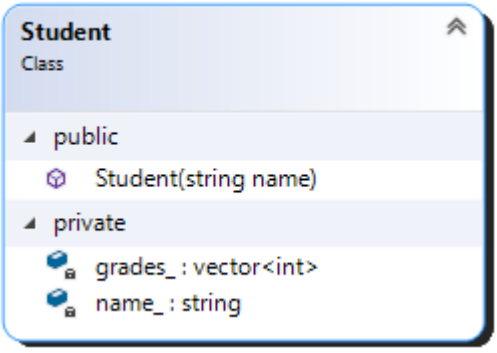
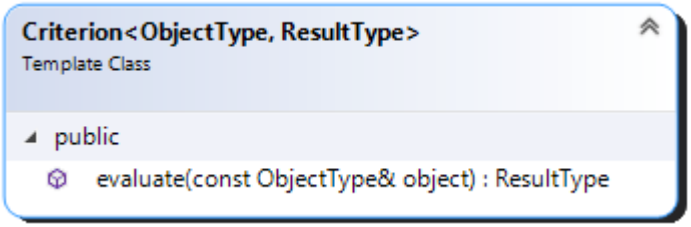
Implementačná príručka k semestrálnej práci

Na jednoduchom príklade (mimo témy tejto semestrálnej práce) budú vysvetlené základy univerzálneho objektového návrhu. Pracovať budeme s triedou `Student` podľa nasledujúceho UML diagramu:



Ako si navrhnúť kritériá?

Kritérium $K_{\Omega(\Pi)}^{NázovKritéria}(\omega) = \kappa: \tau$ je objekt, ktorý získava hodnotu vlastnosti κ (kappa) z iných objektov ω (omega). Môžeme teda navrhnúť jednu abstraktnú triedu reprezentujúcu kritérium s jednou abstraktnou metódou. Nakoľko je podľa analýzy možné ohodnotiť akýkoľvek objekt z množiny Ω (omega), musíme umožniť tento typ špecifikovať až neskôr. To vieme docieliť pomocou šablónového parametra – v našom prípade `ObjectType`. Podobné platí pre návratovú hodnotu τ (tau), ktorú tiež nemôžeme špecifikovať vopred, lebo sa môže v potomkoch rôzne líšiť – špecifikujeme preto druhý šablónový parameter `ResultType`.



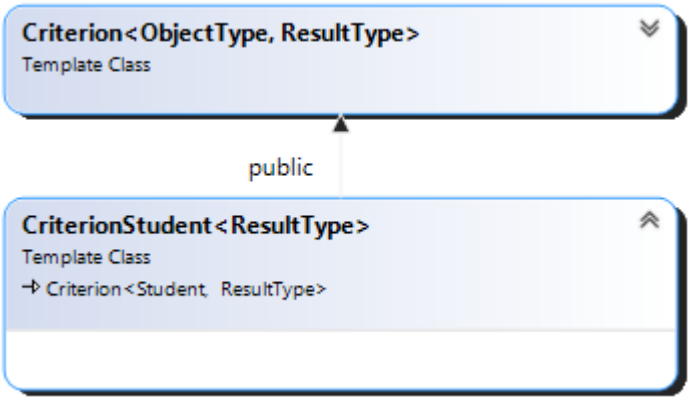
Požadujme nasledujúce kritériá:

