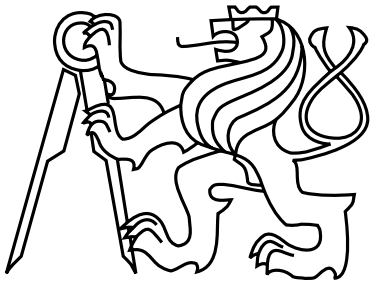
České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Bakalářská práce

# Konfigurovatelný generátor základních struktur VHDL kódu

*Jan Matějů*

Vedoucí práce: Ing. Pavel Kubalík

Studijní program: Elektrotechnika a informatika, dobíhající Bakalářské

Obor: Výpočetní technika

30. května 2010

# Poděkování

Na tomto místě bych rád poděkoval svému vedoucímu Ing. Pavlu Kubalíkovi za trpělivost a cenné připomínky a dále svým rodičům a kolegům za jejich podporu.

# Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Dobříši dne 26. 5. 2010 .....

# Abstract

This bachelor thesis deals with evolution of application called Configurable generator of basic VHDL structures (VHDL SGen). This application defines templates of VHDL structures, which alows us comfortly generate most used VHDL structures, mainly counters and automats. Application can also work with VHDL testbench templates from which can be created VHDL testbenches of existing projects.

For the implementation was chosen programming language called Java, namely the SE version. GUI of the application is created using a graphical library JFC Swing, which was chosen for its independence from the operating system. Within this work was developed a simple VHDL parser, thru which runs the analysis of VHDL structures and templates.

The work includes a research of existing solutions, especially the Xilinx ISE tools and plugin for the editor Notepad++.

# Abstrakt

Tato práce mapuje vývoj aplikace Konfigurovatelný generátor základních struktur VHDL kódu (VHDL SGen). V aplikaci jsou definovány šablony VHDL struktur, na základě kterých lze snadno vytvářet základní struktury VHDL kódu, převážně ty nejpoužívanější, a to čítače a automaty. Aplikace dále umožňuje pracovat s šablonami VHDL testbenchů, ze kterých lze vytvářet testbenche existujících VHDL projektů.

Pro realizaci byl vybrán jazyk Java, konkrétně jeho SE verze. GUI tvoří grafická knihovna JFC Swing, která byla zvolena pro svou nezávislost na operačním systému. V rámci práce vznikl i jednoduchý parser jazyka VHDL, pomocí kterého probíhá analýza VHDL struktur a šablon.

Práce obsahuje také rešerši stávajících řešení, a to konkrétně nástrojů Xilinx ISE a pluginu do programu Notepad++.

# Obsah

# Seznam obrázků

# Kapitola 1

# Úvod

Cílem této práce je vytvořit nástroj pro generování základních struktur VHDL kódu. Navržený nástroj bude umožňovat vytvářet a zpracovávat šablony VHDL struktur. Na základě šablon bude možné vytvořit například: základní strukturu entity, automatu, čítače. Nástroj bude obsahovat i jednoduchý rozpoznávač VHDL kódu, díky kterému bude možné na základě vyplněné entity automaticky vytvořit testbench. Součástí práce bude i rešerše stávajících řešení a několik šablon ukazujících práci s nástrojem.

Jazyk VHDL je jedním z nejpoužívanějších jazyků pro popis hardwarových struktur hradlových polí. Je definován standardem IEEE 1076 z roku 1987. Umožňuje návrh jak logických tak i sekvenčních struktur[8, 7]. Jeho hlavní výhodou je jeho univerzálnost. Při návrhu hardwaru se v naprosté většině projektů používají základní struktury, a to především čítač a automat. Ty mají vždy stejný základ a funkčnost, která se v konkrétních implementací líší zpravidla pouze nepatrně. Například automat vždy obsahuje funkci přechodu mezi stavy a výstupní funkci.

Další typickou a nezbytnou věcí při návrhu logických obvodů před jejich fyzickou realizací je testování funkčnosti pomocí testbenchů. Tvorba testbenche je rutinní záležitost. V těle architektury testbenche musíme:

- přepsat tělo testované entity (porty a generiky) jako komponentu
- podle portů vytvořit výčet signálů
- namapovat komponentu na signály

Tato aplikace by měla návrhářům logických obvodů usnadnit rutinní práci při tvorbě základních struktur a jejich následném testování. Bude umožňovat vytvářet univerzální šablony VHDL struktur a testbenchů, ze kterých lze snadno vytvořit požadovanou strukturu/testbench.

# Kapitola 2

# Popis problému, specifikace cíle

## 2.1 Deklarace záměru

Navrhněte nástroj pro generování zkladních struktur VHDL kódu. Navržený nástroj bude umožňovat vytvářet a zpracovávat šablony VHDL struktur. Na základě šablon bude možné vytvořit například: základní strukturu entity, automatu, čítače. Nástroj bude obsahovat i jednoduchý rozpoznávač VHDL kódu, díky kterému bude možné na základě vyplněné entity automaticky vytvořit testbench. Součástí práce bude i rešerše stávajících řešení a několik šablon ukazujících práci s nástrojem.

## 2.2 Odborný článek

Cílem projektu je vytvořit nástroj, který umožní vývojářům hardwaru usnadnit a zrychlit rutinní práci. Aplikace bude nabízet dvě hlavní funkce, a to generování základních VHDL struktur a generování testbenchů. Obojí bude realizováno pomocí šablon, které si uživatel bude moci sám vytvářet.

Generování základních struktur bude co nejvíce usnadňovat opakující se práci. Neměla by chybět možnost změnit na více místech kódu nějakou část názvu pouze jedním přepsáním onoho názvu. Mějme například entitu „cntr“ obsahující následující porty:

```
cntr_en, cntr_up, cntr_res
```

Pokud se vývojář rozhodne změnit název entity na „citac“, bude pravděpodobně chtít, aby se změnily i názvy jednotlivých portů. Zavede-li se nějaká *globální proměnná*[1], na které bude část názvů portů závislá, stačí potom například v nějakém dialogu změnit „cntr“ na „citac“. Tím se nahradí všechny části názvu, které si v šabloně označíme. Další funkcí, která by byla vhodná, je automatické přídání větví `when` příkazu `case` v závislosti na konkrétním typu nebo signálu. Kvůli tomu bude třeba implementovat analyzátor kódu, který potřebné typy/signály z kódu získá. Tato funkce lze ještě vylepšit tím, že se ke každé větvi přidá předem definovaný kus kódu, společný pro všechny větve.

[1] Proměnná, která bude definována pro celou šablonu.

Generování testbenche bude uskutečněno také pomocí šablony. Uživatel by v šabloně měl mít možnost definovat, kam přesně bude umístěna komponenta testované VHDL struktury. Dále bude moci vybrat místo, kam se vloží signály a signálová přiřazení. Tvorba testbenche se neobejde bez znalosti portů testované entity. Opět se tedy neobejdeme bez analyzátoru kódu, jako v případě vkládání větví `when` příkazu `case`.

Velký důraz je kladen na zachování původní podoby a formátování šablony. Aplikace nesmí formátovat vkládaná data podle sebe, ale musí dodržovat uživatelem definovaná odsazení, komentáře a pod. Pokud vývojář umístí značku pro vložení výčtu portů do komponenty testbenche, musí být všechny porty vloženy se stejným odsazením, jaké má původní značka.

V dnešní době je kladem velký důraz na multiplatformovost aplikace, a to minimálně podpora operačního systému Windows a operačních systémů UNIXového typu. Je proto velmi vhodné zvolit pro vývoj aplikace programovací jazyk, který je pro tvorbu takových aplikací přímo stvořem. Jedním z nejužívanějších jazyků pro tvorbu takových aplikací je bezesporu jazyk Java. Díky JVM je minimálně závislý na OS. Navíc má vlastní grafickou knihovnu SWING, která se skládá z takzvaných *lehkých komponent*[2], tedy komponent nezávislých na operačním systému. O jejich podobu a vykreslování se stará samotná Java. Aplikace potom vypadá na každém operačním systému téměř totožně. Knihovna SWING je tedy pro multiplatformovou aplikaci ideální.

## 2.3 Rešerše stávajících řešení

Pro rešerši jsem vybíral z několika programů a pluginů. Nakonec jsem zvolil dva projekty: komerční Xilinx ISE a nekomerční VHDL plugin do programu Notepad++. Xilinx ISE jsem vybral proto, že je to velmi silný a hojně užívaný nástroj určený k návrhu a testování hardwaru. Je na něm také vedena výuka na této škole. VHDL plugin do programu Notepad++ mi byl doporučen Ing. Kubalíkem jako ukázka rozpracovaného projektu se stejným cílem, jaký má tato práce. Podívejme se tedy na jednotlivé nástroje podrobněji.

### 2.3.1 Xilinx ISE

Xilinx ISE je gigantický projekt firmy Xilinx. K datu vzniku této práce byla k dispozici již jeho 12. verze[3]. Je na něm vedena výuka na naší škole, proto jsem ho zvolil jako zástupce velkých komerčních projektů. Je to sice komerční projekt, ale existuje možnost stáhnout omezenou verzi WebPack zdarma. Pro její stažení ovšem musíme být zaregistrováni. Pokud bychom chtěli placenou verzi, její cena začíná na 3000$. Xilinx ISE je vyvíjen jak pro OS Windows, tak pro OS Linux.

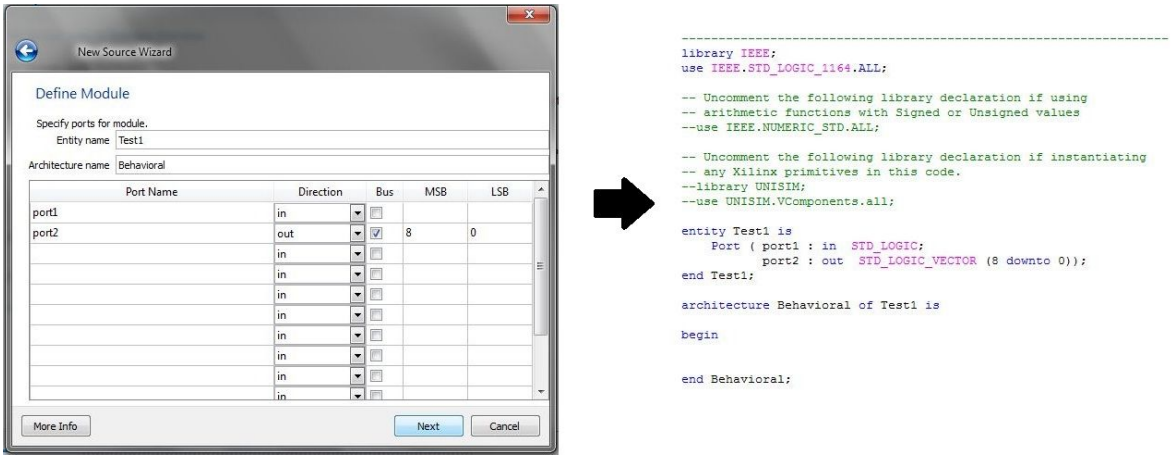
Ihned po zahájení stahování si všimneme první nevýhody - instalační soubory zabírají téměř 3GB místa na disku. Po instalaci Xilinx ISE zabírá rovných 7,5GB (placené verze ještě o 1GB více). Po spuštění programu se musí relativně dlouho čekat, než se zavede do paměti. V tom vidím další zápor. Uživatelské prostředí je kvůli velkému množství funkcí zpočátku dost nepřehledné, nicméně si lze velmi rychle zvyknout. Nebudu rozebírat všechny

[2] Narozdíl od starší knihovny AWT postavené na *těžkých komponentách*. Ta sice dosahuje vyššího výkonu, ale vypadá na každé platformě jinak[3].

[3] Firma Xilinx nabízí na svých stránkách ke stažení ještě tři předchozí verze programu.

jeho úžasné funkce, zaměřím se pouze na usnadňování práce při psaní VHDL kódu a tvorbě testbenchů.

Při zakládání nové struktury máme možnost vyplnit typy a názvy portů, které má budoucí entita obsahovat. V této fázi však nelze využít jinou, již dříve navrženou strukturu. Začínáme tedy „na zelené louce“. Po vygenerování vznikne struktura obsahující výčet námi




Obrázek 2.1: Xilinx ISE - vytvoření entity

definovaných portů a prázdné tělo architektury. Nyní lze využít mnohem zajímavější funkci, a to vložení hotové struktury z množiny existujících šablon. Šablony si můžeme vytvářet i vlastní. Nicméně výraz „šablona“ není úplně na místě, při vkládání totiž nemáme možnost předpřipravenou strukturu upravit. Vloženou strukturu je možné upravovat pouze ručně v kódu, není zde žádná funkce usnadňující vyplňování.


