Programozási nyelvek és paradigmák

Fenyvesi, Róbert
fenyvesr@gmail.com

2019. december 20.

Elméleti kérdések kidolgozása

1. Írd fel egy exportált rutin és egy exportált creation procedure szerződését Hoare-hármasok segítségével, és magyarázd el a különbséget!

Az alapvető különbség, hogy egy r rutin megőrzi az osztály invariánst, míg a creation procedurenek (természetesen) csak az utófeltételében szerepel. Tehát,

$$\begin{array}{rcl} \{Q_m\} & make & \{inv_C \wedge R_m\} \\ \{inv_C \wedge Q_r\} & r & \{inv_C \wedge R_r\}, \end{array}$$

ahol r az exportált rutin, *make* az exportált creation procedure és inv_C az osztály invariánsa. Emellett Q_r és R_r az r rutin, míg Q_m és R_m a creation procedure elő- és utófeltétele.

2. Mi a szerepe a `like` kulcsszónak az Eiffelben?

Öröklődési hierarchiában sokszor okozhat gondot, ha egy típust az alosztályok megfelelő közös ősével szeretnénk meghatározni. Az öröklődés és polimorfizmus miatt Objektum-orientált nyelveknél sok esetben fordulhat elő az az eset, hogy nem ismerjük az egyes változók dinamikus típusát fordítási időben. Ez az ismeret viszont sok esetben fontos lenne. Például a Java-ban az `equals` függvény az `Object` osztályon definiálva a következő szignatúrával rendelkezik,

```
public boolean equals(Object obj)
```

Ebben az esetben egy `String` típusú objektumra bármikor meg lehet hívni az `equals` függvényt bármely `Object`-ből származó paraméterrel (pl. `Integer`)

Az Eiffelben ennek a jelenségnek a kiküszöbölésére vezették be a `like` kulcsszót, mellyel ún. *kapcsolt típusokat* (anchored type) lehet létrehozni. Ezt a kulcsszót többféleképpen is lehet használni. Az előző példában látható jelenség elkerüléséhez a következő szignatúrát írhatjuk Eiffelben,

```
is_equal (other: like Current): BOOLEAN
```

A `Current` az aktuális osztály típusára hivatkozik, így bármely leszármazotton a megfelelő paramétertípussal fog rendelkezni a függvény.

Az osztály típusa mellett hivatkozhatunk azonosító típusára is, így növelve a kód újrafelhasználhatóságát.

```
like <azonosito>
```

Tekintsünk egy egyszerű példát, ahol síelő fiúkat és lányokat akarunk elszállásolni, viszont csak az azonos neműeket szeretnénk egy szobába tenni. Ezt a típusokkal tudjuk elérni a következő módon.

```
class SKIER
feature
  roommate: like Current

  share( other: like roommate ) is
  require
    other /= Void
  do
    roommate := other
  ensure
    other = roommate
  end
end
```

Láthatjuk, hogy a `roomate` attribútum típusa `like Current` azaz meg kell egyezzen az aktuális típussal (A `Current` az aktuális objektumot jelöli. Olyan, mint Java-ban a `this`.). Ez azt jelenti, hogy ha egy `GIRL` osztályt leszármaztatunk a `SKIER` osztályból, akkor annak a `roomate` attribútumoka is `GIRL` típusú kell legyen. Emellett láthatjuk, hogy a `share` metódus paramétere `like roomate` típusú, magyarul csak olyan objektumot adhatunk paraméterként, mely megegyezik a `roomate` típusával. (`GIRL` esetén csak `GIRL` lehet)

Megjegyzés: Létezik kiskapu, van lehetőségünk fiúkat és lányokat egy szobába rakni. Ezt a kiskaput "Polymorphic CAT-call"-nak hívják.

Az `ANY` osztálynak van egy `copy` feature-e, melyet a következőképpen definiált,

```
copy (other: like Current)
```

3. Mi a mixin-inheritance lényege?

Mixin-inheritance amikor egy generikus paraméterből származtatunk le.

```
template <class Base>
class Mixin : public Base  { ... };

class Base { ... };
```

A többszörös öröklődés által okozott kétértelműséget hivatott kiküszöbölni. A mixin-ek linearizálják az öröklést, így oldják fel a problémákat. Különböző nyelvek különbözőképpen alkalmazzák.

Például C++ esetében,

```
template <typename T> class Numeric : public T { ... };
template <typename T> class Comparable : public T { ... };
class Base {};
```

Ekkor a következő módon lehet `Rational` osztályt definiálni,

```
class Rational : Comparable< Numeric<Base> > { ... };
```

Eiffel-ben nem lehetséges mixin-ek definiálása.

4. Milyen szerződések vonatkoznak az Eiffel rutinokra? Hogyan befolyásolja ezt az öröklődés? Hogyan a felüldefiniálás? Magyarázd el a kapcsolódó fogalmakat és szabályokat. Fogalmazd meg a szerződéseket Hoare-hármasok segítségével!

Eiffelben a rutinokra explicit módon az elő- és utófeltétel szerződések vonatkoznak, míg implicit módon az osztály invariáns.

$$\{Q_r \wedge inv_C\} r \{R_r \wedge inv_C\},$$

ahol Q_r és R_R az r elő- és utófeltételei, míg inv_C az osztály invariánsa.

Az öröklődés és polimorfizmus talán legnagyobb kérdése a *variancia*. A variancia két típus helyettesíthetőségét fejezi ki. $A :> B$ az altípus relációt jelenti, vagyis ebben az esetben a B típus az A-nak altípusa.

Kovarianciának nevezzük, ha az általánosabb A típus helyére a speciálisabb B típust behelyettesíthetjük.

Kontravarianciának nevezzük, ha a speciálisabb B típus helyére az általánosabb A típust helyettesíthetjük be.

Invariáns vagy *nonvariáns* a reláció, ha a fentiek közül egyik sem mondható.

