Pattern te modelimit

Design Patterns – The Gang of Four sepse jane percaktuar ne librin e kater autoreve *Design Patterns Elements of Reusable Object Oriented Software*, E.Gamma, R. Helm, R. Johnson, J.Vlissides.

1. Modelimi me *pattern* (struktura)

Nje pattern pershkruan nje problem te perseritur dhe zgjidhjen baze te tij, ne menyre te tille qe zgjidhja te mund te riperdoret vazhdimisht, pa pasur nevoje qe ajo te rimendohet dhe ribehet. Per software, zgjidhja realizohet nepermjet objekteve dhe nderfaqeve ndermjet tyre.
Nje patern perbehet nga kater elemente:

- **emri** – identifikon pattern dhe eshte pershkrues i problemit te modelimit qe trajton, zgjidhjes dhe pasojave. Emertimi i pattern sherben si mjet komunikimi ne nje gjuhe te perbashket per komunitetin e inxhiniereve te software.
- **problemi** – pershkruan situatat kur duhet perdorur pattern. Pershkruan problemin dhe kontekstin. Ndonjehere problemi permban edhe listen e kushteve qe duhen plotesuar perpara se te kete kuptim aplikimi i pattern.
- **zgjidhja** – pershkruan elementet qe perbejne modelimin, lidhjet, pergjegjesite e secilit element dhe njesi me te cilat ato bashkepunojne. Zgjidhja nuk pershkruan nje model apo implementim konkret, sepse ajo eshte si nje template qe mund te aplikohet ne situata te shumta. Pattern ofron nje pershkrim abstrakt te nje problemi modelimi dhe sesi mund te zgjidhet ai nepermjet kombinimit te ca elementeve.
- **pasojat** – jane rezultate apo zgjedhje qe duhet te behen kur aplikohet pattern. Eshte e rendesishme te dihen dhe vleresohen pasojat, sepse shpesh ato percakojtne koston dhe perfitimin e aplikimit te paternit. Kur flitet per pasoje, pergjithesisht pasojat vleresohen ne lidhje me hapesiren dhe kohen e ekzekutimit, sepse keto jane ato qe perbejne koston e ekzekutimit te nje software. Mqs riperdorimi eshte nje faktor shume i rendesishem ne modelimin me objekte, atehere pasojat e zgjedhjes se nje paterni shpesh vleresohen edhe me ndikimin e saj tek fleksibiliteti, zgjerueshmeria dhe zhvendosmeria e sistemit.

Pattern-at e modelimit vendosin nje lidhje mes konteksit ne te cilin ndodh nje problem i caktuar, problemit dhe zgjidhjes se tij. Shumica e problemeve kane shume zgjidhje, por nje zgjidhje eshte efektive vetem nese ajo zbatohet ne kontekstin e duhur, pra zgjidhja duhet te jete ne perputhje me kombinimin e elementeve qe percaktojne kontekstin.
Nje pattern modelimi eshte efektiv kur:

1. *zgjidh nje problem.* Pattern pershkruajne zgjidhje konkrete, jo parime abstrakte apo strategji te zgjidhjes.
2. *zgjidhja e problemit mbeshtetet nga prova.* Patterni nuk pershkruan nje zgjidhje e cila vertetohet teorikisht per korrektesi, por zgjidhja duhet te jete provuar si korrekte ne raste konkrete.
3. *zgjidhja e problemit nuk eshte e kuptueshme menjehere.* Kjo do te thote qe patterni nuk pershkruan zgjidhje nisur vetem nga parimet teorike (zgjidhja ne kete rast do te ishte menjehere e kuptueshme), por perfshin nje zgjidhje indirekte te problemit.
4. *pershkruan marredhenie.* Patter nuk pershkruan/ percakton vetem modulet e nje sistemi, por pershkruan me thelle strukturat dhe mekanizmat perberese te sistemit.

5. *patterni thekson faktorin human.* Sistemet i sherbejne njerezeve, me qellim permiresimin e jeteses. Pattern-at me te mire theksojne vecanerisht estetiken dhe perdorueshmerine e sistemeve software-ike.

Pattern-at e modelimit pershkruajne njohuri te fituara nga njerezit ne kohe, ne menyre te tille qe keto njohuri te mund te riperdoren gjithmone pa riberje te gjerave. Pra, pattern-at e modelimit ndalojne rishpikjen e rrotes gjate modelimit te sistemeve.
Pattern te modelimit nuk kane te bejne me modelimin e listave te lidhura apo tabelave hash qe mund te permblidhen ne klasa dhe te riperdoren dhe nuk jane specifike ne varesi te domainit ku kerkohet zgjidhja. Pattern te modelimit jane pershkrime te menyres se komunikimit te objekteve dhe klasave te zgjedhura per te zgjidhur nje problem te pergjithshem modelimi ne nje kontekst te vecante.
Modelimi me pattern krijon nje software te ri duke gjetur/ perdorur zgjidhje te provuara me pare per probleme te mire percaktuara. Problemi dhe zgjidhja e tij pershkruhen ne nje pattern modelimi, i cili eshte shtuar ne katalogun qe permbledh pattern-at e krijuar nga inxhinieri qe eshte perballur me pare me problemin dhe qe ka dhene zgjidhjen (e provuar). Pra, cdo pattern modelimi ofron nje zgjidhje te provuar per problemin, ose per pjese te problemit.
Perdorimi i patterns lejon riperdorimin e njohurive te fituara gjate zgjidhjeve te problemeve me te cilat perballet komuniteti i inxhiniereve te software.

