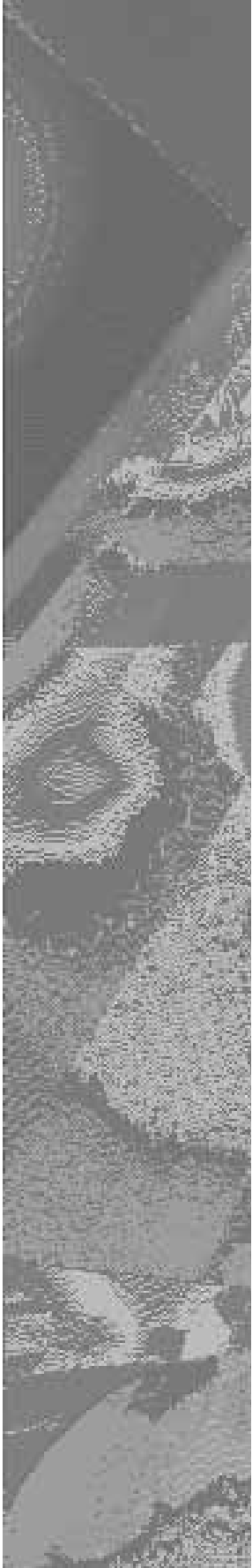
Dragan Milićev, Marko Zarić, Nebojša Pirocanac

Objektno orijentisano modelovanje na jeziku UML

Praktikum

Dragan Milićev

Primeri



Glava 1

Uvod u objektno modelovanje

Apstraktni tipovi podataka

Kompletan kôd za projektovanu biblioteku apstraktnih tipova podataka dat je u nastavku.