I. $\boldsymbol{K^{Meno}}$:
$\Omega \subseteq \{Študenti\}$
$\Pi = \emptyset$
$\kappa = \omega.meno;\ \tau = string$

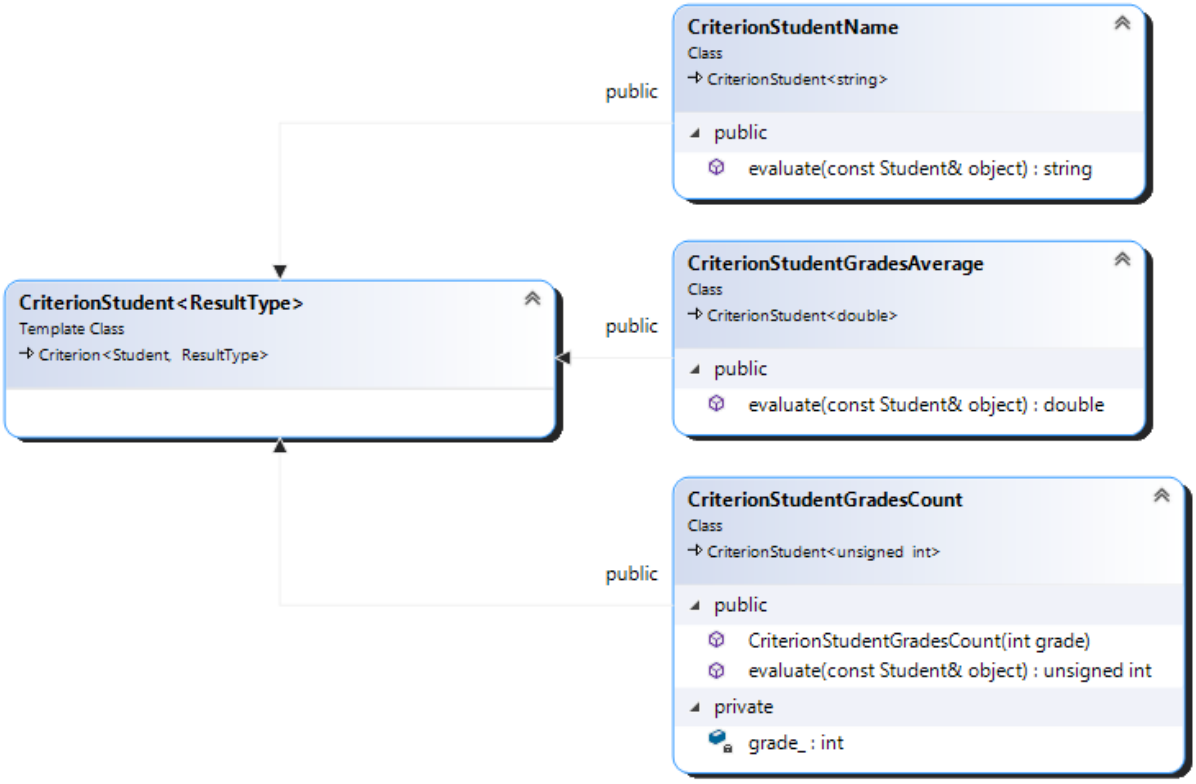
II. $\boldsymbol{K^{PriemerZnamok}}$:
$\Omega \subseteq \{Študenti\};$
$\Pi = \emptyset$
$\kappa = \omega.priemerZnámokŠtudenta; \tau = double$

III. $\boldsymbol{K^{PocetZnamok}}$:
$\Omega \subseteq \{Študenti\}$
$\Pi = (známka \in \{1,2,3,4,5\})$
$\kappa = \omega.početZnámok(známka); \tau = integer \in \langle 0|\infty \rangle$

Pre kritériá potrebujeme vytvoriť samostatné triedy. Vidíme, že množina Ω je stále rovnakého typu, môžeme preto pridať ešte jedného spoločného predka, ktorého jedinou úlohou bude definovať konkrétny typ šablónového parametra `ObjectType`, v našom prípade ako `Student`. Tento objekt nebude definovať žiadne ďalšie atribúty ani metódy.



Názov kritéria (meno, priemer známok, počet známok) sa odrazí v identifikátore konkrétneho potomka. Množina parametrov Π (pí) dodatočne špecifikuje vlastnosť kritéria. Nakoľko je kritérium objekt, svoje vlastnosti uchováva formou atribútov (ak máte atribúty, mali by ste aj definovať konštruktory, ktorými ich inicializujete). Jediné kritérium v našom príklade, ktoré má parametre, je kritérium III. Môžeme preto definovať nasledujúcu objektovú hierarchiu:



Ostáva už iba implementovať telá prekrytých metód `evaluate`. Nakoľko je potrebné pristupovať k atribútom (v našom prípade triedy `Student`), môžeme postupovať dvomi spôsobmi:

Definovať v triede `Student` príslušné metódy a kritérium ich iba zavolá (všimnite si, že takto to je v popise kritéria, napr. pre kritérium K^{Meno} máme definované $\kappa = \omega.priemerZnámokŠtudenta; \tau = double$, čo by viedlo k metóde `double Student::averageGrade()`). Týmto so svojimi atribútmi pracuje iba táto trieda (podporujete princíp ukrývania informácií), avšak musí poskytovať metódy, ktoré jej kód môžu zneprehľadniť. Naviac, ak by ste sa rozhodli pridať ďalšiu funkčnosť (kritérium) v budúcnosti, potom by ste museli modifikovať kód triedy `Student`.

1. Všetko definovať v kritériu. To sa ale musí dostať k hodnotám vlastností v objekte, ktorý ohodnocuje. To je možné docieliť buď gettrami alebo definíciou `friend`. V takomto prípade kód lepšie distribuujete. Oba prístupy cez kritérium však majú ďalšie dôsledky:
    i. Gettre je však možné navrhnúť ako `const`, navyše môžu vracať referenciu na objekty (&), takže nedôjde k pomalému kopírovaniu objektov a zamedzíte ich modifikácii. Ak sa teda rozhodnete v budúcnosti pridať ďalšiu funkčnosť, nemusíte modifikovať objekt, lebo všetko poskytuje (avšak dochádza potenciálne k porušeniu zapúzdrenia – princíp ukrývania informácii).
    ii. Ak využijete `friend`, potom každý potomok kritéria musí byť `friend` s triedou `Student`. Opäť sa dostaneme k problému, že pre pridanie funkčnosti je potrebné modifikovať túto triedu (pridať `friend` deklaráciu), avšak kód je stále distribuovaný a nedochádza k ukrývaniu informácií.