A fent említett tulajdonságok kontextustól függőek. A legfontosabb felhasználási területe ezeknek a tulajdonságoknak az öröklődés során a metódusok specializációja. Egy metódust akkor tudunk típushelyesen specializálni, ha a paraméterei kontravariánsak az ősosztály metódusának paramétereivel, míg a visszatérési típusa kovariáns a ősosztály metódusának visszatérési típusával.

Tekintsük a következő példát! Tegyük fel, hogy $CREATURE :> ANIMAL :> MONKEY$ és $FOOD :> FRUIT :> BANANA$, illetve létezik egy `ANIMAL_FEEDER` osztályunk,

```
class ANIMAL_FEEDER
creation
  make
feature {ANY}
  feed(a : ANIMAL) : FRUIT
  do
    -- implementation
  end
end
```

Ha létre akarjuk hozni a `MY_FEEDER` osztályt és specializálni szeretnénk a `feed` metódust, akkor a paramétere lehet továbbra is `ANIMAL` típusú, vagy `CREATURE`, de `MONKEY` semmiképp. Ugyanis azon a helyen, ahol kicseréljük az `ANIMAL_FEEDER` példányt, az azt használók nem feltétlenül csak `MONKEY` típusú paramétereket adhatnak át. Hasonlóképpen a visszatérési érték csak `FRUIT` és `BANANA` lehet. Az `ANIMAL_FEEDER`-t használók `FRUIT` típust vagy annak altípusait várják értékül.

Ahogy a metódusok paramétereire és visszatérési értékére, a szerződésekre is érvényes a variancia. A feature-ök előfeltételeit lazítani lehet, míg az utófeltételeit megszorítani. Erre a `require else` és `ensure then` kulcsszavakat lehet használni. A `require else` esetén az újonnan megfogalmazott előfeltétel *vagy* kapcsolatban fog állni az eredetivel. Az `ensure then` esetén az utófeltétel szigorodik, *és* kapcsolatban fog állni az eredeti utófeltétellel. Tehát, ha $B <: A$, akkor B minden r rutinjára teljesülnie kell a következőnek

$$\{Q_{rA} \vee Q_{rB}\} r \{R_{rA} \wedge R_{rB}\}$$

Az öröklődés során Eiffelben a leszármazottak megőrzik az ősök osztályinvariánsait. Így a leszármazott invariánsa a leszármazottban megfogalmazott invariáns és ősei invariánsának konjunkciója lesz (összeéselődnek). Tehát, ha $B <: A$, akkor B minden r rutinjára a következő teljesül

$$\{Q_r \wedge inv_A \wedge inv_B\} r \{R_r \wedge inv_A \wedge inv_B\},$$

ahol Q és R az r elő- és utófeltételei, míg inv_j a megfelelő j osztályok invariánsai.

5. Milyen viszonyban kell álljanak az A_i és a B_j típusok ahhoz, hogy fennálljon az alábbi altípus reláció (matematikailag helyesen)? Indokold válaszod! $(A_1 \to A_2) \to A_3 \to A_4 <: (B_1 \to B_2) \to B_3 \to B_4$

Az elhagyott zárójeleket kitéve a kifejezés a következő

$$((A_1 \to A_2) \to (A_3 \to A_4)) <: ((B_1 \to B_2) \to (B_3 \to B_4)).$$

A paraméter kontravariáns, az eredmény kovariáns kell legyen, így

$$(B_1 \to B_2) <: (A_1 \to A_2) \qquad (A_3 \to A_4) <: (B_3 \to B_4).$$

Ugyanezen gondolatmenet alapján,

$$A_1 <: B_1, \qquad B_2 <: A_2, \qquad B_3 <: A_3, \qquad A_4 <: B_4.$$

6. A strukturális altípusosságot hogyan szokás definiálni direktszorzat (tuple) típusokra?

Általánosságban a kovariancia jellemző, vagyis

$$TUPLE[INTEGER, STRING] <: TUPLE[INTEGER, ANY] <: \\ <: TUPLE[INTEGER] <: TUPLE[ANY] <: TUPLE.$$

7. A strukturális altípusosságot hogyan szokás definiálni rekord típusokra?

Általánosságban a kovariancia jellemző, vagyis

$$\{x_1 : \sigma_1, \cdots, x_n : \sigma_n, \cdots, x_m : \sigma_m\} <: \{x_1 : \sigma_1, \cdots, x_n : \sigma_n\}$$

8. Mire való az old kulcsszó az Eiffelben? Magyarázd el, fogalmazd meg a rá vonatkozó szabályokat, illusztráld használatát példán!

Eiffelben az `old` kulcsszót utófeltételekben szokás használni. Egy azonosító régi értékére lehet vele hivatkozni. Tegyük fel, hogy `x` az osztály egy attribútuma. Ekkor megfogalmazhatjuk a következő feature-t.

```
increment
do
  x := x + 1
ensure
  x = old x + 1
end
```

9. Hogyan működik, miért és mire jó az átnevezés az öröklődés során?

Többszörös öröklődés esetén az ősosztályokban lévő megegyező nevű feature-öket átnevezéssel tudjuk megkülönböztetni egymástól.

10. Mit jelent a mixin inheritence? Mi az előnye? Minek az alternatívája?

Lásd a 3. kérdést.

11. Milyen szerződésnek kell megfelelnie a retry kulcsszóval végződő kivételkezelő ágaknak? Fogalmazd meg Hoare-hármas segítségével!

Ha a `rescue` kivételkezelő ág `retry` kulcsszóval rendelkezik, akkor a vezérlés vissza fog térni a r rutin elejére, megpróbálja az utasításokat újra végrehajtani. Ennél fogva a rá vonatkozó szerződés, hogy az ág R_{rescue} utófeltétele azonos vagy erősebb legyen a rutin előfeltételénél $R_{rescue} \subseteq Q_r$.