Pattern te modelimit emertojne, abstraktojne dhe identifikojne aspektet kryesore te nje strukture te modelimit qe lehteson krijimin e objekteve te riperdorshme. Ato identifikojne klasat dhe instancat pjesemarrese, rolet dhe bashkepunetoret dhe shperndarjen e pergjegjesive. Cdo pattern modelimi fokusohet ne nje problem te vecante te modelimit me objekte. Pershkruan kur mund te aplikohet, nese mund te aplikohet pavaresisht kushtezimeve te tjera te modelimit, dhe pasojat qe sjell nese perdoret.
Qellimet kryesore te pattern jane:

- mbeshtetja e riperdorimit te modeleve te suksesshme
- lehtesimi i evolimit te software : shtohen lehte vecori, pa prishur ekzistueset
- ulja e varesise ndermjet elementeve perberese te sistemit
- pra, mbeshtjetja e modelimit me objektiv akomodimin e ndryshimit.

2. Klasifikimi i pattern te modelimit

2.1 Pattern te krijimit te objekteve

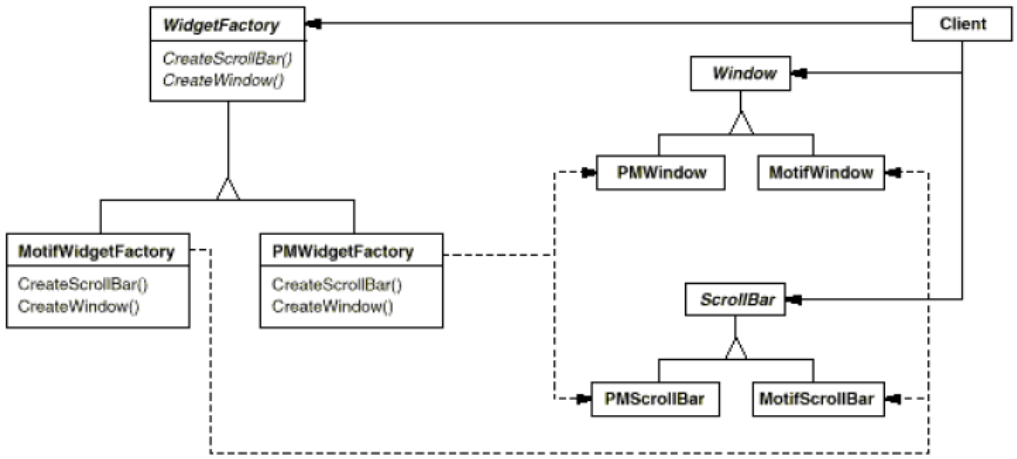
Abstragojne menyren e krijimit, perberjes dhe inicializimit te objekteve. Keto pattern-a dine se cilat jane klasat qe perdor konkretisht sistemi, por edhe sesi krijohen dhe komunikojne instancat e klasave. Pattern-at e krijimit vendosin kufizime per tipin dhe numrin e objekteve qe mund te krijohen ne sistem. Shembuj te pattern-ave te krijimit jane:

Abstract Factory

Pattern-i perdoret kur eshte nevoja te krijohet nje familje objektesh.
Ky pattern ofron nje nderfaqe per krijimin e nje familje objektesh qe kane lidhje me njeri tjetrin ose qe kane varesi me njeri tjetrin, por pa specifikuar klasat konkrete.

Mund te ndodhe qe nderfaqja grafike e nje sistemi te duhet te ofroje shume standarte te paraqitjes, psh. butonat, ashensoret, dritaret (te cilave u referohemi Widgets) duhet te kene pamje te ndryshme. Ne menyre qe aplikimi te jete portabel persa i perket pamjeve te ndryshme qe duhet te ofroje, ai nuk duhet te kete nderfaqen grafike te paracaktuar (hard coded), por duhet te kete mundesi ta gjeneroje ate dinamikisht. N.q.s. do te kishte instanca te klasave specifike per cdo menyre paraqitje te aplikimit, atehere ndryshimet do te ishin shume te veshtira.

Per te zgjidhur problemin me lart, mund te percaktohet nje klase *WidgetFactory* e cila deklaron nderfaqen baze te krijimit te cdo elementi te nderfaqes grafike. Me pas deklarohen klasa abstrakte per cdo element te nderfaqes dhe nenklasa konkrete qe implementojne detajet specifike te seciles paraqitje. Nderfaqja e *WidgetFactory* percakton nje operacion qe kthen nje objekt te ri per cdo klase abstrakte qe i takon elementeve te ndryshem te nderfaqes grafike. Klientet therrasin keto objekte per te perftuar instanca te objekteve te elementeve te nderfaqes, por ata nuk kane dijeni per objektet konkrete qe po perdorin. Prandaj klienet jane te pavarur nga paraqitja konkrete qe kerkohet.



