Zpracoval: houzvjir@fel.cvut.cz

15. Návrhové vzory. Komponentový návrh, volná vazba versus těsná vazba. Oddělení politiky od mechanismu. Doménově specifické jazyky (DSL). (A7B36OMO)

Obsah

Návrhové vzory 2
Návrhový vzor Singleton 2
Návrhový vzor Abstract Factory 3
Návrhový vzor Observer 4
Návrhový vzor Facade 4
Návrhový vzor Adapter 5
Komponentový návrh 7
Volná vazba versus těsná vazba 8
Oddělení politiky od mechanizmu 10
Doménově specifické jazyky 11
Výhody DSL 12
Nevýhody DSL 12

Návrhové vzory

Návrhový vzor představuje obecné řešení problému, který se opakovaně objevuje při návrhu softwaru. Návrhový vzor není knihovnou nebo částí zdrojového kódu, která by se dala přímo vložit do našeho programu. Jedná se o popis či šablonu, jak řešit problém způsobem, který může být použit v různých situacích.

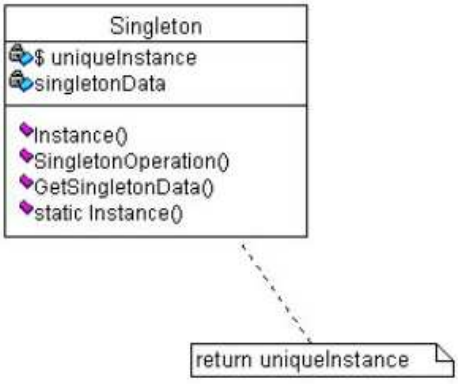
Objektově orientované návrhové vzory typicky ukazují vztahy a interakce mezi třídami a objekty, aniž by určovaly implementaci konkrétní třídy. Algoritmy nejsou považovány za návrhové vzory, protože řeší výpočetní problémy a nikoliv návrhové.

Návrhové vzory dělíme do tří skupin:

- **Tvořivé** – vyzkoušené postupy na řešení problémů s vytvářením objektů. Jsou to např. Abstract Factory, Builder, Factory Method, Prototype, Singleton.
- **Strukturální** - vyzkoušené postupy na řešení problémů při zpracování dat. Jsou to např, Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy,.
- **Behaviorální** - vyzkoušené postupy na řešení problémů s činností objektů a se změnami objektů v čase. Jsou to např. State, Strategy, Visitor, Template Method

Návrhový vzor Singleton

- **Definice:** Zajišťuje, že třída má pouze jednu instanci a poskytuje k ní globální přístup.
- **Problém:** Pro některé třídy je důležité mít pouze jednu instanci (například logovací služba, služba tisku, generátor ID, okenní systém - window manager). Pro zajištění jedné instance bychom mohli použít globální proměnnou, to nám umožní jednoduchý globální přístup, ale nezabrání nám to vytvořit více instancí. Lepší řešení je nechat třídu samotnou držet jedinou instanci sama sebe, tím zabráníme vytvoření jiné instance a zároveň třída může poskytnout globální přístup k této jediné instanci.
- **Řešení:** (Java) Například pro třídu Auto uděláme private konstruktor (aby nebylo možno vytvářet instance), vytvoříme private static Auto auto atribut, který bude držet naši jedinou instanci a public static Auto getInstance() metodu, která bude naši jedinou instanci vracet. Jediná instance se vytvoří při inicializaci třídy (ta má přístup k private konstruktoru), nikdo jiný ke konstruktoru nemá přístup, tudíž nemůže vytvořit jinou instanci a globální přístup k té jediné je zajištěn pomocí statické metody třídy (Auto.getInstance()) odkudkoliv v programu.



