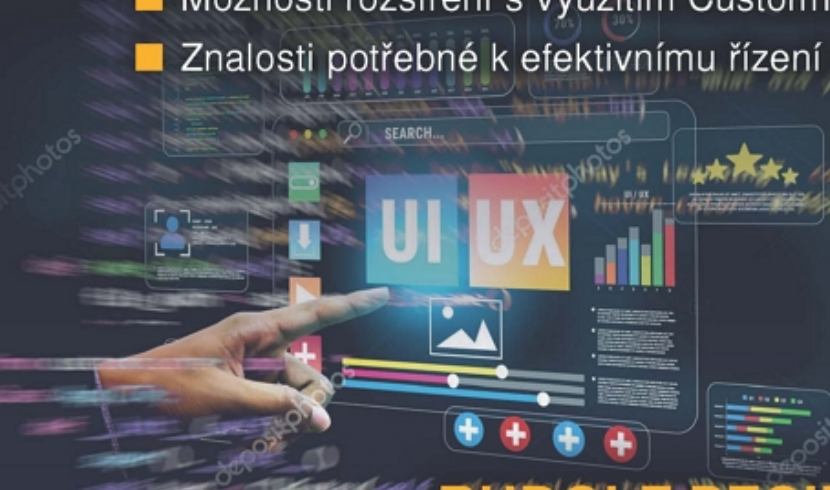
EDICE
myslíme v ...

knihovna programátora

- Plnohodnodnotné GUI s použitím standardní knihovny
- Správa oken, událostí a asynchronního běhu
- Systematický vývoj čisté a rozšiřitelné aplikace
- Možnosti rozšíření s využitím CustomTkinter
- Znalosti potřebné k efektivnímu řízení AI



RUDOLF PECINOVSKÝ

Python

architektura GUI

Návrh aplikací nad knihovnou Tkinter

soubory ke stažení na WWW.GRADA.CZ

GRADA

Podrobný obsah

Kapitola 17 Moderní kabát s CustomTkinter 376
17.1 **Příprava nové verze** 376
17.1.1 Instalace knihoven *CustomTkinter* a *Pillow* 377
17.1.2 **Architektonická stabilita při přechodu** 377
17.2 **Proč CustomTkinter a co přináší nového** 378
17.2.1 Témata a barevné režimy 378
17.2.2 **Práce s grafikou a podpora formátů** 379
17.2.3 Hybridní model a kompatibilita 380
17.3 **Rozšíření sdílených konstant** 381
17.4 Aplikační **okno a typografie** 381
17.4.1 Teoretický základ: okno a písmo 382
17.4.2 Úprava inicializace v modulu **`app_window`** 382
17.4.3 Úprava funkce **`_configure_main_window()`** 383
17.5 Úprava správce grafiky v modulu **`game_manager`** 384
17.5.1 Teoretický základ: nástroje knihoven Pillow a CustomTkinter 384
Modul **`PIL.Image`** a metoda **`Image.open()`** 385
Třída **`io.BytesIO`** 385
Metoda **`resize()`** a vyhlazovací filtr **`LANCZOS`** 385
Třídy **`ImageTk.PhotoImage`** a **`ctk.CTkImage`** 385
17.5.2 **Importy a globální proměnné** 386
17.5.3 Funkce **`_setup_graphics()`** 386
17.5.4 **Pomocná funkce pro dohledání a otevření souboru** 387
17.5.5 Poskytování grafiky pro plátno a **`ctk`** **tlačítka** 388
Funkce **`get_image()`** 388
Funkce **`get_ctk_image()`** 389
17.6 **Předání mezních rozměrů v modulu `map_window`** 390
17.6.1 **Prvotní načtení výchozí mapy** 390
17.6.2 **Dynamická výměna mapy za běhu** 391
17.7 **Úprava postranních panelů v modulu `panels`** 392
17.7.1 **Teorie: posuvná příčka, rolovací rámec a `ctk` tlačítka** 393
`PanedWindow(master=None, **options)` 393
`PanedWindow.add(child, **options)` 393
`CTkScrollableFrame(master, **options)` 393
`CTkButton(master, options)` 394
`CTkButton.configure()` 394
17.7.2 **Uspořádání p**ravého panelu ve funkci **`initialize()`** 394
17.7.3 **Změna předka a initor třídy** 395
17.7.4 **Úprava tlačítek v metodě `update_content()`** 396
17.8 Úprava centrálního panelu 398
17.8.1 Hybridní pojetí centrálního panelu 398
17.8.2 Rozdíly v API widgetu **`CTkComboBox`** 398
17.8.3 Importy a globální deklarace 399
17.8.4 Aktualizace nabídky akcí 399

17.8.5 Úprava vstupního panelu ... 400
17.8.6 **Zpracování výstupu a zneplatnění vstupu** ... 401
17.9 **Konfigurační dialog a záložky** ... 402
17.9.1 Kontejner záložek **`CTkTabview`** a volba typu okna ... 402
17.9.2 Sestavení dialogu ve funkci **`show()`** ... 403
17.9.3 Ovládací prvky na kartách a hybridní kompromisy ... 404
17.9.4 Úprava jednotlivých karet nastavení ... 405
Karta **Vzhled** ... 405
Karta **Žurnál** ... 406
Karta **Informační panel** ... 406
Karta **Statistika** ... 407
17.10 **Oživení témat a initor balíčku** ... 409
17.10.1 **Teoretický základ: Globální řízení vzhledu** ... 409
17.10.2 **Souběžná změna velikosti písem v modulu `settings_manager`** ... 409
17.10.3 **Úprava initoru balíčku** ... 410
17.11 Kontrolní test ... 411
17.12 Shrnutí a soubory pro opakování ... 412
17.12.1 Doprovodné soubory ... 413

Literatura ... 415

Rejstřík ... 416

Část E **Přílohy a** seznamy 421

Příloha A API hry, pro niž **tvoříme** GUI ... 422

Seznam výpisů programů ... 424

Seznam obrázků ... 430

Seznam odboček – podšeděných bloků ... 431

Kapitola 17
Moderní kabát s CustomTkinter



Co se v kapitole naučíte
V závěrečné kapitole modernizujeme vzhled i funkční možnosti uživatelského rozhraní naší aplikace. Vyměníme dosavadní komponenty za modernější zaoblené prvky, vyřešíme posouvání tlačítek, která se nevejdou na panel, i podporu dalších grafických formátů. V kapitole se dozvíte:

- Jak nainstalovat a začlenit moderní nadstavbovou knihovnu *CustomTkinter*.
- Jak uplatnit hybridní model pro souběžné fungování původních komponent knihovny *Tkinter* s novými widgety knihovny *CustomTkinter* v témže okně.
- Jak zapojit grafickou knihovnu *Pillow*, rozšířit podporu o rastrové formáty JPG a WEBP a zajistit proporcionální přizpůsobení rozměrů map i ikon na tlačítkách.
- Jak zefektivnit rozvržení postranních panelů kombinací posuvné příčky `PanedWindow` a rolovacího rámce `CTkScrollableFrame` s využitím návrhového principu kompozice.
- Jak aktivovat podporu pro světlý a tmavý režim a jak zajistit plynulé vektorové škálování rozhraní pro displeje s vysokým rozlišením.

17.1 Příprava nové verze

Jako vždy si na začátku nové kapitoly nejprve připravíme čistý pracovní stůl. Vytvoříme kopii stávajícího balíčku `gui69.v16_canvas` a pojmenujeme ji `v17_custom`.

V této závěrečné verzi již nebudeme nijak rozšiřovat ani doplňovat aplikační rozhraní her, jak jsme to udělali v 6. a 7. kapitole. Naším cílem je výhradně technologická

a vizuální modernizace samotného grafického rozhraní – náhrada zastaralých komponent modernějšími prvky, definitivní vyřešení rolování v panelech a oživení volby barevného režimu (tématu), kterou jsme v nastavení připravili v předchozích kapitolách.

17.1.1 Instalace knihoven *CustomTkinter* a *Pillow*

Na rozdíl od standardní knihovny *Tkinter*, která je přímou součástí instalace jazyka *Python*, představuje *CustomTkinter* externí balíček vyvíjený komunitou. Před zahájením práce jej proto musíme do prostředí doinstalovat.

Ačkoliv *CustomTkinter* staví svou práci s grafikou na knihovně *Pillow*, závislost na této knihovně nebývá na všech platformách striktně vynucena, a proto je bezpečnější nainstalovat obě knihovny současně.

Otevřeme systémový terminál či příkazový řádek ve virtuálním prostředí našeho IDE a spustíme instalátor balíčků `pip`:

```
pip install customtkinter pillow
```

Úspěšnou instalaci si můžeme ověřit rychlým testem v interaktivním režimu *Pythonu*. Povšimněte si, že knihovnu *Pillow* importujeme pod názvem PIL. Jde o označení převzaté z původní knihovny PIL (*Python Imaging Library*), jejímž je *Pillow* moderním a plně kompatibilním nástupcem:

```
import customtkinter as ctk
import PIL
print(f"CustomTkinter: {ctk.__version__}\n"
      f"Pillow:        {PIL.__version__}")
```

Vypíše-li interpret čísla verzí bez chybového hlášení, máme vše potřebné připraveno.

17.1.2 Architektonická stabilita při přechodu

Přechod na novou grafickou knihovnu bývá v mnoha projektech vnímán jako noční můra, která si vynutí kompletní přepsání celé aplikace. V našem případě však naplno zúročíme úsilí, jež jsme od první kapitoly věnovali čistému architektonickému návrhu.

Protože jsme striktně oddělili model hry od zobrazení a komunikaci svěřili prostředníkům a přepravkám, zůstane naprostá většina kódu zcela netknuta:

- **Nezávislí správci a pomocná okna:** Moduly `journal_manager`, `info_panel`, `level_dialog`, `journal_window` i `tooltip` zůstávají zcela netknuté, neboť jejich zodpovědnost nesouvisí s modernizovanými widgety.
- **Správce grafických zdrojů:** Modul `game_manager` sice nemění své základní poslání, ale integruje knihovnu *Pillow* a stává se klíčovým grafickým adaptérem; zobrazovací vrstvu tím zcela odstiňuje od přímé práce se souborovým systémem a novými formáty.

- **Zobrazovací vrstva:** Úpravy se soustředí výhradně na moduly tvořící vzhled a rozvržení (`app_window`, `panels`, `central_panel`, `settings_window`, `map_window` a řídicí `__init__`), kde nasazujeme prvky *CustomTkinter* nebo optimalizujeme rozvržení kontejnerů.

Ukážeme si navíc, že přechod nemusí být radikální ve stylu „všechno, nebo nic“. Knihovna *CustomTkinter* je vystavěna přímo nad knihovnou *Tkinter*, což nám umožní ponechat funkční části (například kreslicí plátno s mapou v modulu `map_window` nebo systémovou nabídku v modulu `menu`) v jejich stávající podobě a modernizovat pouze ty celky, kde to přináší okamžitý užitek.

17.2 Proč CustomTkinter a co přináší nového

Knihovna *Tkinter* slouží spolehlivě již desítky let. Její věk se však začíná projevovat na vizuální stránce. Prvky vypadají na různých operačních systémech odlišně a na monitorech s vysokým rozlišením (displeje High-DPI) bývají texty i obrysy neostré. Řešení stylů prostřednictvím modulu `tkinter.ttk` je navíc poměrně těžkopádné.

Knihovna *CustomTkinter* tento problém řeší. Nejedná se o zcela nový framework, nýbrž o důmyslnou nadstavbu. Knihovna *CustomTkinter* totiž vykresluje vlastní prvky (tlačítka, rámy, vstupní pole) vektorově pomocí základního kreslicího plátna (widget `Canvas`). Díky tomu mají widgety jednotný vzhled na všech platformách a zůstávají absolutně ostré i na monitorech s vysokou hustotou pixelů.

Přechod na plochý styl však nemusí vyhovovat každému. Klasický *Tkinter* využíval komponenty s prostorovým stínováním a plastickými okraji, které uživateli jasně signalizovaly, na co lze kliknout a jaká plocha náleží ke kterému prvku. Plochý styl knihovny *CustomTkinter* často obětuje tuto okamžitou rozpoznatelnost na úkor vizuálního minimalismu.

Typickým příkladem jsou právě záložky: namísto tradičních kartotéčních desek s jasně ohraničenými oušky sází *CustomTkinter* na plochá přepínací tlačítka v záhlaví. Pro některé uživatele tak může být rozhraní paradoxně méně přehledné než původní klasická verze. Vizuální změna tedy nemusí po ergonomické stránce znamenat jednoznačný krok vpřed.

17.2.1 Témata a barevné režimy

Nejvýraznějším benefitem pro uživatele je integrovaná podpora pro světlý a tmavý režim (anglicky *Dark Mode*). V čisté knihovně *Tkinter* představuje přebarvení celé aplikace zdlouhavý a chybami zatížený proces, při němž je nutné explicitně překonfigurovat barvu pozadí i popředí u každého jednotlivého widgetu.

Knihovna *CustomTkinter* naproti tomu nabízí globální funkci `set_appearance_mode()`. Přijme-li tato funkce string `"light"`, `"dark"` nebo `"system"`, automaticky přebarví veškeré své widgety v aplikaci podle definovaného klíče.

Zde naplno zúročíme prozřetelnost z dvanácté kapitoly, v níž jsme do konfiguračního dialogu a správce nastavení zakomponovali přepínač nesoucí název `theme` přesně s těmito třemi volitelnými hodnotami.

V hybridním modelu musíme počítat s tím, že funkce `set_appearance_mode()` automaticky přebarví výhradně `ctk` widgety. Původní `tk` prvky (například textové pole v centrálním panelu či seznam ve statistice) na tuto funkci nereagují a zachovávají si svůj světlý podklad.

To však nepředstavuje nedostatek: v moderních aplikacích (například v textových editorech či vývojových prostředích) je kombinace tmavého pracovního prostředí se světlou plochou dokumentu běžným standardem, jenž navozuje přirozenou metaforu čistého listu papíru položeného na tmavé pracovní desce stolu.

17.2.2 Práce s grafikou a podpora formátů

V kapitole *15 Obrázky na tlačítkách a správa paměti* na straně 338 jsme narazili na výrazný limit třídy `tk.PhotoImage`, která nativně podporuje pouze formáty PNG a GIF. V podšeděném bloku jsem tehdy avizoval, že přechod na nadstavbovou knihovnu otevře cestu k dalším formátům (například JPG).

Tento příslib plní knihovna *CustomTkinter* zavedením vlastní třídy `ctk.CTkImage`. Tato třída vznikla jako obal nad populární externí knihovnou *Pillow* (`PIL.Image`), která se při instalaci balíčku `customtkinter` automaticky stáhne jako závislost.

Nasazení třídy `ctk.CTkImage` přináší tři zásadní výhody:

- **Široké spektrum formátů:** Díky knihovně *Pillow* lze v aplikaci bez dalších úprav načítat soubory formátů JPG, BMP, WEBP i řady dalších.
- **Přizpůsobení měřítku:** Obrázek předaný přes `ctk.CTkImage` se na monitorech High-DPI automaticky a ostře škáluje v souladu s nastaveným měřítkem rozhraní.
- **Podpora světlého a tmavého režimu:** Konstruktor třídy umožňuje předat v argumentech dva různé podklady: `light_image` a `dark_image`. Při přepnutí tématu aplikace automaticky vymění světlou ikonu za tmavou a naopak.

V našem projektu třídu `ctk.CTkImage` plnohodnotně nasadíme na všechna tlačítka v postranních panelech. Společně s knihovnou *Pillow* tím otevřeme aplikaci pro rozsáhlejší grafické podklady v komprimovaných formátech JPG či WEBP a zajistíme jejich ostré a plynulé zobrazení bez ohledu na rozlišení monitoru.

17.2.3 Hybridní model a kompatibilita

Jak již bylo zmíněno, knihovna *CustomTkinter* je vystavěna přímo nad knihovnou *Tkinter*. Z toho plyne obrovská výhoda: obě knihovny lze v jedné aplikaci bez problémů kombinovat.

Naprostá většina moderních prvků nese předponu `CTk` (např. `CTkButton` namísto `Button`, `CTkFrame` namísto `Frame`) a zachovává si při konstrukci téměř identické aplikační rozhraní, jaké známe u původních widgetů. Parametry jako `command`, `text` nebo správa rozmístění přes `pack()` a `grid()` fungují identicky.

Narazíme-li na prvek, jenž v knihovně *CustomTkinter* dosud chybí nebo jehož náhrada nedává z technického hlediska smysl, jednoduše použijeme původní widget z modulu `tkinter`. V naší aplikaci tak v původní podobě vědomě ponecháme:

- `tk.Menu` – systémovou lištu hlavní nabídky, kterou *CustomTkinter* vůbec nenabízí;
- `tk.Text` s posuvníkem `tk.Scrollbar` – výstupní plochu historie v centrálním panelu, kde využíváme barevné formátovací značky (tagy), jež `ctk` komponenta `CTkTextbox` nepodporuje;
- `tk.Canvas` – kreslicí plátno interaktivního plánku umožňující přesnou manipulaci se souřadnicemi prvků přes metodu `coords()`;
- `tk.PanedWindow` – posuvnou vodorovnou příčku pravého sloupce, pro niž v *CustomTkinter* neexistuje přímý ekvivalent;
- `ttk.Spinbox` a `tk.Listbox` – specializované prvky v konfiguračním dialogu, které nadstavba v základní nabídce nemá;
- `tk.Toplevel` – osvědčené sekundární okno pro konfigurační dialog, které zajišťuje stabilní modální uzamčení (`grab_set()`) bez konfliktů se správcem oken.

Naproti tomu prvky z modulu `ttk` (typicky `ttk.Combobox`), které nedokážou přirozeně a bez složitých zásahů do stylů reagovat na změnu barevného tématu, nahradíme jejich modernějšími ekvivalenty.

Prvky obou knihoven mohou vedle sebe bez konfliktu koexistovat v témže aplikačním okně.



Abych se v dalším textu vyhnul neustálému opakování názvů knihoven, zavedu jednotné přívlastky:

tk bude označovat komponenty z původní knihovny *Tkinter* i jejího doplňkového modulu `ttk` (`tk` widget, `tk` tlačítko, `tk` okno, …).

ctk bude označovat komponenty z nadstavbové knihovny *CustomTkinter* (`ctk` widget, `ctk` tlačítko, `ctk` okno, …).

17.3 Rozšíření sdílených konstant

Dříve než se pustíme do úprav jednotlivých oken a zobrazovacích komponent, připravíme si konfigurační hodnoty v modulu `constants`.