- Datoteka `unbound.h`:

```
// Project:  Abstract Data Types
// Module:   Unbound
// File:     unbound.h
// Date:     3.11.1996.
// Author:   Dragan Milicev
// Contents:
//      Class templates: Unbounded
//                       IteratorUnbounded

#ifndef _UNBOUND_
#define _UNBOUND_

////////////////////////////////////////////////////////////////////
// class template Unbounded
////////////////////////////////////////////////////////////////////

template <class T>
class Unbounded {
public:

  Unbounded ();
  Unbounded (const Unbounded<T>&);
 ~Unbounded ();

  Unbounded<T>& operator= (const Unbounded<T>&);

  void append (const T&);
  void insert (const T&, int at=0);
  void remove (const T&);
  void remove (int at=0);
  void clear  ();

  int      isEmpty  () const;
  int      isFull   () const;
  int      length   () const;
  const T& first    () const;
  const T& last     () const;
  const T& itemAt   (int at) const;
  T&       itemAt   (int at);
  int      location (const T&) const;

protected:

  void copy (const Unbounded<T>&);
private:

  friend class  IteratorUnbounded<T>;
  friend struct Element<T>;
```

```
  void remove (Element<T>*);

  Element<T>* head;
  int size;

};



template <class T>
struct Element {
  T t;
  Element<T> *prev, *next;
  Element (const T&);
  Element (const T&, Element<T>* next);
  Element (const T&, Element<T>* prev, Element<T>* next);
};



template<class T>
Element<T>::Element (const T& e) : t(e), prev(0), next(0) {}


template<class T>
Element<T>::Element (const T& e, Element<T> *n)
    : t(e), prev(0), next(n) {
  if (n!=0) n->prev=this;
}


template<class T>
Element<T>::Element (const T& e, Element<T> *p, Element<T> *n)
    : t(e), prev(p), next(n) {
  if (n!=0) n->prev=this;
  if (p!=0) p->next=this;
}


template<class T>
void Unbounded<T>::remove (Element<T>* e) {
  if (e==0) return;
  if (e->next!=0) e->next->prev=e->prev;
  if (e->prev!=0) e->prev->next=e->next;
  else head=e->next;
  delete e;
  size--;
}



template<class T>
void Unbounded<T>::copy (const Unbounded<T>& r) {
  size=0;
  for (Element<T>* cur=r.head; cur!=0; cur=cur->next) append(cur->t);
}


template<class T>
void Unbounded<T>::clear () {
  for (Element<T> *cur=head, *temp=0; cur!=0; cur=temp) {
    temp=cur->next;
    delete cur;
  }
  head=0;
  size=0;
}
```

```
template<class T>
int Unbounded<T>::isEmpty () const { return size==0; }


template<class T>
int Unbounded<T>::isFull () const { return 0; }



template<class T>
int Unbounded<T>::length () const { return size; }


template<class T>
const T& Unbounded<T>::first () const { return itemAt(0); }


template<class T>
const T& Unbounded<T>::last () const { return itemAt(length()-1); }


template<class T>
const T& Unbounded<T>::itemAt (int at) const {
  static T except;
  if (isEmpty()) return except;  // Exception!
  if (at>=length()) at=length()-1;
  if (at<0) at=0;
  int i=0;
  for (Element<T> *cur=head; i<at; cur=cur->next, i++);
  return cur->t;
}


template<class T>
T& Unbounded<T>::itemAt (int at) {
  static T except;
  if (isEmpty()) return except;  // Exception!
  if (at>=length()) at=length()-1;
  if (at<0) at=0;
  int i=0;
  for (Element<T> *cur=head; i<at; cur=cur->next, i++);
  return cur->t;
}


template<class T>
int Unbounded<T>::location (const T& e) const {
  int i=0;
  for (Element<T> *cur=head; cur!=0; cur=cur->next, i++)
    if (cur->t==e) return i;
  return -1;
}


template<class T>
void Unbounded<T>::append (const T& t) {
  if (head==0) head=new Element<T>(t);
  else {
    for (Element<T> *cur=head; cur->next!=0; cur=cur->next);
    new Element<T>(t,cur,0);
  }
  size++;
}
```

```
template<class T>
void Unbounded<T>::insert (const T& t, int at) {
  if ((at>size)||(at<0)) return;
  if (at==0) head=new Element<T>(t,head);
  else if (at==size) { append(t); return; }
  else {
    int i=0;
    for (Element<T> *cur=head; i<at; cur=cur->next, i++);
    new Element<T>(t,cur->prev,cur);
  }
  size++;
}


template<class T>
void Unbounded<T>::remove (int at) {
  if ((at>=size)||(at<0)) return;
  int i=0;
  for (Element<T> *cur=head; i<at; cur=cur->next, i++);
  remove(cur);
}


template<class T>
void Unbounded<T>::remove (const T& t) {
  remove(location(t));
}


template<class T>
Unbounded<T>::Unbounded () : size(0), head(0) {}


template<class T>
Unbounded<T>::Unbounded (const Unbounded<T>& r) : size(0), head(0) {
  copy(r);
}


template<class T>
Unbounded<T>& Unbounded<T>::operator= (const Unbounded<T>& r) {
  clear();
  copy(r);
  return *this;
}

template<class T>
Unbounded<T>::~Unbounded () { clear(); }

////////////////////////////////////////////////////////////////////
// class template IteratorUnbounded
////////////////////////////////////////////////////////////////////

template <class T>
class IteratorUnbounded {
public:

  IteratorUnbounded (const Unbounded<T>*);
  IteratorUnbounded (const IteratorUnbounded<T>&);
 ~IteratorUnbounded ();

  IteratorUnbounded<T>& operator= (const IteratorUnbounded<T>&);
  int operator== (const IteratorUnbounded<T>&);
  int operator!= (const IteratorUnbounded<T>&);

  void reset();
  int  next ();

  int      isDone() const;
  const T* currentItem() const;
```

```
private:

  const Unbounded<T>* theSupplier;
  Element<T>* cur;

};


template <class T>
IteratorUnbounded<T>::IteratorUnbounded (const Unbounded<T>* ub)
  : theSupplier(ub), cur(theSupplier->head) {}


template <class T>
IteratorUnbounded<T>::IteratorUnbounded (const IteratorUnbounded<T>& r)
  : theSupplier(r.theSupplier), cur(r.cur) {}


template <class T>
IteratorUnbounded<T>& IteratorUnbounded<T>::operator= (const
IteratorUnbounded<T>& r) {
  theSupplier=r.theSupplier;
  cur=r.cur;
  return *this;
}


template <class T>
IteratorUnbounded<T>::~IteratorUnbounded () {}


template <class T>
int IteratorUnbounded<T>::operator== (const IteratorUnbounded<T>& r) {
  return (theSupplier==r.theSupplier)&&(cur==r.cur);
}


template <class T>
int IteratorUnbounded<T>::operator!= (const IteratorUnbounded<T>& r) {
  return !(*this==r);
}


template <class T>
void IteratorUnbounded<T>::reset () {
  cur=theSupplier->head;
}


template <class T>
int IteratorUnbounded<T>::next () {
  if (cur!=0) cur=cur->next;
  return !isDone();
}


template <class T>
int IteratorUnbounded<T>::isDone () const {
  return (cur==0);
}


template <class T>
const T* IteratorUnbounded<T>::currentItem () const {
  if (isDone()) return 0;
  else return &(cur->t);
}


#endif
```