Obrázek 2.2: Xilinx ISE - vytvoření testbenche

Při tvorbě testbenche z hotové entity nám Xilinx ISE sám vytvoří na základě původních portů komponentu. Dále podle portů vytvoří odpovídající testovací signály a také namapo-

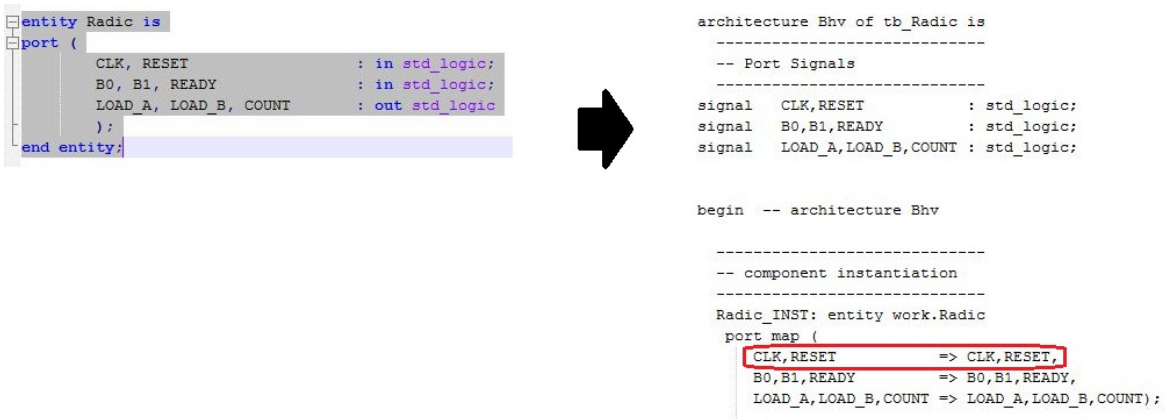
vání portů na signály. Nicméně nám nedává žádnou možnost určit, jak bude předpřipravený testbench vypadat. Nemůžeme například zvolit, kam přesně se umístí komponenta nebo výčet signálů.

### 2.3.2 VHDL plugin do programu Notepad++

VHDL plugin do programu Notepad++ se nejvíce podobá nástroji, který by měl vzniknout v rámci této práce. Narozdíl od předchozích projektů je to extrémně malý a nenáročný program. Než ho začneme používat, musíme ho začlenit do programu Notepad++.

Notepad++ je kvalitní a velmi oblíbený textový editor. Jeho stažení a používání není zpoplatněno[4]. Po instalaci zabírá necelých 10MB prostoru na disku. Obsahuje podporu zvýrazňování syntaxe pro velké množství jazyků. Plně podporuje integraci pluginů. Po instalaci lze s pomocí jednoduchého dialogu v pár krocích přidat VHDL plugin.

Plugin podporuje pouze tvorbu testbenchů. Pokud chceme testbench vytvořit, musíme nejprve v okně Notepadu++ otevřít soubor s testovanou strukturou. Ve struktuře musíme označit celou entitu a poté pomocí menubaru nebo klávesové zkratky provést příkaz *VHDL*




Obrázek 2.3: Notepad++ - generování testbenche

*copy Entity*, který analyzuje a uloží vybranou entitu. Následně můžeme v jakékoliv záložce Notepadu++ pomocí jednoho ze tří příkazů vložit instanciaci, signály a nebo celý testbench. Princip je jednoduchý, nicméně potřeba označit celou entitu kvůli její následné analýze je nepohodlný krok navíc. Plugin také obsahuje chyby. Například pokud máme více portů stejného typu oddělených čárkou, jsou v testbenchi namapovány špatně (viz Obr. 2.3).

### 2.3.3 Shrnutí rešerše

Komerční Xilinx ISE je zcela určitě velmi mocným nástrojem k návrhu hradlových polí. S jeho funkcemi si návrhář hardwaru jistě vystačí. Pokud ho ovšem chceme používat čistě jako editor VHDL kódu, je zbytečně robustní. Podpora šablon VHDL struktur je pouze v

[4] Je vydáván pod veřejnou licencí *GNU General Public License*.

omezené podobě. Plugin programu Notepad++ má nakročeno správným směrem. Bohužel je to nedotažený projekt. Umí generovat jenom testbench a to ještě s chybami.

Oba nástroje mají své klady i zápory. Nicméně ani jeden z nich neumožňuje uživatelům vytvářet a zároveň rychle vyplňovat šablony často se opakujících struktur. To by měl realizovat tento projekt.

# Kapitola 3

# Analýza a návrh řešení

## 3.1 Volba implementačního prostředí

Volba implementačního prostředí zásadně mění způsob vývoje celé aplikace. Je třeba udělat dvě základní rozhodnutí:

- Bude výsledný projekt samostatná aplikace, nebo komponenta do jiného programu?
- Jaký zvolit pogramovací jazyk?

### 3.1.1 Samostatná aplikace versus komponenta

Při tvorbě editoru kódu existují tři cesty, kterými se můžeme ubírat:

1. Použít základ existující aplikace s otevřeným zdrojovým kódem
2. Vytvořit plugin do již existujícího editoru
3. Napsat celou aplikaci sám

#### Použít základ existující aplikace

Použít zdrojový kód existující aplikace může usnadnit práci s modelováním základního chování očekávaného od editoru kódu. Stejně tak se ale můžeme v kódu, který psal někdo jiný, beznadějně ztratit. Navíc není zaručeno, že si vybereme opravdu kvalitní základ.

#### Vytvořit plugin

Napsat plugin do fungující aplikace se jeví jako velmi elegantní řešení, kterým se dnes ubírá velké množství programátorů. Uživateli a časem ověřené aplikace disponují přehledným grafickým prostředím a kvalitním formátováním textu. Většinou nechybí podpora zvýrazňování syntaxe. Je pravděpodobné, že cílový uživatel již program z nějakého důvodu používá, a nebude si na něj muset zvykat. Téměř úplně se odprostíme od vytváření a ladění GUI.

#### Napsat aplikaci sám

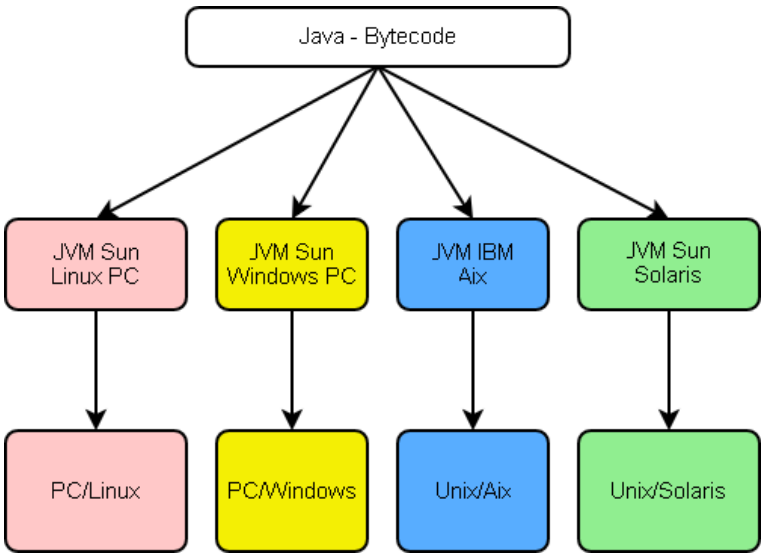
Poslední možností je vytvořit celou aplikaci sám. V takovém případě můžeme vytvořit grafické uživatelské prostředí přesně podle našich potřeb. Nemusíme studovat cizí kód, ani manuály a interfacy potřebné pro vytvoření funkčního pluginu. Tato cesta má ovšem i své nevýhody. Vytvořit přehledné a uživatelsky příjemné GUI jistě není triviální záležitost. Prostředí musí poskytovat základní sadu funkcí, které jsou dnes již standardem. Po vytvoření takové aplikace je nutné otestovat použitelnost jejího grafického prostředí mezi uživateli.

Po zvážení všech výhod a nevýhod jednotlivých možností implementace se jako nejlepší cesta jeví napsat aplikaci jako plugin do již osvědčeného editoru. Já jsem si chtěl ve své bakalářské práci vyzkoušet návrh a vývoj celé aplikace, včetně grafického prostředí. Proto jsem zvolil třetí cestu, tedy napsat aplikaci naprosto nezávislou na jiných editorech.

### 3.1.2 Programovací jazyk

V případě, že aplikace bude komponenta existujícího textového editoru, budeme nuceni programovat v jazyce, který je nadřazenou aplikací vyžadován. Pokud aplikace vznikne jako samostaný projekt, vyvstává velmi důležitá otázka, jaký zvolit programovací jazyk. Vybral jsem si jazyk Java.

Java je odpočátku vyvíjena jako multiplatformový jazyk. Přenositelnost je zajištěna díky tomu, že je Java jazykem interpretovaným. Interpretovaný znamená, že překlad zdrojového kódu neprobíhá přímo do spustitelného souboru, tj. do strojového jazyka počítače. Místo toho se zdrojový kód přeloží do pseudojazyka zvaného byte-code. Tento jazyk je nezávislý




Obrázek 3.1: JVM (převzato z [2])

na cílovém počítači. Spouštění programu probíhá pomocí interpreteru (v Javě je jím JVM). Program lze spustit na každém počítači, pro který je připravena Java platforma. Ta se skládá z Java Core API a již zmiňovaného JVM.

Problém interpretovaných jazyků ve srovnání s kompilovanými jazyky je jejich pomalost. Vše se vytváří dynamicky a uvolňování paměti probíhá automaticky. To je pohodlné pro programátora, pokud píše rozsáhlejší aplikaci. Pohybuje se na vyšší úrovni abstrakce, nemusí řešit alokování a uvolňování paměti. Za to ovšem platíme snížením rychlosti aplikace. Java se tento problém snažila částečně řešit pomocí takzvaných JIT kompilátorů. Ty během spouštění programu překládají byte-code do strojového jazyka konkrétního počítače. Tato metoda ovšem velmi brzdí zavádění programu do paměti. Java tedy přišla ještě s jedním vylepšením, technologií hot-spot. Pomocí technologie hot-spot se do strojového kódu konktrétního počítače překládají pouze kritická místa programu. Existují vyumělkované příklady[1], ve kterých je Java rychlejší než kompilované jazyky (konkrétně jazyk C). V praxi je Java ovšem stále pomalejší.

Bezesporu velkou výhodou Javy je mocná podpora v knihovnách. Java nabízí silné nástroje pro práci s textem. Jmenovitě například třída `Scanner`, která umožňuje rozlišovat tokeny a může tím velmi usnadnit parsování. Dále nám Java nabízí pohodlnou práci s I/O proudy. V této práci bude třeba pracovat se soubory, k tomu nabízí Java bytově a znakově orientované I/O proudy s množstvím různých dekorátorů (například bufferované čtení ze souboru). Poslední užitečnou knihovnou, kterou zde zmíním, je knihovna obsahující kolekce. Kolekce jsou hotové datové struktury všech možných druhů. Jsou v nich realizovány tabulky, stromy, fronty atd. Tyto kolekce jsou v Javě velmi rychlé a efektivní, jejich použití je tedy doporučováno[5].

Má volba padla na Javu právě kvůli velké podpoře v knihovnách a nezávislosti na OS. U aplikace tohoto typu bude rychlostní deficit Javy téměř neznatelný. Projeví se maximálně pomalejším zaváděním aplikace do paměti.

## 3.2 Podoba šablon

Volba šablony je klíčovým bodem celého projektu. Šablona by měla být pro uživatele přehledná, pochopitelná a lehce osvojitelná. Správně návržená šablona může navíc zásadním způsobem usnadnit následnou analýzu a vyplňování šablony. Tato fáze návrhu je tedy velmi důležitá a nesmí být podceněna.

Aplikace bude pracovat se dvěma druhy šablon: šablona VHDL struktury a šablona VHDL testbenche. Na základě šablony VHDL struktury budou vznikat hotové struktury entit, automatů, čítačů a pod. Pomocí šablony VHDL testbenche bude uživatel mít možnost rychle vytvořit entitu určenou k testovaní existující struktury.

### 3.2.1 Šablona VHDL struktury

#### Globální proměnné

První vlastností, kterou by šablona VHDL struktury měla mít, je umožňovat uživateli definovat univerzální proměnnou. Odkaz na tuto proměnnou (dále **globální proměnná**) bude možné použít místo názvu portů, signálů, typů atd. Dále ji bude možné použít jako část názvu jakékoliv jiné proměnné VHDL kódu. Pokud uživatel při vyplňování šablony změní

[1] Viz např. pokus pana Herouta[4].

název globální proměnné, bude tento název nahrazen na všech místech, kde byl použit odkaz na danou proměnnou. Tímto způsobem lze naráz přejmenovat velké množství záznamů. V tom spočívá hlavní smysl zavádění globálních proměnných.

