Algoritmos y Estructuras de Datos II

Práctica 5 – Diseño: invariante de representación y función de abstracción

Notas preliminares

- Para esta práctica vamos a utilizar versiones acotadas de los TADs básicos (cola, diccionario, conjunto, etc). En el anexo I se describen estos TADs.

Ejercicio 1

Para implementar un diccionario acotado (ver anexo I), se propone utilizar un array de tuplas, `Array <tupla <K, V> >`. Escriba el invariante de representación y la función de abstracción correspondientes.

Ejercicio 2

Para mejorar el rendimiento de un modulo que implementa el TAD `ConjuntoAcotado` con Arrays, se le agregó a la estructura un arreglo de booleanos que indica si una posición está siendo usada o no. De esta manera, para eliminar un elemento del conjunto sólo basta marcarla como borrada en dicho arreglo, sin necesidad de eliminarlo efectivamente del arreglo de datos.

```
Modulo ConjAcotadoImpl<T> implenenta ConjAcotado<T> {
    var elems: Array<T>
    var enUso: Array<bool>
}
```

Se pide:

- Escribir el invariante de representación y la función de abstracción
- Escribir los algoritmos para las operaciones `definir` y `obtener` con sus correspondientes pre y postcondición de implementación.

Ejercicio 3

Se quiere implementar el TAD cola acotada, que es una cola que puede contener a lo sumo n elementos.

Como estructura de representación se propone utilizar un *buffer circular*. Esta estructura almacena los k elementos de la cola en posiciones contiguas de un arreglo, aunque no necesariamente en las primeras k posiciones. Para ello, se utilizan dos índices que indican en qué posición empieza y en qué posición termina la cola. Los nuevos elementos se encolan "a continuación" de los actuales tomando módulo n, es decir, si el último elemento de la cola se encuentra en la última posición del arreglo, el próximo elemento a encolar se ubicará en la primera posición del arreglo. Notar que para desencolar el primer elemento de la cola, simplemente se avanza el índice que indica dónde empieza la cola (eventualmente volviendo éste a la primera posición del arreglo). En nuestra implementación, además de esto, se pone en cero la posición que se acaba de liberar. Se propone la siguiente estructura de representación.

```
modulo ColaAcotadaImp<T> implementa ColaAcotada<T> {
    var inicio: int
    var fin: int
    var elems: array<T>
}
```

Se pide:

a) Definir el invariante de representación y la función de abstracción.
b) Escribir todos los algoritmos, con su pre y postcondición de implementación.
c) Por qué tiene sentido utilizar un buffer circular para una cola y no para una pila?

Ejercicio 4

Considere la siguiente especificación de una relación uno/muchos entre alarmas y sensores de una planta industrial: un sensor puede estar asociado a muchas alarmas, y una alarma puede tener muchos sensores asociados.

```
TAD Planta {
    obs alarmas: conj<Alarma>
    obs sensores: conj<tupla<Sensor, Alarma>>

    proc nuevaPlanta(): Planta
        asegura res.alarmas == {}
        asegura res.sensores == {}

    proc agregarAlarma(input p: Planta, in a: Alarma)
        requiere !(a in p.alarmas)
        asegura p.alarmas == old(p).alarmas + {a}
        asegura p.sensores == old(p).sensores

    proc agregarSensor(inout p: Planta, in a: Alarma, in s: Sensor)
        requiere a in p.alarmas
        requiere !(<s,a> in p.sensores)
        asegura p.alarmas == old(p).alarmas
        asegura p.sensores == old(p.sensores) + {<s,a>}
}
```

Se decidió utilizar la siguiente estructura como representación, que permite consultar fácilmente tanto en una dirección (sensores de una alarma) como en la contraria (alarmas de un sensor).

```
modulo PlantaImpl implementa Planta {
    var alarmas: Diccionario<Alarma, Conjunto<Sensor>>
    var Sensores: Diccionario<Sensor, Conjunto<Alarma>>
}
```

Se pide:

a) Escribir formalmente y en castellano el invariante de representación.
b) Escribir la función de abstracción.