- Datoteka `bound.h`:

```
// Project:  Abstract Data Types
// Module:   Bound
// File:     bound.h
// Date:     3.11.1996.
// Author:   Dragan Milicev
// Contents:
//      Class templates: Bounded
//                       IteratorBounded

#ifndef _BOUND_
#define _BOUND_

////////////////////////////////////////////////////////////////////
// class template Bounded
////////////////////////////////////////////////////////////////////

template <class T, int N>
class Bounded {
public:

  Bounded ();
  Bounded (const Bounded<T,N>&);
 ~Bounded ();

  Bounded<T,N>& operator= (const Bounded<T,N>&);

  void append (const T&);
  void insert (const T&, int at=0);
  void remove (const T&);
  void remove (int at=0);
  void clear  ();

  int      isEmpty  () const;
  int      isFull   () const;
  int      length   () const;
  const T& first    () const;
  const T& last     () const;
  const T& itemAt   (int at) const;
  T&       itemAt   (int at);
  int      location (const T&) const;

protected:

  void copy (const Bounded<T,N>&);

private:

  T dep[N];
  int size;

};



template<class T, int N>
void Bounded<T,N>::copy (const Bounded<T,N>& r) {
  size=0;
  for (int i=0; i<r.size; i++) append(r.itemAt(i));
}


template<class T, int N>
void Bounded<T,N>::clear () {
  size=0;
}
```

```
template<class T, int N>
int Bounded<T,N>::isEmpty () const { return size==0; }


template<class T, int N>
int Bounded<T,N>::isFull () const { return size==N; }


template<class T, int N>
int Bounded<T,N>::length () const { return size; }


template<class T, int N>
const T& Bounded<T,N>::first () const { return itemAt(0); }


template<class T, int N>
const T& Bounded<T,N>::last () const { return itemAt(length()-1); }


template<class T, int N>
const T& Bounded<T,N>::itemAt (int at) const {
  static T except;
  if (isEmpty()) return except;  // Exception!
  if (at>=length()) at=length()-1;
  if (at<0) at=0;
  return dep[at];
}


template<class T, int N>
T& Bounded<T,N>::itemAt (int at) {
  static T except;
  if (isEmpty()) return except;  // Exception!
  if (at>=length()) at=length()-1;
  if (at<0) at=0;
  return dep[at];
}


template<class T, int N>
int Bounded<T,N>::location (const T& e) const {
  for (int i=0; i<size; i++) if (dep[i]==e) return i;
  return -1;
}


template<class T, int N>
void Bounded<T,N>::append (const T& t) {
  if (isFull()) return;
  dep[size++]=t;
}


template<class T, int N>
void Bounded<T,N>::insert (const T& t, int at) {
  if (isFull()) return;
  if ((at>size)||(at<0)) return;
  for (int i=size-1; i>=at; i--) dep[i+1]=dep[i];
  dep[at]=t;
  size++;
}


template<class T, int N>
void Bounded<T,N>::remove (int at) {
  if ((at>=size)||(at<0)) return;
  for (int i=at+1; i<size; i++) dep[i-1]=dep[i];
  size--;
}
```

```
template<class T, int N>
void Bounded<T,N>::remove (const T& t) {
  remove(location(t));
}


template<class T, int N>
Bounded<T,N>::Bounded () : size(0) {}


template<class T, int N>
Bounded<T,N>::Bounded (const Bounded<T,N>& r) : size(0) {
  copy(r);
}


template<class T, int N>
Bounded<T,N>& Bounded<T,N>::operator= (const Bounded<T,N>& r) {
  clear();
  copy(r);
  return *this;
}


template<class T, int N>
Bounded<T,N>::~Bounded () { clear(); }



////////////////////////////////////////////////////////////////////
// class template IteratorBounded
////////////////////////////////////////////////////////////////////

template <class T, int N>
class IteratorBounded {
public:

  IteratorBounded (const Bounded<T,N>*);
  IteratorBounded (const IteratorBounded<T,N>&);
 ~IteratorBounded ();

  IteratorBounded<T,N>& operator= (const IteratorBounded<T,N>&);
  int operator== (const IteratorBounded<T,N>&);
  int operator!= (const IteratorBounded<T,N>&);

  void reset();
  int  next ();

  int      isDone() const;
  const T* currentItem() const;

private:

  const Bounded<T,N>* theSupplier;
  int cur;

};



template <class T, int N>
IteratorBounded<T,N>::IteratorBounded (const Bounded<T,N>* b)
  : theSupplier(b), cur(0) {}
```

```
template <class T, int N>
IteratorBounded<T,N>::IteratorBounded (const IteratorBounded<T,N>& r)
  : theSupplier(r.theSupplier), cur(r.cur) {}


template <class T, int N>
IteratorBounded<T,N>& IteratorBounded<T,N>::operator= (const
IteratorBounded<T,N>& r) {
  theSupplier=r.theSupplier;
  cur=r.cur;
  return *this;
}


template <class T, int N>
IteratorBounded<T,N>::~IteratorBounded () {}


template <class T, int N>
int IteratorBounded<T,N>::operator== (const IteratorBounded<T,N>& r) {
  return (theSupplier==r.theSupplier)&&(cur==r.cur);
}


template <class T, int N>
int IteratorBounded<T,N>::operator!= (const IteratorBounded<T,N>& r) {
  return !(*this==r);
}


template <class T, int N>
void IteratorBounded<T,N>::reset () {
  cur=0;
}


template <class T, int N>
int IteratorBounded<T,N>::next () {
  if (!isDone()) cur++;
  return !isDone();
}


template <class T, int N>
int IteratorBounded<T,N>::isDone () const {
  return (cur>=theSupplier->length());
}


template <class T, int N>
const T* IteratorBounded<T,N>::currentItem () const {
  if (isDone()) return 0;
  else return &theSupplier->itemAt(cur);
}


#endif
```