První úprava se týká konfiguračního dialogu nastavení. Původní rozměr `"450x250"` z pasáže *12.4 Rozšíření modulu constants* na straně 272 by pro komponenty knihovny *CustomTkinter* nestačil. Kritická je zejména výška: moderní kontejner záložek disponuje vyšším přepínacím záhlavím a widgety mají větší svislé rozestupy, takže v původních `250` pixelech by ovládací prvky i spodní tlačítko pro zavření přetekly pod spodní okraj okna.

Druhým doplněním je konstanta `BUTTON_ICON_SIZE`. V patnácté kapitole se ikony zobrazovaly v nativní velikosti souboru PNG. Nyní však otevíráme aplikaci pro libovolné formáty přes knihovnu *Pillow*, takže autor hry může k předmětům dodat i rozměrné fotografie ve formátech JPG či WEBP.

Aby takový obrázek neroztáhl postranní panel přes celou obrazovku, zavedeme pro ikony na tlačítkách standardní logický rozměr 48 × 48 pixelů.

Upravený počátek modulu `constants` ukazuje výpis 17.1.

Výpis 17.1: *Úprava modulu `gui69.v17_custom.constants`*

```
1  # Geometrie oken (šířka x výška + x_offset + y_offset)
2  MAIN_GEOMETRY:     str = "770x350+100+100"
3  LEVEL_GEOMETRY:    str = "250x240"
4  JOURNAL_GEOMETRY:  str = "500x400"
5  SETTINGS_GEOMETRY: str = "500x380"
6
7  # Výchozí rozměry ikon předmětů a prostorů na tlačítkách
8  BUTTON_ICON_SIZE: tuple[int, int] = (48, 48)
```

Na řádku 5 je hodnota konstanty `SETTINGS_GEOMETRY` upravena na `"500x380"`. Tato plocha spolehlivě pojme kontejner se záložkami i ovládací tlačítko ve spodní liště.

Na řádku 8 přibyla konstanta `BUTTON_ICON_SIZE` definující dvoudílnou n-tici s požadovanou šířkou a výškou ikony v pixelech. Tuto hodnotu využije správce grafiky při vytváření instancí třídy `ctk.CTkImage`.

17.4 Aplikační okno a typografie

Refaktoraci zahájíme u kořenového aplikačního okna v modulu `app_window`, jemuž propůjčíme modernější základ. Než se však pustíme do úpravy samotného kódu, seznámíme se s novými třídami, které původní objekty nahrazují, a vysvětlíme si, proč musíme k typografii přistoupit takzvanou hybridní cestou.

17.4.1 Teoretický základ: okno a písmo

Základním stavebním kamenem upravené aplikace je třída `ctk.CTk`, jež plně nahrazuje původní `tk.Tk`. Automaticky se stará o ostré vykreslování na monitorech s vysokým rozlišením a nativně podporuje globální změnu barevného režimu (světlý či tmavý).

Při konfiguraci hlavního okna však narazíme na výzvu spojenou s typografií. Jak již bylo zmíněno, v naší aplikaci volíme hybridní přístup. Vedle sebe tak koexistují prvky z knihovny *CustomTkinter* a některé původní widgety knihovny *Tkinter* (například systémová lišta nabídek či naformátovaný textový výstup historie v centrálním panelu). Tyto dva odlišné světy však přistupují k definici a distribuci písma rozdílně.

Zatímco widgety knihovny *Tkinter* spoléhají na pojmenované šablony a globální databázi vlastností (doplňovanou metodou `option_add()`), komponenty knihovny *CustomTkinter* vyžadují přímé předání objektu speciální třídy `ctk.CTkFont`.

Třída `ctk.CTkFont` se od klasické třídy `tkfont.Font` (tu jsme probírali v pasáži *4.4.4 Pojmenovaná písma – šablony písma* na straně 104) liší především tím, že se nespoléhá na vnitřní databázi interpretu *Tcl/Tk*. Z toho plynou dvě zásadní změny v jejím konstruktoru:

- Nevyžaduje rodičovský widget (parametr `master`): Objekt `ctk` písma existuje samostatně a není pevně svázán s jedním konkrétním kořenovým oknem.
- *Neexistuje parametr `name`:* Autor knihovny *CustomTkinter* se rozhodl upustit od textového pojmenovávání písem, na které spoléhal interpret *Tcl/Tk*, a zvolil čistě objektový přístup. Objekt písma tak nedostává žádný vnitřní textový identifikátor. `Ctk` widgetům se do parametru font předává přímo odkaz na vytvořenou instanci (například uloženou v atributu okna).

Z architektonického hlediska se proto jako nejčistší řešení jeví paralelní existence obou přístupů. Původní pojmenovanou šablonu písma s názvem v `c.APP_FONT` nezrušíme, nýbrž ji ponecháme pro `tk` komponenty a pouze vedle ní vytvoříme objekt `ctk` písma pro komponenty nové.

17.4.2 Úprava inicializace v modulu `app_window`

Pojďme nyní přejít k aplikaci právě probraného v modulu `app_window`. Nejprve přidáme import nové knihovny dle výpisu 17.2.

Výpis 17.2: *Rozšíření importů v modulu `gui69.v17_custom.app_window`*

```
1 import tkinter          as     tk
2 import customtkinter    as     ctk
3 import tkinter.font     as     tkfont
4 from   collections.abc  import Callable
5 from   .                import constants     as c
6 from   .                import central_panel as cp
7 from   .                import info_panel    as ip
8 from   .                import panels, menu
9 from   .                import SidePanelsAux, MenuAux
```

Následuje úprava funkce `initialize()`, která je tak jednoduchá, že to skoro nestojí za řeč. Prostě vyměníme třídu, jejíž instancí bude hlavní okno. Místo instance třídy `Tk` z modulu `tkinter` vytvoříme instanci třídy `CTk` z modulu `customtkinter`, neboli místo výrazu `tk.Tk` zadáme `ctk.CTk`.

Popsanou úpravu si můžete prohlédnout ve výpisu 17.3.

Výpis 17.3: *Úprava funkce `initialize()` a vytvoření kořenového okna v modulu `gui69.v17_custom.app_window`*

```
1  def initialize(cmd_process:  Callable[[str], None],
2                 side_aux:     SidePanelsAux,
3                 menu_aux:     MenuAux
4                ) -> tuple[ctk.CTk, Callable[[str, str, bool], None],
5                                    Callable[[], None], Callable[[], None],
6                                    Callable[[tuple[str, ...]], None]]:
7      """Vytvoří hlavní okno a základní rozvržení aplikace.
8      """
9      global root
10     root = ctk.CTk()        # Vyměnili jsme mateřskou třídu aplikačního okna
11     _configure_main_window()
12
13     # Pokračování výpisu je totožné s výpisem 13.4 na straně 310
```

Použití nové knihovny se promítlo přímo do typové anotace v hlavičce funkce – návratový typ tvoří namísto `tk.Tk` nově n-tice s instancí `ctk.CTk`. Tuto změnu kontraktu následně přebírá řídicí initor v modulu `__init__`, jehož funkce `run()` vrací tentýž typ.

Klíčový je ale příkaz na řádku **10**, jenž vytváří kořenové okno.

Výpis jsem zkrátil nejenom závěrečným odkazem na definici zbytku těla funkce v jiném výpisu, ale také zestručněním dokumentačního komentáře. V doprovodných programech je samozřejmě plná verze bez odkazů na jiné definice.

17.4.3 Úprava funkce `_configure_main_window()`

Nyní přistoupíme k úpravě funkce `_configure_main_window()`, v níž založíme zmíněnou paralelní existenci dvou objektů písma.

Původní jmennou šablonu písma s názvem uloženým v konstantě `c.APP_FONT` proto nerušíme, nýbrž ji ponecháme pro staré komponenty a pouze vedle ní vytvoříme `ctk` objekt písma. Kód funkce `_configure_main_window()` obstarávající tuto paralelu ukazuje výpis 17.4.

Jak výpis naznačuje, začátek funkce se nemění. Oproti předchozí verzi se mění jenom závěrečné příkazy.

Definice klasického fontu s názvem ukládaným do konstanty `c.APP_FONT` zůstala (viz řádky **9-11**). Pouze ochranný odkaz na objekt `tkfont.Font` byl přesunut do nového atributu kořenového okna pojmenovaného `legacy_font`, abychom atribut uvolnili pro instanci `ctk` písma typu `ctk.CTkFont` vytvořenou na řádku **14**.

Zatímco klasické prvky si automaticky přečtou původní font z globální databáze nastavené metodou `option_add()`, objekt `root.app_font` definuje výchozí rodinu písma (`"Helvetica"`).

Samotné `ctk` widgety si však velikost neřídí individuálním předáváním písma do jednotlivých tlačítek a polí, nýbrž se dynamicky přizpůsobují globálnímu měřítku rozhraní přes `ctk.set_widget_scaling()`, jak si ukážeme v závěru kapitoly.

Výpis 17.4: *Upravená definice funkce `_configure_main_window()` v modulu `gui69.v17_custom.app_window`*

```
def _configure_main_window():
    """Nastaví titulek i výchozí geometrii okna a připraví mřížku.
    Vytvoří a uchová klasické i ctk písmo pro hybridní běh.
    """
    # Začátek těla je shodný s výpisem 6.6 na straně 150

    # Definice centrálního pojmenovaného písma (šablony) c.APP_FONT
    # Odkaz uložíme do atributu okna, abychom jej chránili před smazáním
    root.legacy_font = tkfont.Font(root, name=c.APP_FONT,
                                   family="Helvetica", size=c.DEFAULT_FONT_SIZE)
    root.option_add("*Font", c.APP_FONT)  # Výchozí nastavení pro celou aplikaci

    # Definice ctk písma pro prvky knihovny CustomTkinter
    root.app_font = ctk.CTkFont(family="Helvetica", size=c.DEFAULT_FONT_SIZE)
```

17.5 Úprava správce grafiky v modulu game_manager

V patnácté kapitole jsme zavedli modul `game_manager` jako výhradního správce obrázků, aby zobrazovací moduly nemusely komunikovat přímo se souborovým systémem. Tehdejší řešení však mělo dvě omezení daná možnostmi třídy `tk.PhotoImage`: akceptovalo pouze formát PNG a neumělo plynule přizpůsobovat velikost rozměrnějším předlohám.

V této podkapitole modul `game_manager` rozšíříme o spolupráci s knihovnou *Pillow*. Umožníme načítání dalších grafických formátů (zejména JPG pro plánky hry), vyřešíme automatické zmenšení velkých map pro potřeby kreslicího plátna a zavedeme funkci vracející `ctk` obrázky pro tlačítka v postranních panelech.

17.5.1 Teoretický základ: nástroje knihoven Pillow a CustomTkinter

Dříve než sáhneme do kódu správce, musíme si důkladně představit čtyři klíčové komponenty, které nám zpracování grafiky zprostředkují.

Modul `PIL.Image` a metoda `Image.open()`

Základním stavebním kamenem knihovny *Pillow* je modul `Image`. Představuje reprezentaci rastrového obrázku přímo v operační paměti nezávisle na konkrétním grafickém rozhraní.

K načtení slouží metoda `Image.open(fp)`. Jejím hlavním přínosem je automatická detekce formátu: metoda sama analyzuje hlavičku binárních dat a bez ohledu na příponu souboru rozpozná, zda jde o PNG, JPG, WEBP, GIF či BMP. Není tedy nutné psát specializovaný kód pro každý formát zvlášť.

Třída `io.BytesIO`

Ve čtrnácté kapitole jsme vyřešili bezpečné načítání datových souborů z balíčků pomocí rozhraní `importlib.resources`. Metoda `read_bytes()` vrací načtený obsah jako surovou sekvenci bajtů (typ `bytes`).

Metoda `Image.open()` však v parametru `fp` očekává souborový objekt (*stream*), ze kterého lze číst metodami jako `read()` nebo `seek()`. Zde přichází ke slovu třída `BytesIO` ze standardního modulu `io`.

Tato třída obalí surová bajtová data v paměti a propůjčí jim chování otevřeného diskového souboru. Můžeme tak načíst libovolný grafický formát přímo z paměti, aniž bychom museli vytvářet dočasné soubory na disku.

Metoda `resize()` a vyhlazovací filtr `LANCZOS`

Zatímco klasický *Tkinter* disponoval pouze celočíselnými operacemi `zoom()` a `subsample()`, knihovna *Pillow* nabízí plnohodnotný matematický resampling pomocí metody:

```
pil_image.resize(size, resample)
```

V parametru `size` přijímá požadovanou dvoudílnou n-tici `(šířka, výška)` v pixelech. Klíčový je však parametr `resample`. V něm předáme vyhlazovací filtr `Image.Resampling.LANCZOS`. Tento algoritmus (založený na sinc interpolaci) přepočítává hodnoty okolních pixelů tak, aby při zmenšení rozsáhlé mapy (např. z rozlišení 1920 × 1080 na 640 × 480) nedošlo ke vzniku zubatých hran a drobné detaily zůstaly čitelné a ostré.

Třídy `ImageTk.PhotoImage` a `ctk.CTkImage`

Na výstupu z modulu `game_manager` musíme uspokojit dva odlišné konzumenty:

- **Kreslicí plátno (`tk.Canvas`) v modulu `map_window`:**
  Kreslicí plátno je `tk` widget, a vyžaduje proto objekt kompatibilní s původním rozhraním. K tomuto účelu slouží třída `PIL.ImageTk.PhotoImage`. Přebírá v argumentu instanci `Image` z knihovny *Pillow* a vytvoří z ní objekt, který plátno bez zaváhání vykreslí.
- **Tlačítka (`ctk.CTkButton`) v modulu `panels`:**
  Tlačítka vyžadují třídu `ctk.CTkImage`. Její konstruktor přijímá v argumentu `light_image` instanci z knihovny *Pillow* a v argumentu `size` logické rozměry tlačítkové ikony (např. `(48, 48)`). Právě tento objekt zajistí vektorové škálování ikony na High-DPI displejích a odstraní dřívější varování v konzoli.

17.5.2 Importy a globální proměnné

Do modulu `game_manager` nejprve doplníme potřebné importy, konkrétně modul `io` pro práci s paměťovými proudy bajtů a třídy `Image` a `ImageTk` z balíčku `PIL` pro načítání obrázků a knihovnu `customtkinter`.

Ke globálním proměnným přidáme druhou vyrovnávací paměť `_ctk_image_cache`, do níž budeme ukládat vytvořené instance `ctk.CTkImage` jako ochranu před správcem paměti. Původní slovník `_image_cache` ale ponecháme, avšak jeho typovou nápovědu upravíme na `ImageTk.PhotoImage`.

Kompletní podobu nového úvodu souboru s importy a deklaracemi globálních proměnných ukazuje výpis 17.5.

Výpis 17.5: *Úprava importů a globálních proměnných v modulu `gui69.v17_custom.game_manager`*

```
1  import importlib
2  import importlib.resources  as     ir
3  import io
4  import tkinter              as     tk
5  import customtkinter        as     ctk
6  from   pathlib              import Path
7  from   tkinter              import filedialog as fd
8  from   PIL                  import Image, ImageTk
9  from   game69.api.protocols import IPortal
10 from   .                    import constants as c
11
12
13 # --- Vnitřní stav modulu ---
14 current_portal:   IPortal | None     # Portál aktuálně hrané hry
15 _image_cache:     dict[str, ImageTk.PhotoImage | None] = {}  # Pro tk.Canvas
16 _ctk_image_cache: dict[str, ctk.CTkImage | None]       = {}  # Pro ctk tlačítka
17 _image_aliases:   dict[str, str] = {}                       # Přezdívky obrázků
18 _place_coords:    dict[str, tuple[int, int, int]] = {}     # Souřadnice prostorů
```

Na řádcích 3, 5 a 8 přibyly importy pro práci s proudem bajtů, nadstavbou *CustomTkinter* a knihovnou *Pillow*.

Na řádku 15 se typová nápověda původního slovníku `_image_cache` změnila na `ImageTk.PhotoImage`. Na řádku 16 vzniklo nové úložiště `_ctk_image_cache` vyhrazené pro `ctk` komponenty.

17.5.3 Funkce `_setup_graphics()`

První upravovanou funkcí v souboru je neveřejná funkce `_setup_graphics()`. Tato funkce se volá při každém spuštění nové hry. Doplníme do ní vymazání nově zavedené mezipaměti `_ctk_image_cache`, aby v paměti nezůstávaly viset ikony předchozí partie. Kód upravené funkce ukazuje výpis 17.6.

Výpis 17.6: Úprava funkce `_setup_graphics()` v modulu `gui69.v17_custom.game_manager`

```
1  def _setup_graphics() -> None:
2      """Vyčistí mezipaměti obrázků a načte případné přezdívky obrázků
3      a souřadnice lokalizačních bodů prostorů pro aktuální hru.
4      """
5      _image_cache.clear()
6      _ctk_image_cache.clear()
7      _image_aliases.clear()
8      _place_coords.clear()
9
10     # Zbytek je totožný s definicí ve výpisu 16.3 na straně 360.
```

Na řádku 6 přibylo volání `_ctk_image_cache.clear()`. Při přepnutí hry se tak obě úložiště spolehlivě vyprázdní ještě předtím, než se začnou načítat grafická data nového světa.

Ostatní řádky načítající aliasy a souřadnice prostorů zůstávají v původním znění z výpisu 16.3 na straně 360.

17.5.4 Pomocná funkce pro dohledání a otevření souboru

Abychom neopakovali kód pro procházení přípon na dvou různých místech, vytvoříme neveřejnou pomocnou funkci `_load_pil_image()`.

Tato funkce vezme název herního objektu, zohlední případný alias ze slovníku `_image_aliases` a postupně prověří přítomnost souborů s příponami `.png`, `.jpg`, `.jpeg` a `.webp`. Narazí-li na existující soubor, přečte jeho bajty, obalí je třídou `BytesIO` a otevře metodou `Image.open()`. Kód funkce ukazuje výpis 17.7.

