**Structural design patterns - Decorator, Proxy a Flyweight**

## **Decorator**

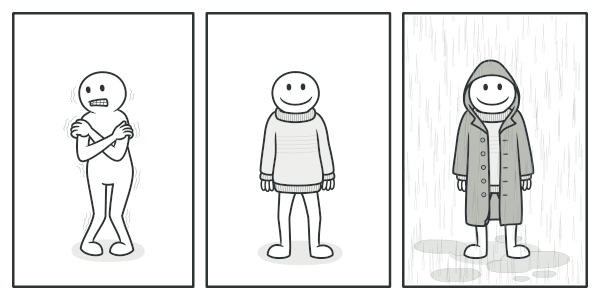
Návrhový vzor decorator je typickým představitelem zásady užití kompozice před dědičností. Za předpokladu, že chceme přidat další dodatečné chování (vlastnost) nějakému objektu, pak pro tuto vlastnost vytvoříme vlastní objekt, do kterého ten původní zabalíme (dekorujeme jej).Tento postup je obzvláště výhodný, pokud mohou existovat objekty s libovolnými kombinacemi těchto vlastností, jelikož při použití prosté dědičnosti by došlo ke kombinatorické explozi počtu tříd. Zároveň by bylo velmi obtížné daný kód udržovat, protože by přidání jedné další vlastnosti znamenalo přidání velkého množství nových tříd.

**Příklad 1:** Představme si, že si chceme uvařit kafe. Máme různé varianty jak si ho připravit např. kafe s cukrem nebo kafe s mlékem. Příprava bude spočívat v tom, že si uvaříme obyčejné kafe a poté si ho budeme podle sebe upravovat (dekorovat) a tak nám vznikne dekorované kafe.

**Příklad 2:** Představte si, že vyrábíte trička. Místo toho aby jste dělali žluté, modré a červené tričko, budete mít jen tričko a to dekorovat žlutou, modrou a červenou.

**Příklad 3:** Představte si, že děláte validátor. Můžeme ho dekorovat validací zespoda (minimální délky), zeshora (maximální délky), kontrolu na velká písmena atd…