- Datoteka `collect.h`:

```
// Project:  Abstract Data Types
// Module:   Collection
// File:     collect.h
// Date:     5.11.1996.
// Author:   Dragan Milicev
// Contents:
//      Class templates: Collection
//                       CollectionB
//                       CollectionU
//                       IteratorCollection
//                       IteratorCollectionB
//                       IteratorCollectionU

#ifndef _COLLECT_
#define _COLLECT_

#include "bound.h"
#include "unbound.h"


////////////////////////////////////////////////////////////////////
// class template Collection
////////////////////////////////////////////////////////////////////

template <class T>
class Collection {
public:

  virtual ~Collection () {}

  Collection<T>& operator= (const Collection<T>&);

  virtual IteratorCollection<T>* createIterator() const =0;

  virtual void add    (const T&) =0;
  virtual void remove (const T&) =0;
  virtual void remove (int at) =0;
  virtual void clear  () =0;

  virtual int      isEmpty  () const =0;
  virtual int      isFull   () const =0;
  virtual int      length   () const =0;
  virtual int      location (const T&) const =0;

protected:

  void copy (const Collection<T>&);

};



////////////////////////////////////////////////////////////////////
// class template IteratorCollection
////////////////////////////////////////////////////////////////////

template <class T>
class IteratorCollection {
public:

  virtual ~IteratorCollection () {}

  virtual void reset() =0;
  virtual int  next () =0;
```

```
  virtual int  isDone() const =0;
  virtual const T* currentItem() const =0;

};



template<class T>
void Collection<T>::copy (const Collection<T>& r) {
  for (IteratorCollection<T>* it=r.createIterator();
       !it->isDone(); it->next())
       if (!isFull()) add(*it->currentItem());
  delete it;
}

template<class T>
Collection<T>& Collection<T>::operator= (const Collection<T>& r) {
  clear();
  copy(r);
  return *this;
}



////////////////////////////////////////////////////////////////////
// class template CollectionB
////////////////////////////////////////////////////////////////////

template <class T, int N>
class CollectionB : public Collection<T> {
public:

  CollectionB () {}
  CollectionB (const Collection<T>&);
  virtual ~CollectionB () {}

  Collection<T>& operator= (const Collection<T>&);

  virtual IteratorCollection<T>* createIterator() const;

  virtual void add    (const T& t)             { rep.append(t); }
  virtual void remove (const T& t)             { rep.remove(t); }
  virtual void remove (int at)                 { rep.remove(at); }
  virtual void clear  ()                       { rep.clear(); }

  virtual int      isEmpty  () const           { return rep.isEmpty(); }
  virtual int      isFull   () const           { return rep.isFull(); }
  virtual int      length   () const           { return rep.length(); }
  virtual int      location (const T& t) const { return rep.location(t); }

private:
  friend class IteratorCollectionB<T,N>;
  Bounded<T,N> rep;
};



////////////////////////////////////////////////////////////////////
// class template IteratorCollectionB
////////////////////////////////////////////////////////////////////

template <class T, int N>
class IteratorCollectionB : public IteratorCollection<T>,
                            private IteratorBounded<T,N> {
```

```
public:

  IteratorCollectionB (const CollectionB<T,N>* c)
                            : IteratorBounded<T,N>(&c->rep) {}
  virtual ~IteratorCollectionB () {}

  virtual void reset() { IteratorBounded<T,N>::reset(); }
  virtual int  next () { return IteratorBounded<T,N>::next(); }

  virtual int  isDone() const { return IteratorBounded<T,N>::isDone(); }
  virtual const T* currentItem() const
                           { return IteratorBounded<T,N>::currentItem(); }

};



template<class T, int N>
CollectionB<T,N>::CollectionB (const Collection<T>& r) {
  copy(r);
}

template<class T, int N>
Collection<T>& CollectionB<T,N>::operator= (const Collection<T>& r) {
  return Collection<T>::operator=(r);
}

template<class T, int N>
IteratorCollection<T>* CollectionB<T,N>::createIterator() const {
  return new IteratorCollectionB<T,N>(this);
}



////////////////////////////////////////////////////////////////////
// class template CollectionU
////////////////////////////////////////////////////////////////////

template <class T>
class CollectionU : public Collection<T> {
public:

  CollectionU () {}
  CollectionU (const Collection<T>&);
  virtual ~CollectionU () {}

  Collection<T>& operator= (const Collection<T>&);

  virtual IteratorCollection<T>* createIterator() const;

  virtual void add    (const T& t)             { rep.append(t); }
  virtual void remove (const T& t)             { rep.remove(t); }
  virtual void remove (int at)                 { rep.remove(at); }
  virtual void clear  ()                       { rep.clear(); }

  virtual int      isEmpty  () const           { return rep.isEmpty(); }
  virtual int      isFull   () const           { return rep.isFull(); }
  virtual int      length   () const           { return rep.length(); }
  virtual int      location (const T& t) const { return rep.location(t); }

private:
  friend class IteratorCollectionU<T>;
  Unbounded<T> rep;
};
```