Emellett a `rescue` kivételkezelő ág elsődleges feladata az osztályinvariáns visszaállítása, vagyis ezeket a következő Hoare-hármassal írhatjuk le

$$\{Igaz\} rescue \{R_{rescue} \wedge inv_C\}.$$

A kivételkezelő ág előfeltételére nem tudunk megszorítást adni, mivel a rutin bármely részénél átválthat a vezérlés.

12. Milyen szerződésnek kell megfelelnie egy ciklusnak? Fogalmazd meg Hoare-hármasok segítségével!

Eiffelben a ciklusok is kiemelt figyelmet kapnak, mivel ezeknek is lehet megkötéseket, kiegészítő információkat adni. A nyelv Programozáselméletből ismert ciklus invariáns és variáns függvény kifejezésére ad lehetőséget. Egy ciklus szintaxisa a következő,

```
from
  -- initalization block
invariant
  -- loop invariant
until
  -- terminal condition
loop
  -- loop body
variant
  -- loop variant
end
```

A ciklus invariáns egy olyan állítás, melynek a ciklus előtt, majd minden iteráció után igaznak kell lennie. A ciklus variáns vagy variáns függvény pedig olyan nemnegatív egész szám, melynek értéke minden iteráció után legalább eggyel csökken. Mivel egy nemnegatív egész szám nem csökkenthető a végtelenségig, illetve mivel kötelező a csökkenés így biztosítva van a terminálás.

Hoare-hármasokkal megfogalmazva,

$$\{Igaz\}INIT\{INV\}$$
$$\{Igaz\}INIT\{VAR \geq 0\}$$
$$\{INV \wedge \neg EXIT\}BODY\{INV\}$$
$$\{INV \wedge \neg EXIT \wedge VAR = v\}BODY\{0 \leq VAR < v\},$$

ahol $INIT$ a ciklus inicializációs blokkja, INV a ciklusinvariánsa, $EXIT$ a kilépési feltétel, $BODY$ a ciklus törzse, VAR a variáns függvény. (link 8.9.22)

13. Mire való az "old"? Mikor, hogyan, mire használjuk? Mutass példát is!

Lásd a 8. kérdést

14. Mit szokás bináris műveletnek nevezni az objektumorientált programozásban?

Objektumorientált programozásban bináris műveletnek nevezzük amikor a művelet paraméterének típusa a fogadó objektum típusával kell megegyezzen, mint mondjuk egyenlőségvizsgálatnál `a.is_equal(b)`.

15. Mi a különbség az `=`, a $\sim$, az `is_equal`, az `equals`, a `standard_is_equal` és az `is_deep_equal` között? Mi közük van egymáshoz?

Az `=` *shallow equality*-t viszgál, vagyis csak a referenciákat hasonlítja össze (címeket). Azt, hogy igazából mit mutatnak nem hasonlítja össze.

Az `is_equal` (ua. mint a $\sim$) újradefiniálható, objektumok attribútumait is összehasonlítja (pl. `point1.is_equal(point2)` akkor és csakis akkor, ha `point1.x = point2.x` és `point1.y = point2.y`. Mondhatjuk, hogy ez a megszokott egyenlőség. A `standard_is_equal` is ugyanígy működik. Fontos, hogy csak attached attribútumokon működik.

`is_deep_equal` *deep equality*-t vizsgál, vagyis az attribútumok attribútumait,stb. is rekurzívan összehasonlítja. (fix implementációjú)

16. Hogyan szabályozza az Eiffel a láthatóságot? Miben különbözik ez a más nyelvekben megszokottól?

Szelektív láthatóság esetén megadhatjuk, mely osztály(ok) számára legyen látható az adott feature. Ilyen esetben az altípusa(i) számára is látható lesz.

```
create {A,B,C} make
feature {A,B} make
```

Megadhatjuk, hogy egy feature nyilvános legyen (`{ANY}` kulcsszóval vagy semmit nem írunk elé), vagy titkos (`{}` vagy `{NONE}` kulcsszavakkal). A titkos feature még azonos típusú változónak sem elérhető, nem olyan, mint a `private`.

```
class BASE
feature {}
  v: INTEGER  --titkos attributum
feature
  query( other: BASE ): INTEGER
  do
    Result := other.v -- forditasi hiba
  end
end

class SUB inherit BASE
feature
  mine: INTEGER
  do
    Result := v -- ok, mivel az objektum attributuma
  end
end
```

Lehet csökkenteni a láthatóságot altípusban.

```
class QUEUE[T]
  inherit {NONE} SEQUENCE[T]
    export {ANY}   hiext, lov, lorem, size;
           {QUEUE} all -- e.g. hirem, hiv, loext
  end
end
class DEQUE[T]
  inherit QUEUE[T]
    export {ANY} hirem, hiv, loext end
end
```

Viszont ez a CAT-problémához (Changed Availability or Type) vezet.

Mit nevezünk *catcall*-nak? Bármilyen hívást egy B osztály f feature-ére, ami a C osztályban fordul elő *catcall*-nak nevezzük, akkor és csak is akkor, ha $D <: B$ esetén a következők közül bármelyik fennáll, ahol D nem a `NONE`.

- D megváltoztatja a láthatóságát f-nek úgy, hogy C ne érje el.
- D redeclares f such that the call's actual argument signature does not conform to the redeclared formal argument signature.
- The version of f in D has a formal argument of a type anchored either to `Current` or to a feature that D redeclares to a type to which the corresponding actual argument's type does not conform.

17. Milyen szerződés vonatkozik egy `retry` nélküli kivételkezelőre? Írd le Hoare-hármassal!

Egy Eiffel rutin végén állhat egy `rescue` klóz, mely akkor fut le, amikor a rutin törzsében kivétel lép fel. A `rescue` klóz célja, hogy a rutin úgy érhessen véget, hogy legalább az osztályinvariáns teljesüljön, ha már az utófeltételt nem sikerült elérni.