Výpis 17.7: Pomocná funkce `_load_pil_image()` v modulu `gui69.v17_custom.game_manager`

```
1  def _load_pil_image(item_name: str) -> Image.Image | None:
2      """Pokusí se dohledat a načíst obrázek zadaného prvku přes Pillow.
3      Hledá soubory s příponami .png, .jpg, .jpeg a .webp.
4      """
5      base_name = _image_aliases.get(item_name, item_name)
6      data_pkg  = f"{current_portal.__name__}.data"
7
8      for ext in (".png", ".jpg", ".jpeg", ".webp"):
9          filename = base_name + ext
10         try:
11             img_path = ir.files(data_pkg) / filename
12             if img_path.is_file():
13                 raw_data = img_path.read_bytes()
14                 return Image.open(io.BytesIO(raw_data))
15         except Exception:
16             pass
17
18     return None
```

Na řádku 5 se zjišťuje případné alternativní jméno souboru.

Cyklus na řádcích 8–16 postupně testuje jednotlivé podporované formáty. Na řádku 13 se načtou surová binární data a na řádku 14 se zabalená do instance `io.BytesIO` předají metodě `Image.open()`. Nenajde-li se žádný vyhovující soubor, vrátí funkce na řádku 18 hodnotu `None`.

17.5.5 Poskytování grafiky pro plátno a `ctk` tlačítka

Nyní zbývá upravit stávající funkci `get_image()` a nadefinovat novou funkci `get_ctk_image()`.

Funkce `get_image()`

V předchozích kapitolách přebírala funkce `get_image(item_name)` pouze název položky a obrázek načetla v jeho původním rozlišení. Spoléhalo se na to, že autor hry dodal soubory PNG připravené přímo na míru.

S přechodem na knihovnu *Pillow* však otevíráme aplikaci pro formáty jako JPG či WEBP, u nichž mohou mít předlohy (zejména rozsáhlé mapy) zcela libovolné rozlišení i odlišný poměr stran. Aby správce grafiky zůstal univerzální a nemusel v sobě obsahovat natvrdo zapsané rozměry kreslicího plátna z jiného modulu, rozšíříme jeho signaturu o nepovinný parametr `max_size: tuple[int, int] | None = None`.

Není-li tento parametr zadán, funkce vrátí obrázek v jeho originální velikosti (to využijeme při načítání ikony okna a postavičky hráče). Obdrží-li naopak v argumentu mezní rozměry, spočítá měřítko tak, aby se obrázek právě vešel do zadaného obdélníku (buď se zvětší, nebo zmenší) se zachováním poměru stran, a metodou `resize()` jej proporcionálně přizpůsobí s filtrem `LANCZOS`. Tím je zaručeno, že se plánek do okna spolehlivě vejde, aniž by došlo k jeho geometrické deformaci.

Se zavedením volitelného parametru `max_size` však vyvstává potřeba upravit ukládání do mezipaměti `_image_cache`. Kdyby totiž klíčem zůstal pouhý název položky `item_name`, požadavek na zmenšenou mapu by v paměti přepsal dříve načtený originál (nebo naopak: po prvotním načtení originálu by následný požadavek na zmenšení vrátil z mezipaměti již uložený nezmenšený obrázek).

Vytvoříme proto složený klíč spojující jméno předmětu s požadovaným rozměrem, aby každá velikostní varianta téhož obrázku získala v mezipaměti svůj vlastní nezávislý záznam.

Jak je celá funkce pro načítání a škálování implementována, najdete ve výpisu 17.8.

Podíváte-li se do výpisu na to, co je zarámováno jako změněné, vidíte, že z minulé verze zbyl netknutý jen závěrečný příkaz.

V hlavičce funkce na řádcích 1–3 přibyl volitelný parametr `max_size`.

Na řádku 7 se na základě přítomnosti parametru `max_size` sestavuje onen avizovaný složený klíč mezipaměti `cache_key`.

Podmínka na řádku 13 prověří přítomnost parametru `max_size`. Je-li zadán, výpočet na řádcích 14–17 určí koeficient zmenšení či zvětšení pro obě osy a stanoví nové celočíselné rozměry.

Liší-li se vypočtené rozměry od originálu (test na řádku 18), metoda `resize()` na řádcích 19–20 obrázek vyhlazeně přepočítá.

Na řádku 22 vzniká instance `ImageTk.PhotoImage`, která je na řádku 23 uložena do mezipaměti a vzápětí vrácena volajícímu.

Výpis 17.8: *Funkce `get_image()` v modulu `gui69.v17_custom.game_manager`*

```
1  def get_image(item_name: str,
2                max_size: tuple[int, int] | None = None
3               ) -> ImageTk.PhotoImage | None:
4      """Vrátí PhotoImage kompatibilní s kreslicím plátnem. Je-li zadán
5      argument max_size, upraví rozměry se zachováním poměru stran.
6      """
7      cache_key = f"{item_name}_{max_size}" if max_size else item_name
8      if cache_key in _image_cache:
9          return _image_cache[cache_key]  # Obrázek požadovaných rozměrů už máme
10
11     pil_img = _load_pil_image(item_name)
12     if pil_img:
13         if max_size:
14             scale = min(max_size[0] / pil_img.width,
15                         max_size[1] / pil_img.height)
16             new_w = max(1, int(pil_img.width  * scale))
17             new_h = max(1, int(pil_img.height * scale))
18             if (new_w, new_h) != pil_img.size:
19                 pil_img = pil_img.resize((new_w, new_h),
20                                          Image.Resampling.LANCZOS)
21
22         img = ImageTk.PhotoImage(pil_img)
23         _image_cache[cache_key] = img
24         return img
25
26     _image_cache[cache_key] = None
27     return None
```

Funkce `get_ctk_image()`

Nová funkce `get_ctk_image()` je určena pro tlačítka. Načtený podklad zabalí do instance `ctk.CTkImage` s požadovanou velikostí. Její kód si můžete prohlédnout ve výpisu 17.9.

Funkce vytváří na řádku 10 instance `ctk.CTkImage`. V argumentu `light_image` předává načtený obrázek a v argumentu `size` hodnotu konstanty `c.BUTTON_ICON_SIZE`. Tím je zaručeno, že ikony tlačítek budou mít jednotný rozměr bez ohledu na fyzické rozlišení zdrojového souboru na disku.

Získaný objekt je na řádku 11 uložen do `_ctk_image_cache` a následně vrácen volajícímu.

Nebyl-li obrázek se zadaným názvem načten, je na řádku 14 tato informace uložena a volajícímu je vrácen prázdný odkaz. Tlačítko tak bude bez obrázku.

Výpis 17.9: *Funkce `get_ctk_image()` v modulu `gui69.v17_custom.game_manager`*

```
def get_ctk_image(item_name: str) -> ctk.CTkImage | None:
    """Vrátí ctk obrázek pro zadaný předmět. Není-li ještě načten,
    vytvoří jej z načtených dat knihovnou Pillow.
    """
    if item_name in _ctk_image_cache:
        return _ctk_image_cache[item_name]

    pil_img = _load_pil_image(item_name)
    if pil_img:
        ctk_img = ctk.CTkImage(light_image=pil_img, size=c.BUTTON_ICON_SIZE)
        _ctk_image_cache[item_name] = ctk_img
        return ctk_img

    _ctk_image_cache[item_name] = None
    return None
```

17.6 Předání mezních rozměrů v modulu `map_window`

Kreslicí plátno pro zobrazení plánku hry zůstává i v této verzi standardním widgetem `tk.Canvas` s pevnými rozměry danými konstantami `c.MAP_WIDTH` a `c.MAP_HEIGHT`. Aby však plánek dokázal využít novou schopnost správce `game_manager` a přizpůsobit i rozsáhlé mapy ve formátech JPG či WEBP, musíme při žádosti o obrazová data mapy tyto mezní rozměry explicitně předat.

Při přípravě grafických podkladů však musíme pamatovat na jednu zásadní podmínku konvence: grafický adaptér sice zajistí vyhlazené zmenšení či zvětšení libovolně rozměrného obrázku na plochu plátna, samotné souřadnice lokalizačních bodů ve slovníku `PLACE_COORDS` však nijak nepřepočítává.

Autor proto musí souřadnice středů prostorů i poloměry jejich tolerance ve svých datech definovat vždy přímo vůči cílovému souřadnicovému systému kreslicího plátna (v našem případě 640 × 480 pixelů), nikoliv vůči původnímu rozlišení zdrojového souboru.

Úprava se dotkne dvou samostatných operací: prvotního vytvoření scény a následné dynamické aktualizace mapy za běhu hry.

17.6.1 Prvotní načtení výchozí mapy

Při vytváření okna a prvotním naplnění scény se volá funkce `_initialize_content()`. V původní verzi z pasáže *16.6.2 Funkce `_initialize_content()`* na straně 363 se obrázek mapy načítal bez dalších parametrů, což vyhovovalo předem připravenému souboru PNG v přesném rozlišení.

Nyní volání `gm.get_image()` rozšíříme o argument `(c.MAP_WIDTH, c.MAP_HEIGHT)`. Správce grafiky tak zajistí proporcionální přizpůsobení podkladu ploše plátna.

Načítání ikony hráče naopak ponecháme bez tohoto argumentu, neboť postavička si má zachovat své původní rozměry.

Kód upravené funkce naleznete ve výpisu 17.10.

Výpis 17.10: Předání rozměrů plátna ve funkci `_initialize_content()` v modulu `gui69.v17_custom.map_window`

```
1  def _initialize_content() -> None:
2      """Načte výchozí obrázky mapy a hráče a vloží je na plátno.
3      """
4      global _map_id, _player_id
5      if _canvas is None:
6          return
7
8      map_img = gm.get_image(c.DEFAULT_MAP_NAME, (c.MAP_WIDTH, c.MAP_HEIGHT))
9      if map_img:
10         _map_id = _canvas.create_image(0, 0, image=map_img, anchor=tk.NW)
11
12     player_img = gm.get_image(c.PLAYER_NAME)
13     if player_img:
14         _player_id = _canvas.create_image(-100, -100, image=player_img)
```

Oproti výpisu 16.6 na straně 363 se změnil pouze příkaz na řádku 8, kde byl volané funkci předán druhý argument – n-tice `(c.MAP_WIDTH, c.MAP_HEIGHT)`. Správce grafiky díky tomu vrátí výchozí plánek přizpůsobený ploše plátna se zachováním poměru stran.

Na řádku 12 je obrázek hráče vyžádán bez zadání mezních rozměrů, takže jeho měřítko zůstává nezměněno.

17.6.2 Dynamická výměna mapy za běhu

Během hry se může stát, že hráč přejde do podzemí či jiné dimenze a herní logika v příznacích (`game.conditions()`) oznámí požadavek na jinou mapu. O výměnu pozadí na plátně se stará funkce `update_map()`.

Zde uplatníme stejný postup: při volání `gm.get_image()` s novým názvem mapy předáme jako druhý argument mezní rozměry plátna. Tím zaručíme, že i nově načtený podklad v libovolném rozlišení či formátu zapadne do vymezeného prostoru.

Upravenou funkci ukazuje výpis 17.11.

Oproti minulé verzi se opět změnil pouze jediný příkaz – získání obrazu na řádku 16, kde se nově vyžádaný obrázek z herních příznaků omezuje rozměry `(c.MAP_WIDTH, c.MAP_HEIGHT)`. Alternativní mapa ve formátu JPG či WEBP tak nepřeteče vymezené plátno ani v případě, že má podstatně vyšší rozlišení než původní grafika. Bude-li naopak menší, tak obrázek přiměřeně zvětší.

Zbytek těla funkce zůstává shodný s verzí z výpisu 16.7 na straně 365.

Výpis 17.11: *Předání rozměrů plátna ve funkci `update_map()` v modulu `gui69.v17_custom.map_window`*

```
def update_map(game: IGame) -> None:
    """Aktualizuje pozici hráče a případně vymění obrázek mapy,
    vyžaduje-li to aktuální stav hry.
    """
    global _current_map_name

    # Není-li plátno zobrazeno, nebo hra již skončila, není co aktualizovat
    if _canvas is None or not game.is_alive():
        return

    # 1. Výměna mapy na pozadí (je-li v příznacích vyžádána nová)
    flags = game.conditions()
    new_map_name = flags.get("map_name", c.DEFAULT_MAP_NAME)

    if new_map_name != _current_map_name:
        map_img = gm.get_image(new_map_name, (c.MAP_WIDTH, c.MAP_HEIGHT))
        if map_img:
            _canvas.itemconfig(_map_id, image=map_img)
            _current_map_name = new_map_name

    # 2. Přesun ikony hráče na aktuální pozici
    curr_place = game.world().current_place().name
    coords_dict = gm.get_place_coords()

    if curr_place in coords_dict:
        # Vyzvedneme x, y a toleranci (tu zde při prostém přesunu ignorujeme)
        x, y, _radius = coords_dict[curr_place]
        _canvas.coords(_player_id, x, y)
    else:
        # Souřadnice neznáme (např. tajná místnost), ikonu raději schováme
        _canvas.coords(_player_id, -100, -100)
```

17.7 Úprava postranních panelů v modulu `panels`

Modul `panels` ukrýval od patnácté kapitoly funkční dluh, a to hned ve dvou ohledech.

- Když se na tlačítka přidají obrázky, jejich výška výrazně naroste. Jakmile by se hráč ocitl v místnosti s větším množstvím předmětů, obsah kontejneru by přetekl mimo obrazovku a k některým tlačítkům by se nedalo dostat.
- Druhým neduhem bylo pevné rozdělení pravého sloupce v poměru 50 : 50, kvůli němuž poloprázdný panel **Sousedé** zbytečně plýtval místem na úkor zaplněného panelu **Předměty**.

Knihovna *CustomTkinter* tento problém řeší nasazením widgetu s nativní podporou rolování.

17.7.1 Teorie: posuvná příčka, rolovací rámec a `ctk` tlačítka

Než přistoupíme k úpravě samotného kódu postranních panelů, seznámíme se se třemi komponentami, které pro realizaci nového rozvržení využijeme.

`PanedWindow(master=None, **options)`
Třída `PanedWindow` z modulu `tkinter` představuje specializovaný kontejner umožňující rozdělit vymezený prostor mezi několik dceřiných panelů a oddělit je interaktivní dělicí příčkou (*sash*). Uživatel může tuto příčku myší uchopit a posouvat, čímž dynamicky mění poměr ploch vyhrazených jednotlivým panelům.

Mezi její nejdůležitější konfigurační parametry patří:

- **`master`** – Odkaz na rodičovský kontejner.
- **`orient`** – Určuje směr řazení dceřiných panelů. Konstanta `tk.HORIZONTAL` řadí panely vedle sebe (příčka je svislá a posouvá se do stran), zatímco konstanta `tk.VERTICAL` řadí panely svisle nad sebe. Dělicí příčka je v tomto případě fyzicky vodorovná, posouvá se však svisle nahoru a dolů.
- **`sashrelief`** – Styl reliéfu dělicí příčky (např. `tk.RAISED`, `tk.SOLID` či `tk.FLAT`). Zajišťuje, aby byla dělicí čára na ploše vizuálně zřetelná.
- **`sashwidth`** – Šířka (tloušťka) příčky v pixelech. Nastavení větší tloušťky usnadňuje její zasažení kurzorem myši.

Dceřiné panely se do kontejneru `PanedWindow` neumisťují prostřednictvím správců `pack()` ani `grid()`, nýbrž se do něj vkládají prostřednictvím metody:

`PanedWindow.add(child, **options)`
Zaregistruje zadaný widget pod správu dělicí příčky. Klíčovým doplňkovým parametrem je:

- **`minsize`** – Minimální rozměr panelu v pixelech (výška při svislé orientaci, šířka při vodorovné). Zabraňuje tomu, aby uživatel při tažení příčky panel zcela sbalil na nulovou velikost. Nastavením této meze zaručíme trvalou viditelnost záhlaví panelu s titulkem.

`CTkScrollableFrame(master, **options)`
Třída `CTkScrollableFrame` z knihovny *CustomTkinter* představuje rámec s integrovaným posuvníkem. Vnitřně kombinuje kreslicí plátno s posuvníkem a stará se o automatický přepočet rolovatelné oblasti při dynamickém přidávání a odebírání prvků. Od klasického rámce `LabelFrame` se liší těmito vlastnostmi:

- **`master`** – Odkaz na rodičovský kontejner.
- **`label_text`** – String s textem nadpisu v záhlaví panelu. Nahrazuje dřívější parametr `text`.
- **Vnitřní okraje:** Výchozí vnitřní odsazení spravuje widget vnitřně. Při volání initoru předka se nepředávají dříve používané argumenty `padx` ani `pady`, jež by u této třídy způsobily chybu.

Při použití metody `add()` je nutné pamatovat na zásadní omezení: **vkládaný widget musí být přímým potomkem daného `PanedWindow`** v hierarchii oken, jinak operace selže s chybou `TclError`.

Právě zde naráží rolovací rámec `CTkScrollableFrame`. Pod kapotou totiž skládá vnější rám, plátno a teprve v něm vnitřní rámec, takže pro `PanedWindow` nepředstavuje přímého potomka. Kdybychom z něj přímo dědili, metoda `add()` by jej odmítla. Z tohoto důvodu uplatníme princip kompozice: třídu `Panel` vytvoříme jako přímý rámec `tk.Frame` a rolovací kontejner `CTkScrollableFrame` do něj vložíme jako jeho obsah.

`CTkButton(master, options)`
Tlačítka v panelech nahradíme třídou `ctk.CTkButton`. Ta zachovává většinu základních parametrů původního widgetu `tk.Button` (jako `text`, `state` či `command`), vyžaduje však obezřetnost při dodatečné změně vlastností.

`CTkButton.configure()`
V klasické knihovně *Tkinter* bylo zvykem pro dodatečnou konfiguraci volat zkrácený alias `config()`. Knihovna *CustomTkinter* tento alias nepodporuje, a proto je nutné při vkládání načtené ikony volat výhradně plný název metody `configure(image=..., compound=tk.TOP)`.

17.7.2 Uspořádání pravého panelu ve funkci `initialize()`

Než se pustíme do úprav útrob modulu `panels`, doplníme mezi importy na začátku modulu import knihovny *CustomTkinter* pod standardní přezdívkou `ctk`.

Poté přistoupíme k úpravě funkce `initialize()`, která obstarává sestavení postranních panelů. Levý panel pro batoh ponecháme jako samostatný kontejner. Pravý sloupec `pan_right` vytvoříme jako instanci `tk.PanedWindow` a oba dceřiné panely do něj vložíme metodou `add()`.

Doplněné importy spolu s kódem upravené funkce `initialize()` si můžete prohlédnout ve výpisu 17.12.

Na řádku `2` je importován modul `customtkinter` pod standardní přezdívkou `ctk`.

Na řádku `15` vzniká levý panel `pan_bag` přímo jako potomek hlavního okna.

