem;
int main() {
    DriveFileSystem fs;
    ServiceLocator::registerService<FileSystem>(&fs);
    Game.run();
}
```

# *Shrnutí*

# Obecné rady a doporučení

- Některé platformy mají doporučenou nebo obvyklou architekturu
  - Dáno např. frameworkem, nebo fungováním dané platformy
  - Typicky web nebo desktopové aplikace: MVC, FE/BE
- Vyhněte se:
  - Všemocné objekty a funkce
  - Cyklické závislosti mezi objekty
  - Prolínání zodpovědností
- Naopak:
  - Oddělte vrstvy (prezentace × simulace × operační systém)
  - Vyjasněte si komunikaci mezi objekty
  - Snažte se o data driven engine (o tom více příště)

# Předvedené vzory

- Komunikace – Command, Observer, Publisher / Subscriber
- Chování – State, Sequence
- Vytváření – Prototype, Builder, (Factory)
- Služby – Singleton, Service Locator, (Dependency Injection)

Některé jsme si ukázali i bez pojmenování.

Doporučený zdroj:

- Game Programming patterns, https://gameprogrammingpatterns.com/