(A `retry` a `rescue` klózban a rutin újrakezdését kezdeményezi.) A szerződést (utófeltételt) teljesíteni nem tudó, és ezért kivételt kiváltó törzset követve lefut a `rescue`, majd a fellépett kivételt propagálja a hívó felé, azaz a hívóban a hívás helyszínén fellép a szóban forgó kivétel. A hívó a szerződést teljesíteni nem tudó objektumot konzisztens állapotban látja (az osztályinvariáns teljesül rá). Az r rutin $rescue_r$ `rescue` klózának szerződése tehát

$$\{Igaz\} rescue_r \{R_{rescue} \wedge inv_C\},$$

ahol inv_C az r osztályának, azaz a C-nek az egyesített invariánsa (a bázistípusokban és a C-ben deklarált invariánsok konjunkciója), R_{rescue} a rescue klóz utófeltétele.

Amennyiben egy rutin nem tartalmaz explicit `rescue` klózt, az `ANY`-ből megörökölt (és esetleg átnevezett) `default_rescue` eljárás lesz implicit módon a rutin `rescue` klóza, melynek `ANY`-beli implementációja üres (`SKIP` utasítás). Egy lehetőség a `rescue`-klózra egy creation-procedure végrehajtása (például a `default_rescue` és `default_create` ugyanolyan módon történő implementálása.).

18. Milyen szerződés köti az alosztályban felüldefiniált műveletet? Írd le Hoare-hármassal!

Lásd a 4. kérdést!

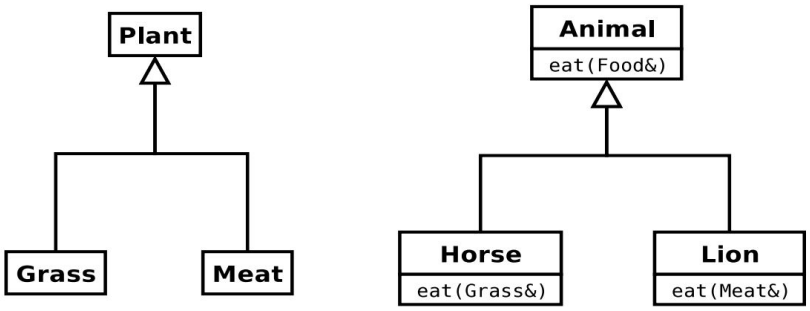
19. Mi a family polimorfizmus alapproblémája?

Lehet, hogy nem ezt tanultuk...

Gyakran szükségünk lehet rá, hogy ne csak osztályok típusparamétereire tehessünk megszorításokat, hanem esetleg az egyes osztályok típusparaméterei közötti összefüggést is leírhassuk. Ez azt jelenti, hogy osztályok egy csoportját (family) szeretnénk együttesen kezelni egy magasabb absztrakciós szinten. Ezen a magas szinten szeretnénk megadni bizonyos kapcsolataikat, majd egy alacsonyabb szinten lehetőséget kapni arra, hogy megadjuk a kombinációit azoknak az osztályokat, amik között az őseik közötti kapcsolatot megengedjük. Ezzel több osztály esetén azt biztosítjuk, hogy az ősosztályoknak csak az általunk megadott kombinációkban állhatnak a leszármazottjai kapcsolatban egymással.

A family polimorfizmus a megszokott polimorfizmus természetes kibővítése azáltal, hogy az öröklődést egy multi-object szintre emeli. Több osztályt egy családba tömörítünk, és ezen családok között határozunk meg polimorfizmust. Természetesen egy család minden tagját egyszerre kezeljük és nem keveredhetnek a családokon belüli osztályok, ha megtörténik, azt hibának tekintjük.

A könyebb érthetőség kedvéért vegyünk egy egyszerű példát. Tekintsünk egy állatkertet mindenfajta szokásos és exotikus állattal (lovak, oroszlánok, pingvinek, tevék, stb.). Minden állatot egy közös `Animal` osztályból érdemes leszármaztatni. Mivel az állatokat etetni kell, így szükséges lesz egy `Food` osztályra és az `Animal` osztály egy `eat(f : Food)` feature-ére. Az állatok különböző osztályai, mint a `Horses` és a `Lion` örökölnek az ős `Animal` osztályból. Emellett a `Food` osztályt is tovább specializálhatjuk, mint `Grass` és `Meat`.



A probléma `Animal` osztály `eat()` feature-nél keletkezik. Különböző állatok különböző táplálékot igényelnek. Az oroszlánok csak húst esznek, míg a lovak csak növényeket.

Különböző táplálék osztályok nem felcserélhetőek. A legtöbb objektumorientált nyelvben nem tudunk ilyesfajta megkötéseket tenni. (link)

20. Milyen esetekben áll fenn altípus reláció paraméterezett típusok között az Eiffelben? Mit jelent itt a `frozen`?

Eiffelben (általában) a típusparaméter is kovariáns ha `class STACK[T]...`, akkor

$$STACK[INTEGER] <: STACK[ANY],$$

de emiatt statikusan nem biztonságos a típusrendszer!

Legyen $D[U]$ és $C[T]$ két paraméterezett típus. $D[U] <: C[T]$ akkor és csakis akkor állhat fenn, ha $D <: C$ és $U <: T$. (link)

A `frozen` kulcsszót háromféleképpen használhatjuk. (link)

Ha egy osztály header-jében alkalmazzuk expliciten a `frozen` kulcsszót, akkor ebből az osztályból nem lehet leszármaztatni.

Ha egy feature deklarálásánál alkalmazzuk a `frozen` kulcsszót, akkor a feature-t nem lehet újradefiniálni egy leszármaztatott osztályban.