Na řádcích `18–19` je vytvořen kontejner `pan_right` jako instance `tk.PanedWindow`. Argument `orient=tk.VERTICAL` nařizuje řadit vložené panely nad sebe, argumenty `sashrelief=tk.RAISED` a `sashwidth=4` definují zřetelnou plastickou vodorovnou příčku.

Na řádcích `20–21` vznikají panely `pan_exits` a `pan_items`, jejichž rodičem je právě tento kontejner.

Na řádcích `22–23` jsou oba panely zaregistrovány do kontejneru voláním metody `add()`. Původní volání metody `pack()` s fixním parametrem `expand=True` tím zcela odpadá.

V těchto voláních jsou použity argumenty `minsize=40`, abychom zaručili, že záhlaví panelu s titulkem zůstane vždy viditelné i při maximálním posunu příčky.

Výpis 17.12: Doplnění importů a úprava funkce `initialize()` v modulu `gui69.v17_custom.panels`

```
1  import tkinter          as     tk
2  import customtkinter    as     ctk
3  from   collections.abc  import Callable
4  from   .                import game_manager as gm
5  from   .                import PanelAux, SidePanelsAux
6  from   .tooltip         import Tooltip
7
8
9  def initialize(parent: tk.Frame, side_aux: SidePanelsAux
10                ) -> tuple[tk.Frame, tk.PanedWindow, Callable[[], None]]:
11     """Vytvoří a uspořádá postranní panely.
12     Vrací (levý_rám, pravý_rám, aktualizační_funkce).
13     """
14     # 1. Vytvoření levého panelu s využitím jeho části přepravky
15     pan_bag = Panel(parent, "Batoh", side_aux.for_bag)
16
17     # 2. Vytvoření pravého sloupce s posuvnou příčkou
18     pan_right = tk.PanedWindow(parent, orient=tk.VERTICAL,
19                                sashrelief=tk.RAISED, sashwidth=4)
20     pan_exits = Panel(pan_right, "Sousedé", side_aux.for_exits)
21     pan_items = Panel(pan_right, "Předměty", side_aux.for_items)
22     pan_right.add(pan_exits, minsize=40)
23     pan_right.add(pan_items, minsize=40)
24
25     # Definice vnitřní funkce update_all je totožná s výpisem 7.9 na straně 176
26
27     return pan_bag, pan_right, update_all
```

17.7.3 Změna předka a initor třídy

Následně upravíme definici samotné třídy Panel. Namísto dědění z `CTkScrollableFrame` zvolíme dědění z obyčejného rámce `tk.Frame`, čímž splníme požadavek `PanedWindow` na přímého potomka.

Samotný rolovací rámec `ctk.CTkScrollableFrame` do něj vložíme kompozicí jako vnitřní prvek.

Kód hlavičky a initoru ukazuje výpis 17.13.

Na řádku 1 dědí třída `Panel` z kontejneru `tk.Frame`.

Na řádku 11 initor inicializuje bázový rámec a na řádku 15 do něj vytvoří rolovací rámec `_scroll_frame` s požadovaným nadpisem v záhlaví.

Na řádku 16 se tento vnitřní rámec roztáhne metodou `pack()` přes celou plochu panelu.

Výpis 17.13: Výměna rodiče třídy `Panel` a její initor v modulu `gui69.v17_custom.panels`

```
class Panel(tk.Frame):
    """Třída pro panely, které umí dynamicky aktualizovat svůj obsah
    zobrazující seznamy položek formou interaktivních tlačítek.
    Využívá kompozici: do základního rámce vkládá rolovací kontejner.
    """

    def __init__(self, parent: tk.Widget, title: str,
                 panel_aux: PanelAux):
        """Vytvoří panel a vloží do něj rolovací rámec s nadpisem.
        """
        super().__init__(parent)
        self._panel_aux = panel_aux  # Uloží obdrženou přepravku

        # Kompozice: vnitřní rolovací rámec vyplní celou plochu panelu
        self._scroll_frame = ctk.CTkScrollableFrame(self, label_text=title)
        self._scroll_frame.pack(fill=tk.BOTH, expand=True)
```

17.7.4 Úprava tlačítek v metodě `update_content()`

Ve druhém kroku vyměníme samotná interaktivní tlačítka uvnitř panelu. Původní třídu `tk.Button` nahradíme třídou `ctk.CTkButton`. Tlačítka nevkládáme do samotného panelu, nýbrž do jeho vnitřního rolovacího rámce `self._scroll_frame`.

S touto výměnou úzce souvisí i způsob přiřazení ikony předmětu. Kdybychom tlačítku předali dřívější instanci `tk.PhotoImage` z funkce `get_image()`, tlačítko by sice obrázek zobrazilo, ale konzole by byla zahlcena varovnými hlášeními a ikona by se na displejích High-DPI neuměla přizpůsobit měřítku.

Obrátíme se proto na nově vytvořenou funkci `get_ctk_image()` v modulu `game_manager`, která vrací instanci třídy `ctk.CTkImage`.

Při dodatečném přiřazení načteného obrázku navíc pamatujeme na pravidlo z teoretické části: namísto dřívějšího aliasu `config()` musíme volat přímo metodu `configure()`.

Úpravu metody zajišťující překreslování tlačítek po každém kroku hry ukazuje výpis 17.14.

Změny se soustředí do těla iteračního cyklu. Na řádku 6 prochází cyklus potomky vnitřního rámce `self._scroll_frame` a ruší předchozí sadu tlačítek.

Na řádcích 19–22 vzniká instance tlačítka `ctk.CTkButton`, jako potomek tohoto rolovacího rámce. Tlačítko přebírá vypočtený stav `'normal'` či `'disabled'` a je mu přiřazen dříve zavedený lambda výraz odesílající herní příkaz.

Na řádku 27 volá metoda novou funkci `gm.get_ctk_image()`. Získá tak grafický objekt třídy `ctk.CTkImage`, jenž garantuje přesné sjednocení velikosti všech ikon na standardní rozměr konstanty `c.BUTTON_ICON_SIZE` bez ohledu na rozlišení zdrojových souborů na disku.

Zjistí-li podmínka na řádku 28, že obrázek k danému předmětu existuje, přiřadí se tlačítku na řádku 29 voláním metody `configure()`. Konstanta `tk.TOP` z modulu `tkinter`,

umisťující obrazová data nad textový název položky, zůstává zachována a bez problémů spolupracuje i s novou knihovnou.

V samém závěru cyklu (řádky 32–34) se k aktivnímu tlačítku připojuje bublinová nápověda. Zde se plně projevuje elegance našeho dřívějšího návrhu z podkapitoly *14.3 Realizace bublinové nápovědy* na straně 318: třída `Tooltip` funguje naprosto beze změny i nad moderním widgetem `ctk.CTkButton`, protože `ctk` komponenty spolehlivě delegují zachytávání událostí myši (`<Enter>`, `<Leave>`) i geometrické dotazy (`winfo_rootx`, `winfo_height`) na své vnitřní kreslicí plátno.

Výpis 17.14: *Metoda `update_content()` ve třídě `Panel` v modulu `gui69.v17_custom.panels`*

```
    def update_content(self, is_alive: bool) -> None:
        """Smaže starý obsah a vytvoří novou sadu tlačítek.
        Požádá správce her o případné ikony k předmětům.
        """
        # 1. Smazání původního obsahu (tlačítek)
        for child in self._scroll_frame.winfo_children():
            child.destroy()

        # 2. Získání aktuálních názvů od zdroje v přepravce
        names = self._panel_aux.get_names()

        # 3. Vytvoření tlačítek pro jednotlivé položky
        for item in names:
            # Tlačítko je aktivní, běží-li hra a je-li položka povolena
            is_enabled = self._panel_aux.is_enabled(item)
            current_state = 'normal' if is_alive and is_enabled else 'disabled'

            # Reakce tlačítka sestaví text příkazu a odešle jej do procesoru
            btn = ctk.CTkButton(self._scroll_frame, text=item,
                                state=current_state,
                                command=lambda k=item: self._panel_aux.execute_cmd(
                                        f"{self._panel_aux.get_action_name()} {k}"))

            # Tlačítka vystředíme (CENTER) a svisle odsadíme (pady)
            btn.pack(anchor=tk.CENTER, pady=2)

            img = gm.get_ctk_image(item)
            if img:
                btn.configure(image=img, compound=tk.TOP)

            # 4. Připojení bublinové nápovědy k tlačítku
            if is_alive:
                prefix = self._panel_aux.get_action_name()
                Tooltip(btn, f"Příkaz: {prefix} {item}")
```

17.8 Úprava centrálního panelu

V centrálním panelu (modul `central_panel`) se setkávají dvě základní složky uživatelského rozhraní: výstupní plocha zobrazující historii hry a vstupní řádek sloužící k zadávání hráčových příkazů.

Centrální panel se stane typickou ukázkou hybridního řešení: zatímco horní výstupní část zůstane beze změny složena z původních `tk` widgetů, spodní vstupní část nahradíme `ctk` widgety, aby spolehlivě fungovala i po přepnutí do tmavého režimu.

17.8.1 Hybridní pojetí centrálního panelu

V předchozích kapitolách jsme v modulu `central_panel` vybudovali systém formátování odpovědí pomocí funkce `_define_styles()`. Textové pole `Text` v něm ve výchozím nastavení vysazuje příkazy hráče modře a tučně, odpovědi hry černě a chybová hlášení tučně tmavě červeně.

Knihovna *CustomTkinter* sice disponuje víceřádkovým textovým polem `ctk.CTkTextbox`, to však vnitřně nepodporuje dříve zavedený mechanismus tagů se specifickým formátováním jednotlivých úseků textu tak, jak jej naše aplikace vyžaduje.

Přechod na `CTkTextbox` by znamenal ztrátu barevného odlišení chybových hlášení. Z tohoto důvodu ponecháme výstupní pole `text_pane` i jeho posuvník `scrollbar` v původní podobě jako standardní widgety knihovny *Tkinter*.

Zůstal-li by však kontejner spodní vstupní části klasickým widgetem `tk.Frame` a návěští klasickým `tk.Label`, po přepnutí aplikace do tmavého režimu by jejich šedé pozadí a černý text vytvořily nevzhledný, nečitelný blok. Spodní kontejner `pane` proto nahradíme třídou `ctk.CTkFrame` s průhledným pozadím (`fg_color="transparent"`) a návěští `lbl` nahradíme `ctk` třídou `ctk.CTkLabel`, jejíž text se v tmavém režimu automaticky přebarví na bílo.

17.8.2 Rozdíly v API widgetu `CTkComboBox`

Původní rozbalovací seznam `ttk.Combobox` neumí reagovat na volání funkce `ctk.set_appearance_mode()`. V tmavém režimu by jeho vstupní řádek i rozbalovací nabídka zůstaly světlé. Nahradíme jej proto `ctk` komponentou `ctk.CTkComboBox`.

Při přechodu na `ctk.CTkComboBox` musíme zohlednit několik odlišností v jeho aplikačním rozhraní:

- **Čištění vstupního pole:** Původní `ttk.Combobox` vymazal svůj obsah voláním metody `delete(0, tk.END)`. Třída `ctk.CTkComboBox` tuto metodu ve svém veřejném rozhraní nenabízí. Vyčištění textu se u ní dosáhne předáním prázdného stringu jako argumentu metodě `set("")`.

- **Aktualizace seznamu hodnot:** Pro změnu seznamu nabízených akcí je nutné namísto dřívějšího zkráceného názvu `config()` volat metodu plným názvem `configure()`, přičemž se jí jako argument `values` předá nová kolekce hodnot.
- **Reakce na výběr z nabídky:** V klasickém *Tkinteru* se výběr z roletky zachytával svázáním virtuální události `<<ComboboxSelected>>`. Třída `ctk.CTkComboBox` namísto toho nabízí přímo argument `command` v konstruktoru.

  Zadané obslužné funkci se navíc při výběru položky předá vybraný text jako argument. Odpadá tak dřívější nutnost zjišťovat text dodatečným voláním metody `get()`.
- **Absence metody `icursor()`:** Původní prvek vyžadoval po programovém dopsání mezery explicitní posun textového kurzoru voláním `entry_field.icursor(tk.END)`. Třída `ctk.CTkComboBox` tuto metodu ve svém veřejném rozhraní nenabízí a po zavolání `set()` zanechá kurzor na konci textu zcela automaticky.

17.8.3 Importy a globální deklarace

Nyní přistoupíme k úpravě samotného kódu vstupní části. Před úpravami pochopitelně doplníme na začátek modulu import modulu `customtkinter`.

Současně bychom měli u globální proměnné `entry_field` upravit typovou anotaci na `ctk.CTkComboBox`. Změny najdete ve výpisu 17.15.

Výpis 17.15: *Rozšíření importů a globálních proměnných v modulu `gui69.v17_custom.central_panel`*

```
1  import tkinter          as     tk
2  import customtkinter    as     ctk
3  from   collections.abc  import Callable
4  from   tkinter          import ttk
5  from   .                import constants as c
6
7
8  # Globální proměnné modulu inicializované uvnitř příslušných funkcí
9  input_pane:  tk.Frame    # Kontejner pro dolní část (návěští + vstupní pole)
10 output_pane: tk.Frame    # Kontejner pro horní část (textové pole + posuvník)
11 entry_field: ctk.CTkComboBox      # Kombinovaný seznam pro zadávání příkazů
12 text_pane:   tk.Text     # Víceřádkové textové pole pro průběžný výpis konverzace
```

Jak rámečky ukazují, import přibyl na řádku 2 a typová anotace se změnila na řádku 11.

17.8.4 Aktualizace nabídky akcí

Procházíme-li dále zdrojovým souborem, tak funkce `initialize()` se změny nedotknou. První funkcí, jejíž kód vyžaduje zásah, je `update_actions()`. Tato funkce slouží k aktualizaci seznamu nabízených akcí v rozbalovací roletce při spuštění či změně hry.

Jak bylo vysvětleno v teoretické části, `ctk` widgety nepodporují dřívější alias `config()`. Pro předání nové kolekce hodnot je nutné zavolat metodu plným názvem `configure()`, jíž se v argumentu `values` předá seznam názvů akcí. Kód upravené funkce najdete ve výpisu 17.16.

Výpis 17.16: *Úprava funkce `update_actions()` v modulu `gui69.v17_custom.central_panel`*

```
1 def update_actions(action_names: tuple[str, ...]) -> None:
2     """Aktualizuje seznam nabízených akcí v rozbalovacím seznamu comboboxu.
3     """
4     entry_field.configure(values=list(action_names))
```

Na řádku 4 předává funkce převedený seznam akcí jako argument `values` metody `configure()`. Původní volání aliasu `config()` je tímto plnohodnotně nahrazeno.

Explicitní volání konstruktoru `list(action_names)` zde není samoúčelné – slouží k přetypování neměnné n-tice z herního API na standardní seznam, který `ctk.CTkComboBox` pro své vnitřní procházení nabízených položek vyžaduje.

17.8.5 Úprava vstupního panelu

Ve funkci `_prepare_input_pane()` vyměníme kontejner za `ctk.CTkFrame`, návěští za `ctk.CTkLabel` a vstupní pole za `ctk.CTkComboBox`. Obsluhu výběru akce navážeme přímo přes parametr `command`.

Díky tomu zjednodušíme i obslužnou funkci `_on_action_selected()`. Ta nově přijímá vybraný text přímo ve svém parametru `choice`. Doplní za něj mezeru a nastaví jej zpět do vstupního pole, aby hráč mohl rovnou psát argument příkazu.

Podobu obou upravených funkcí ukazuje výpis 17.17.

Ve funkci `_prepare_input_pane()` se na řádku 9 vytváří kontejner `pane` jako instance `ctk.CTkFrame` s průhledným pozadím.

Na řádku 12 se do něj vkládá návěští `lbl` a na řádcích 16–17 rozbalovací seznam `entry_field`.

Všimněte si, že v definici na řádku 16 předáváme jako argument pro parametr `command` přímo odkaz na funkci `_on_action_selected`. Zcela tak odpadá dřívější svazování s událostí `<<ComboboxSelected>>`.

Funkce `_on_action_selected()` na řádcích 26–29 pak demonstruje zjednodušení obsluhy. Na řádku 26 definuje parametr `choice`, do nějž widget automaticky předá zvolený název akce jako textový argument. Na řádku 29 se k tomuto stringu připojí mezera a výsledek se do pole nastaví voláním metody `set()`.

Všimněte si, že dřívější volání metody `icursor(tk.END)` z výpis 7.8 na straně 175 zde zcela odpadá, neboť metoda `set()` u `ctk` roletky zanechá textový kurzor připravený na samotném konci automaticky.

Výpis 17.17: Úprava funkce `_prepare_input_pane()` v modulu `gui69.v17_custom.central_panel`

```
1  def _prepare_input_pane(parent: tk.Frame | ctk.CTkFrame,
2                          cmd_process: Callable[[str], None]
3                         ) -> ctk.CTkFrame:
4      """Vytvoří panel obsahující návěští a ctk vstupní roletku.
5      """
6      global entry_field
7
8      # 1. Vytvoření ctk kontejneru pro vstupní část
9      pane = ctk.CTkFrame(parent, fg_color="transparent")
10
11     # 2. Vytvoření a umístění ctk návěští
12     lbl = ctk.CTkLabel(pane, text="Zadejte příkaz: ")
13     lbl.pack(side=tk.LEFT)
14
15     # 3. Vytvoření ctk rozbalovacího seznamu
16     entry_field = ctk.CTkComboBox(pane, command=_on_action_selected)
17     entry_field.pack(side=tk.LEFT, fill=tk.X, expand=tk.YES)
18
19     # 4. Nastavení reakce na stisk klávesy Enter
20     entry_field.bind('<Return>', lambda event: cmd_process(entry_field.get()))
21     entry_field.focus()  # Nastavení fokusu, aby uživatel mohl hned psát
22
23     return pane
24
25
26 def _on_action_selected(choice: str) -> None:
27     """Po výběru akce z roletky doplní mezeru pro zadání argumentu.
28     """
29     entry_field.set(choice + ' ')
```

17.8.6 Zpracování výstupu a zneplatnění vstupu

Závěrečnou funkcí modulu `central_panel`, která pracuje se vstupním polem, je funkce `game_output()`. Zajišťuje formátovaný výpis komunikace do textové historie a po každém tahu manipuluje se vstupním polem `entry_field`.