**Příklad 4:** Streamy v Javě

****

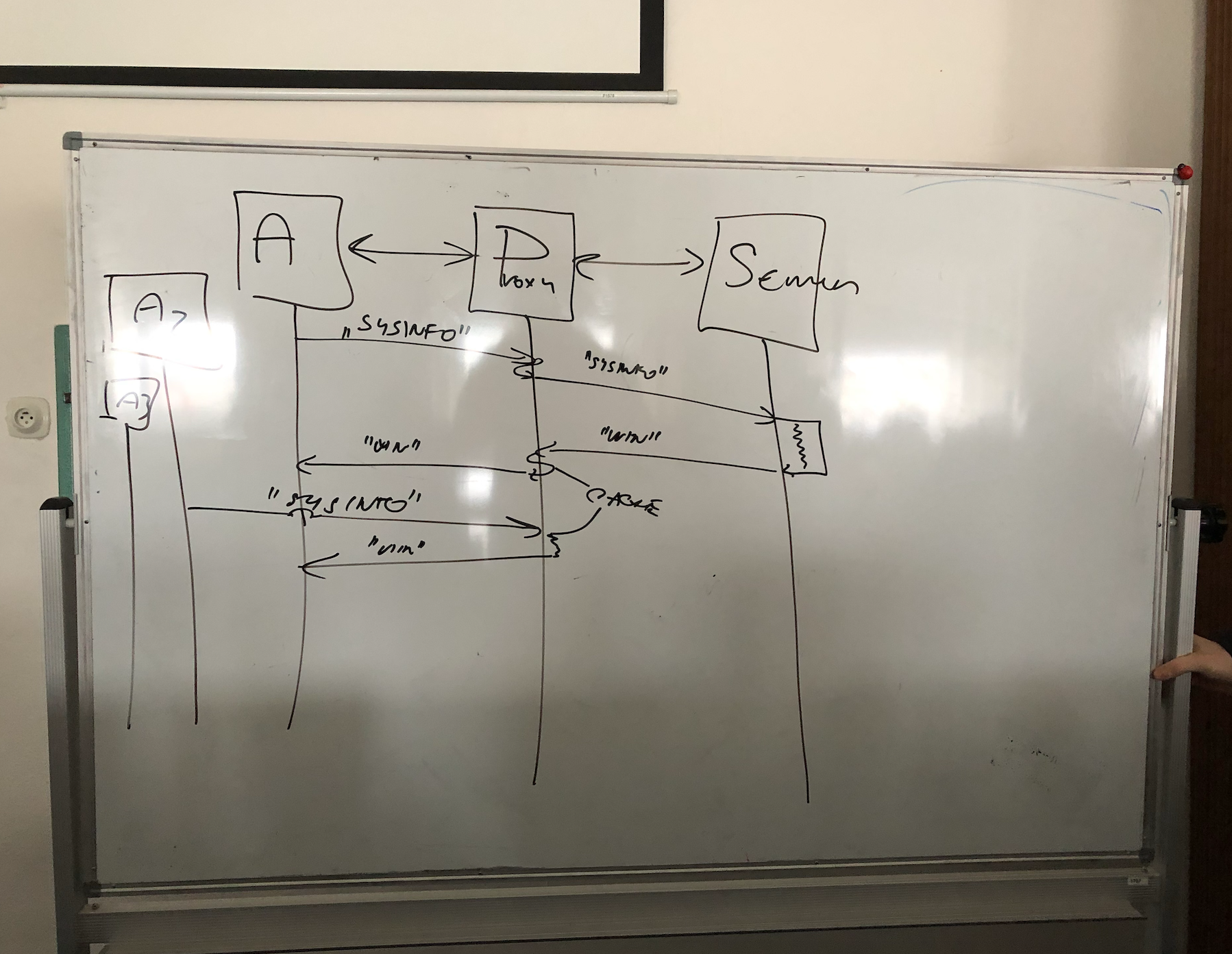
## **Proxy**

Návrhový vzor Proxy se hodí hned v několika případech. Jeho asi nejčastější použití je zapouzdření instance jiného objektu nebo přidání pomocné funkčnosti. Proxy nám tedy umožňuje řídit přístup ať už k celému či částečnému rozhraní objektu přes nějaký jiný zastupující objekt (proxy - zástupce, reprezentant).