Ha egy paraméterezett típus definiálásánál a típusparaméterre alkalmazzuk a `frozen` kulcsszót, akkor a leszármaztatott paraméterezett típus csak ugyanazt a típus paramétert használhatja.

Eiffelben (általában) a típusparaméter is kovariáns kivéve, ha `frozen` a típusparaméter, mert akkor invariáns. Ha `class STACK[frozen T] ...`, akkor nem áll fenn a következő összefüggés

$$STACK[INTEGER] <: STACK[ANY].$$

21. Mit jelent az attached (az Eiffel szabványban „!”) és a detachable (a szabványban „?”)?

Referencia típusok esetén a nullreferencia kérdése, illetve a nullable típusok természetesen Eiffelben is felmerülnek. A nullreferencia Eiffelben `Void` névre hallgat. A nullable vagy non-nullable tulajdonságok egy változó típusának meghatározásánál kapnak szerepet. A nullable Eiffelben `detachable`, míg a non-nullable `attached` kulcsszavakkal fejezhető ki. (link)

```
my_attached_string: STRING
my_detachable_string: detachable STRING

...

my_attached_string := my_detachable_string      -- Invalid
my_detachable_string := my_attached_string      -- Valid
```

A fenti példában láthatjuk, hogy milyen módon feleltethetők meg egymásnak az `attached` és `detachable` változók, illetve, hogy alapvetően minden változó `attached`.

Ha egy változó deklarációjában `attached` szerepel, akkor a fordító megakadályozza, hogy `Void` értéket kapjon, vagy bármi olyan értéket, amit `Void`-ra lehet állítani. (link 8.11.1)

22. Milyen módon kezeli a kivételeket egy rutin, ha nincs benne rescue?

Eiffel-ben az `ANY` osztályban (minden felhasználói osztály ősosztályában) definiálva van egy `default_rescue` nevő metódus, melynek törzse alapértelmezésben csak egy üres utasítás. Ez az a feature, ami minden olyan esetben implicit módon meghívódik, amikor explicite nem adunk meg egy alprogram végén `rescue` részt, és az alprogram működése során mégis kivétel váltódik ki. Mivel minden felhasználói osztály az `ANY` class leszármazottja, ezért minden osztály felüldefiniálhatja ezt a metódust, saját igényei szerint. További öröklés esetén a felüldefiniált feature a leszármazottakra is érvényes lesz. Mivel Eiffel-ben a konstruktorok feladata az osztály-invariáns beállítása, ezért a `default_rescue`-t gyakran a paraméter nélküli konstruktor szinonímájaként szokták definiálni, hogy szervezett pánik esetén biztosítsa az invariáns visszaállítását. (link)

```
class C
creation
  make, -- Other creation procedures if any ...
inherit
  ANY
    redefine
      default_rescue
    end
feature
  make, default_rescue is
    -- No precondition
  do
    -- Appropriate implementation;
    -- Must ensure the invariant
  end -- make, default_rescue
  -- Other features ...
end -- class C
```

23. Mi a `default_create`?

Lehetőség van nem megadni `creation` klózt, ami azt jelenti, hogy a `default_create` alapvető creation procedure kerül meghívásra. A `default_create` az `ANY` osztályban kerül definiálásra, amit minden osztály örököl. Alapvetően egy üres utasítás.

Eiffel-ben az objektumok dinamikusan rendelődnek hozzá a szimbólumokhoz. Lehetőség van konstruktorok megadására is. A nyelv alaptípusaihoz létezik automatikus kezdőérték hozzárendelés. Lokális változók esetén a kezdőérték hozzárendelés a rutin végrehajtása előtt történik meg, attribútumok esetén pedig az objektum létrehozásakor a konstruktor eljárás végrehajtása előtt. A kezdőértékek az egyes típusok esetén a következők,

Típus	Érték
`INTEGER`	0
`REAL`, `DOUBLE`	0.0
`BOOLEAN`	`false`
`CHARACTER`	`null` karakter
`BIT_N`	csupa 0 bitsorozat
Minden referencia típus	`Void`

24. Hogyan szól a Liskov-féle helyettesítési elv (substitution principle)?

Minden osztály legyen helyettesíthető a leszármazott osztályával anélkül, hogy a program helyes működése megváltozna. Vagy pontosabban ha B altípusa A-nek, akkor minden olyan helyen ahol A-t felhasználjuk B-t is minden gond nélkül behelyettesíthetjük anélkül, hogy a programrész tulajdonságai megváltoznának.

Nézzünk egy egyszerű, ám annál híresebb példát: Kör-Ellipszis probléma:

```
class Shape { }

class Ellipse : Shape
{
  public double MajorAxis { get; set; }
  public double MinorAxis { get; set; }
}

class Circle : Ellipse
{

}
```

Az osztályhierarchiában a Kör az Ellipszisből származik, lévén annak egy speciális esete. Na most, ha a fenti állításokat figyelembe vesszük, akkor egy rendkívül érdekes problémával nézünk szembe: tegyük fel, hogy egy olyan függvényt készítünk, amely egy ellipszis tengelyeit is módosítja. Ugye fönt kimondtuk, hogy ilyenkor ezt a függvényt bármely Kör objektumra is meghívhatjuk és most jön a lényeg: mi történik, ha egy Kör objektum tengelyei (az Ellipszisből származik, szóval nincs szigorúan vett sugár) megváltoznak?

Azt bátran elmondhatjuk, hogy onnantól megszűnik Kör lenni, elméletileg legalábbis, de maga a Kör objektum – immár hibásan – még ugyanúgy létezik.

Az elvet a következő szabályokon keresztül lehet betartani:

- Az előfeltétel a leszármazott osztályokban gyengíthetjük
- Az utófeltétel a leszármazott osztályokban erősíthetjük

Nézzük meg, hogy ez hogy működik a gyakorlatban! Térjünk vissza az eredeti Kör-Ellipszis példára, az Ellipszis osztály egy ki nem mondott előfeltétele, hogy a tengelyeket tetszés szerint módosíthatjuk. Ha a Kör osztályban megmondjuk, hogy ezt talán mégsem szeretnénk, azzal erősítjük az eredeti előfeltételt, vagyis a Kör osztályt ilyen módon nem használhatjuk Ellipszisként.