Je třeba určit, kde a jak bude globální proměnná definována a jakým způsobem se na ni v šabloně budeme odkazovat. V první fázi návrhu se počítalo s tím, že proměnná bude zadefinována při prvním výskytu v šabloně. V takovém případě není třeba rozlišovat definici globální proměnné od odkazu na ni. První odkaz je zároveň i definicí. K označení globální proměnné byl vybrán znak '$', který ve VHDL kódu nemá žádný klíčový význam. Pokud by globální proměnné reprezentovaly pouze kompletní názvy proměnných VHDL kódu, stačilo by znakem '$' označit jenom začátek globální proměnné. To by ovšem bylo omezující. Je žádoucí, aby uživatel mohl definovat pomocí globální proměnné pouze část názvu skutečné proměnné. K tomu je nutné označit začátek i konec globální proměnné. Globálně závislé proměnné VHDL kódu potom budou v šabloně aplikace VHDL SGen vypadat následovně:

```
$cntr$
$cntr$_en
my_$cntr$
my_$cntr$_en
```

Globální část názvu může být ve formě prefixu, postfixu i uprostřed názvu. Počítá se také s tím, že aplikace bude nahrazovat i globální proměnné vyskytující se v komentářích.

Kvůli přehlednosti byla zavedena ještě počáteční definice globální proměnné. Umisťuje se na začátek šablony, před klíčové slovo `entity`. Uživatel má lepší přehled o globálních proměnných. Zároveň se tím aplikaci usnadní parsování šablony. Globální proměnná je definována pomocí symbolu '@'. Ten se dává pouze před název globální proměnné. Definice globální proměnné `cntr` vypadá takto:

```
@cntr
```

Ve vygenerované struktuře nesmějí zůstat ani tyto definice, ani případné odkazy uzavřené v dolarech[2]. Při finálním vyplňování šablony jsou proto definice globálních proměnných odstraněny. Pokud uživatel některou globální proměnnou nepřejmenuje, nebo se objeví odkaz na nedefinovanou globální proměnnou, zůstane na místě odkazu původní jméno zbavené dolarové notace.

Na jednoduchém příkladu si ukážeme, jak bude vypadat práce s globálními proměnnými. Definujme jednu proměnnou, jejíž odkazy využijeme na několika místech kódu zároveň.

```
--Sablona ukazujici praci s globalnimi promennymi
@citac    --definice globalni promenne

entity muj_$citac$ is   --odkaz pouzity k pojmenovani entity
port (
          clk             : in std_logic;
          $citac$_reset   : in std_logic; --jmeno portu pomoci odkazu
          $citac$_enable  : in std_logic;
```

[2] Tedy odkazy na definované proměnné.

```
            $citac$_done    : out std_logic;
        );
end muj_$citac$;

architecture muj_$citac$_arch of muj_$citac$ is  --odkaz ve jmenu architektury
...
end muj_$citac$_arch;
```

Pokud uživatel v aplikaci zvolí možnost vytvořit ze šablony konkrétní strukturu VHDL kódu, šablona bude analyzována[3]. Na základě analyzovaných dat se zobrazí dialog s výčtem všech definovaných globálních proměnných (zde tedy pouze `citac`). Uživatel má možnost modifikovat název, nebo ponechat původní. Po stiknutí tlačítka „Generate“ se ze šablony odebere řádek s definicí globální proměnné (`@citac`) a na místa všech odkazů (`$citac$`) se vloží nově zadané pojmenování (popřípadě zůstane původní „citac“). Změníme-li tedy název např. na „counter“, bude výsledná struktura vypadat následovně:

```
--Sablona ukazujici praci s globalnimi promennymi

entity muj_counter is
port (
            clk             : in std_logic;
            counter_reset   : in std_logic;
            counter_enable  : in std_logic;
            counter_done    : out std_logic;
        );
end muj_counter;

architecture muj_counter_arch of muj_counter is
...
end muj_counter_arch;
```

### Generování větví `when`

Další funkcí, kterou aplikace bude nabízet, je automatické generování `when` větví příkazu `case`. Větve se budou generovat na základě znalosti existujícího typu nebo signálu. Uživatel bude moci přesně definovat kam se větve vloží a podle kterého typu/signálu se vygenerují[4].

Definujme příkaz označující místo, kam má být generovaný kód vložen (dále **rozkopírovávací příkaz**). V prvním návrhu byl příkaz ohraničen z obou stran znakem '`$`'. To ovšem způsobovalo zbytečné problémy. Bylo nutné složitě odlišovat rozkopírovávací příkazy od odkazů na globální proměnné. Dále bylo téměř nemožné použít rozkopírovávací příkaz v závislosti na proměnné, která je sama nějakým způsobem závislá na globální proměnné[5]. Byl tedy vybrán znak '`#`'. V aplikaci jsou definovány dva typy rozkopírovávacích příkazů:

[3] Detaily ohledně analýzy kódu jsou popsány v kapitole 3.3.1 .
[4] Zde je samozdřejmě nutné typy a signály znát. Tuto problematiku řeší kapitola 3.3.2 .
[5] Bude vysvětleno později na příkladu.

1. Příkazy generující větve `when` přímo podle hodnot, kterých nabývá konkrétní `type`.
2. Příkazy generující celý blok příkazu `case` v závislosti na typu konkrétního signálu.

Z podoby příkazu musí být jasné, zda se bude rozkopírovávat podle hodnot z tabuky typů nebo signálů. Příkaz musí samozřejmě obsahovat také název konkrétního typu/signálu. Bude se tedy skládat ze dvou částí. První definuje typ příkazu, druhá konkrétní typ/signál.

#### Příkazy generující větve `when` přímo

Pro příkazy pracující s typy bylo zvoleno klíčové slovo `case_body`. Slovo „body“ vyjadřuje fakt, že nevkládáme celý blok příkazu `case`, nýbrž jenom jednotlivé větve. Počet vkládaný větví `when` je dán výčtem hodnot, kterých nabývá konkrétní `type`. Jeho jméno tvoří druhou část příkazu. Klíčové slovo a indentifikátor typu rozděluje znak '`#`'. Celý rozkopírovávací příkaz vypadá následovně:

```
#case_body#identifikator_typu#
```

#### Příkazy generující celý blok příkazu `case`

Příkaz pracující se signály bude označen klíčovým slovem `case`. Při vyplňování šablony bude nahrazen celým blokem příkazu `case`. Jeho druhou část tvoří identifikátor signálu. Klíčové slovo a indetifikátor signálu opět rozděluje znak '`#`'. Celý příkaz vypadá takto:

```
#case#identifikator_signalu#
```

V této fázi aplikace dokáže na základě rozkopírovávacích příkazů vytvářet **prázdné** větve `when` příkazu `case`. Návrhář se často dostává do situace, kdy je potřeba v každé větvi provést totožné nebo velmi podobné sekvence příkazů. Bylo by tedy dobré, kdyby aplikace umožňovala ke každé větvi přidat nějaký předem určený kus kódu. Byla tedy zavedena ještě jedna možná podoba rozkopírovávacího příkazu, která uživateli umožňuje definovat sekvenci příkazů, které budou při rozkopírovávání přídány ke každé větvi. Za název typu/signálu lze vložit ještě složené závorky, do který může návrhář vložit libovolně dlouhý kus kódu.

```
#case_body#identifikator_typu{nejaky kod}#
#case#identifikator_signalu{nejaky kod}#
```

Praktickou ukázku použití rozkopírovávacího příkazu naleznete v příloze D.

### 3.2.2 Šablona VHDL testbenche

Vytváření komponent, generování signálů podle původního výčtu portů a mapování signálů na porty - to jsou tři nejdůležitější a zároveň dostačující věci k tomu, aby uživatel mohl být oproštěn od mechanického a stále se opakujícího procesu vytváření testovacích entit. Některé existující nástroje sice umí vytvořit testbench, ale uživatel nemá možnost ovlivnit jeho podobu. Aplikace VHDL SGen bude pro šablonu VHDL testbeche definovat sadu příkazů, pomocí kterých bude uživateli umožněno vytvořit testovací entitu přesně podle svý představ.

Všechny příkazy jsou z důvodu přehlednosti a snadného parserování z obou stran ohraničeny znakem '$'. Následuje přehled příkazů a kódu, který se na jejich základě vygeneruje.

Pro snažší pochopení vytvořme jednoduchou entitu, z níž budeme odvozovat jednotlivé části testbenche.

```
--Testovana entita
entity moje_entita is
port (
        clk, reset    : in std_logic;
        vystup        : out std_logic;
     );
end moje_entita;

architecture ...
```

Pomocí analyzátoru[6] zjistíme, že entita má název „moje_entita“ a obsahuje porty „clk“, „reset“ a „vystup“. Rozeberme tedy příkazy, které může uživatel použít v šabloně testbenche:

- `$entity$` ... Na všechna místa, na kterých se vyskytuje tento příkaz, bude vloženo jméno testované entity (i do komentářů). V našem případě se tedy všechny značky `$entity$` nahradí řetězcem `moje_entita`.
- `$component$` ... Na místě této značky se v testbenchi vytvoří celá komponenta, obsahující porty a generiky testované entity (pokud jsou). Výsledný kód vypadá následovně:

  ```
  component moje_entita
  port  (
         clk     : in std_logic;
         reset   : in std_logic;
         vystup  : out std_logic
        );
  end component;
  ```

- `$component_port$` ... Tento příkaz je nahrazen výčtem portů testované entity. Jak je partné z názvu, používá se uvnitř těla komponenty. Generovaný kód vypadá následovně:

  ```
  clk    : in std_logic;
  reset  : in std_logic;
  vystup : out std_logic
  ```

- `$component_generic$` ... Tato značka je při gnerování nahrazena výčtem generiků. Princip je stejný jako u výčtu portů.
- `$signals$` ... Příkaz pro vložení signálů. Ty se vytvářejí na základě znalosti portů testované entity. Výstup:

[6] Princip viz kapitola 3.3 .

```
signal clk    :  std_logic;
signal reset  :  std_logic;
signal vystup :  std_logic;
```

- `$port_map$` ... Namapování portů na signály. Výstup:

```
clk    => clk,
reset  => reset,
vystup => vystup
```

- `$generic_map$` ... Namapování generiků na signály. Principielně stejný výstup jako namapování portů na signály.

Všechny generované řetězce dodržují odsazení svých původních značek. V kombinaci se značkou pro vložení jména původní entity si může uživatel vytvořit šablonu přesně podle svých potřeb. Ukázka reálně použitelné šablony je v příloze D.

## 3.3 Analýza VHDL kódu a šablon, vyplňování šablon

Pro úspěšnou a efektivní práci aplikace VHDL SGen bude rozumné napsat analyzátor VHDL kódu. Zvažoval jsem i variantu provádět analýzu pomocí již hotové třídy Scanner, ve výsledku by to ovšem bylo mnohem pracnější a málo efektivní. Navíc jsem v době návrhu dokončoval předmět „Programovací jazyky a překladače“, který se touto problematikou zabývá.

Výsledek kódové analýzy může mít několik podob. Lexikální analyzátor prochází kód a vrací nám jednotlivé lexikální symboly (tokeny), na které narazil. Na základě postupně získávaných tokenů syntaktický analyzátor identifikuje jednotlivé kódové struktury, například porty, signály, procesy, componenty, entity atd. Jeho výstupem zpravidla bývá vnitřní podoba analyzovaného kódu. Nejčastěji se vnitřní podoba ukládá ve formě derivačních stromů. Tato struktura se v praxi využívá pro překlad z jednoho jazyka do jiného (např. do již zmiňovaného byte-code, nebo do strojového jazyka). V našem případě je ovšem zbytečné ukládat celý kód do nějaké vnitřní struktury. Bylo by to navíc obtížně realizovatelné. Při analýze a následném vyplňování šablon je totiž velmi důležité, aby výsledné struktury přesně zachovávaly formát a původní podobu šablon. Jakou rozumnou podobu a obsah dat by tedy měl poskytovat syntaktický analyzátor programu VHDL SGen?

### 3.3.1 Analýza šablony VHDL struktury

Analýzu šablony VHDL struktury budeme dělit na dvě fáze:

- Analýza před otevřením dialogového okna sloužícího k vyplnění šablony
- Analýza při vyplňování šablony podle uživatelem zadaných hodnot

#### Otevření dialogového okna

Od analýzy před otevřením dialogového okna budeme určitě v nějaké formě očekávat výčet globálních proměnných. Pro ten se jako nejideálnější datová struktura nabízí tabulka. V prvním sloupci tabulky bude původní jméno globální proměnné, na jehož základě budeme nahrazovat odkazy v šabloně. V druhém sloupci bude uživatelem zadané nové jméno, kterým mají být odpovídající odkazy nahrazeny. Bezprostředně po analýze bude vhodné druhý sloupec tabulky vyplnit původním jménem. Každý řádek tedy bude obsahovat dva totožné řetězce.

V aplikaci pravděpodobně přibude ještě možnost dynamicky měnit celou základní strukturu kódu, tzn. mít v nějakém dialogovém okně přehled o všech globálních proměnných, portech, generikách, signálech, typech a konstantách. Při volbě tohoto druhu editace bude třeba získat pomocí kódové analýzy všechny potřebné informace. Jako nejšikovnější se opět jeví ukládat všechny informace do tabulek. Podrobněji o nich v kapitole 4.1 .