Všetky prístupy majú svoje +/- a je na programátorovi, aby zvolil správny v jeho danej situácii. Teraz funkčnosť demonštrujeme s využitím prístupu 2.i. Do triedy `Student` pridáme nasledujúce gettre:

```
const std::string& getName() const { return name_; }
const std::vector<int>& getGrades() const { return grades_; }
```

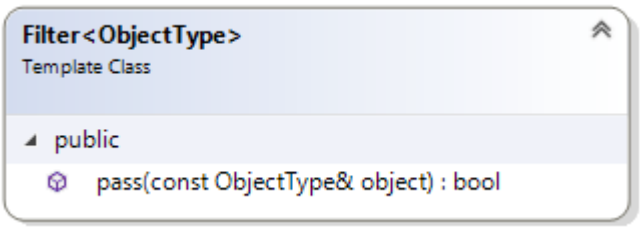
Telá jednotlivých metód `evaluate` v jednotlivých potomkoch by mohli vyzerať nasledujúco (využívame knižnice `algorithm` a `numeric`):

```
std::string CriterionStudentName::evaluate(const Student& object) override {
  return object.getName();
}

double CriterionStudentGradesAverage::evaluate(const Student& object) override {
  return std::accumulate(object.getGrades().cbegin(), object.getGrades().cend(), 0) /
         object.getGrades().size();
}

unsigned int CriterionStudentGradesCount::evaluate(const Student& object) override {
return std::count_if(object.getGrades().cbegin(), object.getGrades().cend(),
                     [&](int i) {return i == grade_; });
}
```

Ako si navrhnúť filtre?

Univerzálne kritériá sa prejavia pri univerzálnom návrhu filtrov. Podľa zadania majú filtre prijať štruktúru údajov a vrátiť vyfiltrovanú štruktúru údajov (podľa vášho návrhu to môže byť pôvodná štruktúra, alebo nová štruktúra s prvkami, ktoré prešli filtrom). V rámci tejto príručky popíšeme, ako filter určí pre **jeden** prvok, či tento prvok filtru vyhovuje alebo nie.

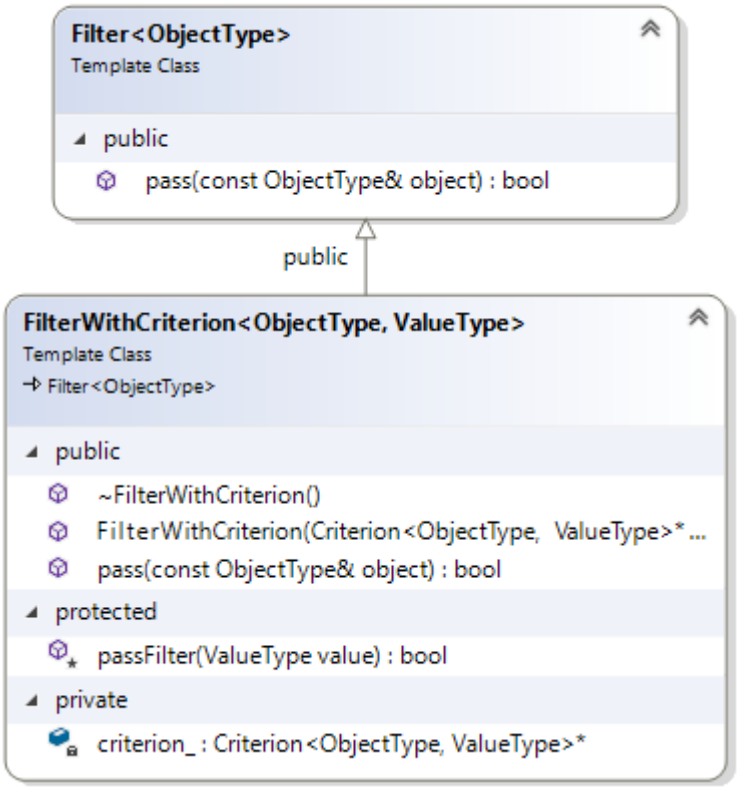
Filter spolupracuje s kritériom a testuje, či hodnota objektu vrátená kritériom je rovná filtrovanej hodnote α (filter $\varphi^{K_{[p,q,..]}}_{\alpha[,p,q,..]}(\Omega)$) alebo či táto hodnota patrí do zadaného intervalu <α, β> (filter $\Phi^{K_{[p,q,..]}}_{\alpha,\beta[,p,q,..]}$). Môžeme opäť definovať spoločného predka pre akékoľvek filtre. Tento predok preberie testovaný objekt a v abstraktnej metóde určí, či filtrom prešiel. Ak chceme filtrovať akékoľvek objekty vopred neznámeho typu, môžeme využiť šablónový parameter `ObjectType`.