25. Mi a kapcsolt (anchored) típus az Eiffelben? Mutass példát arra, hogy miért jó!

Lásd a 2. kérdést!

26. Milyen szerződésnek kell eleget tegyen egy creation procedure? Írd le Hoare-hármas segítségével!

Lehetőség van megadni `creation` klózt. Alapvetően a `creation` klóz feladata az osztályinvariáns megteremtése.

$$\{Q_m\} make \{R_m \wedge inv_C\}$$

ahol $make$ a creation procedure, inv_C az osztály invariánsa, Q_m és R_m a creation procedure elő- és utófeltétele.

27. Mire való az `only` az Eiffelben (illetve a `strip` a jelenlegi implementációkban)?

Az `only` klóz szintaxisa a következő,

$$Only \triangleq only[Feature\_list]$$

Az `only` klóz szintaxisából adódóan csak egy feature utókövetelményének legvégén szerepelhet.

A többi utókövetelmény azt határozza meg, hogy a feature hogyan változtathat meg pár specifikus tulajdonságot, de lehetőség van azt is megadni, hogy miket nem változtat meg (ezt keret problémának nevezik - frame problem). Be tudjuk szűkíteni a feature hatását azzal, ha megkötjük melyik tulajdonságokat nem változtathatja meg.

Egyik lehetőség, ha minden ilyen query-re definiálunk egy `q = old q` szabályt. Ez hosszú távon nem karbantartható, főleg, hogy a szülőosztályok megváltozásával újabb query-k kerülhetnek be.

Erre nyújt megoldást az `only` klóz, amivel meg lehet kötni, hogy egy feature melyik query-ket változtatja meg. Azok a query-k, amik nincsenek felsorolva a feature nem változtathatja meg. Ez a megkötés nem csak az adott verzióra vonatkozik, hanem minden leszármazottra is.

Az `only` klóz állhat üresen is, ekkor "tiszta" rutinról beszélhetünk, ami egyik query-t se változtatja meg.

Egy C osztálybeli `only` klóz akkor és csak akkor helyes, ha minden q_n querry, ami megjelenik a feature listáján teljesíti a következőket,

1. q_n csak egyszer fordul elő a feature listán
2. q_n a végleges neve a C egy q query-jének, argumentumok nélkül
3. Ha C újradeklarálja f-et egy B szülőtől, akkor q_n nem feature-e B-nek.

(link 8.9.11)

`only(x,y)` azt jelenti, hogy a `Current.x` és a `Current.y` megváltozhat a művelet hatására, de például a `Current.z` nem fog (ahol x, y és z az adott osztályban vagy bázisosztályaiban deklarált query-k). A jelenlegi implementációban ezt úgy írhatjuk, hogy `equals(strip(x,y), old strip (x,y))`, ahol a `strip` segítségével projektálhatjuk a `Currentet` a megnevezett mezőkre, az `old` segítségével pedig a műveletbe való belépés pillanatában érvényes értékre hivatkozhatunk. Az objektum belső állapotát meg nem változtató műveleteket only()-val jelölhetjük.

28. Mi a különbség a strukturális és a nominális altípusosság között?

Strukturális altípusosság esetén a felépítés alapján hasonlítunk össze két típust. Két típus egyenlő, ha ugyanaz a struktúrális felépítésük (pl. ugyanazokat a feature-öket tartalmazzák). Egy A típus altípusa B-nek, ha minden résztípusa altípusa a megfeleltetésének B-ben (pl. függvényeknél kontravariáns argumentumok). "Típusok felépítése szerinti induktív definíció". Ezzel olyan típusok is relációba kerülhetnek, amelyeknek "nem kellene".

Nominális altípusosság esetén a név alapján hasonlítunk össze két típust. Ilyen módszerrel két típus ugyanaz, ha a nevük megegyezik. Egy típus akkor altípusa a másiknak, ha ezt explicit kimondtuk (pl. `inherit`, `extends`, stb.).

29. Mit jelent a polymorphic CAT-call?

A CAT jelentése Changed Availability or Type.

CAT-call-nak nevezzük egy B osztály f feature-ére vonatkozó hívást a C osztályban, ha $D <: B$ esetén a következők közül bármelyik fennáll, ahol D nem a `NONE`.

- D megváltoztatja a láthatóságát f-nek úgy, hogy C ne érje el.

- D redeclares f such that the call's actual argument signature does not conform to the redeclared formal argument signature.
- The version of f in D has a formal argument of a type anchored either to `Current` or to a feature that D redeclares to a type to which the corresponding actual argument's type does not conform.

A CAT-call egy futási idejű próbálkozás arra vonatkozóan, hogy végrehajtsunk egy olyan hívást, ami nem hajtható végre az adott elemen.

A típus rendszer feladata, hogy biztosítsa, hogy egy helyes rendszer végrehajtása során soha ne történjen CAT-call.

A CAT egy rövidítés "Changed Availability or Type", ami két nyelvi mechanizmust kapcsol össze, melyeket ha a típus rendszer nem kezel megfelelően, CAT-call történhet.(link 8.25.1)

30. Mit értünk az alatt, ha egy objektumművelet esetén a paraméterek nonvariánsak? Milyen problémát vet fel?

Invariánsak vagy nonvariánsak a paraméterek, ha se nem kovariánsak se nem kontravariánsak.

31. Hogyan, milyen módokon örökölheti meg az A osztály a B osztály egy f feature-ét, ha C és D a B gyermeke, valamint A a C és a D gyermeke? Milyen eseteket különböztethetünk meg?

Ez egy többször öröklés egy őstől.