V této funkci řešíme dvě úpravy spojené s přechodem na `ctk` widget:

- Vyčištění vstupního pole po odeslání příkazu, při němž namísto dřívější metody `delete()` předáme prázdný string jako argument metodě `set("")`.
- Zneplatnění pole po skončení hry a jeho obnova při restartu, kdy namísto dřívějšího aliasu `config()` použijeme volání `configure(state=...)`.

Kód upravené funkce si můžete prohlédnout ve výpisu 17.18.

Ve funkci `game_output()` se na řádku 9 vstupní řádek vyprázdní předáním prázdného stringu metodě `set()`. Tím je nahrazeno dřívější volání metody `delete(0, tk.END)`. Na řádcích 13 a 16 se metodou `configure()` přepíná stav pole (`'disabled'` či `'normal'`).

Výpis 17.18: Úprava funkcí `game_output()` v modulu `gui69.v17_custom.central_panel`

```
def game_output(command: str, answer: str, game_ended: bool = False) -> None:
    """Vypíše do naformátované historie příkaz hráče a odpověď hry.
    Začíná-li odpověď znakem c.ERROR_PREFIX, vypíše ji stylem c.STYLE_ERROR.
    Ukončí-li zadaná odpověď hru, zneplatní vstupní pole.
    """
    # Počátek definice, tj. kroky 1 až 3, se shodují s výpisem 5.8 na straně 139

    # 4. Vyčištění vstupního pole
    entry_field.set("")

    # 5. Zneplatnění vstupního pole po konci hry, nebo jeho obnova
    if game_ended:
        entry_field.configure(state='disabled')
    else:
        # Při startu nebo restartu musíme pole opět aktivovat
        entry_field.configure(state='normal')
        entry_field.focus()
```

17.9 Konfigurační dialog a záložky

Konfigurační dialog (modul `settings_window`) představuje konstrukčně nejsložitější okno celé aplikace. Sdružuje kontejner se čtyřmi tematickými kartami a pestrou škálu ovládacích prvků: táhla, přepínače, zaškrtávací políčka, rozbalovací seznamy i textové seznamy. Při jeho převodu uplatníme `ctk` komponenty a zároveň vyřešíme situace, kdy nadstavbová knihovna některé specializované prvky vůbec nenabízí.

17.9.1 Kontejner záložek `CTkTabview` a volba typu okna

Základ dialogu tvoří sekundární okno, které se zobrazuje nad hlavním oknem aplikace. V duchu hybridního přístupu ponecháme původní třídu `tk.Toplevel`.

Knihovna *CustomTkinter* sice nabízí třídu `ctk.CTkToplevel`, ta se však na operačním systému *Windows* pokouší ovlivňovat barvu systémového záhlaví okna voláním systémového rozhraní. Při kombinaci modálního uzamčení (`grab_set()`) a přepínání barevného schématu to může vést k pádům a zamrznutí aplikace.

Třída `tk.Toplevel` má správu modality (`transient()`, `grab_set()`, `wait_window()`) léty prověřenou a plně stabilní. `Ctk` prvky proto uplatníme až uvnitř tohoto okna.

Zásadní změna v návrhu však přichází s kontejnerem záložek. Původní widget `ttk.Notebook` nahradíme komponentou `ctk.CTkTabview`.

Vizuálně se oba prvky liší: zatímco `ttk.Notebook` vykresluje klasická ouška připomínající kartotéku, `ctk.CTkTabview` používá v záhlaví skupinu plochých tlačítek (obdobu přepínače `CTkSegmentedButton`). Tyto dva prvky se od sebe liší způsobem vytváření a správy jednotlivých záložek:

- **Princip vkládání v `ttk.Notebook`**
  V původním řešení bylo nejprve nutné vytvořit samostatný kontejner `ttk.Frame` jako potomka notebooku a následně jej příkazem

  ```
  notebook.add(frame, text="Název")
  ```

  zaregistrovat jako novou záložku.
- **Princip vytváření v `ctk.CTkTabview`**
  Komponenta `ctk.CTkTabview` vytváří rámce jednotlivých záložek vnitřně ve své vlastní režii. Příkazem

  ```
  tabview.add("Název")
  ```

  vznikne nová záložka a volaná metoda jako svoji návratovou hodnotu vrátí přímo odkaz na vytvořený rámec – kontejner typu `ctk.CTkFrame` reprezentující plochu dané záložky. Odkaz na tento objekt lze dodatečně kdykoliv získat voláním

  ```
  tabview.tab("Název")
  ```

Odpadá tedy ruční vytváření vnějších rámců pro jednotlivé záložky. Pomocné funkci obsluhující obsah záložky jednoduše předáme odkaz na rámec, který pro ni vygeneroval samotný kontejner `CTkTabview`.

17.9.2 Sestavení dialogu ve funkci `show()`

Před úpravou kódu doplníme na začátek modulu `settings_window` import nadstavby:

```
import customtkinter as ctk
```

V samotné funkci `show()` promítneme všechny nově představené konstrukce. Z důvodů, které jsem vysvětlil před chvílí, ponecháme kořenové okno dialogu definované jako instanci třídy `tk.Toplevel`. Pro nastavení jeho výchozí velikosti využijeme zvětšenou hodnotu konstanty `SETTINGS_GEOMETRY`.

Zároveň namísto dosavadního zákazu změny velikosti předáme metodě `resizable()` argumenty `True, True`, aby si uživatel mohl dialogové okno v případě potřeby roztáhnout.

Kontejner záložek nahradíme třídou `ctk.CTkTabview` a jednotlivé karty vytvoříme voláním metody `add()`.

Spodní lištu sestavíme z kontejneru `ctk.CTkFrame` s průhledným pozadím a z tlačítka `ctk.CTkButton`.

Kód upravené funkce `show()` ukazuje výpis 17.19.

V hlavičce funkce je na řádku `1` rozšířena typová nápověda u parametru `parent` na `tk.Tk | ctk.CTk`.

Na řádku `13` povoluje volání metody `resizable()` ruční úpravu rozměrů okna v obou směrech.

Na řádcích `16-17` následuje vytvoření a rozmístění kontejneru záložek `tabview`.

Výpis 17.19: Úprava funkce `show()` v modulu `gui69.v17_custom.settings_window`

```
def show(parent:         tk.Tk | ctk.CTk,       # Rodičovské okno
         settings:       dict[str, tk.Variable],# Slovník sledovacích proměnných
         on_font_change: Callable[[str], None], # Obsluha změny písma
         history:        list[str],             # Seznam názvů odehraných her
         on_play:        Callable[[int], None]  # Obsluha spuštění hry s názvem
        ) -> None:
    """Vytvoří a zobrazí modální okno pro nastavení parametrů aplikace.
    """
    # 1. Vytvoření ctk okna a nastavení základních vlastností
    win = tk.Toplevel(parent)          # Zůstáváme u klasického tk okna
    win.title("Nastavení aplikace")
    win.geometry(c.SETTINGS_GEOMETRY)
    win.resizable(True, True)

    # 2. Vytvoření ctk kontejneru se záložkami
    tabview = ctk.CTkTabview(win)
    tabview.pack(expand=True, fill=tk.BOTH, padx=10, pady=10)

    # 3. Přidání jednotlivých karet a naplnění jejich obsahu
    _create_appearance_tab(tabview.add("Vzhled"), settings, on_font_change)
    _create_journal_tab   (tabview.add("Žurnál"), settings)
    _create_info_tab      (tabview.add("Informační panel"), settings)
    _create_stats_tab     (tabview.add("Statistika"), history, on_play)

    # 4. Spodní lišta s tlačítkem pro zavření
    btn_frame = ctk.CTkFrame(win, fg_color="transparent")
    btn_frame.pack(fill=tk.X, padx=10, pady=(0, 10))
    ctk.CTkButton(btn_frame, text="Zavřít",
                  command=win.destroy).pack(side=tk.RIGHT)

    # 5. Nastavení modality - okno si přivlastní veškerou pozornost
    win.transient(parent)   # Okno je vázáno na rodiče
    win.grab_set()          # Zablokování ostatních oken
    parent.wait_window(win) # Čekání na uzavření dialogu
```

Při plnění záložek na řádcích 20–23 voláme metodu `tabview.add()`, jíž v argumentu předáváme textový název karty. Výsledný odkaz na interní kontejner karty vzápětí předáváme pomocným funkcím obstarávajícím vnitřní obsah.

Na řádcích 26–29 je umístěn spodní kontejner `btn_frame` a v něm tlačítko `Zavřít`.

Nastavení modality na řádcích 32–34 zůstává shodné s původní verzí.

17.9.3 Ovládací prvky na kartách a hybridní kompromisy

Po vyřešení vnějšího rámce dialogu přistoupíme k ovládacím prvkům na jednotlivých kartách. Většinu původních `ttk` komponent nahradíme příslušnými `ctk` protějšky:

- Táhlo `ttk.Scale` nahradí třída `ctk.CTkSlider`. Zde zůstávají parametry `from_`, `to`, `variable` i `command` plně kompatibilní.

- Volby přepínače `ttk.Radiobutton` nahradí třída `ctk.CTkRadioButton`.
- Zaškrtávací políčka `ttk.Checkbutton` nahradí třída `ctk.CTkCheckBox`.

Při návrhu však narazíme na zásadní omezení: knihovna *CustomTkinter* ve své základní nabídce **vůbec neobsahuje** widgety `Spinbox` a `Listbox`.

Zde se naplno projeví přednost námi zvolené hybridní architektury. Namísto složitého doprogramovávání chybějících komponent nebo instalace dalších cizích knihoven jednoduše ponecháme původní widgety:

- Na kartě **Žurnál** zůstane pro volbu úrovně podrobnosti původní prvek `ttk.Spinbox`.
- Na kartě **Statistika** zůstane pro výběr dříve odehrané hry původní seznam `tk.Listbox`.

Oba tyto původní prvky bez jakýchkoliv problémů vložíme přímo do kontejneru `ctk.CTkFrame`, který tvoří plochu dané karty.

17.9.4 Úprava jednotlivých karet nastavení

Dialogové okno s kartami pro jednotlivé skupiny nastavení jsme vytvářeli v kapitole *12 Složitější dialogy* na straně 264. Při refaktoraci pomocných funkcí, jež vytvářejí obsah jednotlivých karet, narazíme na zásadní architektonický rozdíl.

V původní verzi si každá pomocná funkce vytvářela kontejner pro svou kartu sama (`frame = ttk.Frame(notebook)`) a na konci jej vracela jako návratovou hodnotu, aby jej funkce `show()` mohla zaregistrovat do kontejneru `notebook`.

Nyní je tento tok obrácený: kontejner každé záložky vzniká už ve funkci `show()` voláním metody `tabview.add()`. Pomocné funkci je tento hotový rámec předán v argumentu `frame`. Funkce jej pouze naplní prvky a již nic nevrací; její návratový typ je tedy `None`.

Karta Vzhled

Na kartě Vzhled nahradíme původní táhlo `ttk.Scale` prvkem `ctk.CTkSlider` a volby přepínače `ttk.Radiobutton` prvky `ctk.CTkRadioButton`.

Způsob napojení na sledovací proměnné `settings['font_size']` a `settings['theme']` zůstává naprosto shodný. Kód upravené funkce `_create_appearance_tab()` ukazuje výpis 17.20.

V hlavičce funkce na řádcích `1-4` si všimněte parametru `frame` s typovou anotací `ctk.CTkFrame` a návratové hodnoty `None`. Funkce již nevytváří vlastní instanci rámce, nýbrž pracuje přímo s kontejnerem dodaným zvenčí.

Na řádcích `10-13` se vytváří `ctk` táhlo `scale_font`. Přijímá stejné argumenty `from_` a `to` a na změnu hodnoty reaguje voláním zadané obslužné funkce předané v argumentu parametru `command`.

Na řádcích `18-23` iteruje cyklus přes definovanou n-tici témat a vytváří přepínače `ctk.CTkRadioButton`. Všechny sdílejí tutéž sledovací proměnnou předanou v argumentu `variable=settings['theme']`.

Výpis 17.20: Úprava funkce `_create_appearance_tab()` v modulu `gui69.v17_custom.settings_window`

```
1  def _create_appearance_tab(frame: ctk.CTkFrame,
2                             settings: dict[str, tk.Variable],
3                             on_font_change: Callable[[str], None]
4                             ) -> None:
5      """Naplní kartu Vzhled ovládacími prvky.
6      """
7      # Vytvoří ctk táhlo pro nastavení velikosti písma
8      ctk.CTkLabel(frame, text="Velikost písma:") \
9         .pack(anchor=tk.W, pady=(10, 0), padx=10)
10     scale_font = ctk.CTkSlider(frame, from_=8, to=20,
11                                variable=settings['font_size'],
12                                command=on_font_change)
13     scale_font.pack(fill=tk.X, padx=20, pady=5)
14
15     # Vytvoří ctk přepínač pro nastavení barevného tématu
16     ctk.CTkLabel(frame, text="Barevné téma:") \
17        .pack(anchor=tk.W, pady=(15, 5), padx=10)
18     themes = [("Světlé", "light"), ("Tmavé", "dark"), ("Systémové", "system")]
19     for text, val in themes:
20         rb = ctk.CTkRadioButton(frame, text=text, value=val,
21                                 variable=settings['theme'])
22         rb.pack(anchor=tk.W, padx=20, pady=2)
```

Karta Žurnál

Karta **Žurnál** představuje názornou ukázku hybridního soužití: zaškrtávací políčka nahradíme třídou `ctk.CTkCheckBox`, avšak výběr úrovně podrobnosti ponecháme původnímu prvku `ttk.Spinbox`, pro nějž knihovna *CustomTkinter* nemá přímou náhradu.

Celá vnitřní logika obsluhy – včetně zneplatnění spinboxu a synchronizace zatržítka pro okno žurnálu – zůstává plně zachována. Upravenou funkci `_create_journal_tab()` naleznete ve výpisu 17.21.

I zde hlavička na řádcích 1–2 přijímá připravený kontejner `frame` a nevrací žádnou hodnotu.

Po definicích vnitřních funkcí už nenásleduje příkaz na vytvoření rámu.

Na řádcích 12–14 a 24–30 se vytvářejí zaškrtávací políčka `ctk.CTkCheckBox`. Ta spolehlivě spolupracují se stávajícími instancemi `BooleanVar` ze slovníku `settings`.

Důležitý detail se nachází na řádcích 17–20. Widget `spin_level` zůstává instancí `ttk.Spinbox`. Jelikož jde o standardní `ttk` prvek, ve vnitřní funkci se jeho stav nadále přepíná voláním metody `config()`. Tato ukázka dokazuje, že uvnitř `ctk` kontejneru mohou bez jakýchkoliv kolizí existovat i prvky vyžadující původní `tk` rozhraní.

Karta Informační panel

Na kartě **Informační panel** generujeme sérii zaškrtávacích políček na základě konfiguračního seznamu `opts`. Namísto původních prvků `ttk.Checkbutton` zde nasadíme třídu `ctk.CTkCheckBox`.

Výpis 17.21: Upravená funkce `_create_journal_tab()` v modulu `gui69.v17_custom.settings_window`

```
1  def _create_journal_tab(frame: ctk.CTkFrame,
2                          settings: dict[str, tk.Variable]) -> None:
3      """Naplní kartu Žurnál ovládacími prvky.
4      """
5
6      # Definice vnitřních funkcí jsou totožně s výpisem 12.4 na straně 277
7
8      # Definice vnitřních funkcí skončily, pokračují výkonné příkazy
9      # Navázání na proměnnou, aby reagoval i na programové změny
10     settings['log_active'].trace_add('write', toggle_spinbox)
11
12     chk_log = ctk.CTkCheckBox(frame, text="Vytvářet záznam průběhu hry",
13                               variable=settings['log_active'])
14     chk_log.pack(anchor=tk.W, padx=10, pady=(15, 5))
15
16     ctk.CTkLabel(frame, text="Úroveň podrobnosti:").pack(anchor=tk.W, padx=30)
17     spin_level = ttk.Spinbox(frame, from_=1, to=3, increment=1,
18                              state=tk.DISABLED,
19                              textvariable=settings['log_level'])
20     spin_level.pack(anchor=tk.W, padx=30, pady=(0, 15))
21
22     toggle_spinbox()  # Počáteční synchronizace stavu
23
24     ctk.CTkCheckBox(frame, text="Automaticky rolovat žurnál",
25                     variable=settings['auto_scroll']).pack(anchor=tk.W,
26                                                            padx=10, pady=5)
27     ctk.CTkCheckBox(frame, text="Zobrazovat žurnál v samostatném okně",
28                     variable=settings['log_in_window'],
29                     command=on_window_check).pack(anchor=tk.W,
30                                                   padx=10, pady=5)
```

Způsob provázání se sledovacími proměnnými ze slovníku `settings` zůstává naprosto shodný jako ve 12. kapitole. Kód funkce `_create_info_tab()` si můžete prohlédnout ve výpisu 17.22.

Cyklus na řádcích **11–14** prochází n-tice ze seznamu `opts` a generuje zaškrtávací políčka `ctk.CTkCheckBox`. Stringy `'normal'` a `'disabled'` určují počáteční stav přístupnosti políček.

Karta Statistika

Karta **Statistika** pak obsahuje textový seznam odehraných her. Podobně jako v případě spinboxu, ani pro seznam `Listbox` nemá knihovna *CustomTkinter* ekvivalent. Na kartě **Statistika** proto ponecháme klasický widget `tk.Listbox`. Kód této funkce si můžete prohlédnout ve výpisu 17.23.

Výpis 17.22: *Úprava funkce `_create_info_tab()` v modulu `gui69.v17_custom.settings_window`*

```
def _create_info_tab(frame: ctk.CTkFrame,
                     settings: dict[str, tk.Variable]) -> None:
    """Naplní kartu Informační panel zaškrtávacími políčky.
    """
    opts = [("Datum",                       'show_date',      tk.NORMAL),
            ("Aktuální čas",                'show_time',      tk.NORMAL),
            ("Čas aktuální hry",            'show_playtime',  tk.NORMAL),
            ("Nápověda v samostatném okně", 'help_in_window', tk.DISABLED),
            ("Plánek v samostatném okně",   'map_in_window',  tk.NORMAL)]

    for text, key, state in opts:
        chk = ctk.CTkCheckBox(frame, text=text, variable=settings[key],
                              state=state)
        chk.pack(anchor=tk.W, padx=10, pady=8)
```

Výpis 17.23: *Úprava funkce `_create_stats_tab()` v modulu `gui69.v17_custom.settings_window`*

```
def _create_stats_tab(frame: ctk.CTkFrame, history: list[str],
                      on_play: Callable[[int], None]) -> None:
    """Naplní kartu Statistika seznamem odehraných her.
    """
    lbl = ctk.CTkLabel(frame,
                       text="Dvojklikem znovu spusťte dříve odehranou hru:")
    lbl.pack(anchor=tk.W, padx=10, pady=(10, 5))

    listbox = tk.Listbox(frame, selectmode=tk.SINGLE, height=10)
    listbox.pack(fill=tk.BOTH, expand=True, padx=10, pady=(0, 10))

    for item in history:
        listbox.insert(tk.END, item)

    def on_double_click(event: tk.Event) -> None:
        """Přečte vybranou položku a uzavře dialog.
        """
        selection = listbox.curselection()
        if selection:
            on_play(selection[0])
            frame.winfo_toplevel().destroy()

    listbox.bind('<Double-1>', on_double_click)
```

Na řádku 5 do `ctk` kontejneru vkládá klasický `tk.Listbox`.