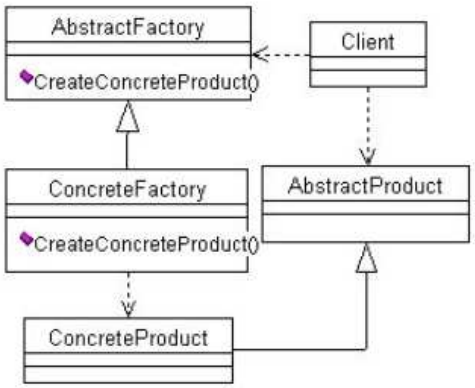
- **Výhody/nevýhody:** Některé služby musí být opravdu jedinečné a často k nim potřebujeme přistupovat, tudíž Singleton je pro takovou situaci vhodný. Bohužel, problémem může být právě globální dostupnost, která zbytečně svazuje program s danou implementací.
- **Použití** Singletonu nemusí být odůvodněno pouze nutností jednoho objektu. Velmi dobrým přístupem je využít tohoto vzoru v době testování, abychom se nemuseli zabývat existencí více instancí jedné třídy.

Dalším příkladem využití vzoru může být řízení přístupu do databáze, kdy chceme zajistit neplýtvání s omezenými zdroji. Obdobné důvody by mohly nastat i u řízení tiskových výstupů nebo zajištění jednoho globálního objektu, který uchovává nastavení aplikace.

Návrhový vzor Abstract Factory

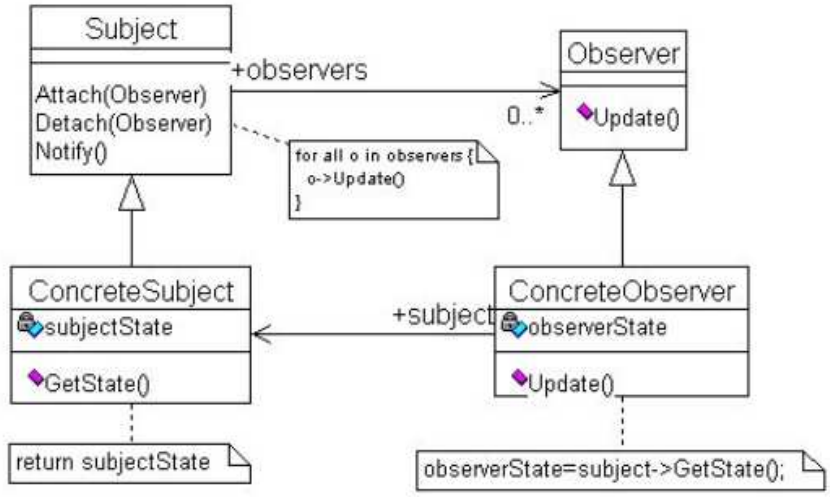
- **Definice**: Poskytuje rozhraní pro vytváření rodin souvisejících nebo závislých objektů aniž bychom specifikovali jejich konkrétní třídu.

- **Problém**: Grafické rozhraní má více tzv. look-and-feels (vzhled). Look-and-feel definuje vzhled a chování jednotlivých prvků rozhraní (tlačítko, scrollbar, ...). Aby aplikace byla přenositelná a bylo možno tyto look-and-feely jednoduše měnit, není možno natvrdo do kódu napsat třídy konkrétního look-and-feelu. Toto je možno vyřešit vytvořením abstraktní továrny, která definuje metodu pro vytvoření každého typu prvku rozhraní a konkrétní implementace továrny pro daný look-and-feel pak vytváří konkrétní instance těchto prvků. Klient není závislý na konkrétní implementaci, volá metody definované v abstratkní továrně pro vytvoření prvků rozhraní a pouhým nahrazením továrny za jinou lze změnit celý look-and-feel aplikace za běhu.
- **Řešení**: (Java) Vytvoříme abstraktní třídu AbstractFactory obsahující metody getButton, getScrollBar, poté vytvoříme konkrétní implementace pro Windows a Mac, třídu WindowsFactory extends AbstractFactory implementující metody v rozhraní pro platformu Windows, třídu MacFactory extends AbstractFactory implementující metody rozhraní pro platformu Mac. Při spuštění programu pak máme proměnnou AbstractFactory, do které vložíme instanci buď WindowsFactory nebo MacFactory a vytváříme rozhraní pomocí dostupných metod, záměnou instance v proměnné pak můžeme změnit vzhled, jednoduše a rychle.