- Ha egyik osztályban sincs implementáció, akkor A-ban implementálható f.
- Ha C és D is implementálja f-et, de azonos módon, akkor `join`-olhatóak.
- Ha C és D is implementálja f-et, de eltérő módon, akkor `undefine` és `redefine` alkalmazhatóak.
- Ha C és D is implementálja f-et más-más néven, akkor mindkét feature-t megörököljük. Lehetséges, hogy szükség lesz `select`

32. Milyen lehetőségeket biztosít egy kovariáns sablonparaméter?

$List[D] <: List[B]$, ha $D <: B$. Innentől már ki lehet használni az altípusos polimorfizmust.

33. Mi a név szerepe, és miért fontos az átnevezés az Eiffelben?

A névvel tudunk azonosítani egy objektumot, feature-t.

Átnevezés szükséges lehet többszörös/ismételt öröklődés használatakor. Ennek segítségével elérhetővé válhat több azonos nevű, de eltérő implementációjú leszármaztatott feature, ha mindegyiket más nevet kap.

34. Mi a különbség a nyilvános és a privát öröklődés között?

Privát öröklődés esetén csak az adott osztály tudja használni a privátan megörökölt metódusokat. A megörökölt metódusok nem lesznek exportálva.

Publikus öröklődés esetén mindenki tudja hívni az osztályon a publikusan megörökölt metódusokat. Az öröklött metódusok export státusza az ősosztályban lévővel lesznek egyenlőek.

35. Milyen problémákat vet fel a többszörös öröklődés?

Az első legszembetűnőbb probléma a névütközés, melyre az egyik megoldást a 9. kérdés szolgáltatja. Másik lehetőség ha az `undefine` kulcsszóval töröljük az egyik őstől származó member-t.

A másik probléma amit felvet a többszörös öröklődés a gyémánt alakú öröklődés. Ennek feloldásában a mixinek segíthetnek, melyet a 3. kérdés tárgyal.

36. Mik az expandált típusok?

`expanded Class exp ...` egy expandált osztály definíció. Olyan típusok, amelyekből a létrehozott példányok nem egy objektumra mutató referenciák lesznek, hanem tényleges objektumok. Tehát bármelyik `exp` típusú változó az `exp` osztály egy példányát fogja tartalmazni. Szemben egy sima osztállyal, amikor a változó csak egy referenciát tartalmaz a tényleges objektumra. Ez megmutatkozik az értékadások esetén, vagy a paraméterátadásnál, ahol expandált típusok esetén másolás történik, nem expandáltaknál pedig csak referenciák átállítása.

37. Hogyan számoljuk ki egy osztály invariánsát? (. . . öröklődés . . .)

Az `invariant` klózban megfogalmazott feltételek illetve a szülőosztályok invariánsainak konjunkciójaként.

38. Mit értünk sekély, illetve mély másoláson?

A sekély másolás (shallow copy) célja, hogy minél kevesebbet másoljon. A referenciákat csak lemásolja, és nem duplikálja a HEAP-en lévő objektumokat. A másolt és az eredeti referencia aliasként működnek ugyanarra a HEAP területre.

A mély másolás (deep copy) lényege, hogy a referenciák által hivatkozott értékeket is lemásolják, és a másolt referenciák ezekre fognak mutatni.

39. Milyen lehetőségek vannak a referenciák nem-ürességének leírására az Eiffelben?

Referencia típusok esetén közvetlenül ráellenőrizhetünk, hogy az adott változó nem nullreferencia-e.

```
if x /= Void then
  -- Any other instructions here that do not assign to x
  x.f (a)
end
```

Referencia típusok esetén használhatjuk az `attached` kulcsszót.

```
if attached x as l_x then
  l_x.f(a)
end
```

A változó deklarációjánál kiköthetjük, hogy soha nem kaphat értékül nullreferenciát (fordítási idejű hibát okoz).

```
my_attached_string: attached STRING
```

40. Mi az F-bounded polimorfizmus? Mutass rá példát Javában!

A bounded polimorfizmust eredetileg a $System_{F<:}$ leírásánál tanulmányozták, de elérhető modern objektumorientált nyelvek (Java, C#, Scala) esetében, mint paraméteres polimorfizmus (template-ek).

Az F-bounded polimorfizmus vagy rekurzív bounded polimorfizmus sokkal pontosabb típusozásra ad lehetőséget rekurzív típusokat alkalmazó függvények esetében. Egy rekurzív típus definiál egy olyan függvényt, ami vagy paraméterében vagy visszatérési értékében hivatkozik a típusra.

Legyen adott egy `Moveable` osztály, melyet egy `moveMeOneInch` függvénnyel el lehet mozdítani egy inch-csel.

```
Moveable {
Moveable move(int x, int y)
}

Moveable moveMeOneInch(Moveable m) {...}
```

`moveMeOneInch` működni fog az `Car`, `Trains`, `Points` osztályokon és mindenen olyan osztályon, ami `Moveable`. Ezzel minden rendben lenne, de a visszatérési értéket cast-olni kell.

```
Car c = ...
Car movedCar = (Car) moveMeOneInch(c);
```

De Java 5 óta írhatjuk a következő F-bounded polimorfizmust.

```
<T extends Moveable> T moveMeOneInch(T m) {...}

Car c = ...
Airplane p = ...

Car movedCar = moveMeOneInch(c);
Airplane movedAirplane = moveMeOneInch(p);
```

Itt már típus helyes lesz a visszatérési érték.

41. Mit jelent a "frame condition" kifejezés? Hogyan írunk ilyet az Eiffelben?

Megszorítások a változókra. Ez az `only` vagy a `strip` segítségével tehető meg.

42. Milyen kifejezések szerepelhetnek értékadás baloldalán az Eiffelben?

Csak írható entitások (változók) állhatnak egy értékadás baloldalán. Ez által a függvényparaméterek például nem szerepelhetnek értékadás bal oldalán, se az osztály memberjei sem. `obj.some_attribute := some_value` nem valid utasítás.