Filtre podľa analýzy v zadaní preberajú kritérium a testujú, či hodnota kritéria vyvolaného nad objektom nadobúda hodnotu alebo patrí do intervalu hodnôt. Nakoľko kritériá majú spoločného predka, môžeme vytvoriť spoločného predka pre filtre testujúce zhodu s hodnotou alebo príslušnosť do intervalu. Uvedomte si, že ak budeme mať spoločného predka, tak potomkovia nebudú musieť kritérium vyhodnocovať – o ohodnotenie sa môže postarať predok a potomkovia len testujú hodnoty. Typ tejto hodnoty musí byť zhodný s návratovým typom kritéria (τ). Nakoľko nie je vo všeobecnosti známy, opäť použijeme šablónový parameter `ValueType` - musí byť typovo zhodný s parametrom `ResultType` v kritériu, preto kritérium v spoločnom predkovi musí byť typu `Criterion<ObjectType, ValueType>*` (musí to byť ukazovateľ na objekt, pretože predok je abstraktný a nie je možné vytvoriť jeho inštanciu – ukazovateľ však môže ukazovať aj na jeho potomka, takže s výhodou využijeme polymorfizmus). V spoločnom predkovi filtrov môžeme kompletne zapúzdriť prácu s kritériom:

- získanie hodnoty (v metóde `pass`)
- manažment životného (v konštruktore si priradí kritérium špecifikované potomkom, v deštruktore ho uvoľní) .

Potomok iba otestuje získanú hodnotu. Môžeme tak prekryť metódu `pass` (definovanú na predkovi) a pre potomkov vynútiť definovanie **chránenej** (z vonku stále používame metódu `pass`) abstraktnej metódy `passFilter` s parametrom `ValueType`, čím sa prekrytie pre nich stane veľmi jednoduchým.