Návrhový vzor Observer

- **Definice:** Definuje one-to-many závislost mezi objekty tak, aby když objekt změní stav, všichni na něm závislí jsou o tom informování.
- **Problém:** Při návrhu se většinou snažíme systém postavit jako sadu oddělených tříd/komponent, které spolu interagují, nicméně tyto třídy spolu potřebují komunikovat co se týče jejich stavu, ale to nemůžeme/nechceme udělat tak, že bychom je natvrdo svázali (těsná vazba). Řešením je vzor pozorovatel, který definuje pozorovaný objekt, který je pozorován více pozorovateli, kteří jsou informování o každé změně v pozorovaném objektu.
- **Řešení:** (Java) Máme rozhraní Observer s metodou update, která je zavolána, když chceme pozorovatele informovat o nějaké změně. Dále máme rozhraní Subject, které má základní metody pro přihlášení k odběru zpráv, odhlášení z odběru a vyvolání notifikace (odeslání zprávy o změně). Konkrétní pozorovatel se pak přihlásí u konkrétního subjektu pro příjem zpráv o změně a pokud se daný objekt změní, zavolá metodu update pozorovatele, který si pak vyžádá změnu stavu metodou např. getState (jakou metodu zavoláme pro zjištění stavu objektu záleží na konkrétní instanci subjektu, toto není definováno v rozhraní Subject, další možností je stav objektu posílat rovnou se zprávou update).



- **Výhody/nevýhody:** Výhodou je volná vazba mezi komponentami, nevýhodou, že pozorovatelé o sobě navzájem nevědí.

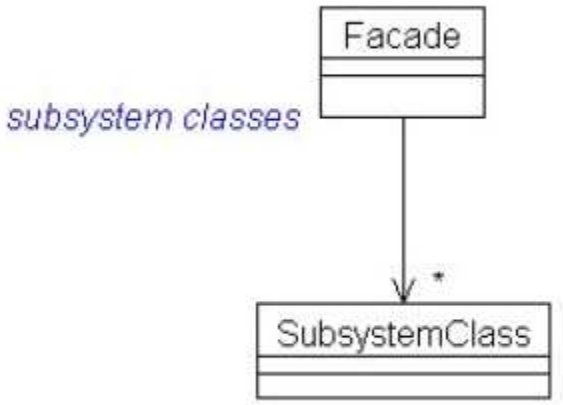
Návrhový vzor Facade

- **Definice:** Poskytuje sjednocené rozhraní k sadě různých rozhraní subsystému. Fasáda definuje rozhraní na vyšší úrovni, které tak činí subsystém jednodušeji použitelným.

- **Problém:** V programu máme určitý subsystém, u kterého bychom rádi měli dvě úrovně přístupu. Jednu pro klienty, kteří pouze vyžadují určitou funkci a druhou pro pokročilé klienty, kteří zároveň vyžadují i různé další informace a funkce. Aby se všichni nemuseli učit a porozumět složitému rozhraní pro pokročilé klienty, vytvoříme

nad subsystémem fasádu, která poskytne zjednodušený pohled na subsystém a je mnohem jednoduše použitelná. Dalším příkladem může být subsystém, který se v průběhu let různými úpravami stal příliš složitý a rádi bychom jeho rozhraní zjednodušili nebo vylepšili.

- **Řešení**: (Java) Implementace je velmi jednoduchá. Klientům poskytneme třídu s nějakým rozhraním (které nějak vhodně abstrahuje a zjednodušuje subsystém) a v této třídě pak veškeré požadavky delegujeme na původní subsystém. Čili funkce zůstává stejná, ale použití je mnohem jednodušší (např. namísto volání pěti metod v určitém pořadí stačí zavolat jednu a ona to udělá za nás).