#### Vyplňování šablony

Po úspěšném vyplnění dialogového okna je třeba dle editovaných tabulek vyplnit šablonu. Vzhledem k faktu, že jsem se rozhodl neukládat celý kód do vnitřní formy, je třeba opět analyzovat původní šablonu a vložit na správná místa získané údaje. Díky tomu je snadné zachovat původní podobu šablony bez větších změn (přesně tak, jak si ji uživatel sestavil). Třída nadřazená syntaktickému analyzátoru pomocí něj bude procházet kód a hledat příkazy k vkládání. Najde všechny odkazy na globální proměnné a nahradí je uživatelem zadanými názvy. Dále zjistí, zda šablona obsahuje rozkopírovávací příkazy. Nalezené příkazy nahradí rozkopírovaným kódem. Pokud by se nepodařilo najít opovídající typ/signál[7], bude příkaz odstraněn.

### 3.3.2 Analýza šablony VHDL testbenche

Abychom mohli vytvořit testbench, potřebujeme znát název a tabulku portů a generiků původní entity (pokud jsou definovány). Dále potřebujeme mít šablonu VHDL testbenche. Na základě těchto informací aplikace vytvoří testbench. Analýza a vyplňování šablony testbenche bude fungovat na stejném principu jako vyplňování šablony VHDL struktury. Analyzátor hledá příkazy k vložení názvu testované entity a jednotlivých struktur kódu[8] a na místa příkazů jsou rovnou vkládána příslušná data z tabulek.

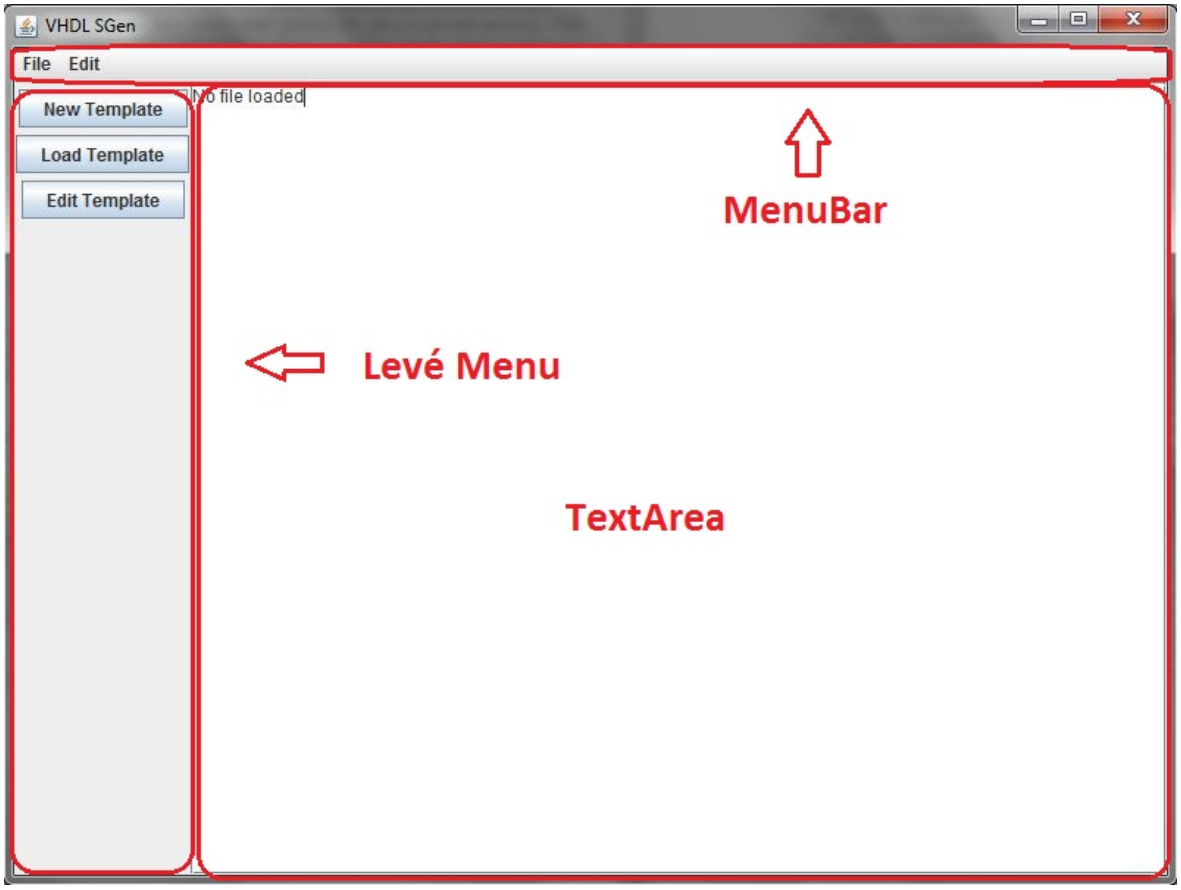
## 3.4 Návrh GUI

Grafické prostředí je u aplikace typu editor kódu stěžejním bodem. Uživatel potřebuje rychlý a přehledný přístup k často používaným funkcím. Ovládání musí být intuitivní. Když potřebujeme provést nějakou akci, mělo by být hned jasné, kam se má kliknout. Velmi vhodné je také zpřístupnit nejpoužívanější funkce aplikace pomocí klávesových zkratek.

[7] Viz kapitola 3.2.1 .
[8] Viz přehled příkazů v kapitole 3.2.2 .

### 3.4.1 Hlavní okno programu

Konkrétní podoba GUI se během vývoje programu několikrát změnila ve snaze vytvořit opravdu přehledné prostředí. Rozložení základních prvků hlavního okna však zůstává od první verze stejné. Základ tvoří klasický rám se základními ovládacími prvky sloužícími k




Obrázek 3.2: Hlavní okno aplikace - základní rozložení prvků

minimalizaci, maximalizaci a zavření okna. V jeho horní části je vložen menubar. Ten již od prvního návrhu obsahuje položku *File* s volbami *Load*, *Save* a *Exit*. V levé části okna je umístěn panel obsahující nejpoužívanější tlačítka. Ten byl v průběhu vývoje aplikace cílem největšího množství změn. Zbytek okna vyplňuje scrollovatelná oblast pro úpravu textu[9].

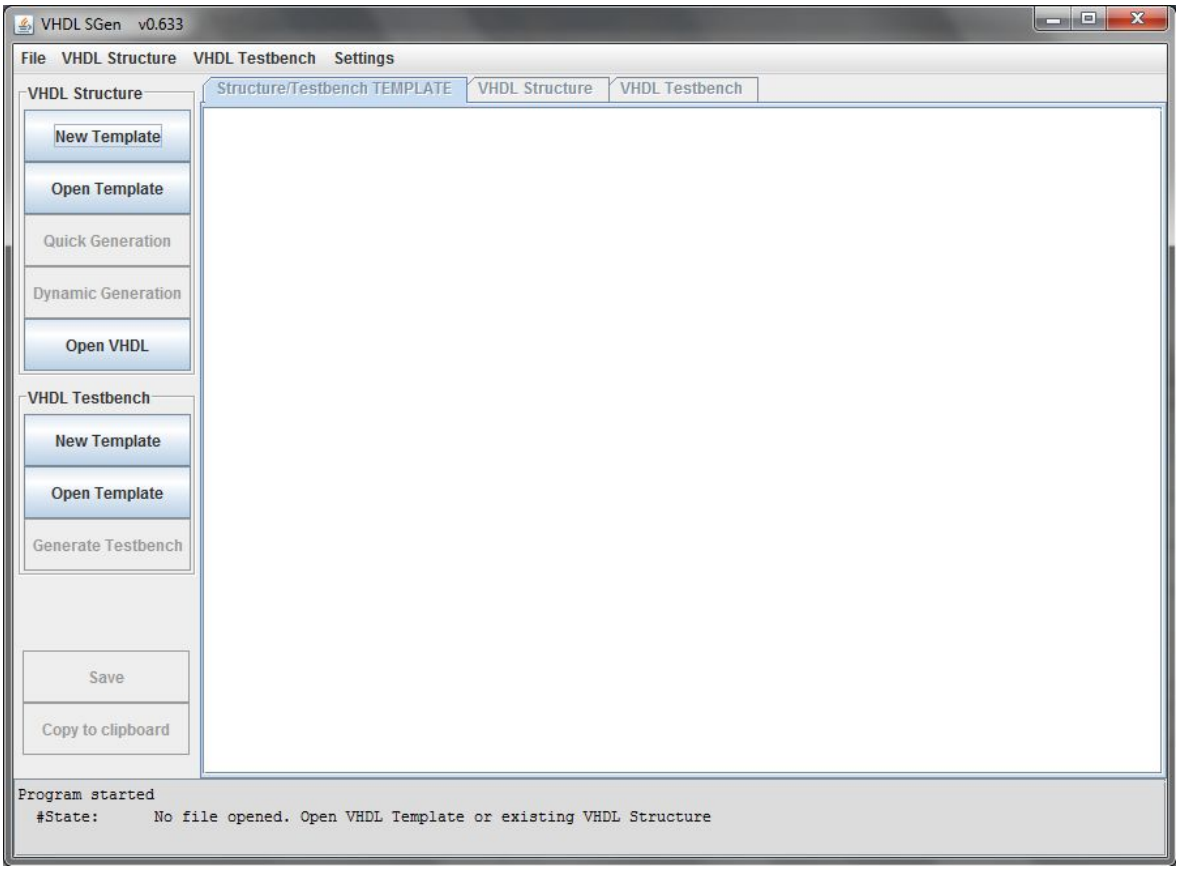
Tento základní koncept se postupně rozrůstal. V dolní části okna přibyl stavový řádek, informující o výsledku posledně prováděné akce. Ten se později změnil na stavový panel, ve kterém lze pomocí scrollbaru prohlížet všechny události, které se během práce s programem staly. Uživatel může zpětně dohledat, jaký editoval soubor, jestli vytvořenou VHDL strukturu již uložil a podobně.

Střídaly a prohazovaly se položky v menubaru. Cílem bylo zpřístupnit pomocí menubaru všechny funkce programu. K původní *File* byly přidány položky *VHDL Structure*, *VHDL*

[9] Tzv. *TextArea*.

*Testbench* a *Settings*. Položka *VHDL Structure* umožňuje práci s šablonami VHDL struktur. Obsahuje položky k načtení a uložení šablony a položku sloužící ke generování struktury. Položka *VHDL Testbench* umožňuje generování a ukládání testbenchů. Poslední položka slouží k nastavení vzhledu aplikace. Lze nastavit velikost, styl, font a barvu písma a pozadí.

Původně jedna textová oblast byla rozšířena o dvě záložky, celkem má tedy tři záložky. První, *Structure/Testbench TEMPLATE*, slouží k zobrazování a editaci šablon VHDL struktur a šablon testbenchů (tedy soubory s příponou .sgs a .sgt[10]). Druhá záložka, *VHDL Structure*, slouží k zobrazování hotových VHDL struktur. Dají se v ní otevřít i struktury určené k následné tvorbě testbenchů. V poslední záložce, *VHDL Testbench*, se zobrazují vygenerované VHDL testbenche.




Obrázek 3.3: Hlavní okno aplikace - finální podoba

Levý panel původně poskytoval pevně danou sadu nejčastěji používaných tlačítek. Všechna tlačítka byla stále aktivní. To se neukázalo jako příliš šťastné řešení. Uživatelé se v takovéto navigaci ztráceli. Po zavedení záložek v textové oblasti byl panel změněn. Zobrazovala se pouze tlačítka, která souvisela s aktuálně vybranou záložkou. Když byla například vybrána záložka *VHDL Structure*, v levém panelu byla k dispozici tlačítka *Load VHDL*, *Save VHDL* a *Generate Testbench*. Nicméně ani toto řešení s v praxi neukázalo jako dobré. Neustálá změna

[10] Viz kapitola 4

nabízených možností na uživatele působila matoucím dojmem. Jako nejlepší řešení se nakonec ukázalo rozdělení tlačítek do několika logických skupin. Skupiny shromažďují tlačítka, která k sobě intuitivně patří. Pokud nějaké tlačítko zrovna nelze použít, je znepřístupněno. O znepřístupnění rozhoduje aktuálně vybraná záložka a přípona aktuálně zobrazeného souboru. V dolní části se nachází tlačítko *Copy to clipboard*, které je určeno k rychlému uložení celého obsahu textové oblasti v aktuálně vybrané záložce. To je užitečné v případech, kdy uživatel chce vygenerovaný kód např. otestovat v simulátoru. Výsledné grafické prostředí je vidět na obrázku 3.3 .

### 3.4.2 Způsob vyplňování šablon

GUI vyvíjené aplikace musí poskytovat také grafickou podporu pro vyplňování šablon. Ta bude realizována pomocí dialogových oken. Aplikace VHDL SGen bude nabízet dva módy vyplňování:

1. Rychlé vyplňování pomocí jednoduchého dialogu.
2. Dynamická editace.

#### Rychlé vyplňování

Pokud uživatel zvolí možnost rychlého vyplnění šablony, zobrazí se mu jednoduchý dialog obsahující pouze výčet globálních proměnných. Dialog umožnuje pouze měnit jména globálních proměnných. Po stisknutí tlačítka *Generate Structure* se vygeneruje příslušná struktura. Viz obrázek 3.4 .