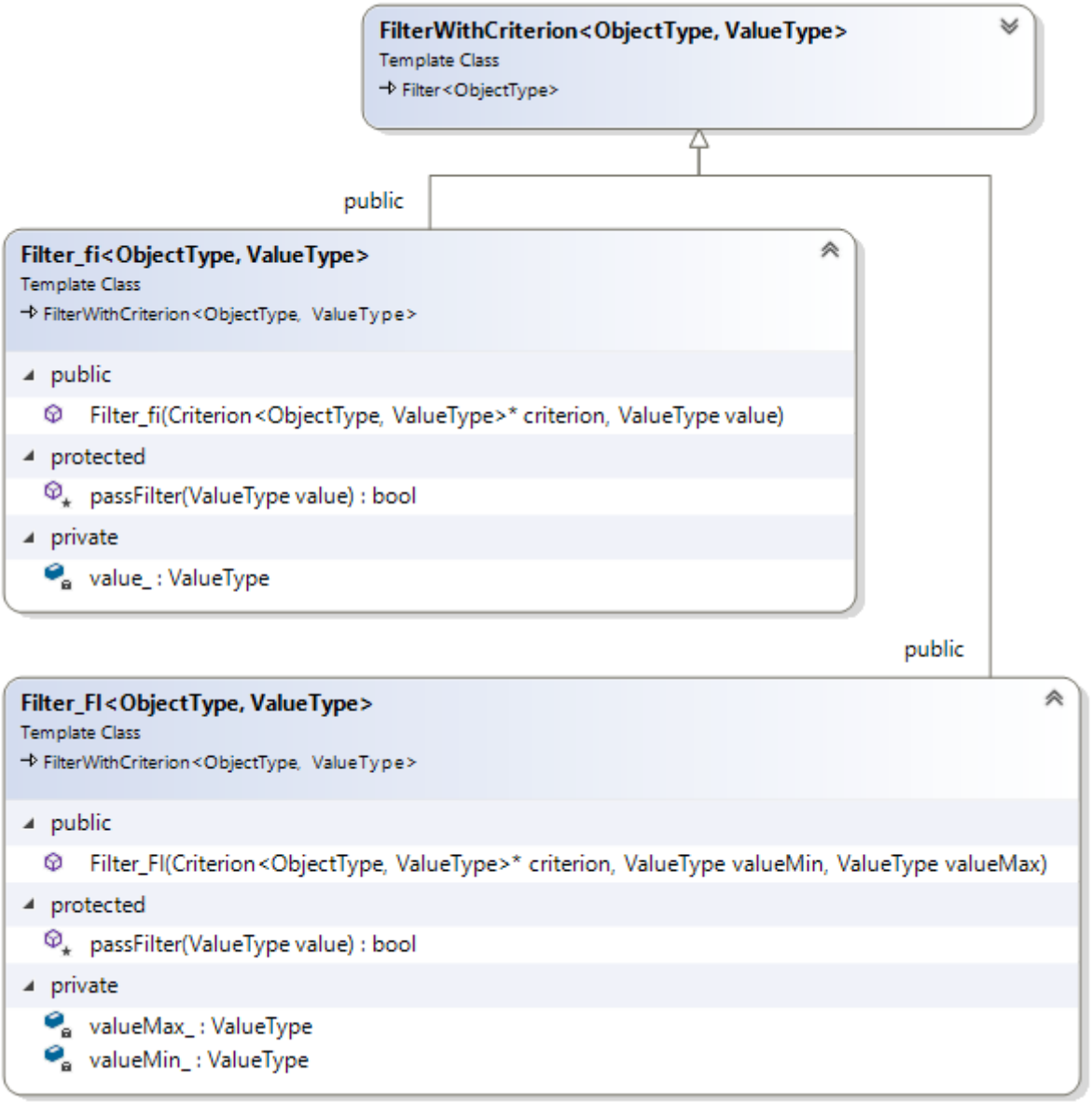


Univerzálna operácia operácie `pass` tak môže vyzerať nasledujúco. Pomocou kritéria necháme ohodnotiť objekt (dali sme si pozor na šablónové parametre, takže určite sú typovo kompatibilní). Túto hodnotu potom pošleme do metódu passFilter, ktorú prekryje príslušný potomok.

```
bool FilterWithCriterion::pass(const ObjectType& object) override {
  return passFilter(criterion_->evaluate(object));
}
```

S takto pripraveným predkom môžeme pripraviť dvoch potomkov. Ich úlohou je len prekryť metódu passFilter a zistiť, či sa daná hodnota zhoduje s parametrom filtra alebo patrí do intervalu

definovaným parametrami filtra. Podobne, ako bolo uvedené pri kritériách, filter je objekt a jeho „parametre“ sú jeho atribúty. Tieto stačí spolu s konkrétnym filtrom nastaviť v konštruktore.



Metódy `passFilter` potom môžu byť prekryté nasledovne:

```
bool Filter_fi::passFilter(ValueType value) override {
  return value == value_;
}

bool Filter_FI::passFilter(ValueType value) override {
  return value >= valueMin_ && value <= valueMax_;
}
```

Úroveň univerzálnych filtrov je hotová. Ak by sme chceli vytvoriť filter pre študentov, ktorých meno sa zhoduje so zadaným $\boldsymbol{F^{Meno}}$: $\varphi^{K^{Meno}}$ (všimnite si, že definícia Vám presne hovorí, ktorý filter treba vytvoriť), tak môžeme pripraviť objekt

```
class FilterStudentName : public Filter_fi<Student, std::string> {
public:
  FilterStudentName(std::string name) :
    Filter_fi(new CriterionStudentName(), name)
```

```
  {}
};
```

Niekde v kóde potom, keby som chcel vyfiltrovať Michalov, tak vytvorím objekt:

```
FilterStudentName filterStudentMichal("Michal");
```

Ak by sme chceli vytvoriť filter pre študentov, ktorých priemer je v nejakom rozpätí $\boldsymbol{F^{PriemerZnamok}}.\Phi^{K^{PriemerZnamok}}$, tak môžeme pripraviť objekt

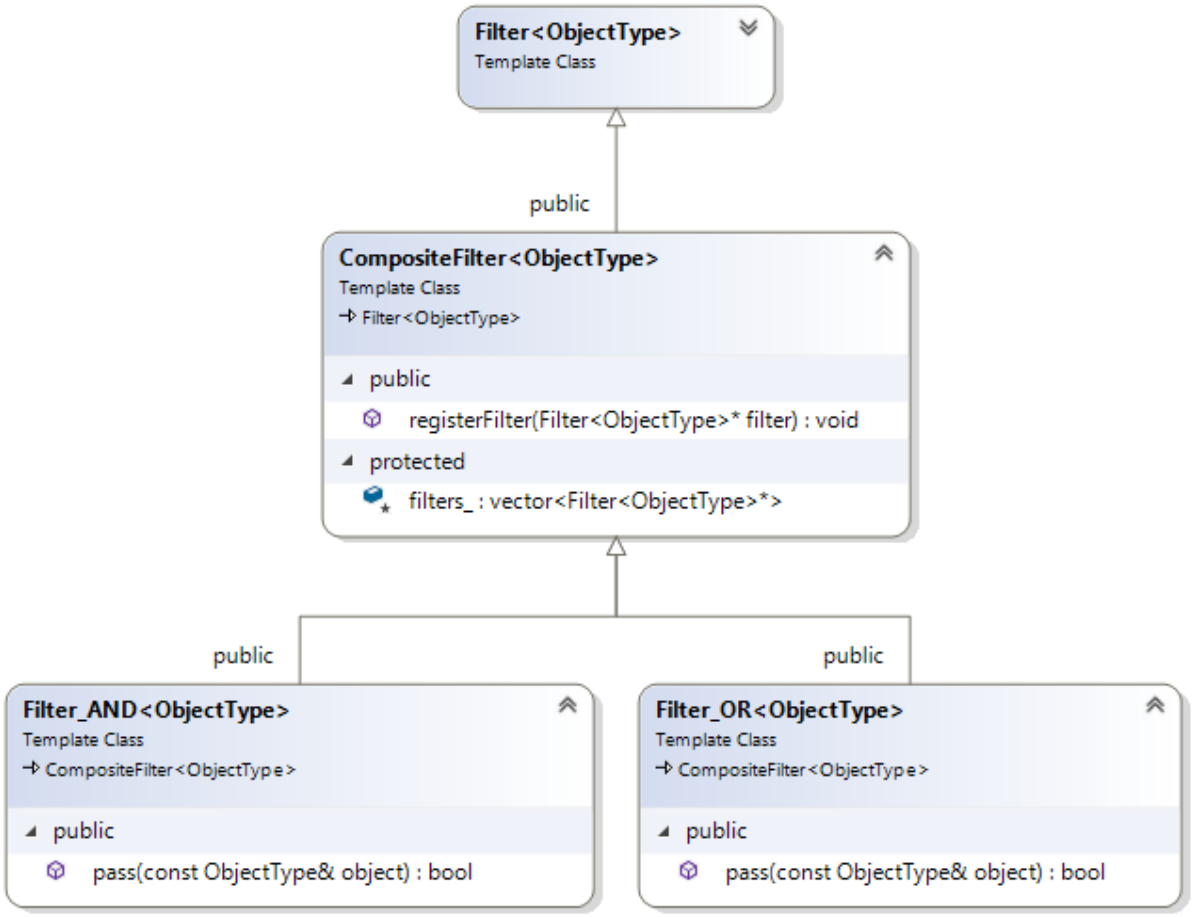
```
class FilterStudentGradeAverage : public Filter_FI<Student, double> {
public:
  FilterStudentGradeAverage(double minAverage, double maxAverage) :
    Filter_FI(new CriterionStudentGradesAverage(), minAverage, maxAverage)
  {}
};
```