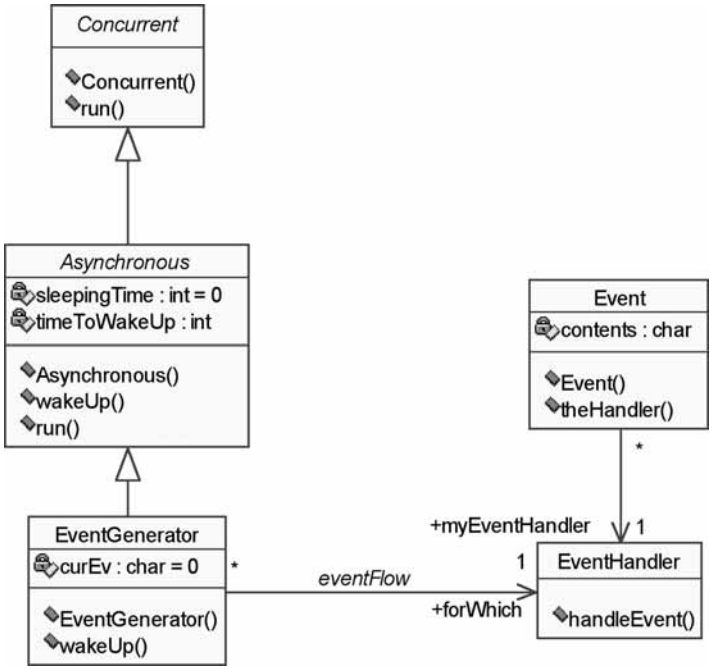
```
//////////////////////////////////////////////////////////////////////
// class template IteratorCollectionU
//////////////////////////////////////////////////////////////////////

template <class T>
class IteratorCollectionU : public IteratorCollection<T>,
                            private IteratorUnbounded<T> {
public:

  IteratorCollectionU (const CollectionU<T>* c)
                               : IteratorUnbounded<T>(&c->rep) {}
  virtual ~IteratorCollectionU () {}

  virtual void reset() { IteratorUnbounded<T>::reset(); }
  virtual int  next () { return IteratorUnbounded<T>::next(); }

  virtual int  isDone() const { return IteratorUnbounded<T>::isDone(); }
  virtual const T* currentItem() const
                              { return IteratorUnbounded<T>::currentItem(); }

};



template<class T>
CollectionU<T>::CollectionU (const Collection<T>& r) {
  copy(r);
}

template<class T>
Collection<T>& CollectionU<T>::operator= (const Collection<T>& r) {
  return Collection<T>::operator=(r);
}

template<class T>
IteratorCollection<T>* CollectionU<T>::createIterator() const {
  return new IteratorCollectionU<T>(this);
}


#endif
```