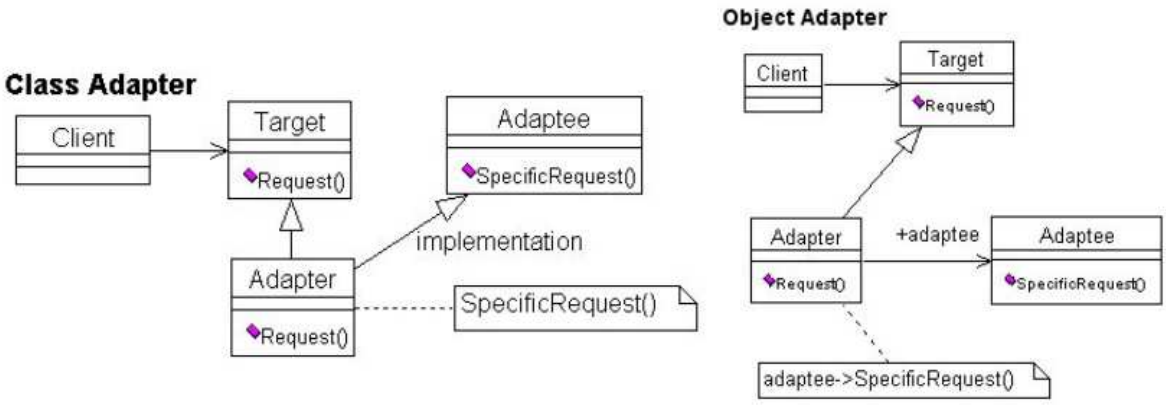


- **Výhody/nevýhody:** Výhodou je snížení vazeb mezi klientem a subsystémem, podporuje vrstvení systému a možnost poskytnout určitým klientům vhodnější rozhraní.

Návrhový vzor Adapter

- **Definice**: Konvertuje rozhraní třídy na rozhraní jiné třídy, které je očekáváno klientem. Adaptér umožňuje spolupráci tříd, které by spolu jinak nemohli spolupracovat vzhledem k nekompatibilním rozhraním.

- **Problém**: Potřebujeme využít nějakou třídu, jejíž rozhraní s námi není kompatibilní. V takovém případě můžeme vytvořit adaptér, který upraví rozhraní dané třídy (poskytne nám vhodné rozhraní a volání deleguje na původní třídu) pro naše potřeby.
- **Řešení**: (Java) Implementačně existují dva přístupy (Adapter Class vs. Adapter Object). Adapter Class se implementuje tak, že máme nějaké požadované rozhraní a adaptované rozhraní a od obou podědíme náš adaptér. Adapter Object se implementuje pomocí kompozice, kdy dědíme pouze od požadovaného rozhraní a instanci adaptované třídy máme uloženou v atributech (delegujeme na ní volání).



- **Výhody/nevýhody:** Zvyšuje znovupoužitelnost tříd díky oddělení rozhraní a implementace.

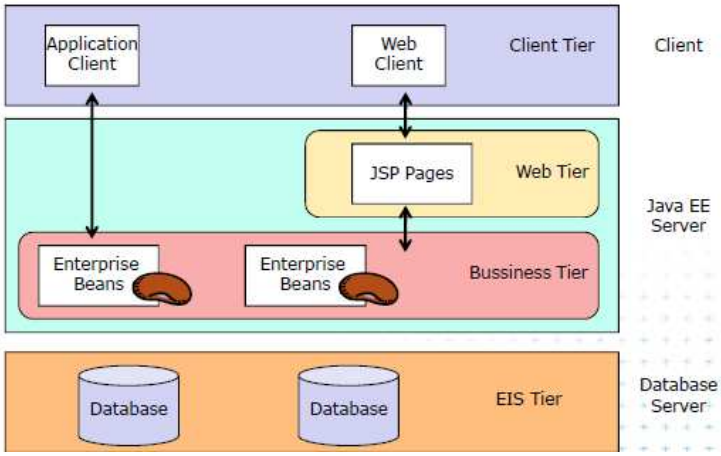
Další návrhové vzory jsou slušně popsány např. zde: http://objekty.vse.cz/Objekty/Vzory.

Komponentový návrh

Hlavní myšlenkou komponentového návrhu je tzv. **separation of concerns** ("oddělení starostí"), to znamená rozdělení celkové funkčnosti systému na jednotlivé **komponenty**, které budou sdružovat spolu související funkce (tzv. high cohesion - princip viz. GRASP). Tím vytvoříme sadu **volně vázaných** (tzv. **low coupling** - princip viz. GRASP) komponent a zároveň komponenty, které jsou dobře **zapouzdřené** (encapsulated) a **soudržné** (cohesive). Výhodou takového přístupu je možnost znovu využít již vytvořené komponenty, jasné oddělení povinností (zároveň se tím dobře oddělují chyby, chybu můžeme hledat v dané komponentě a nemusíme řešit, jaké vztahy a s jakými komponentami má) a celkově jednodušší správu systému. Nevýhodou je vyšší režie, namísto přímého volání voláme přes různé prostředníky zajišťující komunikaci komponent. Příkladem takového komponentového návrhu (architektury) je **EJB** (session management, transaction management, db connection management, use authentication and authorization, asynchronous masaging) nebo **SOA** (Service-Oriented Architecture).