A member-eket csak metódusokkal vagy úgy nevezett *assigner command* definiálásával lehet módosítani. *Assigner command* használatával a fenti értékadás már lehetséges.

```
some_attribute: SOME_TYPE assign set_some_attribute
```

Ezzel automatikusan definiálunk egy setter-t a member-hez.

43. Mire jó a `debug` utasítás az Eiffelben?

A `debug` szintaxisa a következő,

$$Debug \triangleq debug["("Key\_list")"]Compound\ end$$

Az Eiffel támogatja a debug-olást. Ehhez megadhatjuk globálisan vagy egyesével, hogy mely `Key_list`-beli elemek legyenek aktívak, vagyis az ezekre vonatkozó `Compaund` klóz végrehajtódnak.

```
debug("MY_DEBUG_FLAG")
  -- Debug code is here.
end
```

A fenti kódrészlet megfelel a C-ben megszokott kódrészleteknek.

```
#ifdef MY_DEBUG_FLAG
  /* Debug code is here */
#endif
```

44. Hogyan történik a paraméterátadás az Eiffelben? Térj ki az expandált típusokra is!

Általános esetben, nem expandált típusok esetén referencia szerinti paraméterátadás történik, azaz nem másolódik maga a bemeneti paraméter, csak egy referenciát kapunk rá, de ezt nem állíthatjuk át. Viszont műveleteket hívhatunk a paraméterként kapott objektumon.

Expandált típusok esetében, mivel ezek stack-allokált objektumok, így amikor egy ilyen típust adunk át, akkor érték szerinti paraméterátadás történik, azaz másolódik.

45. Milyen elvárások fogalmazhatók meg egy egyenlőségvizsgálat műveletre?

Matematikai értelemben egy egyenlőség reláció reflexív (1), szimmetrikus (2), tranzitív (3)

$$x = x \tag{1}$$

$$x = y \implies y = x \tag{2}$$

$$x = y \land y = z \implies x = z \tag{3}$$

Ha A és B különböző típusú, akkor nem lehetnek egyenlőek.

Ha A és B megegyező típusúak, akkor implementáció függő lehet az egyenlőség. A strukturális egyenlőség azt várja el, hogy a két objektum ugyanolyan tartalmú legyen. A referenciális egyenlőségnek csak annyi a feltétele, hogy azonos elemre mutasson a két referencia.

46. Mi a különbség egy exportált és egy nem exportált feature szerződésében? Mutasd be Hoare-hármas segítségével!

Egy exportált r feature esetében az osztálynak kívülről konzisztensnek kell maradnia, vagyis az osztály invariánst meg kell tartani

$$\{Q_r \land inv_C\} r \{R_r \land inv_C\},$$

ahol Q_r és R_r az r feature elő- és utófeltétele, míg inv_C az osztályinvariáns.

Egy nem exportált r feature nem elérhető az osztályon kívülről, így ennek nem kell az invariánst megtartania.

$$\{Q_r\} r \{R_r\},$$

ahol Q_r és R_r az r feature elő- és utófeltétele.

Érdemes figyelni, hogy egy exportált metódus nem feltétlenül használható rögtön nem-exportáltként, hiszen a meghívása előtt az osztályinvariánst garantálni kell, míg nem-exportált esetben ez nem szükséges.

47. Mit értünk multimethodon?

Bizonyos objektumorientált nyelvekben a dinamikus kötés nem egyetlen kitüntetett paraméter dinamikus típusán működik, hanem az összes paraméter dinamikus típusát figyelembe veszi. Például az Eiffelben az `a.b(c)` rutinhívásnál az `a` és a `c` változók statikus típusa alapján dől el, hogy a `b` rutin hívása értelmes-e, és az `a` változó dinamikus típusa alapján dől el, hogy a művelet mely osztályban definiált implementációja hajtódik végre. Egy *multimethodokat* (azaz multiple dispatchet) támogató nyelvben az `a` és a `c` dinamikus típusa alapján választódik ki a végrehajtandó kód (azaz az a statikus típusában, illetve leszármazottaiban több olyan `b` művelet is lehet, amelynek a paramétere a `c` statikus típusának valamilyen altípusába tartozik).

A multimethodokat támogató nyelvekben az úgynevezett bináris műveletek (azaz amikor a művelet paraméterének típusa a fogadó objektum típusával kell megegyezzen, mint mondjuk egyenlőségvizsgálatnál) megvalósítása megoldott, hiszen minden osztályban megírhatjuk az arra az osztályra jellemző (és a fogadó és paraméter objektumot ugyanazzal típusozó) implementációt, és a dispatch során a legjobban illeszkedő implementáció hívódik meg. A szimmetriáról a multiple dispatch gondoskodik.

48. Mik a virtuális típusok? Mire jók?

Alternatíva a típusparaméterekre. Az ősosztályban a típusparaméterek, mint tagok vannak deklarálva, és az absztrakt alosztályokban lehet specializálni őket. A konkrét alosztályban meg kell mondani, hogy ezek a típusok mit jelentenek.

49. Mit nevezünk parametrikus polimorfizmusnak?

Általános viselkedés leírása típusparaméterek segítségével. Így az általános kód különböző konkrét típusokra is képes a működésre.

```
class List<A> {   ...   }
List<Integer> integerList;
List<String>  stringList ;
```

50. Mit nevezünk altípusos polimorfizmusnak?

A Liskov-féle helyettesítési elv szerint, ha $B <: A$, akkor az B típusú objektumok használhatóak úgy, mintha A típusúak lennének.

```
class Animal {}
class Dog extends Animal {}
class Cat extends Animal {}

void foo(Animal a) {}

foo(new Dog())
foo(new Cat())
foo(new Animal())
```