Creational Patterns:

- Builder:
  - Nejčastěji řeší případ, kdy by měl konstruktor příliš mnoho parametrů a jejich vyplňování by bylo problematické a nespolehlivé.
  - Jedná se v podstatě o objekt pro tvorbu objektů.
  - Pro plynulost (fluent) může vracet sám sebe v každé metodě a řetězit volání metod.
- Factories:
  - Method:
    - Hlavní výhodou je lépe pojmenovat „konstruktor“
    - V jiných jazycích je často problém, že není možné mít dva stejné konstruktory se stejnými vstupy (i když jsou jinak pojmenovány)
  - Factory:
    - Oproti method je nevýhoda, že nemůžu mít privátní konstruktory
    - Výhoda je stejná - přesnější pojmenování „konstruktoru“
  - Obecně se dá říct, že je dobré návrhový vzor využít při potřebě přesněji pojmenovat konstruktor
- Prototype:
  - Návrhový vzor umožňuje vytvářet nové instance na základě existujících instancí, tzv. prototypů, které naklonuje. (Na tomto principu je postavené např. Vytváření instancí v javascriptu).
  - Možné využít například pokud konstruktor provádí složitou logiku a klonování je výkonově efektivnější, než znovu a znovu vytvářet
- Singleton:
  - Vzor který umožňuje zajistit globální přístup k (jediné) instanci nějaké třídy
  - Existuje pouze jedna instance (privátní konstruktor)

Structural Patterns:

- Adapter:
  - Používá se při práci s komponentou, která má nestabilní nebo s mojí aplikací nekompatibilní rozhraní. Umožňuje komponentu obalit vlastním rozhraním a tak aplikaci úplně odstínit od rozhraní původního.
  - Vzor je v podstatě prostředník mezi mým rozhraním a rozhraním komponenty (které je pro aplikaci neznámé)
- Bridge:
  - Základní princip je oddělení měnícího se rozhraní od měnící se implementace tohoto rozhraní.
  - Narozdíl od Adapteru, který odděluje hotové, neměnné rozhraní od nějaké implementace, tak Bridge počítá s tím že se bude měnit jak implementace, tak i abstraktní rozhraní, pomocí kterého klient s komponentou pracuje (a mezi těmito pohyblivými částmi se staví „most“)
- Composite:
  - Ideální řešení pro situace, kdy se pracuje se stromovou strukturou. (Například rekurzivně zanořené navigační menu, kdy každá položka

může být buď položka menu, nebo další podmenu)… (nebo například adresářová struktura)
    - Poskytnu stejné rozhraní jak položkám, tak dalším podmenu, které obsahují další položky - s těmito složenými objekty pak pracuju jako s jednoduchými objekty.
    - Nemá smysl zkoušet vymýšlet jinou reprezentaci takových (stromových) struktur
- Decorator:
    - Pokud potřebuju skupině tříd předat další funkcionalitu, ale zdědění není vhodné (např. Knihovna třetí strany se zapečetěnými třídami…, porušil bych pravidlo že každá odvozená třída musí jít použít ve všech případech, kdy je použita třída základní)
    - Používá se například v GUI Aplikacích jako např. posuvníky na krajích obrazovky - abych ho nemusel implementovat na každý Control, tak vytvořím dekorativní, který control obalí, vykreslí posuvníky a přidá funkcionalitu, zbytek funkcionality se deleguje na původní třídu
    - Dekorátor převezme zodpovědnost pouze za specifický úkol a o zbytek se stará původní třída
- Facade:
    - Vytváří se k vytvoření jednotného rozhraní pro celou logickou skupinu tříd, které se tak sdruží do subsystému
    - Pokud spolu třídy logicky souvisí tak je můžu sdružit a získat jednotné rozhraní…
- Flyweight:
    - Šetří paměť při úkolech, kde potřebuju vytvořit velký počet instancí

Behavioral Patterns:

- Chain of responsibility:
    - Umožňuje oddělit odesílatele požadavku o d jeho příjemců, kterých může být více.
    - Požadavek je předáván mezi příjemci až dokud nedorazí k tomu, který má kompetenci jej vyřídit (!implementace by měla počítat se situací že nebude nalezen!)
    - Někdy se kombinuje se vzorem Composite (Tree of responsibility)
    - Některé frameworks úpužívají chain of responsibility na implementaci událostnímho modelu
- Interpreter:
    - Definuje jakým způsobem implementovat interpretaci nějakého jazyka pomocí OOP (vlastní programovací jazyk, vyhodnocení matematického výrazu, html)
    - Vzor řeší jak kód jazyka reprezentovat jako množinu objektů a jak jej následně spustit.
- Iterator:
    - Zavádí samostatný objekt, který umožňuje jednoduché lineární procházení kolekcemi, aniž bych musel znát vnitřní strukturu těch kolekcí.
    - Někdy se místo označení iterator používá enumerator
    - Většinou poskytuje rozhraní obsahující: Current (prvek na aktuální pozici), next (přesune kurzor na další prvek v kolekci), reset (přesune

kurzor na první pozici)

- Mediator:
  - Zavádí prostředníka mezi přímou komunikací několika objektů. Tím snižuje počet vazeb mezi objekty a reguluje jejich odpovědnost.
  - Množina objektů komunikuje pouze s mediátorem a on s nimi - tím se snižuje počet vazeb
  - Sníží se potřeba znalosti objektů mezi sebou a kód se jednodušší
  - Např. chat, formulářové prvky (nemusí na sebe vzájemně odkazovat a budou komunikovat s jedním (mediatorem) objektem, který podle aktuálních akcí upravuje stav formuláře)
- Memento:
  - Řeší uložení vnitřního stavu objektu, aniž by byl porušen princip zapouzdření.
  - Je možné si doimplementovat historii (undo / redo) - ale v základní verzi vzoru se jedná pouze o uložení jednoho stavu na který se lze vrátit
- Observer:
  - Umožňuje objektu spravovat řadu pozorovatelů, kteří reagují na změnu jeho stavu voláním svých metod
  - Objekt by neměl vědět o objektech které jsou na něm závislé, protože ho to zesložiťuje a znepřehledňuje
  - Využívá se v systémech, které jsou založené na zpracování událostí
- Proxy (zástupce):
  - Zapouzdření instance jiného objektu, nebo přidání pomocné funkčnosti
  - Umožňuje řídit přístup k celému nebo částečnému rozhraní objektu, přes jiný zástupný objekt
  - Je několik variant
  - Proxy má stejný interface jako základní objekt
  - Pro vytvoření proxy replikuju existující interface objektu
  - Přidám funkcionalitu redefinovaným funkcím
- State:
  - Umožňuje objektu změnit své chování, které je závislé na stavu objektu.
  - Nahrazuje složité větvení uvnitř objektu
- Strategy:
  - Umožňuje za běhu aplikace vyměnit algoritmus za jiný bez nutnosti změny kódu programu
  - Algoritmy jsou přehledně zabalené jako moduly
  - Bez použití strategii by bylo nutné implementovat switch nebo nějaké větvení což s narůstajícím počtem algoritmů může být nepřehledné a každý nový algoritmus vyžaduje zásah do kódu
- Template:
  - Definuje kostru algoritmu (jeho jednotlivé kroky), potom potomci kroky implementují a představují zaměnitelné algoritmy
  - V podstatě vytvořím vzorový algoritmus, kdy metody a vlastnosti uvnitř algoritmu nejsou implementované, ale implementují je až potomci (což ve Swiftu trochu komplikuje nemožnost psaní abstraktních tříd, ale je to samozřejmě možné)