Obrázek 3.4: VHDL SGen - jednoduchý dialog

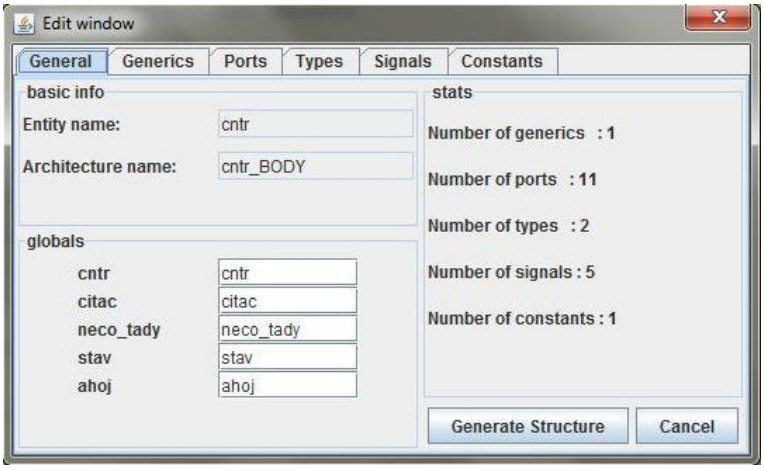
#### Dynamická editace

Dynamické editační okno poskytuje uživateli mnohem komplexnější pohled na celou šablonu. Je členěno do několika záložek. Nabízí, stejně jako jednoduché editační okno, možnost přejmenovat globální proměnné. Tuto základní funkčnost ale dále rozšiřuje o velké množství dalších možností.

První záložka je logicky členěna na tři části. První část obsahuje jméno entity a architektury. Jména lze editovat (pokud nejsou závislá na některé globální proměnné[11]). Pod ní

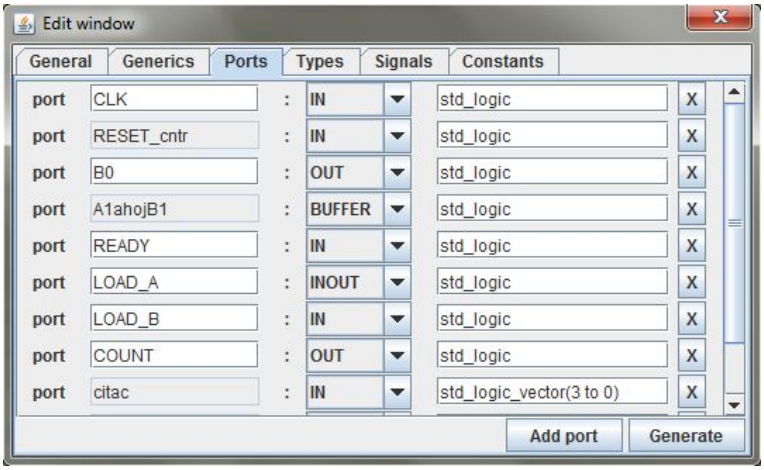
[11] To bude blíže popsáno v kapitole 4 .

se nachází výčet globálních proměnných. V pravé části je zobrazen počet generiků, portů, signálů, typů a konstant nacházejících se v editované šabloně. V pravém dolním rohu se nachází tlačítko *Generate Structure* a tlačítko *Cancel*. První záložka je na obrázku 3.5 .




Obrázek 3.5: VHDL SGen - dynamické editační okno 1

Zbývajících pět záložek obsahuje výčty generiků, portů, signálů, typů a konstant. Všechny mají totožnou strukturu. Většinu panelu zabírá přehled jednotlivých řádků. Řádek obsahuje jméno konkrétní proměnné, její typ, orientaci, šířku a další údaje(v závislosti na tom, zda se jedná o generik, port, signál, typ nebo konstantu). Na konci každého řádku je tlačítko sloužící k jeho odebrání. V dolní části je panel obsahující tlačítko pro přidání nového řádku a tlačítko pro okamžité vygenerování struktury. Na obrázku 3.6 je ukázáno, jak může vypadat záložka portů. Zašedlá jména portů jsou závislá na nějaké globální proměnné.




Obrázek 3.6: VHDL SGen - dynamické editační okno 2

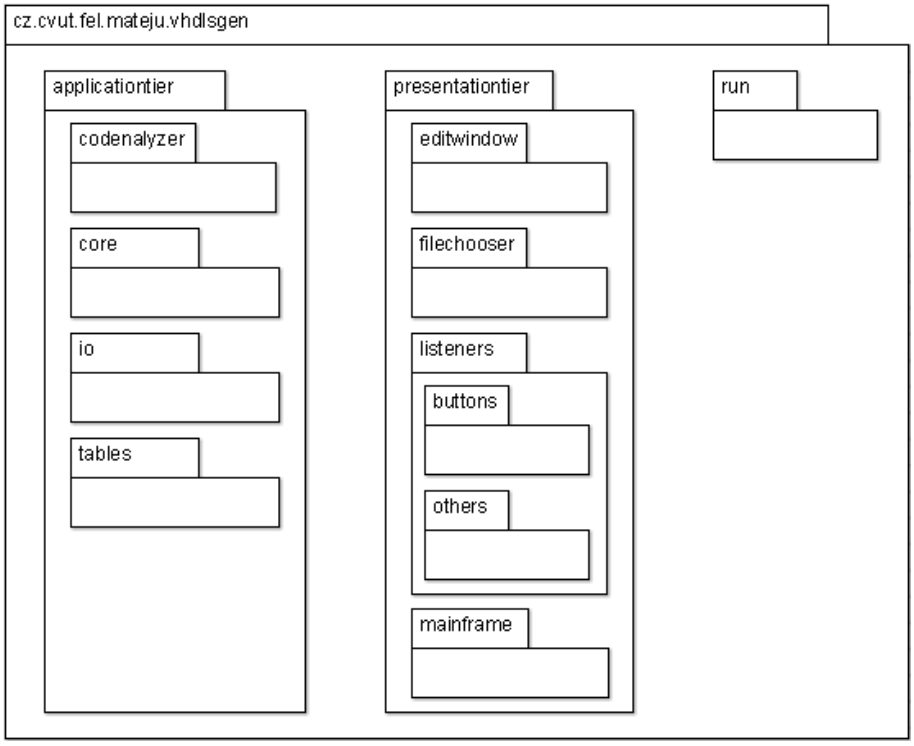
Obsah editačního okna je namapován na tabulky vzniklé při analýze šablony. Jakákoliv

změna provedená uživatelem je ihned uložena do tabulek a okno je na základě nových údajů překresleno. Proto název „dynamické editační okno“.

# Kapitola 4

# Realizace

V této kapitole se pokusím popsat realizaci nejzajímavějších částí mé práce. Během psaní programu jsem často využíval dokumentaci k Javě[10] a Java tutoriál[11]. Inspirací při psaní kódového analyzátoru mi byla ukázková práce pana Troníčka[6] a dále tutoriál jazyka VHDL[9]. Vývoj aplikace probíhal ve vývojovém prostředí *Netbeans IDE*. Finální verze




Obrázek 4.1: UML diagram - přehled balíčků

aplikace obsahuje 10 balíčků, 58 tříd a přibližně 6400 řádek kódu. Celý kód je zdokumentován pomocí javadoc komentářů. Aplikace je rozdělena na dvě základní vrstvy:

- Aplikační vrstva - obsahuje editační jádro, analyzátor, tabulky a třídy pro čtení/zápis do souboru
- Prezentační vrstva - obsahuje hlavní okno, editační okno, posluchače a další třídy

Struktura souborů je znázorněna na obrázku 4.1 . Následující části kapitoly poskytují podrobnější pohled na vybrané třídy a balíčky.

## 4.1 Analyzátor VHDL kódu

Hlavní třídou, koordinující práci s kódem, je třída `EditingCore`. Její instance je vytvořena ve chvíli, kdy je poprvé potřeba analyzovat nějaký kus kódu. Třída `EditingCore` vytváří a udržuje instance tříd z několika balíčků. Pomocí kódového analyzátoru vytváří, udržuje a aktualizuje tabulky globálních proměnných, portů, signálů atd.

### 4.1.1 Třídy analyzující kód

Analyzátor tvoří celkem pět tříd. Třída `CodeAnalyzerExpcetion` je odvozena od třídy `Exception` a představuje typ výjimky, která je produkována třídami kódového analyzátoru. Třída `LexicalSymbol` představuje výčet lexikálních symbolů jazyka VHDL. Pomocí této třídy lexikální analyzátor zjišťuje, zda získaný token odpovídá některému z klíčových slov.

Třída `LexicalAnalyzer` je schopná procházet a analyzovat zdrojový kód v podobě vstupního souboru nebo řetězce. Obsahuje metodu `nextLexicalSymbol()`, která po každém zavolání vrátí následující lexikální symbol. Pokud je získaným symbolem indetifikátor, jeho jméno získáme pomocí metody `getString()`. V případě nalezení konstanty získáme její hodnotu metodou `getConst()`. Dále byly implementovány podpůrné metody usnadňující vyplňování šablon. Metody `getOffset()` a `getIndex()` vrací znakovou vzdálenost aktuálně získaného tokenu od začátku kódu/řádku.

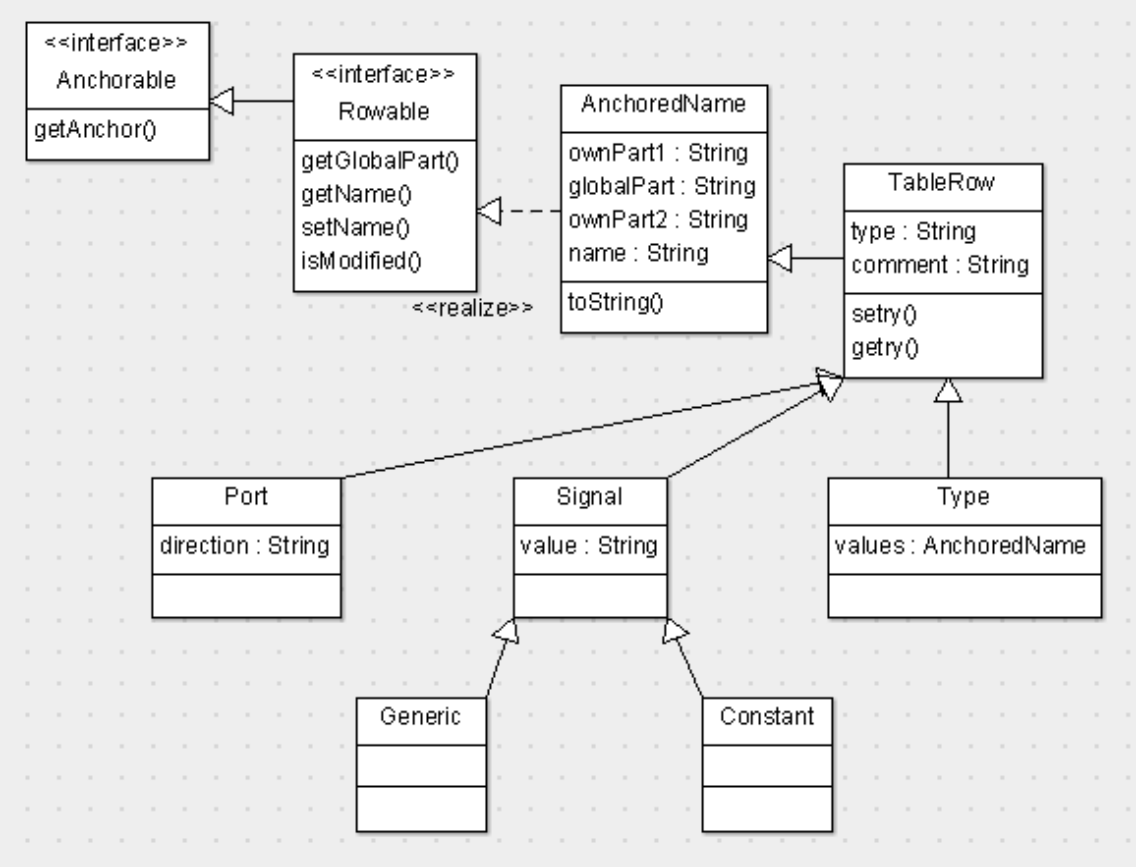
Třída `SyntaxAnalyzer` dokáže pomocí třídy `LexicalAnalyzer` rozlišovat v kódu jednotlivé syntaktické konstrukce jazyka VHDL. Obsahuje především metody sestavující z analyzovaného kódu tabulky portů, generiků, signálů, typů a konstant. Z dalších veřejných metod jmenujme například metodu `skipTo(LexicalSymbol l)`, která preskočí na první výskyt zadaného symbolu.

