CERN School of Computing

Design Patterns for Concurrent Objects

Marek Biskup

Warsaw University

CERN School of Computing

Outline

- Design patterns
- Synchronization patterns
- Concurrency patterns
- Event-driven programming

CERN School of Computing

Design Patterns

- **Some software design problems reappear in many application developments**
- **Successful solutions to such problems may be reused**
- **Design Patterns – documented reusable solution**
  - Not a finished design that can be transferred into code
  - Idea of how to solve a problem in a particular context
- **Mostly related to Object-Oriented design**
- **Give us common vocabulary to discuss O-O design**

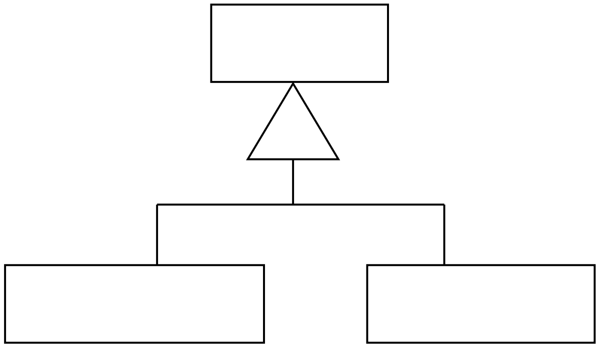
CERN School of Computing

Example – Proxy Pattern

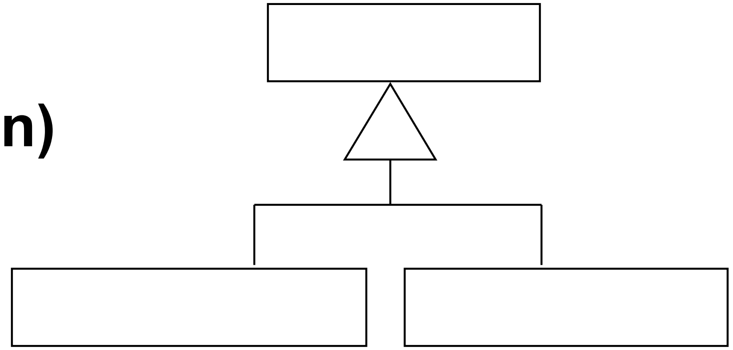
- **Proxy: An object acting as an interface to another object**
  - All methods of the original object but doing something ”in between”
- **CacheProxy**
  - Results of expensive operations are cached so that the next call for the same data will return immediately
- **RemoteProxy (e.g. JavaRMI stubs)**
  - Client calls a method of a local stub
  - The stubs sends method parameters to the server
  - The server executes the method
  - Results are returned from the stub as if the call were local

Example – Abstract Factory

- **Decouples creation of objects from the implementation**
- **Example: Bad GUI desing – code dependent on the platform**
  - Windows code:
    - Button b = new WinButton();
  - Linux code:
    - Button b = new QtButton();



- **AbstractFactory – platform independent code**
    - Button b = guiFactory.getButton();



- **Initialization (once in the application)**
    - guiFactory = new WinGuiFactory();

CERN School of Computing

Synchronization

- Access to the same memory by several threads has to be synchronized
- Example: increasing a value

```
int value; // global
void increase() {
    int x = value
    x++;
    value = x;
}
```

- Executed on two processors simultaneously may increase the value by one instead of two
- There are standard means of synchronization: mutexes (lock and unlock), monitors (as in java), semaphores

CERN School of Computing

Locking

- **Critical section has to be guarded by a lock**
- **When leaving the critical section the lock must be released, but:**
  - There are different execution paths
  - Exceptions may be thrown
- **Failing to release a lock results in a deadlock**
- **Such an error is difficult to spot**
- **And the code is difficult to maintain**

```
class Counter {
  bool increment(int) {
    lock.lock();
    if ( … ) {
      // …
      lock.unlock();
      return false;
    }
    // …
    lock.unlock();
    return true;
  }
};
```

CERN School of Computing

Scoped Locking Pattern

- **Solution: Use a class that:**
  - Acquires the lock in the constructor
  - Releases the lock in the destructor
- **Then the lock will always be released**
- **Use that pattern even if you have a critical section inside a block:**

```
class MutexGuard {
  Mutex* lock;
public:
  MutexGuard(Mutex& aLock)
          : lock(aLock) {
    lock->lock();
  }
  ~MutexGuard() {
    lock->unlock();
  }
};
```

```
mutex lock;
void f() {
  // do whatever
  {
    MutexGuard(lock);
    // critical section
    // ...
  }
}
```

```
class counter {
  Mutex lock;
  bool increment() {
    MutexGuard(lock);
    // critical section
  } // guard will be released in ~counter()
};
```

CERN School of Computing

Locking in Java

- **Use *synchronized* statement**
- **With an explicit lock: no local objects, so no destructors will be called**
  - Cannot use the Scoped Locking Pattern
- **Solution: the *finally* statement**
- **Finally block is always executed when leaving a try block:**
  - Normal leaving
  - After the return statement
  - When an exception is thrown

```
synchronized int aFunction() {
  // synchronized code
}
```

```
public int aFunction()
        throws MyException {
  lock.lock();
  try {
    if (A < 0)
      throw new MyException(A);
    if (A == 1)
      return 1;
    A++;
    return 0;
  } finally {
    lock.unlock();
  }
}
```