Ejercicio 5 *(Planilla de actividades justificacion)*

Un consultor independiente desea mantener una planilla con las actividades que realiza cada mes en cada uno de los proyectos en los que participa. La planilla que desea mantener se describe con el siguiente TAD.

```
TAD Planilla {
    obs actividades: conj<Actividad>
    obs proyectoDe: dict<Actividad, Proyecto>
    obs mesDe: dict<Actividad, int>
    obs horaDe: dict<Actividad, int>

    proc nuevaPlanilla(): Planilla

    proc totProyxMes(in p: Planilla, in m: Mes, in r: Proyecto): int

    proc agregar(
        inout p: Planilla,
        in a: Actividad,
        in r: Proyecto,
        in mes: int,
        in horas: int
    )
}

Actividad es string
Proyecto es string
```

Se propone la siguiente estructura para representar dicho TAD

```
        modulo PlanillaImpl implementa Planilla {
            var detalle: Diccionario<
                Actividad, struct<proy: Proyecto, mes: int, horas: int>
            >
            var horasPorMes: Diccionario<proyecto, Array<int>>
        }

        Actividad es string
        Proyecto es string
```

Se pide:

a) Escribir formalmente y en castellano el invariante de representación.
b) Escribir la función de abstracción.

Ejercicio 6 *(Alta fiesta)* ★

El salón Alta Fiesta se encuentra últimamente en dificultades para coordinar el desarrollo de las reuniones que se realizan en sus facilidades. Hoy en día las fiestas se desarrollan de una manera muy particular. Los invitados llegan en grupos, por lo general numerosos, aunque también a veces de una sola persona. Que un invitado llegue sin un regalo está considerado una falta grave a las buenas costumbres. Tanto es así que a dichos individuos no se les permite el ingreso a las fiestas. Lo que sí se permite es que los invitados hagan regalos en grupo: los invitados que llegan en grupo traen un único regalo de parte de todos. Como es habitual, sólo se permite la entrada a la fiesta a aquellas personas que han sido invitadas.

Al ingresar un grupo de invitados a la fiesta, éste se identifica con un nombre: por ejemplo “Los amigos de la secundaria” o “La familia de la novia”. Este nombre se usa por ejemplo para los juegos grupales que van a

hacer los animadores durante la fiesta. Igualmente, dado que el comportamiento de las personas en masa no siempre es civilizado, se quiere poder saber de manera eficiente cuál es el nombre del grupo más numeroso para poder seguirle el rastro.

Además, se desea tener un registro de todos los regalos, junto con el grupo de personas que lo hicieron, y se desea conocer en todo momento qué personas se encuentran ya en la fiesta.

Se nos ha encomendado realizar un sistema que permite llevar el control del estado de la fiesta en un momento dado. La cátedra ha realizado la especificación del sistema y ha armado una estructura de representación para el diseño del mismo.

```
TAD AltaFiesta {
    obs invitados: conj<Persona>
    obs pendientes: conj<Persona>
    obs grupoDe: dict<Persona, Grupo>
    obs regaloDe: dict<Persona, Regalo>

    proc iniciarFiesta(in personas: Conjunto<Persona>): AltaFiesta

    proc lleganInvitados(
        inout a: AltaFiesta,
        in c: Conjunto<Persona>,
        in g: Grupo,
        in r: Regalo
    )
}
```

```
modulo AltaFiestaImpl implementa AltaFiesta {
    var invitados: Conjunto<Persona>
    var presentes: Conjunto<Persona>
    var grupoDe: Diccionario<Grupo, Conjunto<Persona>>
    var regaloDeGrupo: Diccionario<Grupo, Regalo>
}

Persona es string
Grupo es string
Regalo es string
```

Informalmente, esta representación cumple las siguiente propiedades:

- En *invitados* están todos los invitados a la fiesta, incluyendo también a aquellos que ya llegaron.
- En *presentes* están los invitados que ya llegaron a la fiesta.
- En *grupoDe* se encuentra, para cada identificador de grupo i, las personas que al llegar agrupadas se identificaron como i.
- En *regaloDeGrupo* se encuentra qué regalo trajo cada grupo.