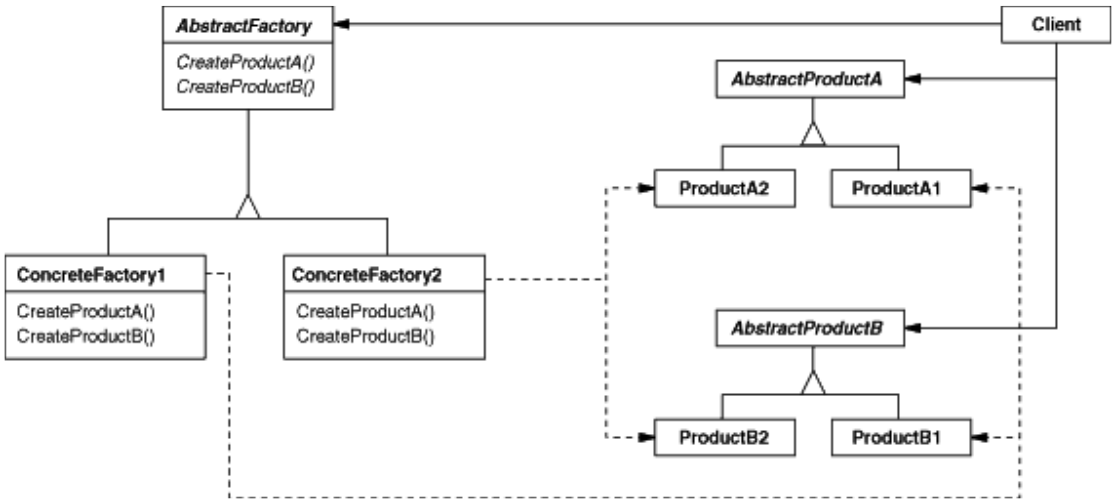
Per cdo menyre paraqitjeje grafike ka nga nje nenklase konkrete te *WidgetFactory.* Cdo nenklase implementon operacionet e krijimit te elementeve grafike qe kerkohen konkretisht. Psh. operacioni *CreateScrollBar()* tek klasa *MotifWidgetFactory* krijon dhe kthen nje ashensor *Motif (*Motif eshte nje toolkit per te zhvilluar programme me GUI per Windows ose Unix), ndersa operacioni korrespondues tek klasa *PMWidgetFactory* kthen nje ashensor per *Presentation Manager* (nje tjeter lloj kontrollesh te nderfaqes grafike). Klientet krijojne elementet konkrete te nderfaqes grafike vetem nepermjet nderfaqes *WidgetFactory* dhe nuk kane njohuri per klasat qe implementojne elementet konkrete (sipas rastit), me pamje te ndryshme. Pra klientet duhet vetem te regjistrohen tek nje nderfaqe e percaktuar nga nje klase abstrakte dhe jo tek nje klase konkrete.

Pattern *Abstract Factory* mund te perdoret ne rastet kur :

- Sistemi duhet te jete i pavarur nga menyra sesi krijohen, perbehen dhe perfaqesohen produktet e tij
- Sistemi duhet te konfigurohet si nje nga produktet e nje familjeje produktesh

- Nje familje e produktesh qe kane lidhje me njeri tjetrin konceptohen qe te perdoren sebashku dhe ky kufizim duhet te imponohet
- Duhet te krijohet nje librari me klasa dhe duhet te ekspozohen vetem nderfaqet e klasave, jo implementimet.

Struktura e *Abstract Factory* eshte si me poshte :



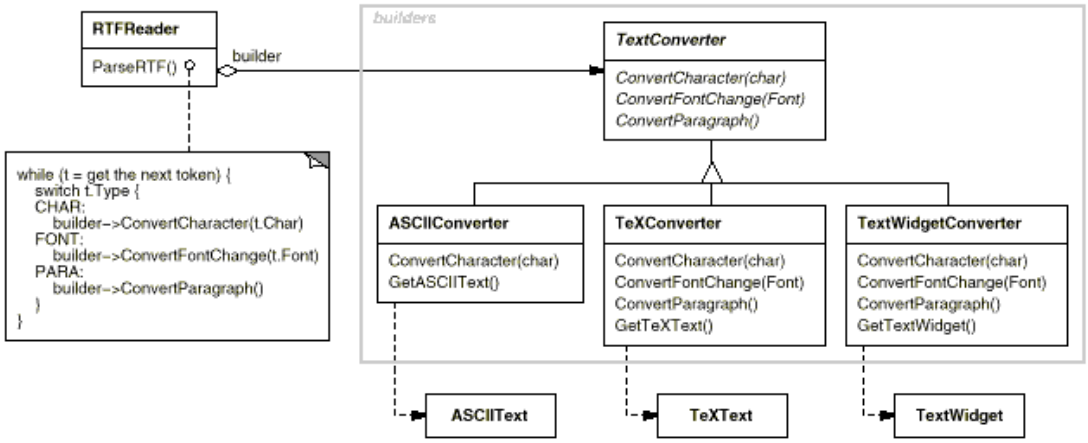
Pjesemarresit jane :