Práci s analyzátorem kódu uceluje a zjednodušuje třída `CodeAnalyzer`, která je odvozená od třídy `SyntaxAnalyzer`. Poskytuje například metodu `fillTables()` umožňující vyplnit všechny tabulky najednou.

### 4.1.2 Tabulky získaných údajů

Když se v programovacím jazyce Java mluví o tabulkách, nabízí se řešit daný problém užitím knihovní třídy `HashMap` nebo `TreeMap`. Datová struktura *Map* je realizována jako množina dvojic klíč-hodnota. K identifikaci hodnot v tabulce může ovšem být použit pouze jeden klíč. To je hlavním důvodem, proč jsem se rozhodl jít cestou realizace vlastních tabulek.

Kvůli jednotnému zacházení s odlišnými řádky tabulek[1] byla vytvořena celá hierarchie tříd (viz obrázek 4.2). Na vrcholu se nachází rozhraní `Anchorable` vnucující řádkům schopnost vrátit informaci o své kotvě. **Kotva** je původní jméno proměnné v kódu. Skládá se z




Obrázek 4.2: UML diagram - hierarchie tříd reprezentujících jednotlivé řádky

prefixu, globání části a postfixu. Pokud proměnná není závislá na žádné globální proměnné, její název obsahuje pouze prefix. Název závislé proměnné[2] obsahuje globální část a volitelně i prefix a postfix. Z toho vychází rozhraní `Rowable` (rozšiřující rozhraní `Anchorable`) a následně třída `AnchoredName`, která rozhraní implementuje. Třída obsahuje stringovou reprezentaci prefixu, globální části a postfixu. Dále implementuje metody `getGlobalPart()`, `getName()`, `setName()` a `isModified()`. Všechny operují se jménem řádku. Jejich názvy dostatečně vypovídají o jejich funkčnosti.

Další třídy odvozené od třídy `AnchoredName` reprezentují konkrétní proměnné VHDL kódu a šablon. Nesou informace, které jsou pro ně typické. Nejdůležitější metodou, kterou všechny implementují, je metoda `toString()`. Prostřednictvím této metody jsou tabulky pohodlně tisknuty bez nutnosti dodatečného formátování. Přehled všech těchto tříd je na obrázku 4.2 .

[1] Tzv. *mnohotvarost*
[2] Viz kapitola 3.2.1 .

Samotná tabulka je reprezentována třídou `Table`, která je odvozena od knihovní třídy `Vector`. Do ní se vkládají jednotlivé řádky. Kromě všech metod třídy `Vector` obsahuje ještě následující metody:

- `findAnchor(String s)`

  Vyhledává v tabulce řádek s odpovídající kotvou (tj. původní název).

- `findName(String s)`

  Vyhledává v tabulce řádek s odpovídajícím jménem.

- `nextModified()`

  Postupně odebírá z tabulky řádky, u který bylo modifikováno jméno.

- `toString()`

  Převede celou tabulku do stringu. Volá na každý řádek tabulky metodu `toString()` a výsledek spojuje v jeden formátovaný řetězec.

- `longestLength()`

  Vrací v počtu znaků délku nejdelšího názvu. Toho se využívá kvůli zarovnání výpisu při volání metody `toString()`.

## 4.2 Generování kódu

Generování kódu řídí třída `EditingCore`. Využívá k tomu singleton[3] `CodeBuilder`, který obsahuje metody schopné analyzovat a vyplňovat šablony. K tomu je samozřejmě ve většině případů využit analyzátor VHDL kódu. Odebírání částí kódu a vkládání nových probíhá pomocí metod, které poskytuje knihovní třída `StringBuffer`. Následuje výčet některých metod, které obsahuje třída `CodeBuilder`. Všechny metody dostávají jako parametr upravovaný kód a vrací upravený kód (v přehledu tedy tento fakt nebude z důvodu přehlednosti zmiňován).

- `removeHeader()`

  Odstraní definice globálních proměnných.

- `insertTables(Table[] tables)`

  Vloží do kódu upravený seznam generiků, portů, signálů, typů, a konstant.

- `replaceAnchor(String anchor, String newName)`

  Nahradí všechny tokeny *anchor* novým řetězcem. Jejich vyhledávání probíhá prostřednictvím třídy `CodeAnalyzer`. Používá se pouze při editaci prostřednictvím dynamického okna.

[3] Třída, od které lze vytvořit pouze jednu instanci.

- `replaceDident(String anchor, String newName)`

  Tato metoda nahrazuje všechny odkazy na globální proměnnou *anchor* novým jménem. Nepoužívá analyzátor VHDL kódu. K hledání využívá metodu `indexOf()` z knihovní třídy `StringBuffer`.

- `unpackCase(Table typeTable, Table signalTable)`

  Hledá v šabloně rozbalovací příkazy. Pokud k příkazu existuje opovídající signál/typ, jsou do kódu rozkopírovány větve `when`. Pokud k příkazu odpovídající signál/typ neexistuje, je rozkopírovávací příkaz z kódu odstraněn. Aby metoda mohla rozhodnout, zda signál/typ existuje či ne, musí dostat jako parametr referenci na příslušné tabulky.

- `createTestbench(String entityName, Table portTable, Table genericTable)`

  Metoda kompletně vyplní šablonu testbenche. Používá k tomu údaje, které dostane jako parametr.

## 4.3 GUI

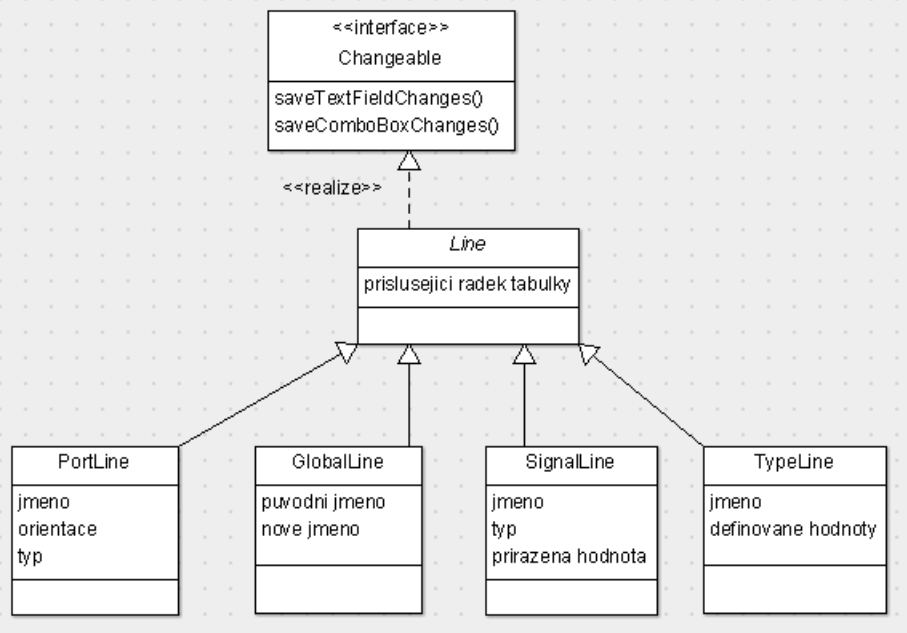
Grafické prostředí aplikace je celé psané „ručně“. Nebyly použity žádné automatické generátory typu WYSIWYG. Vzhledem k častým modifikacím vzhledu a vzájemné provázanosti komponent by bylo téměř nemožné takto vzniklé prostředí udržovat a vylepšovat. Popíši zde hlavně nejzajímavější část, a tou je dynamické editační okno.

Dynamické editační okno je úzce spojeno s tabulkami hodnot získaných při analýze šablony. Uživatel v něm má přehled o podstatných částech kódu. Současně má možnost většinu údajů editovat. Dynamickému editačnímu oknu je ve zdrojových souborech věnován celý jeden balíček tříd. Hlavní třída má název `EditWindow` a rozšiřuje knihovní třídu `JDialog`. Okno je členěno do záložek, jeho základem je tedy `JTabbedPane`. Na něm jsou v jednotlivých záložkách umístěny panely obsahující řádky výčtu portů, signálu atd.

Porty, signály, generiky a další proměnné VHDL kódu se od sebe v několika bodech liší. Bylo tedy třeba jednotlivé řádky realizovat jako samostatné třídy. Tyto třídy mají společného předka, abstraktní třídu `Line`, která rozšiřuje knihovní třídu `JPanel` a implementuje rozhraní `Changeable` (viz obrázek 4.3). Rozhraní `Changeable` nutí třídám, které ho implementují, schopnost reagovat na změny provedené uživatelem. Tato vlastnost je poté u tříd vyvolávána pomocí posluchačů událostí[4].

Třídy rozšiřující abstraktní třídu `Line` se jmenují `PortLine`, `GlobalLine`, `SignalLine` a `TypeLine`. Generiky a konstanty nemají vlastní třídu, protože obsahují totožné údaje jako signály. Třída `EditWindow` obsahuje přetíženou funkci `addLine()`, pomocí které jsou jednotlivé řádky vkládány do záložek dle svého typu. Pokud uživatel některý řádek smaže, vymaže se odpovídající údaj v tabulkách a okno je na základě aktualizovaných tabulek znova celé překresleno. Není tedy třeba implementovat funkci `deleteLine()`. S editačním oknem v době jeho lifetimu komunikuje a manipuluje třída `EditingCore`.

[4] Návrhový vzor *Observer*.




Obrázek 4.3: UML diagram - třídy představující řádky v GUI

Nezanedbatelné množství času zabrala realizace na první pohled samozřejmých vlastností, které očekáváme od každého editoru kódu. Jednou z nich je například změna velikosti, barvy a fontu písma v textové oblasti. Java má ve svých knihovnách dialog sloužící k výběru barvy[5]. Pomocí něj se změna barvy písma realizuje snadno. Java už ale ve svých knihovnách nenabízí žádný dialog sloužící k nastavení písma. Pokusil jsem se jeden primitivní napsat, ale nebyl vyhovující. Nakonec se ukázalo jako nejefektivnější řešení stáhnout hotovou třídu `JFontChooser`[6]. Je to jediná třída v programu, kterou jsem nenapsal a zároveň není součástí Java Core API. Další z očekávaných vlastností editoru je, aby si uživatel nemohl omylem přemazat z aktuální záložky rozpracovaný kód jiným kódem. Těchto „drobností“ se během vývoje aplikace objevilo opravdu mnoho. Z tohoto pohledu by bývalo bylo mnohem jednodušší pojmout aplikaci jako plugin pro již hotový editor, který všechny zmiňované funkce nabízí.

[5] Třída `JColorChooser`.
[6] Zdroj [1].

# Kapitola 5

# Testování

## 5.1 Testování funkčních jednotek

Testování funkčních jednotek[1] jsem aplikoval převážně na třídy analyzující kód. Vytvořil jsem sadu poloautomatických testů. Testovalo se například, zda je lexikální analyzátor schopen analyzovat jakýkoliv VHDL kód. Zda se v kódu nevyskytují lexikální symboly, které ještě nezná. Podobným způsobem se testoval syntaktický analyzátor. Zkoušely se různé druhy generiků, portů, signálů, typů a konstant. Byly testovány neúplné a chybné vstupy a následně se zkoumaly reakce analyzátoru. Dále byly testovány všechny druhy šablon.

Tímto způsobem testování byly odhaleny chyby, které by se zpětně dohledávaly jen velmi obtížně. Zcela určitě se vyplatí investovat čas do psaní unit testů.

## 5.2 Testování GUI

Testování grafického prostředí aplikace probíhalo ze dvou důvodů:

1. **Ověření funkčnosti aplikace.** V softwarovém inženýrství se tento způsob testování nazývá *Black-box testing*. Aplikace je testována jako celek. Kontroluje se, zda z konkrétních akcí a vstupů dostáváme očekávané výsledky. Zda se aplikace chová tak, jak je očekáváno. U aplikace VHDL SGen probíhalo testování pouze manuálně. Nebyly použity žádné automatické testy. Jednotlivé prototypy aplikace byly testovány po každé iteraci.

2. **Zjišťování, zda je aplikace uživatelsky přívětivá.** U aplikace typu editor kódu je velmi důležité, aby ovládání bylo pochopitelné, intuitivní a rychlé. Uživatel, který bude aplikaci VHDL SGen používat, si bude chtít usnadnit práci. Nestojí o složitý neohrabaný nástroj. Toto testování mi tedy přišlo nejdůležitější. Povedlo se získat dva testery z řad kolegů, kteří zkoušeli navržená prostředí a poskytovali cennou zpětnou vazbu. Na základě těchto testů bylo grafické prostředí několikrát méně či více předěláno.

[1] Tzv. *White-box testing*. V Javě se dá s výhodou využít JUnit testů.

Pomocí těchto způsobů testování bylo v průběhu vývoje aplikace odhaleno a následně opraveno největší množství chyb. Zároveň se povedlo navigaci v GUI dostat do přehledného a uceleného stavu.

Uživatelské testy probíhaly na několika platformách. Testovalo se především v OS *Windows XP, Windows 7, Ubuntu 9.10* a *Kubuntu 9.10*. Aplikace byla od začátku vyvíjena tak, aby na žádném ze systémů nenastal problém[2]. Aplikace na všech systémech běžela dle očekávání. Neprojevilo se žádné netypické chování.