Se pide:

a) Realizar el invariante de representación del módulo. Escribirlo en lenguaje formal y en castellano.
b) Escribir la función de abstracción. De considerarse necesario, explicarla en castellano.
c) Escribir el algoritmo de la función llegaGrupo marcando claramente los puntos de su programa en que alguna parte del invariante de representación se rompe, indicando a qué parte se refiere, y también aquellos puntos donde éste se reestablece.

Ejercicio 7

Considerar el siguiente TAD que modela el comportamiento de una oficina del Estado que procesa trámites. Cada trámite está identificado por un ID y se le asigna una categoría al momento de ingresar. Las categorías de la oficina no son fijas, y pueden agregarse nuevas categorías en cualquier momento. En cualquier momento se puede dar prioridad a una categoría. Todos los trámites pendientes que pertenecen a una categoría prioritaria se consideran prioritarios (Notar que en esta oficina, como buena dependencia estatal, un trámite nunca concluye):

```
TAD Oficina {
    obs categorias: conj<Categoria>
    obs tramitesPend: seq<Id>
    obs catsPrioritarias: conj<Categoria>
    obs categoriaDe: dict<Id, Categoria>

    proc nuevaOficina(): Oficina

    proc agregaCategoria(inout o: Oficina, in c: Categoria)
        asegura o.categorias == old(o).categorias + {c}
        asegura o.tramitesPend == old(o).tramitesPend
        asegura o.catsPrioritarias == old(o).catsPrioritarias

    proc agregaTramite(inout o: Oficina, in i: Id, in c: Categoria)
        requiere c in o.categorias && i not in o.tramitesPend
        asegura o.categorias == old(o).categorias
        asegura o.tramitesPend == old(o).tramitesPend + {i}
        asegura o.categoriaDe == setKey(old(o).categoriaDe, i, c)

    proc priorizaCategoria(inout o: Oficina, in c: Categoria)
        requiere c in o.categorias
        asegura o.catsPrioritarias == old(o).catsPrioritarias + {c}
        asegura o.categorias == old(o).categorias
        asegura o.tramitesPend == old(o).tramitesPend
        asegura o.categoriaDe == old(o).categoriaDe

    proc pendientesPrioritarios(in o: Oficina): seq<Id>
        asegura forall i: Id ::
            i in res <==> i in o.categoriaDe && o.categoriaDe[i] in o.catsPrioritarias
}
```

Se decidió utilizar la siguiente estructura como representación:

```
        modulo OficinaImpl implementa Oficina {
            var catPrioritarias: Conjunto<Categoria>
            var tramites: Diccionario<Id, Categoria>
            var tramCat: Diccionario<Categoria, Conjunto<Id>>
            var pendPrioritarios: Conjunto<Id>
            var pendientes: Conjunto<Id>
        }
```

Informalmente, *catPrioritarias* representa el conjunto de todas las categorías a las que se ha dado prioridad, *tramites* asocia a cada trámite su categoría mientras que *tramCat* describe todos los trámites asociados a cada

categoría. *pendPrioritarios* contiene la secuencia de trámites pendientes que tienen una categoría prioritaria mientras que *pendientes* contiene todos los trámites pendientes (incluso a los prioritarios).

a) Escribir en castellano y formalmente el invariante de representación.
b) Escribir formalmente la función de abstracción.