- **AbstractFactory** (WidgetFactory ne shembullin e sistemit me paraqitje te ndryshme grafike). Krijon nje nderfaqe per operacionet qe krijojne objekte te produkteve abstrakte.
- **ConcreteFactory** (MotifWidgetFactory, PMWidgetFactory). Implementon operacionet per te krijuar objekte te produkteve konkrete.
- **AbstractProduct** (Window, ScrollBar). Percakton nderfaqen e tipit te nje objekti te produktit.
- **ConcreteProduct** (MotifWindow, MotifScrollbar). Percakton nje objekt te produktit per t'u krijuar nga objekti perkates konkret *factory* dhe implementon nderfaqen *AbstractProduct*.
- **Client**. Perdor vetem nderfaqen e deklaruar nga klasat *AbstractFactory* dhe *AbstractProdukt*.

Kur perdoret ky *pattern*, normalisht ne run-time krijohet nje instance e vetme e klases *ConcreteFactory*. Kjo instance krijon objektet e produkteve qe kane nje implementim te caktuar. Per te krijuar objekte te ndryshme produktesh, klientet duhet te perdorin nje tjeter *ConcreteFactory*.

Builder

Pattern-i *Builder* perdoret kur eshte nevoja qe nje objekt te krijohet ne disa faza/ hapa.
Ai ndan krijimin e nje objekti kompleks nga paraqitja e tij, ne menyre qe i njejti proces ndertimi i objekteve te krijoje paraqitje te ndryshme.
Shembull do te ishte lexuesi i nje dokumenti ne formatin RTF qe do t'i duhej te konvertonte dokumentin ne formate te ndryshme te tjera text. Numri i formateve ne te cilat mund te konvertohet dokumenti eshte i papercaktuar, keshtu qe duhet gjetur nje menyre fleksible per ta realizuar kete, pa ndryshuar lexuesin e dokumentit.



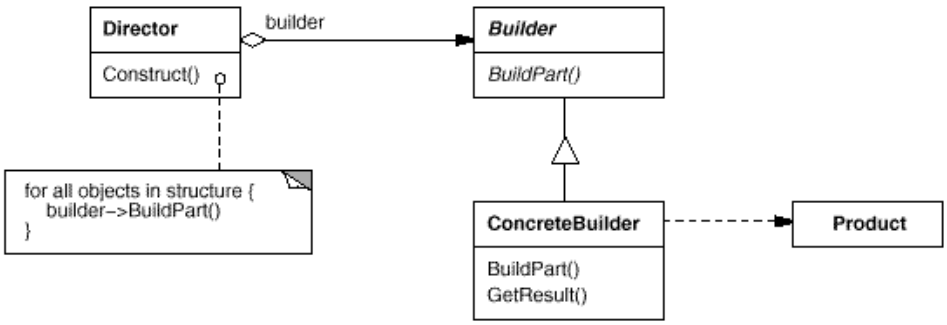
Nje zgjidhje do te ishte konfigurimi i klases *RTFReader* me nje objekt *TextConverteri* qe konverton RTF ne forma te ndryshme teksti. Objekti *RTFReader* parson dokumentin RTF dhe me pas perdor objektin *TextConverter* per te bere konvertimin. Sa here qe *RTFReader* njeh nje *token* RTF ai i dergon nje kerkese *TextConverter* per te konvertuar *token*. Objektet e fundit jane pergjegjese per te konvertuar te dhenat dhe per ta paraqitur *token* ne nje format te caktuar.
Nenklasat e *TextConverter* specializohen per te kryer konvertime ne formate te ndryshme. Psh. klasa *ASCIIConverter* i injoron kerkesat per te konvertuar ne formate te tjera tekst pervecse atij *plain*. Nje objekt *TeXText* implementon te gjitha veprimet qe duhen per te prodhuar nje paraqitje ne formatin TeX, me te gjitha elementet stilistike te nevojshem.
Çdo klase konvertuese merr mekanizmin per krijimin dhe asemblimin e nje objekti kompleks dhe e vendos mbrapa nje nderfaqeje abstrakte. Konvertuesi eshte i ndryshem nga lexuesi qe eshte pergjegjes per parsing te nje dokumenti RTF.

Pattern *Builder* i konsideron te gjitha lidhjet dhe marredheniet e siperpermendura per objektet e perfshira. Cdo klase konvertuese me lart quhet *builder* dhe objekti lexues (reader) quhet *drejtues* (director). Patterni i aplikuar ne situaten e konvertimit ndan algoritmin e interpretimit te nje formati tekst (d.m.th. parseri i dokumentave RTF) nga menyra e krijimit dhe paraqitjes te nje formati te konvertuar. Kjo lejon riperdorimin e algoritmit te parsimit te *RTFReader* per krijimin paraqitjeve te ndryshme tekstuale te dokumentave RTF – mjafton qe RTFReader te konfigurohet te komunikoje me nenklasa te ndryshme te *TextConverter.*

Pattern *Builder* perdoret ne rastet kur :