Niekde v kóde potom, keby som chcel vyfiltrovať študentov s priemerom 1 – 1,44, tak vytvorím objekt:

```
FilterStudentGradeAverage filterStudentGradeAverage(1, 1.44);
```

Všimnite si, že filtre pre ten istý typ objektu, majú spoločného predka. Môžete tak spraviť **kompozitný filter**, ktorý sa bude skladať z filtrov (má v sebe ich zoznam – v tejto ukážke je vector, vy použite vlastnú implementáciu!). Metódu pass potom môžete prekryť dvomi spôsobmi:

- **AND**: Testovaný objekt musí splniť všetky testy v zozname (prejsť metódou pass vo všetkých filtroch).
- **OR**: Testovaný objekt musí splniť aspoň jeden test v zozname (prejsť metódou pass aspoň v jednom filtry).



Ak by ste teraz chceli spraviť zaujímavý filter, ktorý vyfiltruje všetkých Petrov a Jany s priemerom medzi 1 – 2, tak je to výraz $(meno = \text{Peter} \vee meno = \text{Jana}) \wedge priemer \in \langle 1|2 \rangle$. Objektovo si ho poskladáte (všimnite si, že robíte strom):

```
FilterStudentName filterPeter("Peter");
FilterStudentName filterJana("Jana");
FilterStudentGradeAverage filterPriemer12(1, 2);

Filter_OR<Student> filterMeno;
filterMeno.registerFilter(&filterPeter);
filterMeno.registerFilter(&filterJana);

Filter_AND<Student> zaujimavyFilter;
zaujimavyFilter.registerFilter(&filterMeno);
zaujimavyFilter.registerFilter(&filterPriemer12);
```

Následne aplikujte operáciu pass zo zaujímavého filtra na vami testovaného študenta.

Ešte raz zdôrazňujeme, že takto definovaný filtre otestuje iba jednu položku. Ak budete chcieť vyfiltrovať celú údajovú štruktúru (ako bolo spomenuté na začiatku tejto kapitoly), musíte filter aplikovať na všetky jej prvky a podľa toho, či testovaný prvok filter splnil alebo nie, vykonať skutočné filtrovanie.

Všimnite si veľkú prednosť tohto prístupu. Ak si správne navrhnete objekty, potom bude Vaše používateľské rozhranie iba o tom, aby ste vytvorili správny kompozitný filter. Môžete sa napr. pýtať – chceš použiť aj tento filter? Aj tento? A čo tento? Takto zoskladaný filter (fantázii sa medze nekladú) potom aplikujete na filtrovanú štruktúry jediným algoritmom. Kedykoľvek budete chcieť pridať filtrovanie podľa niečoho iného (ale prezentovať údaje z objektu nejako inak), tak si iba vytvoríte nové

potomka triedy kritérium a vložíte ho do pre vás zaujímavého filtra (podľa hodnoty alebo intervalu). Takáto úprava sa však zvyšku vašej aplikácie nedotkne.

Ako si navrhnúť výberové kritériá?