CERN School of Computing

Intra-Component Call

- **Often there is need to call another method from the same component (class)**
- **If component methods are synchronized the same lock will be acquired twice**
  - Deadlock if the lock is not reentrant
  - Unnecessary overhead with reentrant locks

```
class buffer {
Mutex lock;
public:
  void put(int a) {
    MutexGuard(lock);
    A[num++] = a;
    if (num == N)
      flush(); // deadlock!
  }
  void flush() {
    MutexGuard(lock);
    // write the contents to a file
  }
};
```

CERN School of Computing

Thread-Safe Interface

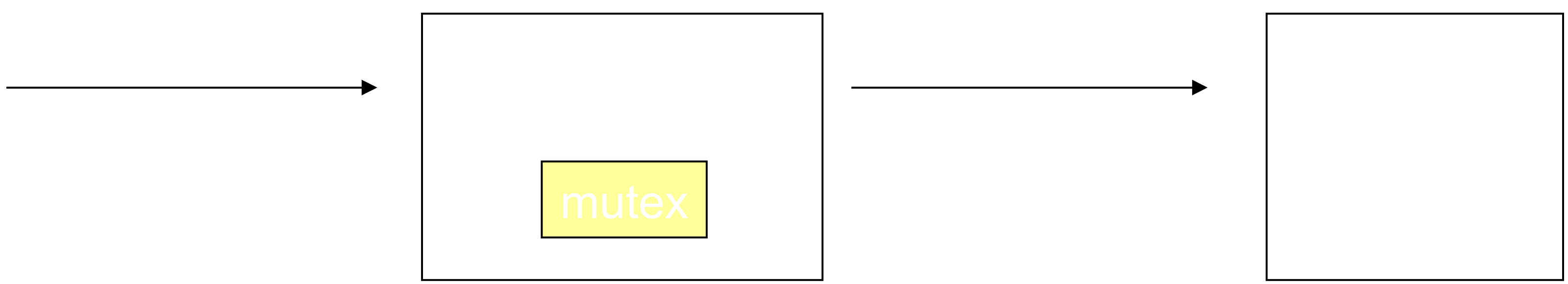
- **Structure the component into two layers**
  - Interface methods – acquire a lock and call an implementation method
  - Implementation methods – assume that the lock has already been acquired and do the job
- **Implementation methods may only call other implementation methods (and not interface ones)**
- **But:**
  - Still be careful when calling other component methods!
  - It adds a bit of overhead (more methods, twice more function calls)

```
class buffer {
Mutex lock;
public:
  void put(int a) {
    MutexGuard(lock);    putImp();
  }
  void flush() {
    MutexGuard(lock);    flushImp();
  }
protected:
  void putImp() {
    A[num++] = a;
    if (num == N)
      flushImp(); // implementation
  }
  void flushImp() {
    // write the contents to a file
  }
};
```

CERN School of Computing

Synchronizing Proxy

- **Synchronize access to an object without changing its code**
- **Solution: Synchronizing Proxy Pattern**
  - The proxy contains:
    - all the methods from our object
    - a mutex
  - Call the proxy instead of the object itself
  - Proxy synchronizes access to the object



CERN School of Computing

Read-Write Lock Pattern

- **Consider a concurrent dictionary:**
  - Many threads can read at the same time
  - Exclusive access when writing

- Better locking strategy – a lock with two operations:
  - readLock(); – blocks when the write-lock is taken
  - writeLock(); – blocks when the read or write-lock is taken

ReadWriteLock

- Starvation problem:
  - When readers keep on coming, writers will wait forever

- Solution:
  - e.g.: don't allow readers when a writer is waiting

CERN School of Computing

Initializing a value

- **Think of a value that should be initialized when accessed for the first time**

```
// no synchronization

class Foo {
   private Helper = null;
   public Helper getHelper() {
     if (helper == null)
         helper = new Helper();
     return helper;
   }
   // other functions and members...
}
```

```
// too much synchronization (?)

class Foo {
   private Helper = null;
   public synchronized Helper getHelper() {
     if (helper == null)
         helper = new Helper();
     return helper;
   }
   // other functions and members...
}
```

CERN School of Computing

Double Checked Locking

- **Can we do it better?**

```
// double checked locking optimization

class Foo {
   private Helper = null;
   public Helper getHelper() {
     if (helper == null) {
         synchronized(this) {
            if (helper == null)
               helper = new Helper();
         }
      return helper;
   }   // ...
}
```

Double check: several processes may enter here

Scenario:

Thread A:

helper == **null**;

helper = **new** Helper();

compilers may assign the pointer value before **A** has finished executing the constructor!

Thread B:

helper != **null**; //!

useHelper // not initialized!

CRASH!