Ejercicio 8

(Colegio secundario). Dado el siguiente TAD:

```
TAD Estudiante ES int

TAD Secundario {
    obs estudiantes: conj<Estudiante>
    obs faltas: dict<Estudiante, int>
    obs notas: dict<Estudiante, seq<int> >

    proc NuevoSecundario(in es: Conjunto<Estudiante>): Secundario
        requiere |es| > 0
        asegura res.estudiantes == es
        asegura forall e: Estudiante :: e in es ==>L
            (e in res.faltas &&L res.faltas[e] == 0)
        asegura forall e: Estudiante :: e in es ==>L
            (e in res.notas &&L res.notas[e] == [])

    proc RegistrarNota(inout s: Secundario, in e: Estudiante, in nota: int)
        requiere e in s.estudiantes
        requiere 0 <= nota <= 10
        asegura s.estudiantes == old(s).estudiantes
        asegura s.faltas == old(s).faltas
        asegura s.notas == setKey(old(s).notas, e, old(s).notas[e] + [nota])

    proc RegistrarFalta(inout s: Secundario, in e: Estudiante)
        requiere e in s.estudiantes
        asegura s.alumnos == old(s).alumnos
        asegura s.faltas == setKey(old(s).faltas, e, old(s).faltas[e] + 1)
        asegura s.notas == old(s).notas
}
```

Se propone la siguiente estructura de representación:

```
        modulo SecundarioImpl implementa Secundario {
            var estudiantes: Conjunto<Estudiante>
            var faltas: Diccionario<Estudiante, int>
            var notas: Array< Conj<Estudiante> >
            var notasPorEstudiante: Diccionario<Estudiante, Array<int> >
        }
```

Donde:

- En *estudiantes* están todos los estudiantes del colegio secundario
- En *faltas* tenemos para estudiante la cantidad de faltas que tiene hasta el momento
- En *notas* tenemos en la posición i-ésima a los estudiantes que tienen nota i

- En *notasPorEstudiante* tenemos para cada estudiante la cantidad de notas con valor i-ésimo tienen

Se pide:

1. Escribir en castellano y formalmente el invariante de representación
2. Escribir la función de abstracción

Ejercicio 9

(Fila del banco). Se decidió diseñar el módulo para representar a la fila de un banco introducida en la guía de especificación de tipos abstractos de datos. Presentamos una posible especificación del TAD:

```
TAD Fila {
    obs fila: seq<Persona>
    obs atendidos: conj<Persona>
    obs colados: conj<Persona>
    obs retirados: conj<Persona>

    proc AbrirVentanilla(): Fila
        asegura res.fila = []
        asegura res.atendidos = {}
        asegura res.colados = {}
        asegura res.retirados = {}

    proc Llegar(inout f: Fila, in p: Persona)
        requiere p not in f.fila
        asegura old(f).fila = tail(f.fila)
        asegura p = head(f.fila)
        asegura f.atendidos = old(f).atendidos
        asegura f.colados = old(f).colados
        asegura f.retirados = old(f).retirados

    proc Atender(inout f: Fila)
        requiere exists p: Persona :: p in f.fila
        asegura f.fila = tail(old(f).fila)
        asegura f.atentidos = old(f).atendidos + {p}
        asegura f.colados = old(f).colados - {p}
        asegura f.retirados = old(f).retirados

    proc Esperando?(in f: Fila, in p: Persona): bool
        asegura res == true <==> p in f.fila

    proc Vacía?(in f: Fila): bool
        asegura res == true <==> |f.fila| == 0

    proc Posición(in f: Fila, in p: Persona): int
        requiere p in f.fila
        asegura res == posición(f.fila, p)

    proc Longitud(in f: Fila): int
        asegura res == longitud(f.fila)

    proc Retirarse(inout f: Fila, in p: Persona)
        requiere p in f.fila
        asegura f.fila = old(f).fila[0..posicion(old(f).fila, q)-1] +
                         old(f).fila[posicion(old(f).fila, q)+1..|old(res).fila|-1]
```

```
        asegura f.atentidos = old(f).atendidos
        asegura f.colados = old(f).colados - {p}
        asegura f.retirados = old(f).retirados + {p}

    proc Colarse(inout f: Fila, in p: Persona, in q: Persona)
        requiere p not in fila
        requiere q in fila
        asegura f.fila = old(f).fila[0..posicion(old(f).fila, q)-1] + [p] +
                         old(f).fila[posicion(old(f).fila, q)..|old(res).fila|]
        asegura f.atentidos = old(f).atendidos
        asegura f.colados = old(f).colados + {p}
        asegura f.retirados = old(f).retirados

    proc SeColó?(in f: Fila, in p: Persona): bool
        asegura res == true <==> p in f.colados

    proc Entró?(in f: Fila, in p: Persona): bool
        asegura res == true <==> (p in f.fila || p in (f.atendidos + f.retirados))

    proc FueAtendido?(in f: Fila, in p: Persona): bool
        asegura res == true <==> p in f.atendidos

    aux posición(ps: seq<Persona>, p: Persona): int
        sum : 0 <= i < |ps| :: if p = ps[i] then i else 0 fi
}
```