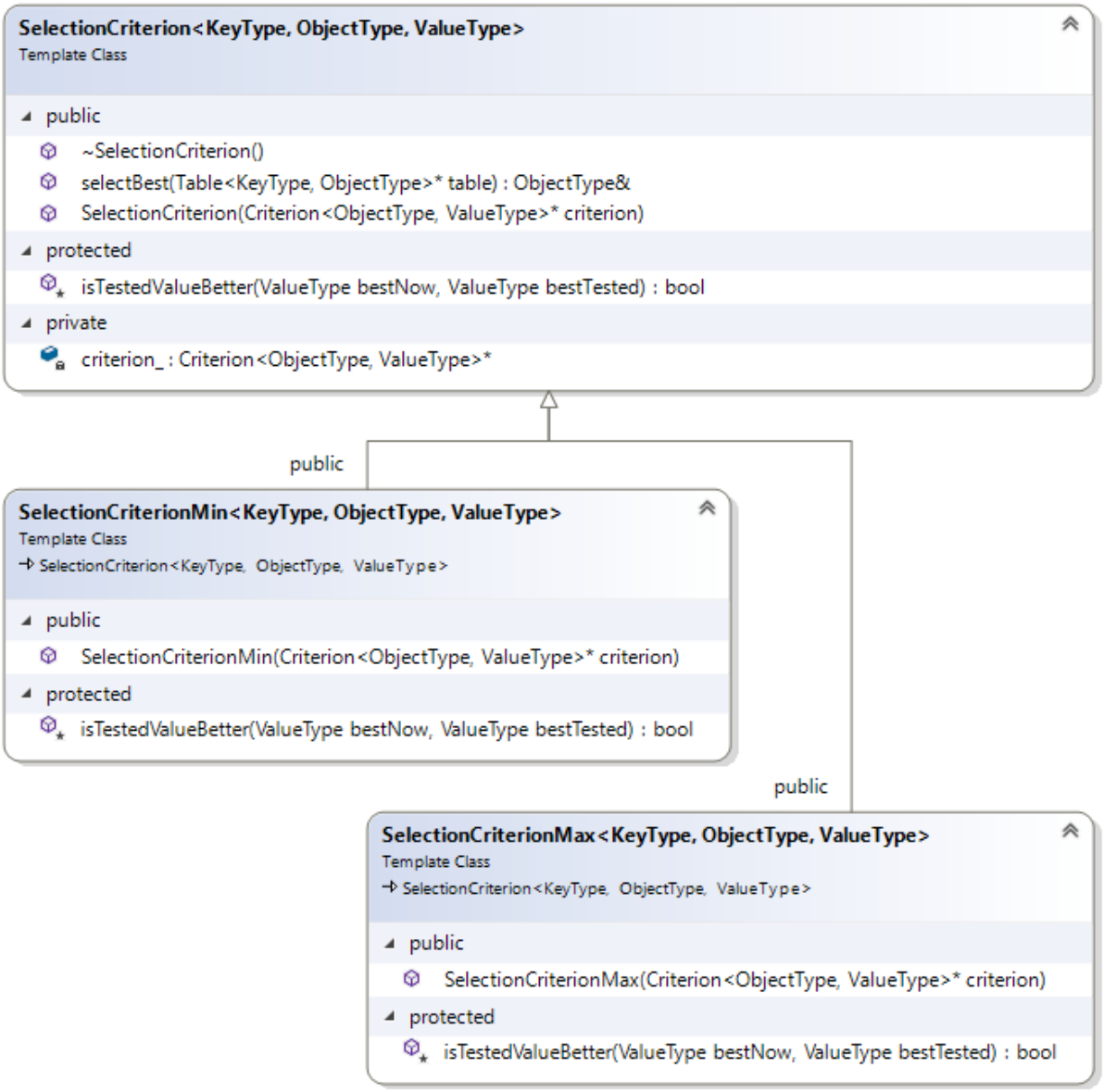
Výberové kritérium je špeciálny druh filtra, ktorý vyfiltruje práve jeden prvok - taký s najväčšou $\mathrm{M}^K(\Omega)$ alebo najmenšou $\mu^K(\Omega)$ hodnotou nejakého kritéria K. Návrh môžeme urobiť veľmi podobne ako pri filtroch, avšak s tým rozdielom, že vieme, že budeme určite pracovať s hodnotou a nie intervalom. Algoritmicky sa jedná o veľmi jednoduchú úlohu – nájsť minimum (alebo maximum). Porovnávanú hodnotu Vám vždy vráti kritérium, ktoré vo výberovom kritériu použijete.

Návrh na univerzálnej úrovni je možné spraviť, avšak je závislý na štruktúre, v ktorej máte údaje uložené. Ak sa jedná o tabuľku, potom iterátory, ktoré iterujú tabuľkou vracajú v metóde `operator*` ukazovateľ na triedu `TableItem<K,T>`. Ak máte údaje v inej štruktúre, musíte si návrh analogicky upraviť. Kritérium má dva šablónové parametre – typ objektu (`ObjectType`) a typ návratovej hodnoty (`ResultType`). Tabuľka má tiež dva parametre – typ kľúča (`K`) a typ dát (`T`). Aby sme mohli pomocou kritériá ohodnocovať prvky tabuľky, tak musí byť typ prvku tabuľky a typ objektu kritéria taký istý. Hodnota, ktorú budeme porovnávať bude zhodná s typom hodnoty kritéria. Na type kľúča nezáleží.