Obsluha dvojkliku myši definovaná na řádcích 15–21 přečte vybraný index hry, spustí ji a zavoláním `frame.winfo_toplevel().destroy()` bezpečně uzavře nadřazené dialogové okno.

17.10 Oživení témat a initor balíčku

V této podkapitole propojíme všechny dosud upravené součásti dohromady. Zprovozníme přepínání barevného vzhledu a dokončíme integraci v initoru balíčku. Tím dosáhneme stavu, kdy uživatelské rozhraní dokáže plynule a v reálném čase přecházet mezi světlým a tmavým režimem.

17.10.1 Teoretický základ: Globální řízení vzhledu

Knihovna *CustomTkinter* spravuje barevné schéma celé aplikace prostřednictvím globální funkce `ctk.set_appearance_mode()`. Tato funkce přijímá jako textový argument jeden ze tří stringů:

- **`"light"`** – Vynutí světlý režim pro všechny `ctk` widgety bez ohledu na nastavení operačního systému.
- **`"dark"`** – Vynutí tmavý režim (tmavé pozadí oken, světlé texty, přizpůsobené barvy tlačítek a posuvníků).
- **`"system"`** – Převezme aktuální systémové nastavení operačního systému. Přepne-li uživatel v systému tmavý motiv, aplikace se okamžitě a bez restartu přebarví.

Právě zde naplno zúročíme návrh konfiguračního dialogu z kapitoly *12 Složitější dialogy* na straně 264. V něm jsme do sledovací proměnné `settings['theme']` uložili přesně tyto tři hodnoty (`light`, `dark`, `system`). Zbývá pouze zařídit, aby se při každé změně této proměnné automaticky zavolala funkce `ctk.set_appearance_mode()`.

Kromě barevného schématu nabízí knihovna *CustomTkinter* také globální nástroj pro přizpůsobení velikosti celého rozhraní: funkci `ctk.set_widget_scaling()`. Tato funkce přijímá jako číselný argument koeficient měřítka (např. hodnotu `1.0` pro výchozí velikost, `1.5` pro 150% zvětšení nebo `0.75` pro zmenšení na tři čtvrtiny).

Na rozdíl od klasického *Tkinteru*, kde bylo nutné měnit velikost písma v bodech, přeškáluje tato funkce všechny `ctk` komponenty proporcionálně jako celek – tedy včetně jejich vnitřního odsazení, výšky tlačítek a zaškrtávacích políček. Vydělíme-li hodnotu z našeho táhla (8 až 20 bodů) základní velikostí 12 bodů, získáme přesný poměr, jímž okamžitě a bez nutnosti zasahovat do jednotlivých widgetů změníme měřítko všech `ctk` prvků v celé aplikaci.

17.10.2 Souběžná změna velikosti písem v modulu `settings_manager`

Než propojíme samotná témata, vyřešíme souběžnou změnu velikosti písma. Úprava začne jako obyčejně již v importech přidáním standardního příkazu

```
import customtkinter as ctk
```

V pasáži *17.4.1 Teoretický základ: okno a písmo* na straně 382 jsme zavedli paralelní existenci dvou objektů písma: pojmenované šablony s názvem v `c.APP_FONT` pro `tk` prvky a objektu `app_font` pro `ctk` prvky.

Posune-li uživatel táhlo velikosti písma v konfiguračním dialogu, zavolá se funkce `on_font_change()`. V původní verzi měnila pouze velikost písma šablony `c.APP_FONT`. Aby se nová velikost promítla i na `ctk` tlačítkách a návěštích, musíme oslovit i `ctk` písma. Jednou z možností je zavolat funkci `ctk.set_widget_scaling()`, které v argumentu předáme vypočtený poměr nové velikosti vůči základu 12 bodů.

Kód upravené funkce ukazuje výpis 17.24.

Výpis 17.24: *Úprava funkce `on_font_change()` v modulu `gui69.v17_custom.settings_manager`*

```
1 def on_font_change(size_str: str) -> None:
2     """Reaguje na posun táhla a dynamicky mění velikost obou písem.
3     """
4     new_size = int(float(size_str))
5     app_font = tkfont.nametofont(c.APP_FONT)
6     app_font.config(size=new_size)
7     ctk.set_widget_scaling(new_size / c.DEFAULT_FONT_SIZE)
```

Funkce nejprve na řádcích 4–6 přepočte a nastaví velikost pojmenovaného `tk` písma.

Příkaz na řádku 7 zajistí okamžité a plynulé přeškálování všech `ctk` komponent v aplikaci.

Obě rodiny widgetů se tak zvětší či zmenší zcela synchronně.

17.10.3 Úprava initoru balíčku

Závěrečný krok celé refaktorace se odehrává v initoru balíčku. V něm propojíme řízení témat a zajistíme spuštění aplikace s novým kořenovým oknem.

V úvodním bloku modulu nejprve doplníme import

```
import customtkinter as ctk
```

a u globální proměnné `window` upravíme typovou anotaci na `ctk.CTk` v deklaraci

```
window: ctk.CTk      # Aplikační okno
```

V samotné řídicí funkci `run()` provedeme v 5. sekci dvě úpravy související se sledovací proměnnou uloženou ve slovníku `sm.app_settings` pod klíčem `'theme'`:

- Zavoláním `ctk.set_appearance_mode()` nastavíme počáteční téma podle obsahu této sledovací proměnné. Tím zaručíme, že se okno otevře v režimu, který si uživatel zvolil při minulém běhu aplikace nebo ve výchozím systémovém nastavení.
- Pomocí metody `trace_add()` pověsíme na tuto proměnnou posluchače. Jakmile uživatel v dialogu nastavení klikne na jinou volbu, obslužná lambda funkce hned zjistí nově nastavenou hodnotu a předá ji funkci `ctk.set_appearance_mode()`.

Kompletní podobu upravené funkce `run()` ukazuje výpis 17.25.

Výpis 17.25: Úprava funkce `run()` v modulu `gui69.v17_custom.__init__`

```
1  def run(portal: IPortal) -> ctk.CTk:
2      """Vytvoří aplikační okno a odstartuje hru dodanou zadaným portálem.
3      """
4      global window, game_output, update_gui, clear_output, update_actions
5
6      # První čtyři body jsou totožné s výpisem 12.18 na straně 295
7
8      # 5. Téma, ochrana modality a navázání posluchačů na sledovací proměnné
9      theme_var = sm.app_settings['theme']  # Proměnná uchovávající aktuální téma
10     ctk.set_appearance_mode(theme_var.get())  # Nastaví se poslední zadané
11     # Po zápisu nového tématu je pomocí get() zjistí a nechá je nastavit
12     theme_var.trace_add('write',
13                         lambda *args: ctk.set_appearance_mode(theme_var.get()))
14
15     # Rozezpívání němé proměnné z prvku Spinbox (úroveň podrobnosti)
16     window.logging_level.trace_add('write',
17         lambda *args: jm.set_log_level(window.logging_level.get()))
18
19     window.lift()       # Umístí okno nad všechna ostatní okna
20     window.update()     # Vynutí vykreslení okna
21
22     # 6. Sjednocený start hry (předáváme celý portál kvůli historii)
23     _start_new_game(portal)
24
25     return window       # Odkaz pro spuštění smyčky událostí
```

V hlavičce funkce na řádku 1 deklaruje návratový typ novou třídu `ctk.CTk`.

Sekce *1* až *4* jsou totožné s výpisem 12.18 na straně 295, takže jsem je v tomto výpisu nahradil odkazem.

Na začátek sekce *5* se vloží několik nových příkazů. Nejprve se na řádku 9 načte odkaz na sledovací proměnnou témat, v níž je uloženo požadované téma.

Zde je kriticky důležité dodržet posloupnost inicializačních kroků: načtení proměnné ze slovníku `sm.app_settings` smí proběhnout až v sekci *5*, tedy teprve poté, co sekce *4* zavolala funkci `sm.initialize(window)`, jež tento konfigurační slovník reálně vytvořila a naplnila.

Na řádku 10 se pak na zjištěnou hodnotu téma nastaví.

Na řádcích 12–13 se na proměnnou témat pověsí posluchač, který po každé zadané změně tématu zjistí, jaká je tato hodnota, a pak příslušné téma zavoláním metody `ctk.set_appearance_mode()` nastaví.

17.11 Kontrolní test

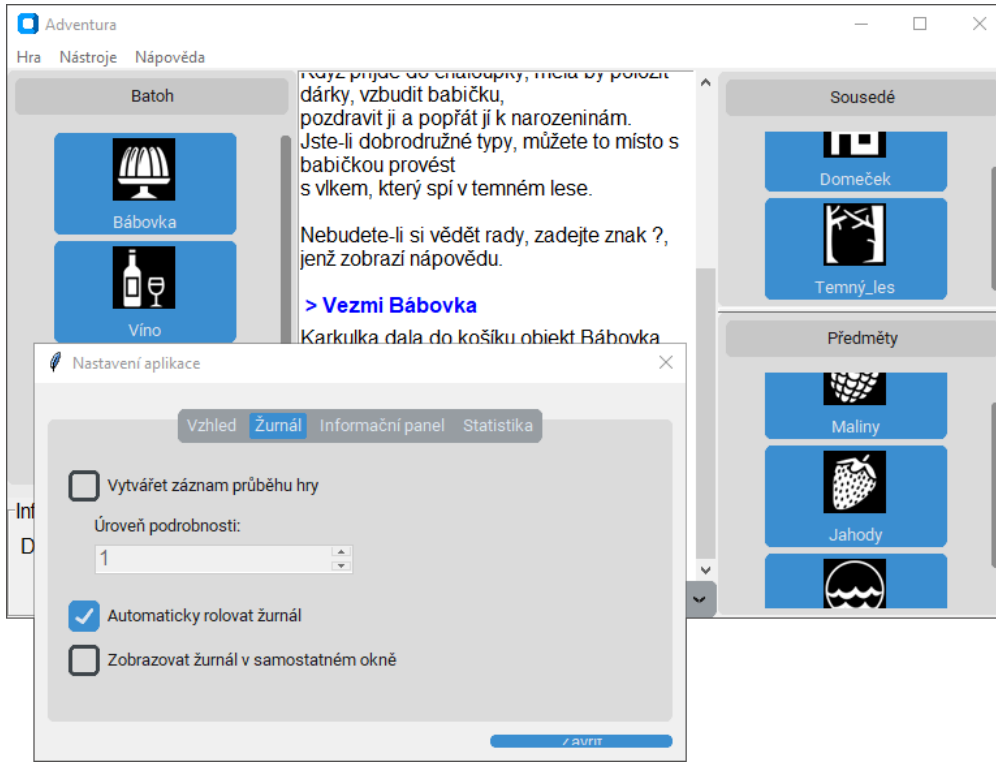
Vše je připraveno ke kontrolnímu testu spuštěnému ve *Windows* zadáním příkazu

```
python -m gui69.test v17_custom
```

v *Linuxu* a *macOS* zadáním příkazu

```
python3 -m gui69.test v17_custom
```

Vizuální proměnu aplikace si můžete prohlédnout na obrázku 17.1.



Obrázek 17.1:
Aplikační okno zobrazující obrázky předmětů a sousedních prostorů

17.12 Shrnutí a soubory pro opakování

V této závěrečné kapitole jsme modernizovali grafické uživatelské rozhraní naší aplikace a definitivně vyřešili prostorové i formátové limity předchozích verzí. Nejdůležitější probrané principy jsou:

- **Nadstavba *CustomTkinter* a vektorové vykreslování:**
  Seznámili jsme se s knihovnou *CustomTkinter*, která vykresluje zaoblené prvky vektorově na kreslicím plátně, čímž zajišťuje jednotný moderní vzhled a ostrost na monitorech s vysokým rozlišením.
- **Hybridní architektura:**
  Ukázali jsme si, že není nutné aplikaci přepisovat od základů. Komponenty z knihovny *CustomTkinter* mohou bez konfliktů koexistovat s klasickými widgety

knihovny *Tkinter* v témže okně (systémová lišta nabídek `Menu`, naformátovaný výstup v poli `Text`, kreslicí plátno `Canvas` či dělicí kontejner `PanedWindow`).

- **Globální správa témat a škálování:**
  Propojili jsme sledovací proměnné nastavení s globální funkcí `set_appearance_mode()` pro přepínání mezi světlým, tmavým a systémovým režimem a funkcí `set_widget_scaling()` pro proporcionální změnu velikosti celého rozhraní.
- **Integrace knihovny *Pillow* a podpora rastrových formátů:**
  Zapojili jsme externí knihovnu *Pillow*, rozšířili podporu o formáty JPG a WEBP a zavedli proporcionální přizpůsobení rozměrů map metodou `resize()` s vyhlazovacím filtrem `LANCZOS`.
- **Standardizace rozměrů ikon na tlačítkách:**
  Díky třídě `ctk.CTkImage` a sdílené konstantě `BUTTON_ICON_SIZE` jsme zaručili, že ikony na tlačítkách mají jednotné logické rozměry bez ohledu na fyzické rozlišení souborů na disku.
- **Kombinace posuvné příčky a rolovacích rámců:**
  Vyřešili jsme riziko přetečení seznamů tlačítek nasazením rolovacího rámce `CTkScrollableFrame`. Pravý sloupec jsme osadili svisle posuvnou vodorovnou příčkou `PanedWindow` a záludnost metody `add()` jsme vyřešili uplatněním návrhového principu kompozice.

17.12.1 Doprovodné soubory

Zadávané příkazy i obsah výpisů se záznamem komunikace s interpretem probíhající v této kapitole najdete v souborech nazvaných `m17__CustomTkinter`.

V této kapitole byly v balíčku `gui69.v17_custom` postupně upraveny následující moduly:

- `constants` – byla upravena hodnota konstanty `SETTINGS_GEOMETRY` s geometrií dialogu nastavení a zavedena výchozí velikost ikon tlačítek `BUTTON_ICON_SIZE`.
- `app_window` – nahradil bázové aplikační okno třídou `ctk.CTk` a zavedl souběžnou správu klasického písma `legacy_font` i moderního písma `app_font`.
- `game_manager` – zapojil knihovnu *Pillow* pro načítání formátů JPG a WEBP, rozšířil funkci `get_image()` o volitelný parametr `max_size` pro proporcionální škálování se zachováním poměru stran a přidal funkci `get_ctk_image()` vracející objekty `ctk.CTkImage` s rozměrem `BUTTON_ICON_SIZE`.
- `map_window` – ve funkcích `_initialize_content()` a `update_map()` doplnil předání mezních rozměrů plátna `(c.MAP_WIDTH, c.MAP_HEIGHT)` do volání `gm.get_image()`, čímž umožnil zobrazení rozsáhlých mapových podkladů v libovolném rozlišení.
- `panels` – ve funkci `initialize()` nahradil pevné rozdělení pravého sloupce kontejnerem `tk.PanedWindow` se svisle posuvnou vodorovnou příčkou. Třída `Panel` byla

převedena na kompozici s rolovacím rámcem `ctk.CTkScrollableFrame` a tlačítka byla nahrazena komponentami `ctk.CTkButton` s ikonami.

- `central_panel` – vstupní řádek byl převeden na moderní prvky `ctk.CTkFrame`, `ctk.CTkLabel` a `ctk.CTkComboBox`, čímž byla zaručena jeho čitelnost i v tmavém režimu; výstupní plocha zůstala hybridně klasická s barevnými formátovacími tagy.
- `settings_window` – kontejner záložek byl nahrazen komponentou `ctk.CTkTabview` a ovládací prvky na kartách zaoblenými `ctk` widgety (`CTkSlider`, `CTkRadioButton`, `CTkCheckBox`), přičemž chybějící prvky `ttk.Spinbox` a `tk.Listbox` byly ponechány hybridně.
- `settings_manager` – funkce `on_font_change()` byla rozšířena o volání `ctk.set_widget_scaling()`, které zajišťuje souběžné škálování `ctk` komponent spolu s pojmenovaným písmem.
- `__init__` – v řídicí funkci `run()` propojil počáteční nastavení tématu s funkcí `ctk.set_appearance_mode()`, navázal sledování proměnné `theme` a zajistil spuštění aplikace s novým kořenovým oknem.

Těla modulů `info_panel`, `journal_manager`, `journal_window`, `level_dialog`, `menu` a `tooltip` zůstala v této kapitole nezměněna.

Literatura

[1] FOWLER, Martin. *Refactoring. Improving the Design of Existing Code.* Boston: Addison-Wesley, © 2000. 430 s. (Přeloženo v [2]) ISBN 0-201-48567-2.

[2] FOWLER, Martin. *Refaktoring. Zlepšení existujícího kódu.* Praha: Grada Publishing, © 2003. 394 s. (Překlad [1]) ISBN 80-247-0299-1.

[3] GAMMA E., HELM R., JOHNSON R., VLISSIDES J.: *Design patterns: elements of reusable object-oriented software.* Boston: Addison-Wesley, 2001. ISBN 0-201-63361-2.

[4] PECINOVSKÝ R.: *Začínáme programovat v jazyku Python*. Praha: Grada Publishing, © 2020. ISBN 978-80-247-1237-1.

[5] PECINOVSKÝ R.: *Začínáme programovat v jazyku Python – 2. přepracované a rozšířené vydání*. Praha: Grada Publishing, © 2022, ISBN 978-80-271-4752-6 (ePub), ISBN 978-80-271-4751-9 (pdf), ISBN 978-80-271-3609-4 (print).

[6] PECINOVSKÝ Rudolf: *Začínáme programovat v jazyku Python – Třetí, aktualizované a doplněné vydání*. Praha: Grada Publishing, 2024, ISBN 978-80-271-7447-8 (ePub), 978-80-271-7446-1 (pdf), 978-80-271-5467-8 (print).