Se propone la siguiente estructura de representación:

```
modulo FilaImpl implementa Fila {
    var entraron: Conjunto<Persona>
    var fila: Cola<Persona>
    var colados: Conjunto<Persona>
    var atendidos: Conjunto<Persona>
}
```

Donde:

- *Entraron* es un conjunto con todas las personas que alguna vez estuvieron en la fila.
- *Colados* son las personas que están actualmente en la fila y se colaron al llegar.
- *Atendidos* son las personas que fueron atendidas en el banco.

Se pide:

1. Escribir en castellano y formalmente el invariante de representación
2. Escribir la función de abstracción

1. Anexo: TADs acotados

```
TAD ConjuntoAcotado<T> {
    obs elems: conj<T>
    obs cap: int

    proc conjVacio(in c: int): Conjunto<T>
        asegura res.cap == c && res.elems == {}

    proc pertenece(in c: Conjunto<T>, in T e): bool
        asegura res == true <==> e in c.elems

    proc agregar(input c: Conjunto<T>, in e: T)
        requiere |c.elems| < c.cap
        asegura c.elems == old(c).elems + {e}

    proc sacar(inout c: Conjunto<T>, in e: T)
        asegura c.elems == old(c).elems - {e}

    proc unir(inout c: Conjunto<T>, in c': Conjunto<T>)
        requiere |c.elems| + |c'.elems| <= c.cap
        asegura c.elems == old(c).elems + c'.elems

    proc restar(inout c: Conjunto<T>, in c': Conjunto<T>)
        asegura c.elems == old(c).elems - c'.elems
}
```

```
TAD DiccionarioAcotado<K, V> {
    obs data: dict<K, V>
    obs cap: int

    proc diccionarioVacio(in c: int): Diccionario<K, V>
        asegura res.cap == c && res.data == {}

    proc esta(d: Diccionario<K, V>, k: K): bool
        asegura res == true <==> k in d.data

    proc definir(inout d: Diccionario<K, V>, in k: K k, in v: V)
        requiere |d.data| < d.cap
        asegura d == setKey(old(d), k, v)

    proc obtener(in d: Diccionario<K, V>, in k: K): V
        requiere k in d.data
        asegura res == d.data[k]

    proc borrar(inout d: Diccionario<K, V>, in k: K)
        requiere k in d.data
        asegura d == delKey(old(d), k)
}
```

```
TAD ColaAcotada<T> {
    obs s: seq<T>
    obs cap: int
```

```
    proc colaVacía(in c: int): Cola<T>
        asegura res.cap == c && res.s == []

    proc vacia(in c: Cola<T>): bool
        asegura res == true <==> c.s == []

    proc encolar(inout c: Cola<T>, in e: T)
        requiere |c.s| < c.cap
        asegura c.s == old(c).s + [e]

    proc desencolar(inout c: Cola<T>): T
        requiere c.s != []
        asegura c.s == old(c).s[1..|old(c).s|]
        asegura res == old(c)[0]
}
```