[2] Typické jsou například problémy se souborovými cestami.

# Kapitola 6

# Závěr

V rámci této práce vznikla aplikace Konfigurovatelný generátor základních struktur jazyka VHDL (VHDL SGen). Tato aplikace usnadňuje návrhářům hardwaru práci při tvorbě neustále se opakujících struktur, jakými jsou např. automaty a čítače. Mezi nejvýznamnější realizované funkce patří automatické generování celých bloků příkazů `case` na základě znalosti konkrétního signálu. Podstatnou součástí aplikace je také generátor testbenchů. V něm vidím největší přínos práce. Ve srovnání s generátory popsanými v rešerši poskytuje tato aplikace mnohem širší možnosti. Uživatel si může vytvořit šablonu přesně podle svých představ. Poté lze na jejím základě rychle generovat testbenche libovolných struktur a entit.

Aplikace by jistě mohla nabízet více možností při generování struktur, například příkazy k výpisu výstupů, automatické generování některých typických procesů a podobně. Grafické uživatelské prostředí by potřebovalo více nezávislých testerů, kteří by dopomohli k vytvoření intuitivní a snadno ovladatelné aplikace. Další chybějící funkcí je zvýrazňování syntaxe. V těchto bodech vidím nějvětší prostor k budoucímu pokračování práce.

Čas strávený nad touto aplikací mi poskytl výrazné prohloubení znalostí v oblasti VHDL kódu a především programovacího jazyka Java. Prakticky jsem si vyzkoušel návrh komplexního uživatelského prostředí, což považuji za velký přínos. Dále jsem v rámci práce vytvořil parser jazyka VHDL, na kterém jsem si prakticky vyzkoušel návrh, realizaci a ladění kódového analyzátoru.

# Literatura


[1] SourceForge.
http://sourceforge.jp/.

[2] Wikimedia Commons.
http://commons.wikimedia.org/wiki/Main_Page.

[3] P. Herout. *Java - grafické uživatelské prostředí a čeština.* Kopp, 2007.

[4] P. Herout. *Učebnice jazyka Java.* Kopp, 2008.

[5] Ing. Jiří Daněček, Ing. Martin Bloch. Programování v jazyku Java - přednášky.
https://webdev.felk.cvut.cz/courses/Y36PJV/.

[6] Ing. Zdeněk Troníček, Ph.D. Programovací jazyky a překladače - vzorová semestrální práce.
http://service.felk.cvut.cz/courses/X36PJP/tronicek/.

[7] J. Pinker, M. Poupa. *Číslicové systémy a jazyk VHDL.* BEN, 2006.

[8] M. Novotný, M. Bečvář, M. Daněk, J. Schmidt. Praktika návrhu číslicových obvodů - přednášky.
https://service.felk.cvut.cz/courses/X36PNO/.

[9] Prof. Dr.-Ing. Wolfram H. Glauert . VHDL Tutorial.
http://www.vhdl-online.de/tutorial/.

[10] Sun Microsystems. Dokumentace k Java 6 SE API, 2008.
http://java.sun.com/javase/6/docs/api.

[11] Sun Microsystems. The Java Tutorials, 2008.
http://java.sun.com/docs/books/tutorial/.

# Příloha A

# Seznam použitých zkratek

**API** Application Programing Interface

**AWT** Abstract Windowing Toolkit

**GUI** Graphical User Interface

**I/O** Input/Output

**JFC** Java Foundation Classes

**JIT** Just In Time

**JRE** Java Runtime Environment

**JVM** Java Virtual Machine

**OS** Operační Systém

**SE** Standard Edition

**Swing** Grafická knihovna jazyka Java

**VHDL** VHSIC Hardware Description Language

**VHDL SGen** VHDL Structure Generator

**VHSIC** Very High Speed Integrated Circuits

**WYSIWYG** What You See Is What You Get

# Příloha B

# Instalační a uživatelská příručka

## B.1 Požadavky

Ke spuštění aplikace je nutné mít v počítači nainstalované JRE 6.0 .

## B.2 Instalace a spouštění

Stačí překopírovat složku *VHDL SGen*, která se nachází v kořenovém adresáři přiloženého CD, do libovolné složky, ze které chcete aplikaci spouštět. Na operačním systému windows lze aplikaci spustit přímo poklikáním na soubor *VHDL_ SGen.jar*. V operačních systémech UNIXového typu je doporučeno spouštět aplikaci z příkazové řádky. Nacházíte-li se v adresáři *VHDL SGen*, lze aplikaci spustit následujícím příkazem:

```
java -jar VHDL_SGen
```

## B.3 Popis práce s aplikací

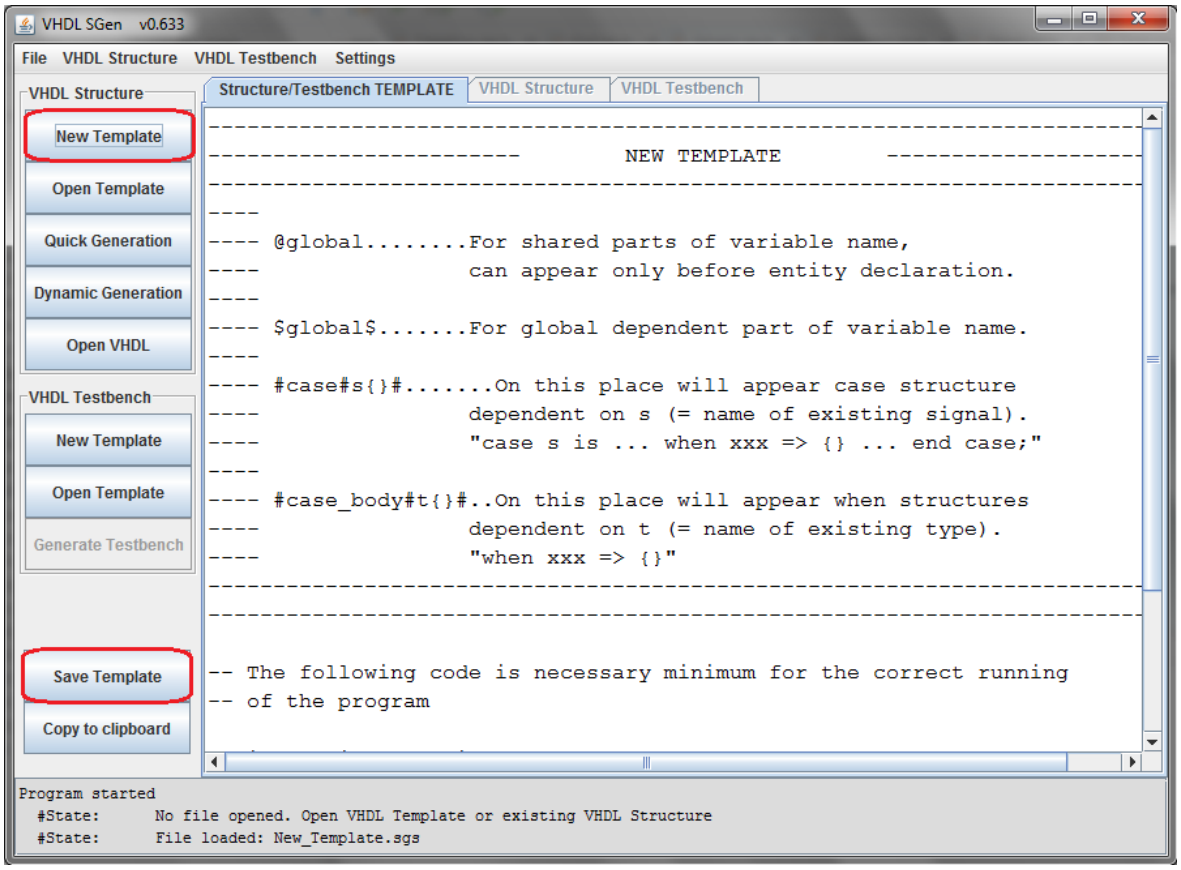
### B.3.1 Práce se šablonami VHDL struktur

#### Vytvoření šablony

Založit novou šablonu VHDL struktury lze kliknutím na tlačítko *New Template* v oblasti *VHDL Structure* (Obr. B.1). Alternativně lze použít klávesovou zkratku **CTRL-N**. V záložce *Structure/Testbench TEMPLATE* se otevře nový soubor obsahující přehled značek, které lze ke tvorbě šablony využít. Hotovou šablonu uložíte stiknutím tlačítka *Save Template* v dolní části levého menu (Obr. B.1). Alternativně lze použít klávesovou zkratku **CTRL-S**. Šablony VHDL struktur se ukládají do souboru s příponou „.sgs“.

V šabloně VHDL struktury se dají použít následující značky a příkazy:

- `@jmeno` ... Definice globální proměnné.
- `$jmeno$` ... Odkaz na globální proměnnou.



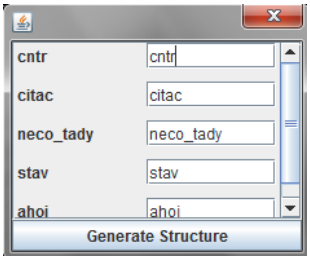

Obrázek B.1: Hlavní okno - tvorba šablony VHDL struktury

- `#case#jmeno_signalu{vkladana data}#` ... Příkaz k vytvoření bloku příkazu `case`. Jednotlivé větve se vytvoří podle výčtu hodnot, kterých nabývá typ zadaného signálu.
- `#case_body#jmeno_typu{vkladana data}#` ... Příkaz k vytvoření větví `when` příkazu `case`. Jednotlivé větve se vytvoří podle výčtu hodnot, kterých nabývá zadaný typ.

### Vyplnění šablony

K vyplnění šablony lze v aplikaci VHDL SGen využít dvou módů. V módu **Quick Generation** máte možnost vyplnit pouze jména globálních proměnných. V módu **Dynamic Generation** můžete měnit jména globálních proměnných, přejmenovávat entitu a architekturu, upravovat, přidávat a odebírat řádky portů, signálů, generiků, typů a konstant. Před vyplňováním je třeba zvolenou šablonu v programu otevřít. K tomu slouží tlačítko *Open Template* v oblasti *VHDL Structure*. Alternativně lze použít klávesovou zkratku **CTRL-R**.

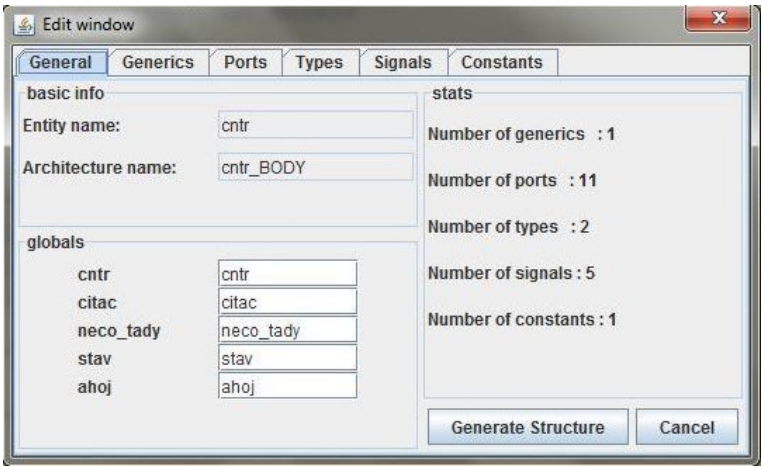
- **Quick Generation** Rychlou editaci spustíte kliknutím na tlačítko *Quick Generation*. Alternativně lze použít klávesovou zkratku **ALT-G**. Po kliknutí se vám zobrazí dialog s editovatelným seznamem globálních proměnných (Obr. B.2). Ty můžete dle libosti




Obrázek B.2: Dialogové okno rychlého generování

přejmenovávat. Po stisknutí tlačítka *Generate Structure* se v záložce *VHDL Structure* zobrazí vygenerovaná struktura.

- **Dynamic Generation** Dynamickou editaci spustíte kliknutím na tlačítko *Dynamic Generation*. Alternativně lze použít klávesovou zkratku **ALT-E**. Po kliknutí se otevře dynamické editační okno (Obr. B.3). Záložka *General* obsahuje jméno entity a architektury, editovatelnou tabulku globálních proměnných a přehled o počtu portů,




Obrázek B.3: Dynamické editační okno - záložka *General*

signálů atd. Další záložky obsahují seznamy seznamy generiků, portů, signálů, typů a konstant. Pomocí tlačítka *Add xxx* lze přidávat řádky. Křízkem na konci řádku lze řádek smazat. Stisknutím tlačítka *Generate* se v záložce *VHDL Structure* hlavního okna zobrazí vygenerovaná struktura.