[7] PECINOVSKÝ Rudolf: *Python 3.14 – Díl 1: Algoritmické konstrukce*. Praha: Grada Publishing, © 2025, ISBN 978–80–271–8342-5 (ePub), 978–80–271–6341-8 (pdf), 978-80-271-6061-7 (print).

[8] PECINOVSKÝ Rudolf: *Python 3.14 – Díl 2: Objektové konstrukce*. Praha: Grada Publishing, © 2025, ISBN 978–80–271–8363-0 (ePub), 978–80–271–8362-3 (pdf), 978-80-271-6062-4 (print).

[9] PECINOVSKÝ Rudolf: *Python – knihovny pro práci s daty pro verzi 3.11.* Praha: Grada Publishing, © 2022. ISBN 978-80-271-6716-6 (ePub), 978-80-271-6715-9 (pdf), 978-80-271-0659-2 (print).

Rejstřík

_

__init__, 94

6

69_INP, 42
69_IWD, 42

A

alokátor, 36
API, 34
aplikace
  obsluhovaná, 34
app_window, 94, 113

B

balíček
  game69.g0_demo, 45
  spuštění, 83
barva, 101
bod
  typografický, 103

C

callback, 70
central_panel, 94, 114
constants, 94, 112

D

dědění
  jmenovité, 33
  strukturální, 33
dialog
  přemístitelný, 53

E

ERROR_PREFIX, 124, 138

F

Fasáda
  návrhový vzor, 48
fokus, 69
font, 101
  systémový, 102
fronta událostí, 58
funkce
  versus metoda, 35

G

game69, 42
  BasicActions, 165
  IAction, 166
  IGame, 45
    all_actions(), 165
    basic_actions(), 165
  INamed, 166
  IPortal
    SCENARIOS(), 117
  ScenarioStep, 117
Garbage Collector, 342
geometrie
  nastavování, 62, 64
GUI, 34
gui69, 43
gui69.test, 118

H

hra
  architektura, 144
  execute_command(), 46
  is_alive(), 45
  konverzační, 40

I

IDLE, 35
IGame, 45
import
  absolutní, 96
  relativní, 96
index
  stringový, 128
initor, 36
input(), 60
interface, 33
inverze řízení, 99
IPortal, 45

J

jedináček
  třídní, 92
Jedináček, 92
JOURNAL_GEOMETRY, 247

K

konstruktor, 36
kontrakt, 33

L

lessons, 42
LEVEL_GEOMETRY, 247

M

MAIN_GEOMETRY, 151
mainloop(), 60, 62, 92
menu, 94
metoda
  versus funkce, 35
modalita, 48

narušení, 246
modul
__init__, 94
__main__, 86
app_window, 94, 113, 148
central_panel, 94, 114
constants, 94, 112, 150
game_manager, 201
gui69.test, 118
level_dialog, 248
menu, 94
panels, 94, 152
pathlib, 202
settings_manager, 283
settings_window, 272
tooltip, 318
modules, 42

N

nápověda
bublinová (tooltip), 316
nastavení
obsluhy události, 70
reakce na událost, 70
navázat
funkci na událost, 70

O

obsluha
obsluhovaná aplikace, 34
události, 73
obsluha události, 57
nastavení, 70
, 3, 441
odstavec, 134
okno
kořenové, 61
modální, 49, 245
nemodální, 49, 245
sekundární, 244
os
fdopen(), 330

P

pack(), 67
panel
centrální, 90
postranní, 90, 143
panels, 94
parametr
**options, 50
pathlib, 202
Path, 202
exists(), 202
písmo, 101
rodina, 101
řez, 103
systémové, 102
pixel, 100
fyzický, 100
logický, 100
pkg_level, 86
políčko
zaškrtávací, 221
portál, 45
posuvník, 131
obousměrná vazba, 131
orientace, 131
program
řízený událostmi, 58
programování
asynchronní, 304
synchronní, 304
proměnná
sledovací, 220
volná, 161
prostor, 40
protokol, 33
IPortal, 45
přepínač, 221
přepravka
PanelAux, 169
SidePanelsAux, 169
Příkaz
návrhový vzor, 70

Q

quit(), 60

R

reakce
na událost, 70
REPL, 35
rozhraní, 33
API, 34
signatura, 33
uživatelské, 34
rozhraní:, 33
run(), 96

Ř

řádek
logický, 134
zobrazený, 134

S

SCENARIOS, 117
scénář, 117
HAPPY, 117
SETTINGS_GEOMETRY, 272
signatura, 33
singleton, 92
smyčka událostí, 58
správce (controller), 99
správce geometrie.
správce rozmístění
správce rozmístění, 67
pack, 67
přehled, 108
strom
tříd, 61
widgetů, 62

T

tag, 134
Tcl, 38
tempfile
mkstemp(), 330
terminologie, 29
test
poloautomatický, 116
text
zalamování, 134
Tk, 38
tkinter, 39
BooleanVar, 220
Button, 159
Canvas, 355
coords(), 356
create_image(), 355
create_oval(), 355
create_rectangle(), 355
delete(), 356
itemconfig(), 356

constants, 65
DoubleVar, 220
Entry, 68, 127
  delete(), 68, 128
  get(), 68, 128
  insert(), 128
filedialog, 201, 230
  askdirectory(), 201
  askopenfilename(), 230
  asksaveasfilename(), 230
Frame, 107
IntVar, 220
LabelFrame, 151
Listbox, 266, 271
  curselection(), 271
  delete(), 271
  get(), 271
  insert(), 271
Menu, 182
  add_cascade(), 183, 222
  add_command(), 183
  add_checkbutton(), 221
  add_radiobutton(), 222
  add_separator(), 183
  delete(), 287
  entryconfig(), 223
  post(), 322
messagebox
  askokcancel(), 50
  askquestion(), 50
  askretrycancel(), 50
  askyesno(), 50
  askyesnocancel(), 50
  showerror(), 49
  showinfo(), 49
  showwarning(), 49
Misc, 61
  after(), 305
  after_cancel(), 318
  bind(), 70
  clipboard_append(), 321
  clipboard_clear(), 321
  config(), 78
  destroy(), 93, 152
  focus_set(), 69
  grab_release(), 246
  grab_set(), 246
  grid(), 110
  grid_columnconfigure(), 109
  grid_rowconfigure(), 109
  selection_get(), 321
  update(), 93
  update_idletasks(), 93
  wait_window(), 245
  winfo_children(), 152
  winfo_rootx(), 317
  winfo_rooty(), 317
  winfo_screenheight(), 318
  winfo_screenwidth(), 318
pack(), 67
PhotoImage, 341
  subsample(), 341
  zoom(), 341
Scrollbar, 131
  set(), 131
simpledialog
  askfloat(), 51
  askinteger(), 51
  askstring(), 51
StringVar, 220
Text, 76, 133
  delete(), 136
  insert(), 77, 136
  konstruktor, 135
  see(), 77, 136
  selection_get(), 325
  tag_config(), 136
  yview(), 131
Tk, 62
  lift(), 62
  mainloop(), 60, 62, 92
  quit(), 60
Toplevel, 244
ttk, 39, 166
  Combobox, 167
    delete(), 167
    get(), 167
    icursor(), 168
    set(), 167
  Checkbutton, 265, 268
  Notebook, 265, 268
    add(), 268
  Radiobutton, 265, 269
  Scale, 265, 269
  Separator, 247
  Spinbox, 265, 270
  Variable
    trace_add(), 224
  Wm
    deiconify(), 253
    geometry(), 62
    iconphoto(), 348
    overrideredirect(), 317
    protocol(), 253
    resizable(), 245
    title(), 62
    transient(), 245
    withdraw(), 253
třída
  Panel, 152, 155
  Tooltip, 318
TUI, 34
typing
  Protocol, 33

U

událost, 57
  <Button-1>, 322
  <Button-3>, 322
  <Double-1>, 280
  <Enter>, 316
  <Key>, 80
  <Leave>, 316
  <Return>, 73
  fronta, 58
  nastavení obsluhy události, 70
  obsluha, 57
    prázdná, 80
  obsluha události, 73
  program řízený událostmi, 58
  přerušení (break), 80
  smyčka událostí, 58
UI, 34
util, 42
uzávěr (closure), 161

V

volání
  zpětné, 70
výpis
  YYY
    xxx, 445, 447, 449, 451, 454
vysílač události, 57
vzor
  návrhový
    Fasáda, 48
    Jedináček, 92
    Prostřednik, 95
    Příkaz, 70

W

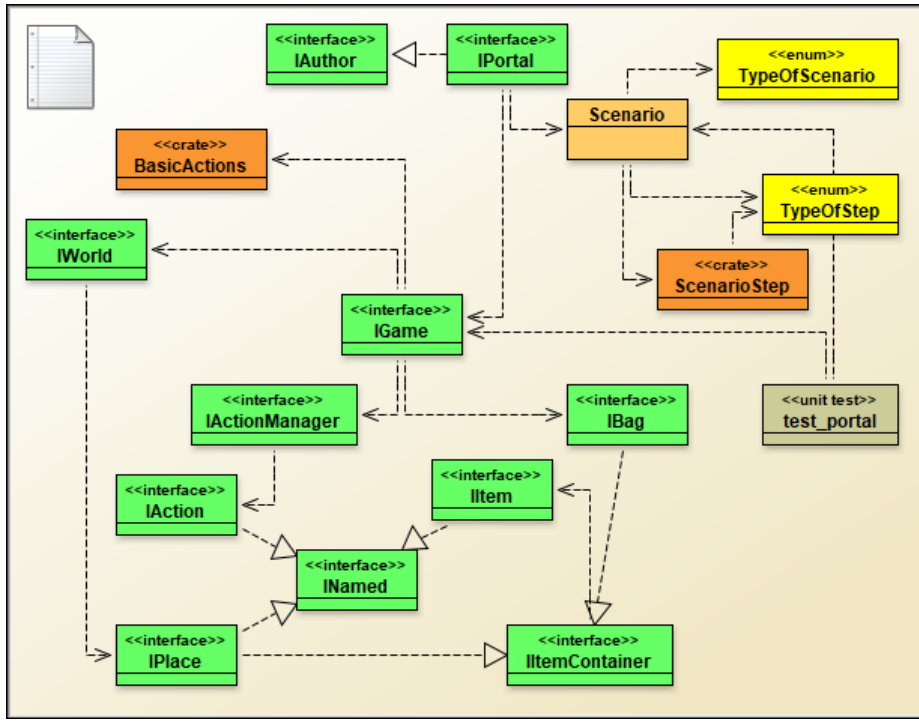
wait_variable(), 60
wait_visibility(), 60
wait_window(), 60
widget, 38, 56
  logický stav, 67
  orientace, 67
  směr roztahování, 66
  ukotvení, 65
  zalomování textu, 66

Část E
Přílohy a seznamy

Příloha A API hry, pro niž **tvoříme** GUI 422
Seznam výpisů programů 424
Seznam obrá**zků** 430
Seznam odboček – podšeděných bloků 431

Příloha A
API hry, pro niž tvoříme GUI

Základní představu o architektuře aplikace, pro niž budeme vytvářet GUI, by měl poskytnout obrázek A.1 s diagramem tříd popisujícím koncept API.



Obrázek A.1:
Ideový diagram tříd aplikace, pro niž vytváříme GUI

Těm, co si netykají s UML bych jenom připomněl, že v diagramu tříd je u některých tříd použit tzv. **stereotyp**, což je text umístěný ve «francouzských uvozovkách». Ten se používá k zvýraznění nějaké speciální vlastnosti, kterou daná entita má.

V našem diagramu jsou použity čtyři stereotypy:

`«crate»` Třída vyhovující návrhovému vzoru *Přepravka* (*crate*), který definuje všechny atributy dané třídy jako veřejné konstanty inicializované při vytváření instance. V *Pythonu* tomu vyhovují pojmenované n-tice a datové třídy (třídy s dekorované `@dataclass`).

`«enum»` Výčtový typ – třída mající konečný počet předem známých instancí.

`«interface»` Datový typ reprezentující rozhraní. V *Pythonu* se rozhraní deklarují jako protokoly, tj. třídy, jejichž bezprostředním předkem je abstraktní třída `typing.Protocol`.

`«unit test»` Třída, jejíž instance realizují tzv. **jednotkové testy**, tj. texty, které testují zadané programové jednotky – většinou třídy.

Uspořádání těchto tříd v balíčku `game69.api` je následující:

- Třída `BacicActions` je definována v initoru balíčku – modulu `game69.api.__init__`.
- Všechny protokoly jsou definovány v modulu `game69.api.protocols`. Pro jistotu bych dodal, že v *Pythonu* mohou v případě potřeby implementovat protokol nejenom třídy, ale i moduly.
- Třídy `Scenario` a `ScenarioStep` jsou definovány v modulu `game69.api.scenario`.
- Třídy `TypeOfScenario` a `TypeOfStep` najdete v modulu `game69.api.scen_types`, kde je s nimi definována i pomocná třída `SubtypeOfStep`, kterou ale používá pouze testovací program, a tak jsem ji v diagramu neuváděl.
- Poslední nezmíněnou entitou v diagramu na obrázku A.1 je `test_portal`, což není v našem projektu třída, ale modul v balíčku `game69.api.tests`. Te je volán funkcí `test()` v initoru balíčku, prostřednictvím níž je možno testovat vyvíjenou hru v jednotlivých etapách vývoje.

Nebudu zde detailně rozebírat funkci jednotlivých protokolů a tříd, protože zájemci najdou ve zdrojovém kódu podrobné dokumentační komentáře.

Seznam výpisů programů

Výpis 1.1: Funkce versus metoda ... 35
Výpis 2.1: Definice funkcí z modulu `gui69.v01_text_ui.__init__`: funkce `run()` spouštějící hru a funkce `multirun()` umožňující opakované spuštění hry ... 46
Výpis 2.2: Použití funkce `run()` k demonstraci spuštění hry v REPL ... 47
Výpis 2.3: Volání informačních funkcí modulu `tkinter.messagebox` ... 49
Výpis 2.4: Volání zjišťovacích funkcí modulu `tkinter.simpledialog` ... 51
Výpis 2.5: Definice funkce `run()` a `multirun()` z balíčku `gui69.v02_primitive` spouštějící hru ovládanou prostřednictvím primitivního GUI ... 52
Výpis 2.6: Funkce `use_standard()` a `use_sticky()` z balíčku `gui69.v02_primitive` nastavující sadu dialogů používaných v primitivním GUI ... 53
Výpis 2.7: Demonstrační zavolání funkce `run()` z balíčku `gui69.v02_primitive` ke spuštění hry v REPL ... 54
Výpis 3.1: Symbolické naznačení principu funkce smyčky událostí nad jejich frontou ... 59
Výpis 3.2: Tělo modulu `modules.m03_01_Tk` demonstrující vytvoření instance třídy `tkinter.Tk` a jednoduché manipulace s ní ... 63
Výpis 3.3: Vložení widgetu typu `Entry` do okna a nastavení reakce na zadaná data v modulu `modules.m03_02_Entry` ... 72
Výpis 3.4: Experimenty s modulem `modules.m03_02_Entry` ... 75
Výpis 3.5: Maximálně jednoduché, ale netriviální GUI dané hry v modulu `gui69.v03_text_entry.__init__` ... 79
Výpis 3.6: Přímé spuštění initoru `gui69.v03_text_entry.__init__` ... 84
Výpis 4.1: Definice těla initoru balíčku – modulu `gui69.v04_central.__init__` ... 97
Výpis 4.2: Definice těla spouštěcího modulu `gui69.v04_central.__main__` ... 98
Výpis 4.3: Tělo modulu `gui69.v04_central.constants` ... 112
Výpis 4.4: Definice těla modulu `gui69.v04_central.app_window` ... 113
Výpis 4.5: Definice těla modulu `gui69.v04_central.central_panel` ... 115
Výpis 4.6: Definice funkcí modulu `gui69.test` realizujících poloautomatický test ... 118
Výpis 4.7: Definice druhé části modulu `gui69.test` řešící spuštění testu z příkazového řádku ... 120
Výpis 5.1: Doplnění nových konstant do modulu `gui69.v05_formatted.constants` ... 124
Výpis 5.2: Definice importů a deklarace globálních proměnných v modulu `gui69.v05_formatted.central_panel` ... 125
Výpis 5.3: Definice funkce `initialize()` v modulu `gui69.v05_formatted.central_panel` ... 126
Výpis 5.4: Definice funkce `_prepare_input_pane()` v modulu `gui69.v05_formatted.central_panel` ... 129
Výpis 5.5: Definice dočasné podoby funkce `_prepare_output_pane()` v modulu `gui69.v05_formatted.central_panel` ... 130