1. Algoritmi i krijimit te nje objekti kompleks duhet te jete i pavarur nga pjeset qe implementojne perberjen e objektit.
2. Procesi i ndertimit te paraqitjes/menyres se outputit te objektit duhet te ofroje mundesi per paraqitje te ndryshme te tij

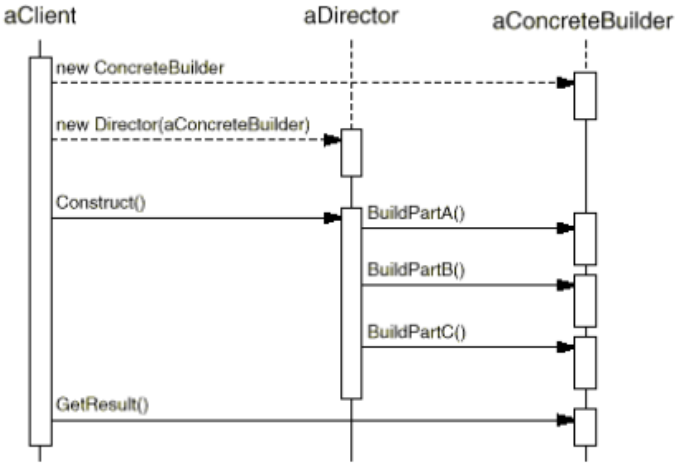
Struktura e ketij pattern eshte :



Pjesemarresit ne kete pattern jane:

- **Builder** (psh TextConverter). Specifikon nje nderfaqe abstrakte per krijimin e pjeseve te nje objekti Product.
- **ConcreteBuilder** (psh ASCIIConverter, TeXConverter, TextWidgetConverter). Nderton dhe mbledh pjeset e produktit duke implementuar nderfaqen *Builder*, percakton menyren e paraqitjes te cilen edhe e realizon dhe ofron nje nderfaqe per te terhequr objektin product (psh GetASCIIText, GetTextWidget).
- **Director** (RTFReader). Krijon nje objekt duke perdorur nderfaqen *Builder*.
- **Product** (ASCIIText, TeXText, TextWidget). Perfaqeson objektin kompleks qe do te ndertohet. Klasa *ConcreteBuilder* nderton paraqitjen e brendshme te produktit dhe percakton procesin nepermjet te cilit do te asemblohen pjeset. Per me teper *Product* perfshin dhe klasat qe percaktojne pjeset perberese, duke perfshire nderfaqet per asemblimin e pjeseve ne nje produkt te plote final.

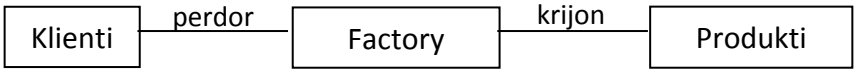
Ne kete pattern, klienti krijon objektin drejtues (director) dhe e konfiguron me objektin e duhur *Builder*. Drejtuesi njofton *builder* sa here qe duhet te ndertohen pjese te produktit. Builder trajton kerkesen dhe i shton pjeset e duhura produktit. Klienti terheq produktin nga *builder*. Bashkeverpimi mes objekteve te perfshire ne pattern paraqitet nepermjet diagrames se sekuences se meposhtme:



Factory Method (Virtual Constructor)

Nje aspekt i rendesishem i modelimit te sistemit eshte menyra sesi krijohen objektet. Shpesh vemendja perqendrohet tek modeli i objekteve dhe nderveprimi mes instancave dhe, mund te ndodhe qe te anashkalohet menyra sesi do te krijohen objektet gjate ekzekutimit te sistemit.