### B.3.2 Tvorba testbenche

#### Vytvoření šablony

Založit novou šablonu VHDL struktury lze kliknutím na tlačítko *New Template* v oblasti *VHDL Testbench*. Alternativně lze použít klávesovou zkratku **CTRL-M**. V záložce

*Structure/Testbench TEMPLATE* se otevře nový soubor obsahující přehled značek, které lze ke tvorbě šablony využít. Hotovou šablonu uložíte stiknutím tlačítka *Save Template* (klávesová zkratka **CTRL-S**) v dolní části levého menu. Šablony VHDL struktur se ukládají do souboru s příponou „.sgt“.

V šabloně VHDL testbenche se dají použít následující příkazy:

- `$entity$` ... Na všechna místa, na kterých se vyskytuje tento příkaz, bude vloženo jméno testované entity (i do komentářů).
- `$component$` ... Na místě této značky se v testbenchi vytvoří celá komponenta, obsahující porty a generiky testované entity (pokud jsou).
- `$component_port$` ... Tento příkaz je nahrazen výčtem portů testované entity. Jak je partné z názvu, používá se uvnitř těla komponenty.
- `$component_generic$` ... Tato značka je při gnerování nahrazena výčtem generiků.
- `$signals$` ... Příkaz pro vložení signálů.
- `$port_map$` ... Namapování portů na signály.
- `$generic_map$` ... Namapování generiků na signály.

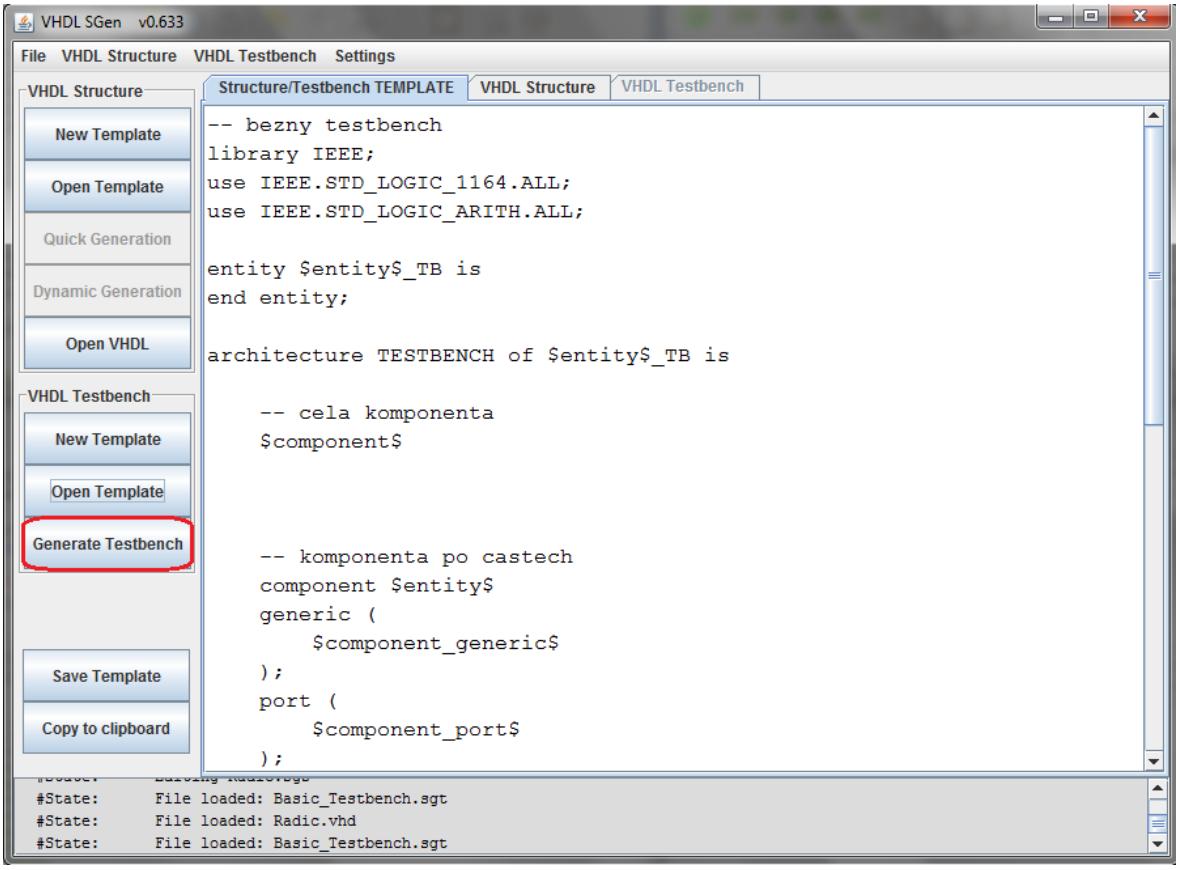
#### Generování testbenche

K vytvoření testbenche existující struktury je třeba otevřít šablonu testbenche nebo testovaný VHDL soubor. Šablona testbenche lze otevřít pomocí tlačítka *Open Template* (klávesová zkratka **CTRL-T**) v oblasti *VHDL Testbench*. VHDL soubor testované entity lze otevřít pomocí tlačítka *Open VHDL* (klávesová zkratka **CTRL-O**) v oblasti *VHDL Structure*.

Pokud máte otevřenou šablonu testbenche nebo VHDL strukturu, zaktivní se tlačítko *Generate Testbench* (Obr. B.4). Po stisknutí tlačítka (klávesová zkratka **ALT-T**) Vás program nechá vybrat druhý soubor (podle toho, který máte načtený). Pokud soubor vyberete, vygenerovaný testbench se zobrazí v záložce *VHDL Testbench* hlavního okna.

### B.3.3 Nastavení prostředí

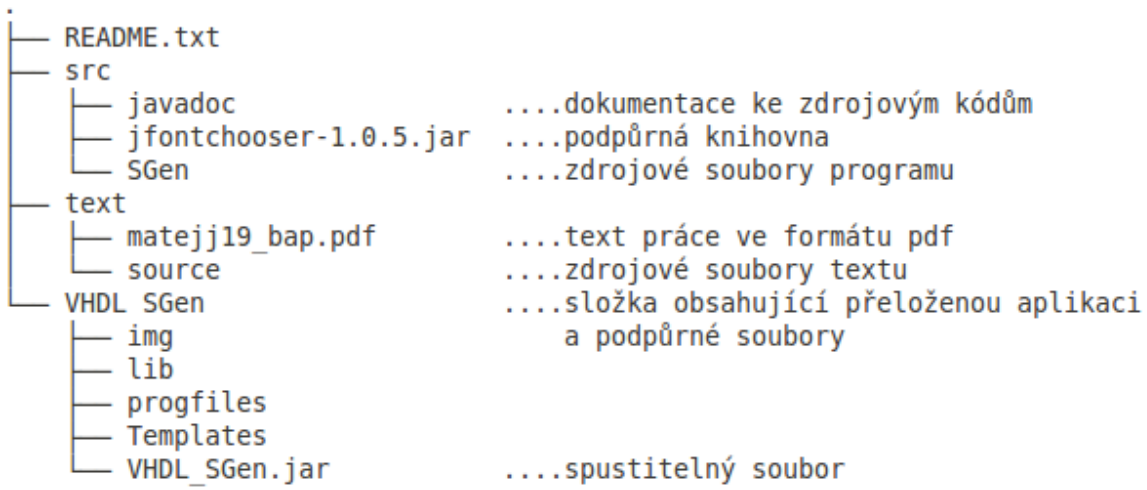
V menubaru v položce *Settings* lze nastavit typ písma, barvu písma a barvu pozadí textové oblasti.




Obrázek B.4: Hlavní okno - tlačítko *Generate Testbench*

# Příloha C

# Obsah přiloženého CD




Obrázek C.1: Obsah přiloženého CD

# Příloha D

# Ukázky šablon

## Šablona VHDL struktury - automat

### Šablona automatu

```
-- Ukazkova sablona automatu
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
-- Definice globalni promenne
@cntr

-- Odkazy na globalni promennou
entity $cntr$ is
port (
        $cntr$_out    : out std_logic_vector(3 to 0);
        $cntr$_en     : in std_logic;
        $cntr$_count  : in std_logic_vector(7 downto 0)
);
end entity;


architecture $cntr$_BODY of cntr is

-- Typ, podle ktereho se rozkopiruje prikaz case_body
type next_state is (WAIT_B0, WAIT_B1, POCITEJ);

-- Typ stav, jehoz hodnot muze nabyvat signal stav_akt
type stav is ('X', 'Z', '0');
signal stav_akt : stav;

begin
```

```
    -- Ukazka prikazu case
    -- Podle hodnot, kterych nabyva typ  stav signalu stav_akt se vytvori
    -- jednotlive vetve when
    DALSI_STAV : process (B0, B1, READY, STAV)
    begin
            #case#stav_akt#
    end process;

    -- Ukazka prikazu case_body
    -- Podle typu muj_typ se vytvori vetve when obsahujici vse, co je vlozeno
    -- mezi slozenymi zavorkami
    DALSI_STAV : process (B0, B1, READY, STAV)
    begin
            case stav_next is
                    #case_body#next_state{stav_next <=}#
            end case;
    end process;

    ...

end architecture;
```

### Vygenerovaná struktura

```
-- Struktura vygenerovana programem VHDL SGen podle puvodni sablony
-- V dialogovem okne byla globalni promenna cntr prejmenovana na citac
-- Definice globalni promenne byla automaticky odstranena

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;

entity citac is
port (
        citac_out   : out std_logic_vector(3 to 0);
        citac_en    : in std_logic;
        citac_count : in std_logic_vector(7 downto 0)
);
end entity;


architecture citac_BODY of citac is

type next_state is (WAIT_B0, WAIT_B1, POCITEJ);
type stav       is ('X', 'Z', '0');
```

```
signal stav_akt : stav;

begin
    -- Misto, kde se puvodne nachazel prikaz case
    DALSI_STAV : process (B0, B1, READY, STAV)
    begin
            case stav_akt is
                                when 'X'    =>
                                when 'Z'    =>
                                when '0'    =>
                                when others =>
            end case;
    end process;

   -- Misto, kde se puvodne nachazel prikaz case_body
    DALSI_STAV : process (B0, B1, READY, STAV)
    begin
            case stav_next is
                    when WAIT_B0 => stav_next <=
                    when WAIT_B1 => stav_next <=
                    when POCITEJ => stav_next <=
                    when others  => stav_next <=
            end case;
    end process;

    ...

end architecture;
```

## Šablona VHDL testbenche

### Testovaná entita

```
-- Priklad jednoduche testovane entity
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;

entity muj_counter is
port (
           clk             : in std_logic;
           counter_reset   : in std_logic;
           counter_enable  : in std_logic;
           counter_done    : out std_logic;
```

```
        );
end muj_counter;

architecture arch of muj_counter is
...
end arch;
```

**Šablona, podle které se bude tvořit testbench**

```
-- Ukazka sablony bezneho testbenche
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;

-- Na miste znacky entity bude skutecne jmeno testovane entity
entity $entity$_TB is
end entity;

architecture TESTBENCH of $entity$_TB is

    -- Na toto misto bude vlozena cela komponenta
    $component$

    -- Alternativne lze komponentu vkladat po castech
    component $entity$
    -- Zde by mohl byt jeste prikaz $component_generic$ slouzi k vlozeni generiku
    port (
          $component_port$
    );
    end component;

    -- Zde budou vloženy signály
    $signals$

begin

    mojePortMap : $entity$ PORT MAP(
          -- Namapovani portu na signaly
          $port_map$
    );

    -- Zde by mohl byt jeste prikaz $generic_map$ slouzici k namapovani generiku

    -- hodinovy signal
    clk_sig : process
    begin
```

```
        CLK<= '1';
        wait for 50 ns;
        CLK <= '0';
        wait for 50 ns;
    end process;

end architecture;
```

### Výsledný testbench

```
-- Vystup programu VHDL SGen - vysledny testbench
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;

-- Na miste znacky entity bude skutecne jmeno testovane entity
entity muj_counter_TB is
end entity;

architecture TESTBENCH of muj_counter_TB is

    -- Na toto misto bude vlozena cela komponenta
    component muj_counter
    port (
            clk            : in std_logic;
            counter_reset  : in std_logic;
            counter_enable : in std_logic;
            counter_done   : out std_logic
     );
    end component;

    -- Alternativne lze komponentu vkladat po castech
    component muj_counter
    port (
          clk            : in std_logic;
          counter_reset  : in std_logic;
          counter_enable : in std_logic;
          counter_done   : out std_logic
    );
    end component;

    -- Zde budou vloženy signály
    signal clk            :  std_logic;
    signal counter_reset  :  std_logic;
    signal counter_enable :  std_logic;
    signal counter_done   :  std_logic;
```

```
begin

    mojePortMap : muj_counter PORT MAP(
          -- Namapovani portu na signaly
          clk            => clk,
          counter_reset  => counter_reset,
          counter_enable => counter_enable,
          counter_done   => counter_done
    );

    -- hodinovy signal
    clk_sig : process
    begin
        CLK<= '1';
        wait for 50 ns;
        CLK <= '0';
        wait for 50 ns;
    end process;

end architecture;
```