Zadatak

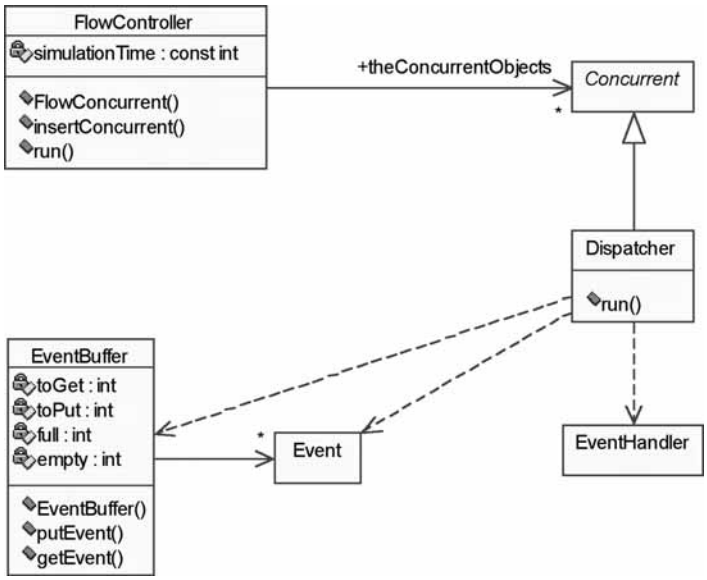
Realizovati jednostavan sistem za simulaciju orijentisan na događaje. U jezgru sistema leži apstrakcija konkurentnog objekta `Concurrent`, koja predstavlja apstraktnu osnovnu klasu za klase čiji se objekti izvršavaju „konkurentno“ u sistemu. Posebna vrsta konkurentnih objekata su tzv. asinhroni objekti, predstavljeni klasom `Asynchronous`, koji „periodično“ obavljaju neki posao. Posebna vrsta ovakvih objekata su „generatori događaja“, predstavljeni klasom `EventGenerator`. Svaki generator događaja vezan je za jednog „primaoca događaja“, predstavljenog klasom `EventHandler`, kome generiše događaje. Događaji se slažu u bafer događaja, objekat klase `EventBuffer`, dok jedan konkurentni objekat `Dispatcher` uzima događaje iz ovog bafera i prosleđuje ih primaocima kojima su namenjeni.

Rešenje

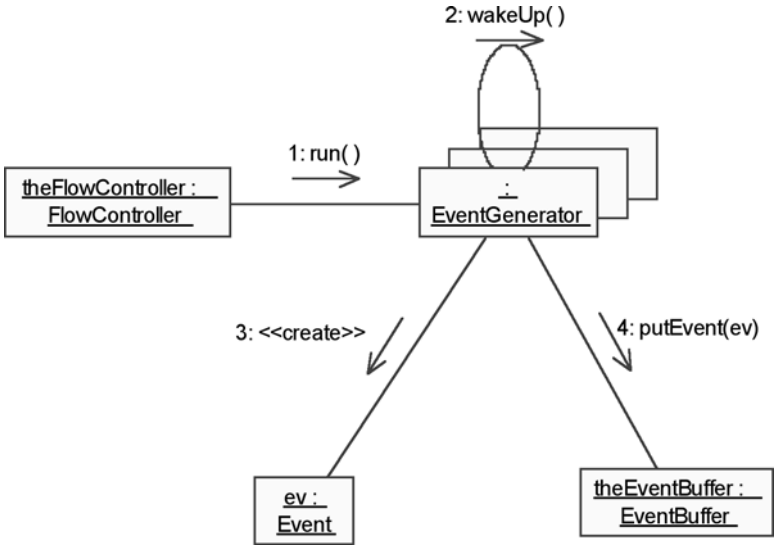
Dijagram klasa koji prikazuje ključne apstrakcije u sistemu:

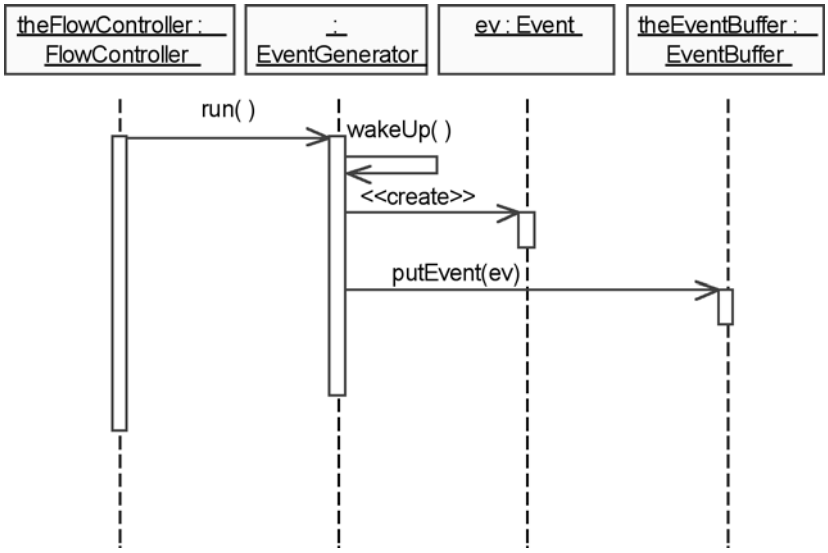


Dijagram klasa koji prikazuje implementacione klase:

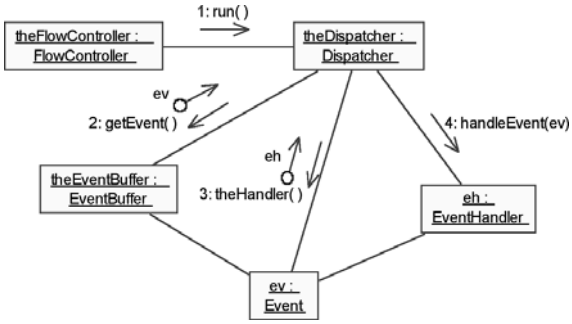


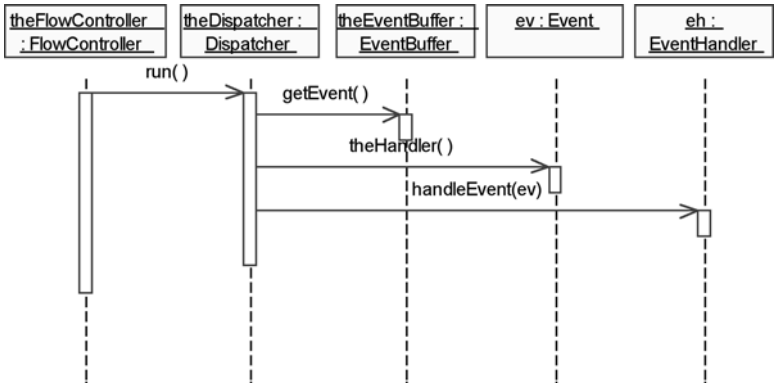
Dijagrami interakcije (kolaboracije i sekvence) koji opisuju prvi ključni mehanizam u sistemu – generisanje događaja:





Dijagrami interakcije (kolaboracije i sekvence) koji opisuju drugi ključni mehanizam u sistemu – obradu događaja:





Izvorni kôd:

```
// Project:     simpleev.prj


// ************************************************
// Constants:

const int tEvGen = 5;  // time constant for EventGenerator
const int maxEvGens =10; // maximum number of EventGenerators
const int maxEvHand =10; // maximum number of EventHandlers
const int maxConcur = maxEvGens+1; // maximum number of Concurrent objects
const int maxEvents =10; // maximum number of Events


// ************************************************
// Class definitions:

class Concurrent {
public:
              Concurrent ();
  virtual void run () = 0;
};
```

```
class Asynchronous : public Concurrent {
public:
               Asynchronous (int timeToWakeUp);
  virtual void run ();

protected:
  virtual void wakeUp () = 0;

private:
        int sleepingTime;
  const int timeToWakeUp;
};



class EventHandler;    // forward

class EventGenerator : public Asynchronous {
public:
               EventGenerator (EventHandler* forWhich);
protected:
  virtual void  wakeUp ();

private:
  EventHandler* forWhich;
  char          curEv;
};



class Dispatcher : public Concurrent {
public:
  virtual void run ();
};



class FlowController {
public:
       FlowController   (int simulationTime);
  void insertConcurrent (Concurrent* toInsert);

  void run ();

private:
  Concurrent* theConcurrentObjects [maxConcur];
  int         numOfConcurrentObjects;
  const int   simulationTime;
};



class Event {
public:
               Event      (EventHandler* forWhich, char contents);
  EventHandler* theHandler ();

private:
  EventHandler* myEventHandler;
  char          contents;
};
```

```
class EventBuffer {
public:
        EventBuffer ();
  void   putEvent    (Event* toPut);
  Event* getEvent    ();

private:
  Event*  theEventQueue [maxEvents];
  int     toGet,toPut;
  int     full,empty;
};



class EventHandler {
public:
  void handleEvent (Event* toHandle);
};




// ************************************************
// Static objects definitions:

FlowController theFlowController(100000);
Dispatcher theDispatcher;
EventBuffer theEventBuffer;


// Event handlers:
EventHandler   EventHandler1, EventHandler2, EventHandler3;

// Event generators:
EventGenerator EventGenerator1(&EventHandler1),
               EventGenerator2(&EventHandler1),
               EventGenerator3(&EventHandler2),
               EventGenerator4(&EventHandler2),
               EventGenerator5(&EventHandler3);




// ************************************************
// Class implementations:

Concurrent::Concurrent () { theFlowController.insertConcurrent(this); }


Asynchronous::Asynchronous (int t) : timeToWakeUp(t), sleepingTime(0) {}

void Asynchronous::run () {
  if (++sleepingTime==timeToWakeUp) {
    wakeUp();
    sleepingTime=0;
  }
}



EventGenerator::EventGenerator (EventHandler* fw) : Asynchronous(tEvGen),
                                                     forWhich(fw), curEv(0) {}

void EventGenerator::wakeUp () {
  theEventBuffer.putEvent(new Event(forWhich,curEv));
  curEv=(curEv+1)%128;
}
```

```
void Dispatcher::run () {
  Event* ev=theEventBuffer.getEvent();
  if (ev!=0) ev->theHandler()->handleEvent(ev);
}


FlowController::FlowController (int s) : simulationTime(s),
                                         numOfConcurrentObjects(0) {}

void FlowController::insertConcurrent (Concurrent* c) {
  if (numOfConcurrentObjects<maxConcur)
    theConcurrentObjects[numOfConcurrentObjects++]=c;
};

void FlowController::run () {
  for (int t=0; t<simulationTime; t++)
    for (int i=0; i<numOfConcurrentObjects; i++)
      theConcurrentObjects[i]->run();
}



Event::Event (EventHandler* fw, char c) : myEventHandler(fw), contents(c) {}

EventHandler* Event::theHandler () { return myEventHandler; }



EventBuffer::EventBuffer () : toGet(0), toPut(0), full(0), empty(1) {}

void EventBuffer::putEvent (Event* e) {
  if (full==1) {  // buffer full!
    delete e; return;
  }
  theEventQueue[toPut]=e;
  toPut=(toPut+1)%maxEvents;
  if (toPut==toGet) full=1;
  empty=0;
}

Event* EventBuffer::getEvent () {
  if (empty==1) return 0; // buffer empty!
  Event* e=theEventQueue[toGet];
  toGet=(toGet+1)%maxEvents;
  if (toGet==toPut) empty=1;
  full=0;
  return e;
}



void EventHandler::handleEvent (Event* e) {
  // here is the place for event handling!
  delete e;
}



// ************************************************
// Main program:

void main () {
  theFlowController.run();
}
```