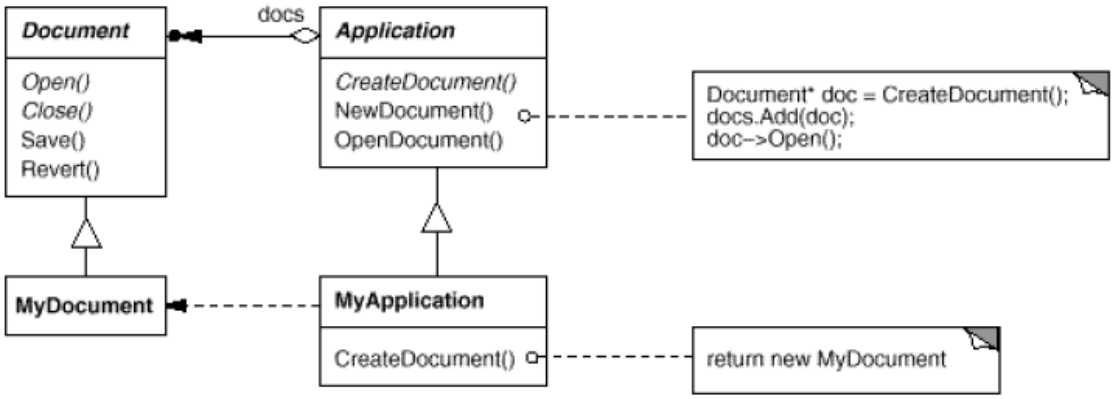
Pra, nuk eshte vetem e rendesisshme te identifikohet se cfare do te beje nje objekt, por edhe menyra sesi ai krijohet.
Gjuhet e programimit me objekte ofrojne funksionalitete te inicializimit te objekteve si psh. konstruktoret dhe prirja eshte qe te perdoren direkt keto funsionalitete, pa u kujdesur per pasojat. Mbiperdorimi i ketyre funksionaliteteve e ben sistemin konsiderueshem jo flexibel sepse krijohet lidhje e ngushte mes krijuesit te klases dhe klases se krijuar. Kjo lidhje e tepruar con ne ciftim te larte dhe shume te veshtire per t'u menaxhuar n.q.s. sistemi do te perballet me ndryshime.
Nevoja per te zgjidhur problemin e çiftimit qe vjen menyra e inicializimit te objekteve haset tek pothuajse te gjithe sistemet dhe menyrat e zgjidhjes jane pothuajse te njejtat. Nje nga menyrat e zgjidhjes eshte patterni *Factory* i cili permban objekte te specializuara vetem per krijimin e objekteve te tjere, ne menyre te ngjashme me nje fabrike ne boten reale.
Ne formen e vet standarte, ky pattern percakton tre pjesemarres kryesore: klientin, nje *factory* dhe nje produkt. Klienti eshte nje objekt qe kerkon/ka nevoje per nje instance te nje objekti tjeter (produktit). Klienti nuk e krijon vete instancen e produktit, por ia delegon kete pergjegjesi objektit *factory*. Objekti *factory* njoftohet qe duhet te krijoje nje instance te objektit produkt dhe pas ketij njoftimi, krijon instancen e objektit produkt dhe ia kalon klientit. Lidhja logjike mes objekteve eshte:



Objekti *Factory* abstrakton plotesisht krijimin dhe inicializimin e produktit nga klienti. Pra klienti nuk ka dijeni sesi incializohet dhe krijohet produkti. Klienti eshte indiferent ndaj ndryshimeve qe mund te ndodhin ne te ardhmen tek produkti.
N.q.s. nderfaqja e produktit konsiderohet e pandryshueshme, atehere *Factory* mund te krijoje objekte te tipeve te ndryshme Produkt (pra mund te kete disa klasa qe percaktojne produkte).
Ky pattern percakton nje nderfaqe per krijimin e nje objekti, por e le ne dore te nenklasave te vendosin se cilen klase te inicializojne.

Framework-et perdorin klasa abstrakte per te percaktuar dhe mirembajtur lidhjet ndermjet objekteve. Shpesh *framework* eshte pergjegjese dhe per krijimin e ketyre objekteve.
Le te konsiderojme nje *framework* per aplikime qe mund t'i shfaqe perdoruesve shume dokumente. Dy abstraktimet kryesore ne kete *framework* jane klasat *Application* dhe *Document*. Te dyja klasat jane abstrakte dhe klientet duhet te krijojne nenklasa te tyre per te realizuar implementimin specifik. Psh. per te krijuar nje aplikim qe mundeson vizatime, do te percaktoheshin klasat *VizatoApplication* dhe *VizatoDocument*. Klasa *Application* do te jete pergjegjese per menaxhimin e dokumentave dhe do t'i krijoje ato sipas kerkeses – kur perdoruesi zgjedh komandat *E Re* ose *Hap* nga nje menu e caktuar.

M.q.s. nenklasa konkrete *Document* qe duhet te inicializohet eshte specifike sipas aplikimit, atehere klasa *Application* nuk mund te parashikoje se cilat nenklasa te *Document* te inicializoje. Klasa *Application* mund vetem te kuptoje qe duhet krijuar nje *Document* i ri, por jo cilin tip. Keshtu qe *framework* duhet te inicializoje klasat, por ai njeh vetem klasat abstrakte te cilat nuk mund te inicializohen.
Pattern *Factory Method* ofron nje zgjidhje per problemin e siper permendur. Ai enkapsulon informacionet se cilat nenklasa *Document* duhet te inicializohen dhe e i kalon keto informacione tek *framework.*



Nenklasat e *Application* percaktojne nje metode abstrakte *CreateDocument* i cili kthen nenklasesn e duhur *Document*. Pasi inicializohet nje nenklase *Application* ajo mund te inicializoje dokumentat specifike pa pasur nevoje te dije klasat e tyre. Metoda *CreateDocument* quhet metode prodhuese (*factory)* sepse eshte pergjegjese per krijimin e objektit.