Podobne, ako pri filtroch, ak aj pri výberových kritériách budeme mať spoločného predka, tak potomkovia nebudú musieť kritérium vyhodnocovať – o ohodnotenie sa môže postarať predok a potomkovia len testujú hodnoty – našli sme lepšiu? Typ tejto hodnoty musí byť zhodný s návratovým typom kritéria (τ). Nakoľko nie je vo všeobecnosti známy, opäť použijeme šablónový parameter `ValueType` - musí byť typovo zhodný s parametrom `ResultType` v kritériu, preto kritérium v spoločnom predkovi musí byť typu `Criterion<ObjectType, ValueType>*` (zopakujeme, že to musí byť ukazovateľ na objekt, pretože predok je abstraktný a nie je možné vytvoriť jeho inštanciu – ukazovateľ však môže ukazovať aj na jeho potomka, takže s výhodou využijeme polymorfizmus). V spoločnom predkovi filtrov môžeme kompletne zapúzdriť prácu s kritériom:

- získanie hodnoty (v metóde `selectBest`)
- manažment životného (v konštruktore si priradí kritérium špecifikované potomkom, v deštruktore ho uvoľní) .

Algoritmus na nájdenie najlepšieho prvku (najmenšieho alebo najväčšieho) je možné zapúzdriť do metódy `selectBest` (všimnite si návratovú hodnotu typu referencia, aby sa nájdený objekt zbytočne nekopíroval). Aby ste nemuseli pomocou vetvenia kódu testovať, či je aktuálny prvok lepší alebo horší, môžete si pridať chránenú virtuálnu metódu `isTestedValueBetter`, v ktorej otestujete, či ste našli lepšiu alebo horšiu hodnotu. Túto metódu prekryje potomok a otestuje v našom prípade väčší alebo menší (avšak mohli by ste otestovať čokoľvek iné). Návrh by teda mohol vyzerať nasledujúco:



Nakoľko sa v našich štruktúrach ako objekty nachádzajú ukazovatele, musíme dereferovať objekty pri prístupe k nim (`accessData`) nemám Telo cyklu, ktorým prechádzate všetky prvky tabuľky by mohlo vyzerať napr. takto:

```
ObjectType& testedObject = *tableItem->accessData(); // pozor na *!!!
ValueType testedValue = criterion_->evaluate(testedObject);
if (isTestedValueBetter(bestValue, testedValue)) {
  bestValue = testedValue;
  bestObject = testedObject;
}
```

`bestValue` a `bestObject` sú najlepšie nájdené hodnoty a objekt (typu `ObjectType`&), ktoré je potrebné pred cyklom (alebo v prvej iterácii cyklu) vhodne nastaviť. `tableItem` je premenná, ktorá v každej otočke cyklu nadobúda ďalšiu hodnotu prvku tabuľky. Po cykle je najlepší objekt v premennej `bestObject`.

Prekrytie metód u potomkov je potom jednoduché:

```
bool SelectionCriterionMin::isTestedValueBetter(
  ValueType bestNow, ValueType bestTested) override {
```

```
    return bestTested < bestNow;
}

bool SelectionCriterionMax::isTestedValueBetter(
  ValueType bestNow, ValueType bestTested) override {
    return bestTested > bestNow;
}
```

Ak by sme chceli vytvoriť výberový filter pre študentov s najlepším priemerom $\boldsymbol{V^{MinPriemerZnamok} = \mu^{K^{PriemerZnámok}}}$ (všimnite si, že definícia Vám presne hovorí, ktoré kritérium treba použiť), tak môžeme pripraviť objekt

```
template <typename K>
class SelectionBestStudentAverage : public SelectionCriterionMin<K, Student, double> {
public:
  SelectionBestStudentAverage() :
    SelectionCriterionMin<K, Student, double>(new CriterionStudentGradesAverage())
  {}
};
```

Ak by sme mali tabuľku študentov `Table<int, Student*>* students` (všimnite si, že ako dáta sú ukazovatele na objekty!), tak nájdenie študenta s najlepším priemerom známok by bolo jednoduché:

```
SelectionBestStudentAverage<int> selectBestStudent;
Student& bestStudent = selectBestStudent.selectBest(students);
```

Záver

Semestrálna práca sa opiera o kritériá, filtre a výberové kritériá, ktorých implementáciu je do veľkej miery možné realizovať podľa tohto návodu. Ak sa však rozhodnete, že nechcete implementovať algoritmy takto univerzálne, na výsledné hodnotenie semestrálnej práce to nebude mať negatívny vplyv (ak bude riešenie funkčné a spĺňať ostatné požiadavky na neho kladené).