Výpis 5.6: Definice finální podoby funkce `_prepare_output_pane()` v modulu `gui69.v05_formatted.central_panel` ... 132
Výpis 5.7: Definice finální podoby funkce `_define_styles()` v modulu `gui69.v05_formatted.central_panel` ... 137
Výpis 5.8: Definice finální podoby funkce `game_output()` v modulu `game69.v05_formatted.central_panel` ... 139
Výpis 6.1: Společné importy v modulu `gui69.v06_panels.__init__` ... 144
Výpis 6.2: Definice funkce `_get_names()` v modulu `gui69.v06_panels.__init__` ... 145
Výpis 6.3: Definice funkce `run()` v modulu `gui69.v06_panels.__init__` ... 146
Výpis 6.4: Definice funkce `process_command()` v modulu `gui69.v06_panels.__init__` ... 147
Výpis 6.5: Definice funkce `initialize()` v modulu `gui69.v06_panels.app_window` ... 149
Výpis 6.6: Definice funkce `_configure_main_window()` v modulu `gui69.v06_panels.app_window` ... 150
Výpis 6.7: Aktualizovaný počátek těla modulu `gui69.v06_panels.constants` ... 151
Výpis 6.8: Definice metody `initialize()` v modulu `gui69.v06_panels.panels` ... 154
Výpis 6.9: Definice třídy `Panel` v modulu `gui69.v06_panels.panels` ... 155
Výpis 7.1: Demonstrace chování volných proměnných v uzávěrech ... 162
Výpis 7.2: Demonstrace uložení aktuální hodnoty volné proměnné ... 162
Výpis 7.3: Definice třídy `BasicAction` v modulu `game69.api` ... 165
Výpis 7.4: Doplnění importů a definice tříd `PanelAux` a `SidePanelsAux` v modulu `gui69.v07_buttons.__init__` ... 169
Výpis 7.5: Upravená definice funkce `run()` v modulu `gui69.v07_buttons.__init__` ... 171
Výpis 7.6: Definice funkce `initialize()` v modulu `gui69.v07_buttons.app_window` ... 173
Výpis 7.7: Upravený počátek těla modulu `gui69.v07_buttons.central_panel` ... 174
Výpis 7.8: Aktualizované definice dalších funkcí modulu `gui69.v07_buttons.central_panel` ... 175
Výpis 7.9: Definice funkce `initialize()` v modulu `gui69.v07_buttons.panels` ... 176
Výpis 7.10: Upravená definice třídy `Panel` v modulu `gui69.v07_buttons.panels` ... 177
Výpis 8.1: Úvodní část těla modulu `gui69.v08_menu.menu` ... 185
Výpis 8.2: Dokončení těla modulu `gui69.v08_menu.menu` ... 187
Výpis 8.3: Úprava funkce `initialize()` a definice funkce `clear_output()` v modulu `gui69.v08_menu.central_panel` ... 188
Výpis 8.4: Rozšíření importů v modulu `gui69.v08_menu.app_window` ... 189
Výpis 8.5: Úprava funkce `initialize()` v modulu `gui69.v08_menu.app_window` ... 190
Výpis 8.6: Definice globálních proměnných a přepravek v modulu `gui69.v08_menu.__init__` ... 191
Výpis 8.7: Aktualizovaná definice funkce `run()` v modulu `gui69.v08_menu.__init__` ... 193
Výpis 8.8: Aktualizovaná definice funkce `process_command()` v modulu `gui69.v08_menu.__init__` ... 193
Výpis 8.9: Definice pomocných funkcí v modulu `gui69.v08_menu.__init__` ... 194
Výpis 9.1: Začátek těla modulu `gui69.v09_files.game_manager` ... 203
Výpis 9.2: Definice funkce `_load_portal_from_path()` v modulu `gui69.v09_files.game_manager` ... 204
Výpis 9.3: Aktualizace funkce `_add_game_menu()` v modulu `gui69.v09_files.menu` ... 206
Výpis 9.4: Aktualizovaná definice funkcí `initialize()` a `update_actions()` v modulu `gui69.v09_files.central_panel` ... 207

Výpis 9.5: Aktualizovaná definice funkce `_prepare_input_pane()` v modulu `gui69.v09_files.central_panel` ... 208
Výpis 9.6: Upravený konec definice funkce `game_output()` v modulu `gui69.v09_files.central_panel` ... 209
Výpis 9.7: Aktualizovaná funkce `initialize()` v modulu `gui69.v09_files.app_window` ... 210
Výpis 9.8: Aktualizované importy a globální deklarace v modulu `game69.v09_files.__init__` ... 211
Výpis 9.9: Aktualizované definice funkce `run()` v modulu `game69.v09_files.__init__` ... 212
Výpis 9.10: Aktualizovaná funkce `_prepare_side_aux()` v modulu `game69.v09_files.__init__` ... 214
Výpis 9.11: Aktualizovaná funkce `_prepare_menu_aux()` a nově zavedené pomocné funkce v modulu `game69.v09_files.__init__` ... 215
Výpis 9.12: Úprava metody `update_content()` v modulu `game69.v09_files.panels` ... 216
Výpis 10.1: Upravená funkce `initialize()` v modulu `gui69.v10_journal.menu` ... 225
Výpis 10.2: Upravená funkce `_add_tools_menu()` v modulu `gui69.v10_journal.menu` ... 226
Výpis 10.3: Definice funkce `_add_journal_submenu()` v modulu `gui69.v10_journal.menu` ... 228
Výpis 10.4: Definice funkce `_show_journal()` v modulu `gui69.v10_journal.menu` ... 229
Výpis 10.5: Začátek těla modulu `gui69.v10_journal.journal_manager` ... 232
Výpis 10.6: Funkce `set_log_level()` a `toggle_log()` v modulu `gui69.v10_journal.journal_manager` ... 233
Výpis 10.7: Definice funkcí `register_display_listener()` a `log_event()` v modulu `gui69.v10_journal.journal_manager` ... 235
Výpis 10.8: Definice funkce `log_delimiter()` v modulu `gui69.v10_journal.journal_manager` ... 236
Výpis 10.9: Aktualizované importy a globální deklarace v modulu `game69.v10_journal.__init__` ... 237
Výpis 10.10: Aktualizované definice přepravky `MenuAux` a funkce `run()` v modulu `game69.v10_journal.__init__` ... 238
Výpis 10.11: Aktualizovaná definice funkce `process_command()` v modulu `game69.v10_journal.__init__` ... 239
Výpis 10.12: Aktualizované pomocné funkce v modulu `game69.v10_journal.__init__` ... 240
Výpis 10.13: Záznam průběhu hry v souboru na disku ... 241
Výpis 11.1: Aktualizovaný počátek těla modulu `gui69.v11_windows.constants` ... 247
Výpis 11.2: Definice funkce `show()` pro výběr podrobnosti žurnálu v modulu `gui69.v11_windows.level_dialog` ... 249
Výpis 11.3: Aktualizovaná definice funkce `toggle_log()` v modulu `gui69.v11_windows.journal_manager` ... 251
Výpis 11.4: Funkce `get_content()` přidaná do modulu `gui69.v11_windows.journal_manager` ... 252
Výpis 11.5: První část těla modulu `gui69.v11_windows.journal_window` ... 256
Výpis 11.6: Druhá část těla modulu `gui69.v11_windows.journal_window` ... 257
Výpis 11.7: Aktualizovaná definice funkce `_add_journal_submenu()` v modulu `gui69.v11_windows.menu` ... 258
Výpis 11.8: Úvodní importy a globální proměnné v modulu `gui69.v11_windows.__init__` ... 259
Výpis 11.9: Úprava funkce `run()` v modulu `gui69.v11_windows.__init__` ... 260

Výpis 11.10: Úprava funkce `_prepare_menu_aux()` v modulu `gui69.v11_windows.__init__` 261
Výpis 12.1: Aktualizovaný počátek těla modulu `gui69.v12_dialog.constants` 272
Výpis 12.2: Začátek těla modulu s definicí funkce `show()` v modulu `gui69.v12_dialog.settings_window` 273
Výpis 12.3: Funkce `_create_appearance_tab()` v modulu `gui69.v12_dialog.settings_window` 275
Výpis 12.4: Funkce `_create_journal_tab()` v modulu `gui69.v12_dialog.settings_window` 277
Výpis 12.5: Funkce `_create_info_tab()` v modulu `gui69.v12_dialog.settings_window` 279
Výpis 12.6: Funkce `_create_stats_tab()` v modulu `gui69.v12_dialog.settings_window` 280
Výpis 12.7: Úvodní deklarace a definice funkce `initialize()` v modulu `gui69.v12_dialog.settings_manager` 284
Výpis 12.8: Definice podpůrných funkcí v modulu `gui69.v12_dialog.settings_manager` 285
Výpis 12.9: Aktualizovaná funkce `initialize()` v modulu `gui69.v12_dialog.menu` 287
Výpis 12.10: Aktualizovaná funkce `_add_game_menu()` v modulu `gui69.v12_dialog.menu` 288
Výpis 12.11: Aktualizovaná funkce `_add_tools_menu()` v modulu `gui69.v12_dialog.menu` 289
Výpis 12.12: Aktualizovaná funkce `_add_journal_submenu()` v modulu `gui69.v12_dialog.menu` 290
Výpis 12.13: Aktualizované definice globálních proměnných v modulu `gui69.v12_dialog.journal_manager` 291
Výpis 12.14: Definice funkcí `set_use_level_dialog()` a `toggle_log()` v modulu `gui69.v12_dialog.journal_manager` 292
Výpis 12.15: Úpravy v modulu `gui69.v12_dialog.journal_window` 293
Výpis 12.16: Importy a deklarace v modulu `gui69.v12_dialog.__init__` 294
Výpis 12.17: Definice přepravky `MenuAux` v modulu `gui69.v12_dialog.__init__` 294
Výpis 12.18: Aktualizovaná funkce `run()` v modulu `gui69.v12_dialog.__init__` 295
Výpis 12.19: Funkce `_toggle_journal_window()` v modulu `gui69.v12_dialog.__init__` 296
Výpis 12.20: Aktualizace přípravy nabídky a potlačení vyskakovacího dialogu ve funkci `_prepare_menu_aux()` v modulu `gui69.v12_dialog.__init__` 298
Výpis 12.21: Úprava spouštěcích funkcí v modulu `gui69.v12_dialog.__init__` 299
Výpis 13.1: Úvodní deklarace a definice funkce `initialize()` v modulu `gui69.v13_info.info_panel` 306
Výpis 13.2: Definice funkcí `reset_timer()` a `_tick()` s asynchronní smyčkou v modulu `gui69.v13_info.info_panel` 308
Výpis 13.3: Rozšíření importů v modulu `gui69.v13_info.app_window` 309
Výpis 13.4: Integrace informačního panelu ve funkci `initialize()` v modulu `gui69.v13_info.app_window` 310
Výpis 13.5: Aktualizované importy v modulu `gui69.v13_info.__init__` 311
Výpis 13.6: Reset stopek ve funkci `_start_new_game()` modulu `gui69.v13_info.__init__` 311
Výpis 13.7: Zpřístupnění zaškrtávacích políček ve funkci `_create_info_tab()` v modulu `gui69.v13_info.settings_window` 312

Výpis 14.1: Definice třídy Tooltip v modulu gui69.v14_help.tooltip – první část 319
Výpis 14.2: Definice metody _show() ve třídě Tooltip v modulu gui69.v14_help.tooltip 320
Výpis 14.3: Aktualizovaná sada importů v modulu gui69.v14_help.panels 323
Výpis 14.4: Úprava metody update_content() ve třídě Panel v modulu gui69.v14_help.panels 323
Výpis 14.5: Tvorba kontextové nabídky v modulu gui69.v14_help.central_panel 324
Výpis 14.6: Tvorba kontextové nabídky ve funkci show() v modulu gui69.v14_help.journal_window 326
Výpis 14.7: Doplněná sekce s názvy souborů v modulu gui69.v14_help.constants 327
Výpis 14.8: Aktualizovaná deklarace přepravky MenuAux v modulu gui69.v14_help.__init__ 331
Výpis 14.9: Aktualizovaná funkce _prepare_menu_aux() v modulu gui69.v14_help.__init__ 331
Výpis 14.10: Nová funkce _show_game_help() v modulu gui69.v14_help.__init__ chráněná proti problémům při distribuci 333
Výpis 14.11: Úprava funkce _add_help_menu() v modulu gui69.v14_help.menu 334
Výpis 15.1: Úprava vnitřního stavu a spouštěcích funkcí v modulu gui69.v15_images.game_manager 344
Výpis 15.2: Definice funkcí _setup_graphics() a get_image() pro správu grafiky v modulu gui69.v15_images.game_manager 346
Výpis 15.3: Aktualizace importů v modulu gui69.v15_images.panels 347
Výpis 15.4: Aktualizace metody update_content() ve třídě Panel v modulu gui69.v15_images.panels 347
Výpis 15.5: Doplnění konstanty ICON_NAME v modulu gui69.v15_images.constants 349
Výpis 15.6: Úprava funkce _start_new_game() v modulu gui69.v15_images.__init__ 349
Výpis 16.1: Doplnění dalších konstant v modulu gui69.v16_canvas.constants 357
Výpis 16.2: Úprava vnitřního stavu v modulu gui69.v16_canvas.game_manager 359
Výpis 16.3: Rozšíření funkce _setup_graphics() o správu souřadnic v modulu gui69.v16_canvas.game_manager 360
Výpis 16.4: Importy a globální proměnné v modulu gui69.v16_canvas.map_window 361
Výpis 16.5: Definice funkcí show(), hide() a _on_close() v modulu gui69.v16_canvas.map_window 362
Výpis 16.6: Definice funkce _initialize_content() v modulu gui69.v16_canvas.map_window 363
Výpis 16.7: Definice funkce update_map() v modulu gui69.v16_canvas.map_window 365
Výpis 16.8: Doplnění interaktivity definicí funkce _on_canvas_click() v modulu gui69.v16_canvas.map_window 367
Výpis 16.9: Definice funkce _create_info_tab() v modulu gui69.v16_canvas.settings_window 368
Výpis 16.10: Zpřístupnění položky plánku ve funkci _add_tools_menu() v modulu gui69.v16_canvas.menu 369
Výpis 16.11: Aktualizace typu přepravky MenuAux v modulu gui69.v16_canvas.__init__ 370
Výpis 16.12: Aktualizace mapy ve funkci process_command() v modulu gui69.v16_canvas.__init__ 371
Výpis 16.13: Definice funkce _toggle_map_window() v modulu gui69.v16_canvas.__init__ 371

Výpis 16.14: Úprava funkce `_prepare_menu_aux()` v modulu `gui69.v16_canvas.__init__` ... 373
Výpis 17.1: Úprava modulu `gui69.v17_custom.constants` ... 381
Výpis 17.2: Rozšíření importů v modulu `gui69.v17_custom.app_window` ... 382
Výpis 17.3: Úprava funkce `initialize()` a vytvoření kořenového okna v modulu `gui69.v17_custom.app_window` ... 383
Výpis 17.4: Upravená definice funkce `_configure_main_window()` v modulu `gui69.v17_custom.app_window` ... 384
Výpis 17.5: Úprava importů a globálních proměnných v modulu `gui69.v17_custom.game_manager` ... 386
Výpis 17.6: Úprava funkce `_setup_graphics()` v modulu `gui69.v17_custom.game_manager` ... 387
Výpis 17.7: Pomocná funkce `_load_pil_image()` v modulu `gui69.v17_custom.game_manager` ... 387
Výpis 17.8: Funkce `get_image()` v modulu `gui69.v17_custom.game_manager` ... 389
Výpis 17.9: Funkce `get_ctk_image()` v modulu `gui69.v17_custom.game_manager` ... 390
Výpis 17.10: Předání rozměrů plátna ve funkci `_initialize_content()` v modulu `gui69.v17_custom.map_window` ... 391
Výpis 17.11: Předání rozměrů plátna ve funkci `update_map()` v modulu `gui69.v17_custom.map_window` ... 392
Výpis 17.12: Doplnění importů a úprava funkce `initialize()` v modulu `gui69.v17_custom.panels` ... 395
Výpis 17.13: Výměna rodiče třídy `Panel` a její initor v modulu `gui69.v17_custom.panels` ... 396
Výpis 17.14: Metoda `update_content()` ve třídě `Panel` v modulu `gui69.v17_custom.panels` ... 397
Výpis 17.15: Rozšíření importů a globálních proměnných v modulu `gui69.v17_custom.central_panel` ... 399
Výpis 17.16: Úprava funkce `update_actions()` v modulu `gui69.v17_custom.central_panel` ... 400
Výpis 17.17: Úprava funkce `_prepare_input_pane()` v modulu `gui69.v17_custom.central_panel` ... 401
Výpis 17.18: Úprava funkcí `game_output()` v modulu `gui69.v17_custom.central_panel` ... 402
Výpis 17.19: Úprava funkce `show()` v modulu `gui69.v17_custom.settings_window` ... 404
Výpis 17.20: Úprava funkce `_create_appearance_tab()` v modulu `gui69.v17_custom.settings_window` ... 406
Výpis 17.21: Upravená funkce `_create_journal_tab()` v modulu `gui69.v17_custom.settings_window` ... 407
Výpis 17.22: Úprava funkce `_create_info_tab()`v modulu `gui69.v17_custom.settings_window` ... 408
Výpis 17.23: Úprava funkce `_create_stats_tab()` v modulu `gui69.v17_custom.settings_window` ... 408
Výpis 17.24: Úprava funkce `on_font_change()` v modulu `gui69.v17_custom.settings_manager` ... 410
Výpis 17.25: Úprava funkce `run()` v modulu `gui69.v17_custom.__init__` ... 411

Seznam obrázků

Obrázek 2.1: Dialogové okno s výzvou k zadání údajů....50
Obrázek 2.2: Dialogové okno s výzvou k zadání údajů....51
Obrázek 2.3: Upozornění na chybné zadání číselné hodnoty....51
Obrázek 2.4: Průběh komunikace s hrou při použití primitivního GUI....54
Obrázek 3.1: Hraní hry s jednoduchým netriviálním GUI....83
Obrázek 4.1: Cílová podoba GUI hry, které dosáhneme v kapitole 16....91
Obrázek 4.2: Testování funkce GUI v poloautomatickém režimu; uživatel právě zadává vlastní příkaz. 119
Obrázek 5.1: Obrázek průběhu testu se špatnými zadáními příkazů a viditelným posuvníkem 140
Obrázek 6.1: Diagram tříd aplikace, pro niž vytvářím GUI....143
Obrázek 6.2: Aplikační okno verze `v06_panels` s postranními panely 156
Obrázek 7.1: Aplikační okno verze `v07_buttons` s postranními panely tlačítek....179
Obrázek 8.1: Aplikační okno verze `v08_menu` s rozbalenou nabídkou **Nápověda** 195
Obrázek 9.1: Aplikační okno verze `v09_files` při zadávání jiné hry 217
Obrázek 11.1: Dialogové okno pro nastavení úrovně podrobností 250
Obrázek 11.2: Aplikační okno verze `v11_windows` při zobrazování žurnálu....262
Obrázek 12.1: Podoby karet dialogového okna **Nastavení aplikace** a aplikační okno verze `v12_dialog` při výběru následně spouštěné hry 301
Obrázek 13.1: Aplikační okno verze `v13_info` po začlenění informačního panelu....313
Obrázek 14.1: Aplikační okno verze `v14_help` s výběrem textu a vyskakujícím popiskem 335
Obrázek 15.1: Aplikační okno verze `v15_images` zobrazující obrázky předmětů a sousedních prostorů....350
Obrázek 16.1: Obrázek aplikačního okna částečně překrytý oknem s plánkem hry....374
Obrázek 17.1: Aplikační okno zobrazující obrázky předmětů a sousedních prostorů....412

Seznam odboček – podšeděných bloků

Odbočka – podšeděný blok 30
Návrhový vzor Fasáda 48
Návrhový vzor Prostředník (Mediator) 95
Fyzický versus logický pixel 100
Co jsou to typografické body 103
Kód, který páchne (Code Smell) 283
Jak se měnila plánovaná funkcionalita 327
Úskalí rolovacích panelů v knihovně *Tkinter* 348
Historické ohlédnutí 354