Pattern *Factory Method* perdoret ne rastet kur:

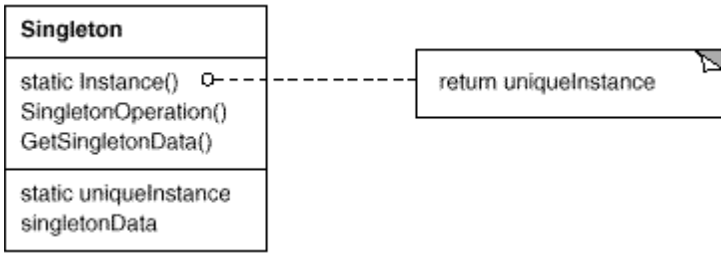
- nje klase nuk mund te parashikoje objektet qe duhet te krijoje
- nje klase ka nevoje qe te jene nenklasat e saj ato qe specifikojne objektet qe ajo do te krijoje

Objektet pjesmarres ne kete pattern jane :

- **Product** (Document). Percakton nderfaqen e objekteve qe duhet te krijoje metoda *factory*.
- **ConcreteProduct** (MyDocument). Implementon nderfaqen *Product.*
- **Creator** (Application). Deklaron metoden *factory* e cila kthen nje objekt te tipit *Product*. Objekti *Creator* mund te percaktoje dhe nje implementim default per metoten qe te ktheje nje objekt default *ConcreteProduct*. Nga ana tjeter, objekti *Creator* mund te krijoje nje objekt *Product*.
- **ConcreteCreator** (MyApplication). Mbishkruan metoden *factory* per te kthyer nje instance te te *ConcreteProduct.*

Singleton

Ky pattern siguron qe nje klase ka vetem nje instance dhe ve ne dispozicion nje pike globale aksesi per te.
Ndonjehere eshte e rendesishme qe per nje klase te ekzistoje vetem nje instance ne sistem. Psh., edhe pse ne sistem mund te kete te lidhur shume printera, duhet te ekzistoje vetem nje spooler. Duhet te kete vetem nje sistem skedaresh dhe nje manager te dritareve. Nese per keto objekte do te kishim variabla globale, ato do te ishin te aksesueshem ne te gjithe sistemin, por nuk ka asnje mekanizem qe pengon krijimin e me shume sesa nje klase.
Nje zgjidhje me e mire eshte qe klasa te jete vete pergjegjese per t'u siguruar qe ne sistem nuk do te kete me shume sesa nje instance te saj. Kjo realizohet me pattern *Singleton*. Klasa percepton kerkesat per objekte te shumefishte, ndalon krijimin e tyre dhe kthen nje metode aksesi tek e vetmja instance e saj.

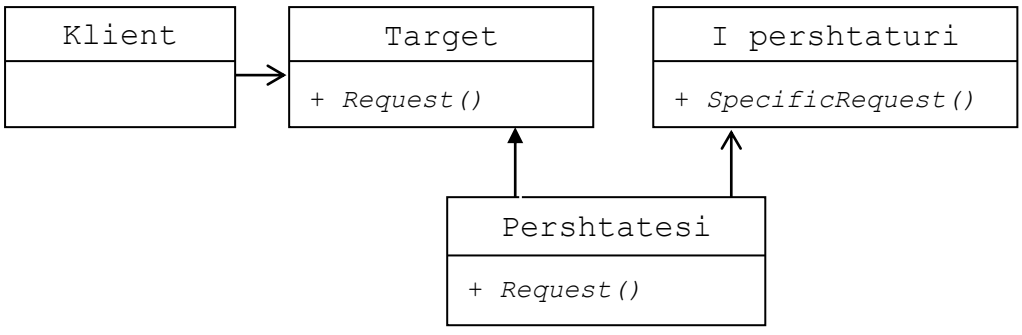


2.2 Pattern strukturore

Pattern strukturore trajtojne perberjen e objekteve dhe klasave, menyren sesi ato kombinohen per te formuar struktura me te medha. Keto paterna perdorin trashegimine per ndertimin e nderfaqeve apo per implementimet. Psh. pattern qe perdorin trashegimine e shumefishte krijojne klasa qe kombinojne karakterisitikat e disa klasave prej nga trashegojne. Keto lloje pattern jane te perdorshme ne rastet kur duhet te krijohen librari te perbera nga klasa qe jane zhvilluar pavaresisht njera tjetres. Shembull tjeter i pattern strukturore eshte *Adapter*. Nje *Adapter* ben pershtatje te nderfaqeve duke ofruar keshtu nje abstragim uniform te disa nderfaqeve.
Pattern strukturore nuk ofrojne kompozime apo implementime, por pershkruajne menyra sesi mund te kompozohen objektet per te realizuar funksionalitete te reja.

Adapter (Pershtatesi)