*Příklad od Mandíka ve škole.*

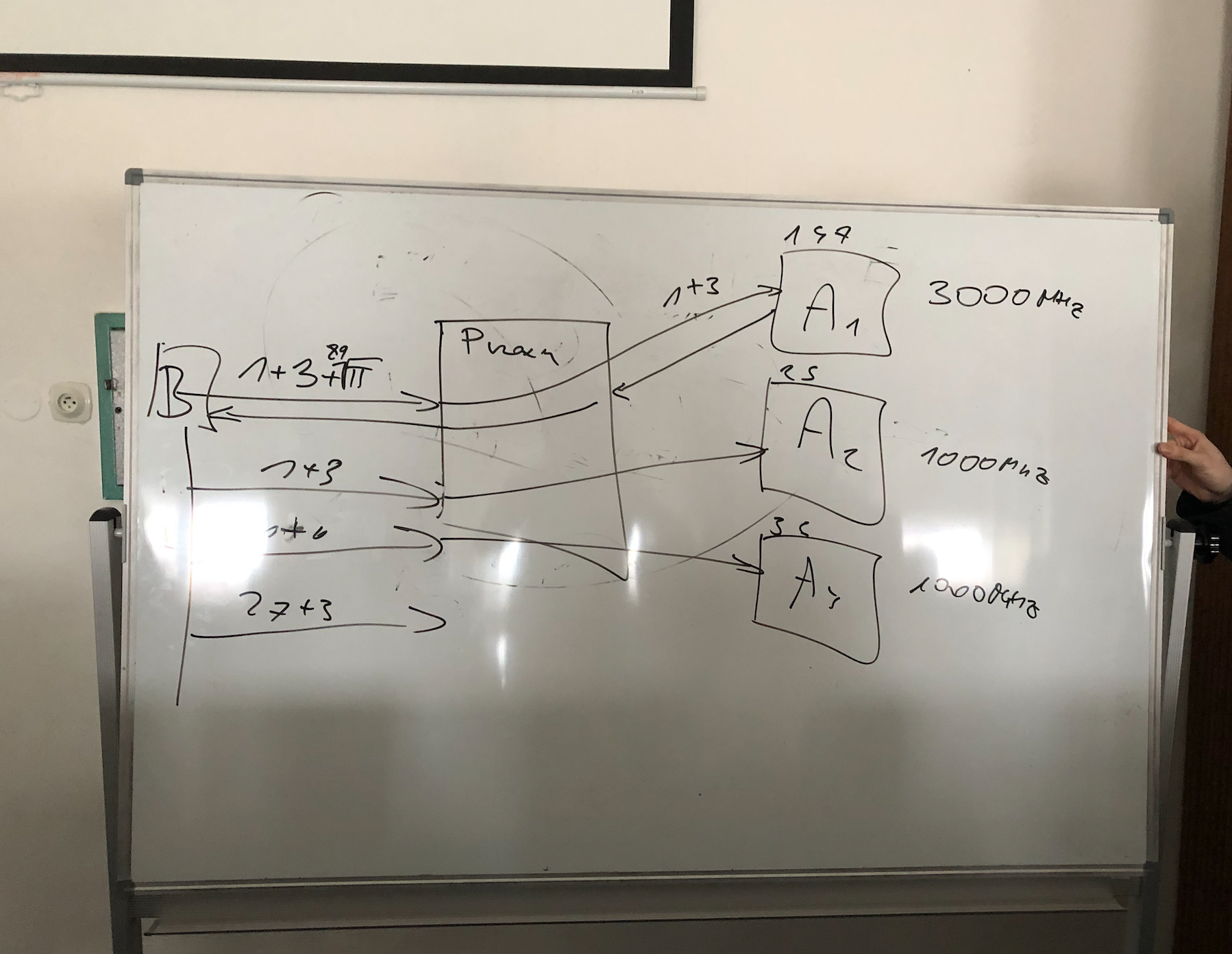
*Proxy - caching*

Proxy pracuje na principu cachování, čímž ulevuje serveru. V praxi to znamená, že do sebe požadovaná data (otázku) načte po prvním dotazu uloží a při dalším stejném dotazu je načítá ze své cache. Příkladem může být například server ministerstva školství, jenž pravděpodobně dostal po ohlášení vlády, že školní výuka je z důvodu nemoci dočasně přerušena, tisíce až desetitisíce žádostí o načtení. Neboť by to server sám o sobě pravděpodobně nezvládl, nastal moment, kdy proxy zachrání den.



*Reverse Proxy - load balancer*

Reverse Proxy funguje na principu množiny serverů, jenž všechny jsou co se týče obsahu identické. Zátěž je rozdělena na základě zatížení jednotlivých serverů, přičemž může být orientována dle výkonu serveru, či jeho již stávajícího vytížení. Zatížení není rovnoměrně rozvržené, pokud se nejedná o náhodný, nebo sériový (postupný výběr serveru - nejdříve první, pak druhý, pak třetí, pak opět první server). Počítáme s tím, že požadavky posílané na server jsou si v náročnosti výpočtu rovné. Příkladem reverse proxy bude například Steam.



### **Remote proxy**

Pokud bychom chtěli komunikovat s objektem, ale na jiném virtuálním stroji, případně na úplně jiném počítači, můžeme využít právě **Remote Proxy**. Jedná se o zástupce vzdáleného objektu, ve kterém je skryta veškerá komunikace.

**Vzdálený zástupce** zastupuje objekt umístěný někde jinde. Jeho úkolem je zprostředkovat komunikaci s tímto objektem tak, aby přes Remote Proxy mohli uživatelé komunikovat se vzdáleným (originálním) objektem aniž by se museli starat o serverovou komunikaci (tu musí zařídit Proxy objekt). Zároveň však nesmí zakrývat komunikaci. Tj. například při selhání spojení musí vyhodit odpovídající výjimku se kterou musí uživatel počítat a případně ji vhodně zpracovat. Uživatel tak může se vzdáleným zástupcem komunikovat skoro jako s normálním objektem.