Schéma integrace založené na službách (SOA):

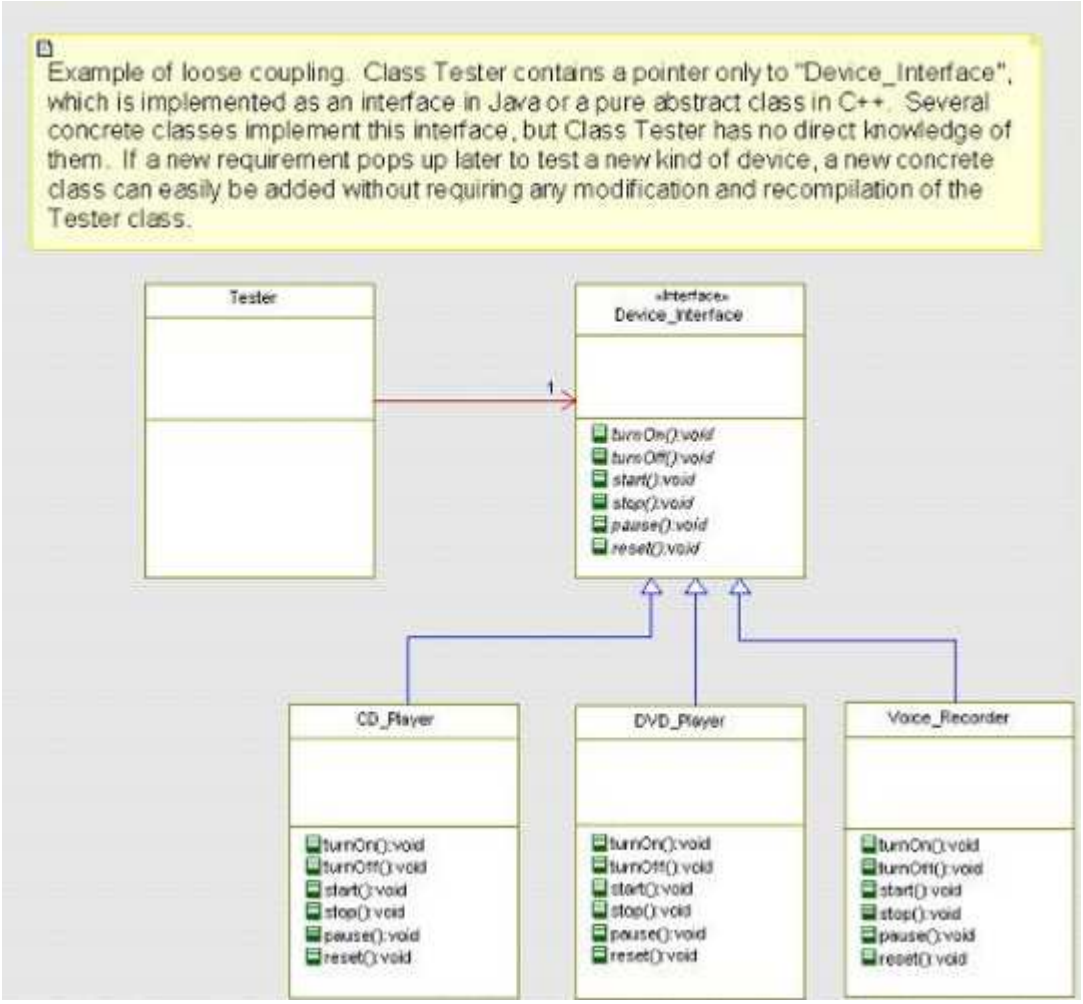
Schéma EJB:



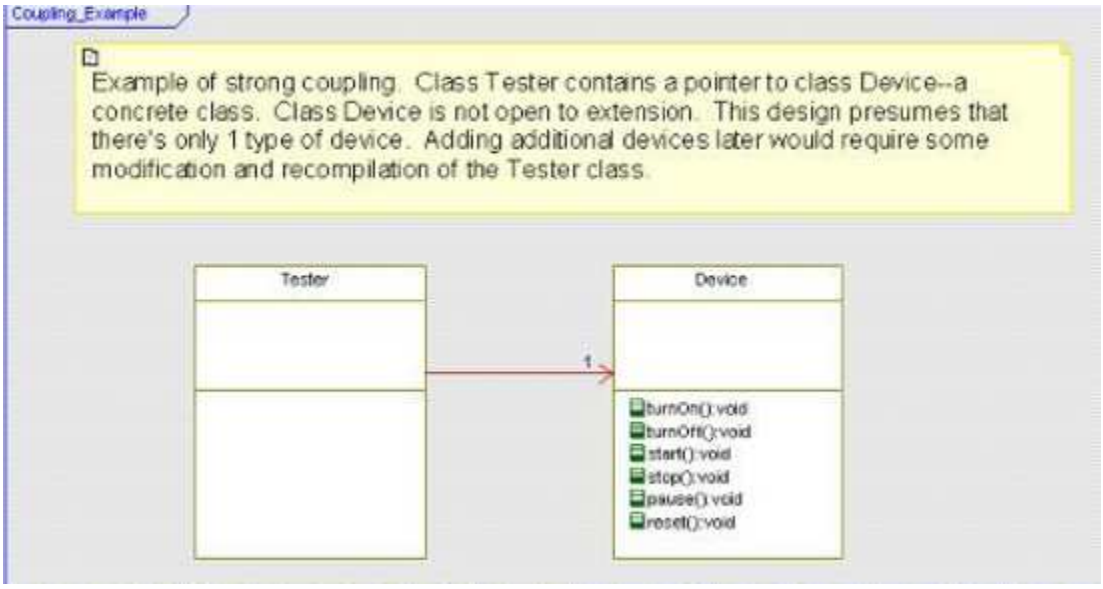
Volná vazba versus těsná vazba

Vazba je **míra znalosti jedné třídy o druhé**. Tím není myšleno znalost metod a atributů třídy (tohoto se týka zapouzdření), ale spíše znalost třídy samotné (znalost toho, že vůbec existuje).

- **Volná vazba** (loose coupling) - závisla třída **neobsahuje** ukazatel/referenci na konkrétní třídu s požadovaným chováním, ale obsahuje referenci pouze na **rozhraní** (interface), které může být implementováno různými třídami. To umožňuje **rozšiřitelnost**, nová konkrétní implementace může být jednoduše přidána, aniž by bylo nutné znovu kompilovat nebo nějak měnit závislou třídu.



- **Těsná vazba** (tight coupling) - závislá třída **obsahuje** ukazatel/referenci přímo na konkrétní třídu, která poskytuje požadované chování.



Kritéria pro hodnocení vazeb:

(podle http://d3s.mff.cuni.cz/teaching/programming_practices/lecture07.html)

- velikost
  - odpovídá počtu vazeb mezi moduly
  - parametry metod, počet public metod, ...
- viditelnost
  - vazby pokud možno co nejvíce explicitní
  - parametry metody vs. globální data
- flexibilita
  - jednoduchost změny vazeb mezi moduly
  - USB konektor vs. pájený spoj

Oddělení politiky od mechanizmu

Způsob návrhu, který říká, že **mechanizmus** (implementace) **by neměl diktovat nebo příliš omezovat politiky** (pravidla).

Příkladem budiž oddělení mechanizmu a politik u čipových karet. Mechanizmus (čtečka, zámek) nerozhoduje o tom, kdo má a nemá přístup. Toto rozhodnutí dělá centralizovaný server, který pravděpodobně rozhoduje na základě pravidel v databázi (kdo má kam přístup - politiky). Konkrétní rozhodnutí, kdo má kam přístup, mohou být **nezávislé** přidávána a odebírána pouze změnou databáze. A pokud by i schéma databáze bylo nevyhovující, lze ho nahradit, zatímco mechanizmus (zámky, čtečky) funguje nezávislé dál.

Dalším příkladem a doporučením je používání konfiguračních souborů, aby nebylo nutné při změně parametrů modifikovat zdrojový kód programu , což je vždy dražší.