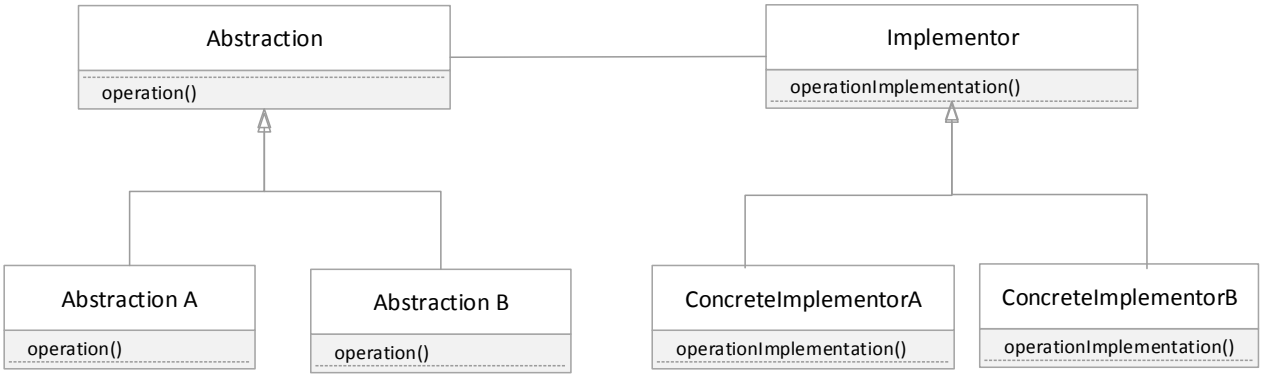
Qellimi i ketij pattern eshte te konvertoje nderfaqen e nje klase ne nderfaqen qe pret nje klient. Pershtatesi ben qe klasa te cilat kane nderfaqe te paperputhshme, e rrjedhimisht nuk do te mund te komunikojne ashtu sic jane, te nderverprojne sebashku. Pergjithesisht ky pattern nuk perdoret kur ndertohet nje sistem i ri, por kur duhet te akomodohen ndryshime ne kerkesa. Ky pattern pershkruhet me objektet me poshte dhe nderverprimin mes tyre:



Supozoni qe eshte perdorur nje librari ku funksioni Shto merr si parametra dy numra te plote dhe kthen shumen e tyre. Kur merret nje version me i ri i librarise kuptohet qe funksioni Shto ka ndryshuar dhe merr si argumenta dy numra dhjetore. Nje menyre per te pershtatur kete ndryshim do te ishte ndryshimi i te gjithe kodit ku eshte perdorur ky funksion, ndersa menyra tjeter eshte perdorimi i patternit Pershtates.

Bridge

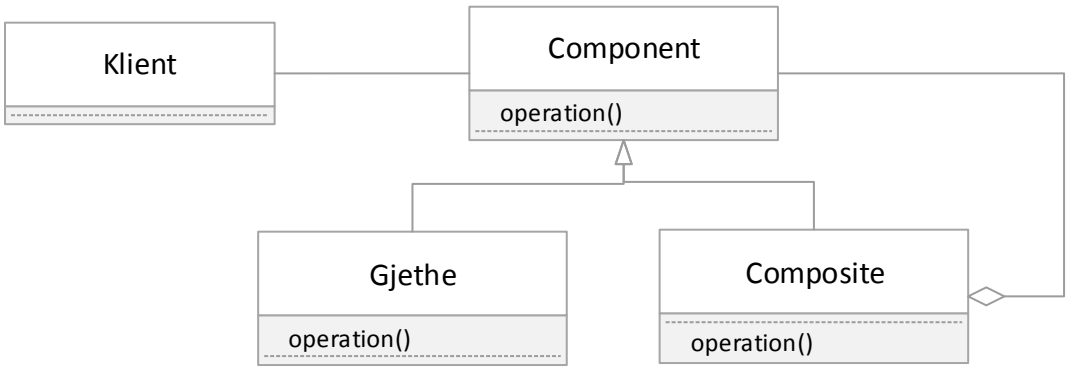
Ky pattern ka si qellim te ndaje abstragimin nga implementimi i tij, ne menyre qe ndryshimet e tyre te jene te pavarura.
Objektet ne nje model qe perdor pattern-in *Bridge* do te ishin si me poshte:



Sic tregohet ne figure, abstragimi dhe implementimi jane te ndare nga njeri tjetri, por lidhen nepermjet nje ure.

Composite

Patterni *Composite* perdoret kur duhet te trajtohet nje grup objektesh, ku te gjithe kane te njejten nderfaqe. Atehere, objektet trajtohen si nje peme, ku objektet e vecanta (gjethe) si dhe objektet e perbera (pjesa tjeter e pemes) sillen njesoj. Objektet e perbera i delegojne kerkesat tek objektet gjethe.
Objektet ne nje pattern te tille, do te ishin organizuar si ne figure:



Ky pattern do te ishte i pershtatshem per te punuar ne nje program me objektet qe paraqesin sistemin e skedareve, I cili eshte ne trajte peme.

Pattern-a te tjere

Pattern te tjere strukturore jane:

Aggregate -
Container -

Proxy -
Pipes and filters -

2.3 Pattern te sjelljes

Pattern-at e sjelljes karakterizojne menyrat ne te cilat klasat apo objektet nderveprojne dhe shperndajne pergjegjesite. Ato trajtojne algoritmat dhe rrjedhen komplekse te kontrollit e cila eshte e veshtire per t'u menaxhuar gjate kohes se ekzekutimit te programit. Ato e zhvendosin fokusin nga rrjedha e kontrollit dhe lejojne perqendrimin tek menyra sesi nderveprojne objektet. Pattern te sjelljes perdorin trashegimine per te shperndare funksionalitetet ndermjet klasave. Ato perdorin edhe kompozimin e objekteve per te realizuar detyra/pune e nuk mund te behen nga nje objekt i vetem.

Patterna te sjelljes jane:
Chain of responsibility -
Command -
Event Listener -
Interpreter -
Iterator -
Mediator -
Visitor -
Single-serving visitor pattern -
Hierarchical visitor patter -