Samotná implementace **Vzdáleného zástupce** se dá realizovat různě. Je poněkud náročnější, protože musíme zajistit už zmíněnou komunikaci mezi dvěma objekty. Vždy však musíme při implementaci počítat s možností selhání komunikace se vzdáleným objektem.

### **Virtual proxy**

Častý případ využití Proxy je **virtuální zástupce**. V naší aplikaci se může velmi často stát, že máme nějaký objekt, jehož vytváření nebo případná správa by byla náročná příp. by trvalo dlouho a bylo by neefektivní načítat ho celý, když pak použijeme jen jeho část. Virtuální zástupce se snaží vytvořit instanci spravované třídy až když je to potřeba.

Virtual Proxy se nám i v tomto případě stará o jiný objekt (zastupuje ho). Vytvoření objektu ale přenechává až na poslední chvíli (pokud ho vůbec potřebujeme vytvořit) a chování objektu jen předstírá. Tj. navenek může mít stejné rozhraní, ale funkčnost jen deleguje na originální objekt, případně přidává nějakou další funkčnost a k zastoupenému objektu přistupuje či ho vytvoří jen tehdy, je-li to opravdu nutné. Typicky se může jednat o nějaký obrázek. Jeho vykreslení bude potřeba třeba jen jednou nebo vůbec. Naše aplikace se ale zeptá na jeho rozměry nebo na další metadata (třeba i v případě, že ho později vůbec nevykreslí). Bylo by neefektivní načítat obrázek do paměti, když ho poté ani nevyužijeme.

### **Protection proxy**

V návrhu aplikace se můžeme někdy setkat s případem, kdy by využití celého API (veřejného rozhraní) objektu mohlo mít fatální dopad. Pro příklad to může být třeba situace, kdy každý objekt má k našemu klíčovému objektu jiná přístupová práva nebo prostě jen potřebujeme zjednodušit rozhraní objektu, aby ho někdo neznalý našeho systému nepoužil špatně, což by mohlo mít v některých případech dokonce fatální následky.

V obou výše zmíněných případech si definujeme novou Proxy třídu, která si spravuje instanci objektu, ale navenek nabízí jen podmnožinu metod objektu, který zastupuje. Můžeme tak ovlivnit, ke kterým metodám cílového objektu bude mít uživatel přístup a ke kterým nikoli. V některých případech, kdy nám jde jen o omezení rozhraní, ho lze nahradit jednoduchým rozhraním, což je efektivnější, ale zároveň méně bezpečné. Uživatel má v tomto případě stále přístup k celému objektu pouze pod zjednodušeným rozhraním. Může si tak celkem jednoduše zjistit, o jaký objekt jde.

## 

## **Flyweight**

Návrhový vzor, který šetří paměť při úkolech, pro které potřebujeme vytvořit velký počet instancí. Základní myšlenkou u tohoto návrhového vzoru je rozdělení třídy na dvě části.

**1)Vnitřní stav objektu:**

- je společná pro všechny třídy

-Tento stav si o sobě instance pamatuje sama.

(Pro příklad si představme, že programujeme hru a máme nepřátele ve světě. Jednotliví nepřátelé jsou naprosto totožní = vypadají stejně, mají stejnou sílu, inteligenci, zbraně a podobně.)

-Není důvod, abychom si tyto informace pamatovali pro každý objekt zvlášť a zabírali tím paměť.

**2)Vnější stav objektu:**

- je pro každý objekt jiný.

(Př/ Souřadnice nepřítele. Tyto informace dostane volaná metoda přímo ve svých parametrech. Objekt si tedy tuto informaci nepamatuje a musí někde získat.)

*“Pokud máme například mapu skládající se z různých druhů políček, lze každý druh reprezentovat jednou instancí obsahující společné vlastnosti celého druhu.”*