Doménově specifické jazyky

Zdroj: http://cs.wikipedia.org/wiki/Dom%C3%A9nov%C4%9B_specifick%C3%BD_jazyk

Doménově specifický jazyk je programovací jazyk, který je prostřednictvím vhodné abstrakce a výrazového slovníku zaměřen na omezenou, konkrétní problémovou doménu.

Jako příklad doménově specifických jazyků uveďme mini-jazyky systému UNIX, mezi něž patří shellové nástroje awk a sed nebo nástroje pro tvorbu software make, yacc.

Doménově specifické jazyky jsou stejně jako jiné jazyky o komunikaci. Pomáhají přemostit propast mezi zúčastněnými stranami softwarového vývoje a programátory. DSL může zjednodušit porozumění komplikovaného bloku kódu, a tak zlepšit produktivitu vývojářů. DSL může zlepšit komunikaci s doménovými experty. Script napsaný v DSL je snadno čitelný pro doménové experty, kteří mohou porozumět tomu, jak jsou jejich myšlenky předkládány systému.

Klíčové vlastnosti, které vymezují rozdíl mezi DSL a obecným programovacím jazykem jsou:

- **Zaměření na doménu**: Limitované jazyky jsou užitečné pouze, pokud jsou zaměřeny na malou část problémové domény. V DSL nemůžeme vybudovat celý systém. To co dělá DSL užitečné je úzké zaměření na konkrétní problém.
- **Limitovaná výrazovost** (limited expressivity): Obecné programovací jazyky nabízejí množství vlastností umožňující řešit téměř libovolný problém, avšak na nižší úrovni abstrakce za použití často složitých konstrukcí, než by umožnil specifický přístup úzce zaměřený na úzce zaměřenou část problému. DSL disponuje pouze minimem vlastností pokrývající podporu problémové domény.
- **Povaha jazyka** (language nature): DSL je programovací jazyk a jako takový by měl mít smysl pro plynulost. Specifickou povahu jazyka vyjadřuje nejen volání jednotlivých metod, ale i způsob jakým jsou tyto metody na sebe navázány. Povahou jazyka míníme celkový kontext, který vznikne spojením volání metod do větších na sebe navazujících celků.

Interní DSL je zvláštní použití obecného programovacího jazyka. Script v interním DSL je validní kód tohoto jazyka, ale používá pouze podmnožinu jazykových vlastností ve specifickém stylu. V mnoha případech je interní DSL implementován jako tenká vrstva nad abstrakcí obecného programovacího jazyka.

Externí DSL je oddělený od jazyka použitého v aplikaci. Externí DSL má vlastní syntaxi, ale může používat syntaxi jiného jazyka (XML). Script v externím DSL je obvykle transformován do jazyka aplikace použitím kompilátoru nebo interpretu. Příkladem externích Doménově specifických jazyků jsou regulární výrazy, SQL, Awk, XML.

Výhody DSL

- **Zlepšení produktivity vývojářů** Programový kód v DSL je čitelnější než při použití knihovního API. Tato čitelnost není jen estetickou vlastností. Pro vývojáře je snazší porozumět programovému kódu, najít chybu a modifikovat kód.
- Limitovaná výrazovost DSL dělá těžší psát špatný kód a snadnější vidět chyby.
- Ze stejného důvodu jako podporujeme jasná jména proměnných, psaní dokumentace k metodám a jasné programové konstrukce, bychom měli podporovat používání DSL.
- **Komunikace s doménovými experty** – DSLnám umožní lépe komunikovat s doménovými experty, kteří nemusí znát obecný programovací jazyk. Doménoví experti mohou být dokonce zapojeni do vývoje software.
- **Změny kontextu za běhu systému** Přesunutí logiky systému z doby kompilace, kde by byla běžně načtena systémem do doby běhu systému je častým důvodem pro návrh DSL. Definováním DSL získáme možnost měnit logiku systému v různých prostředích.

Nevýhody DSL

- Cena za učení specifického jazyka
- Náklady na vytvoření DSL
- Uzavřenost jazyka – může se stát, že postupným přidáváním funkcí vznikem Touringově kompletní jazyk. Je lepší tomu předejít vytvořením více malých DSL než dopustit vytvoření obecného jazyka.