CERN School of Computing

D.C.L. is an AntiPattern

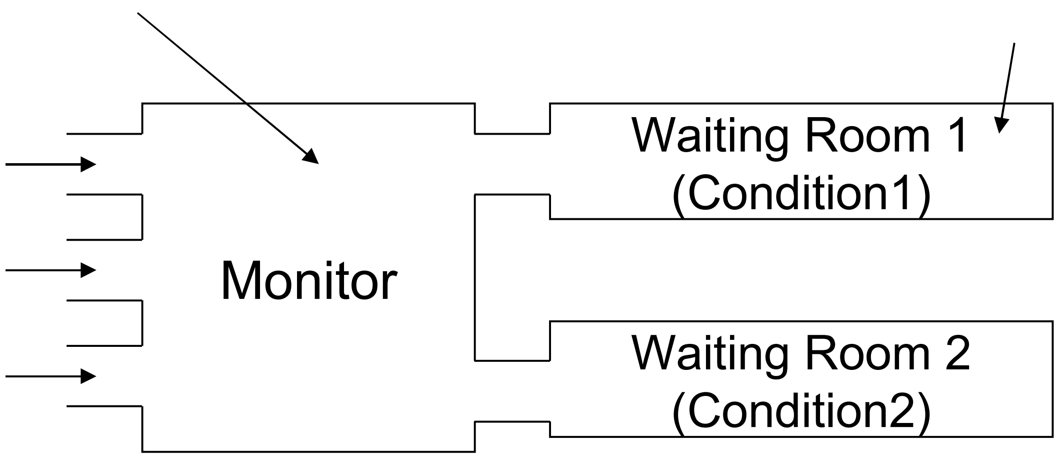
- **In most programming languages (java, c++) using D.C.L optimization may result in a crash**
- **Correct code is the previous one:**

```
// correct code

class Foo {
   private Helper = null;
   public synchronized Helper getHelper() {
     if (helper == null)
         helper = new Helper();
      return helper;
   }
   // other functions and members...
}
```

CERN School of Computing

Monitor object

- **As presented in the previous lecture:**
  - a lock that synchronizes access to the object
  - Condition variables for waiting for specific events
  - Notify() wakes up waiting processes

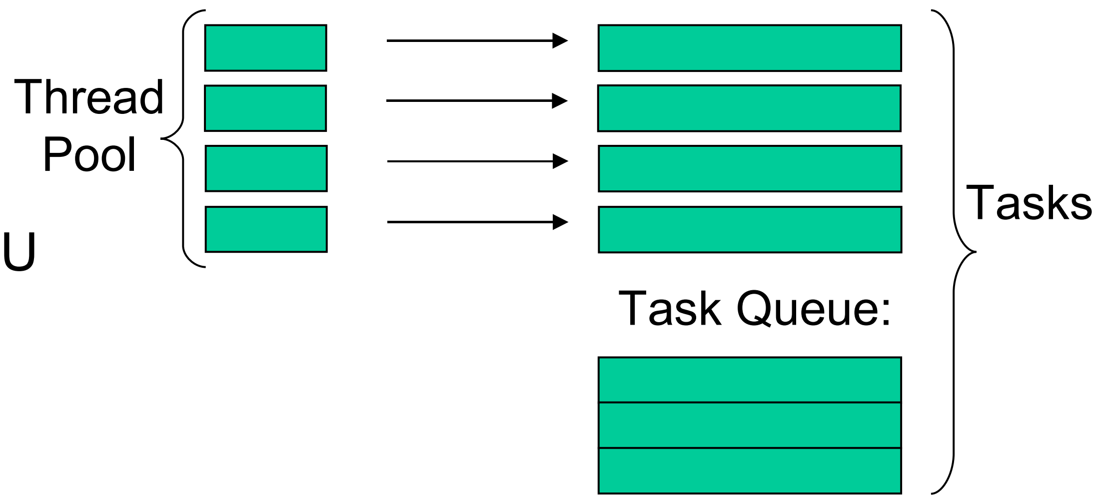


CERN School of Computing

Thread Pool Pattern

- **Steady stream of tasks to be performed, e.g.**
  - A web server handling many clients
  - A database server
- **Single thread:**
  - Not optimal on multiCPU systems
  - Asynchronous IO is difficult
- **Thread per request**
  - Simplifies programming
  - Creating a thread is an expensive operation

Avoid the expense of creating a new thread by reusing existing ones.



•A thread that finished a task takes another one from the queue.
•If there are no more tasks the thread may terminate or sleep.

Thread Pool in Java

- **ThreadExecutor interface**
  - execute(Runnable command)
- **Decouples submitting a task and running it**
- **Several executors Possible:**
  - Direct – just starts the task (synchronously)
  - Thread per task – creates a new thread for each task
  - ThreadPoolExecutor
    - corePoolSize – minimum # of threads
    - maximumPoolSize – maximum # of threads
    - keepAliveTime – threads wait at most this amount of time

```
Executor executor = anExecutor;
executor.execute(new RunnableTask1());
executor.execute(new RunnableTask2());
```

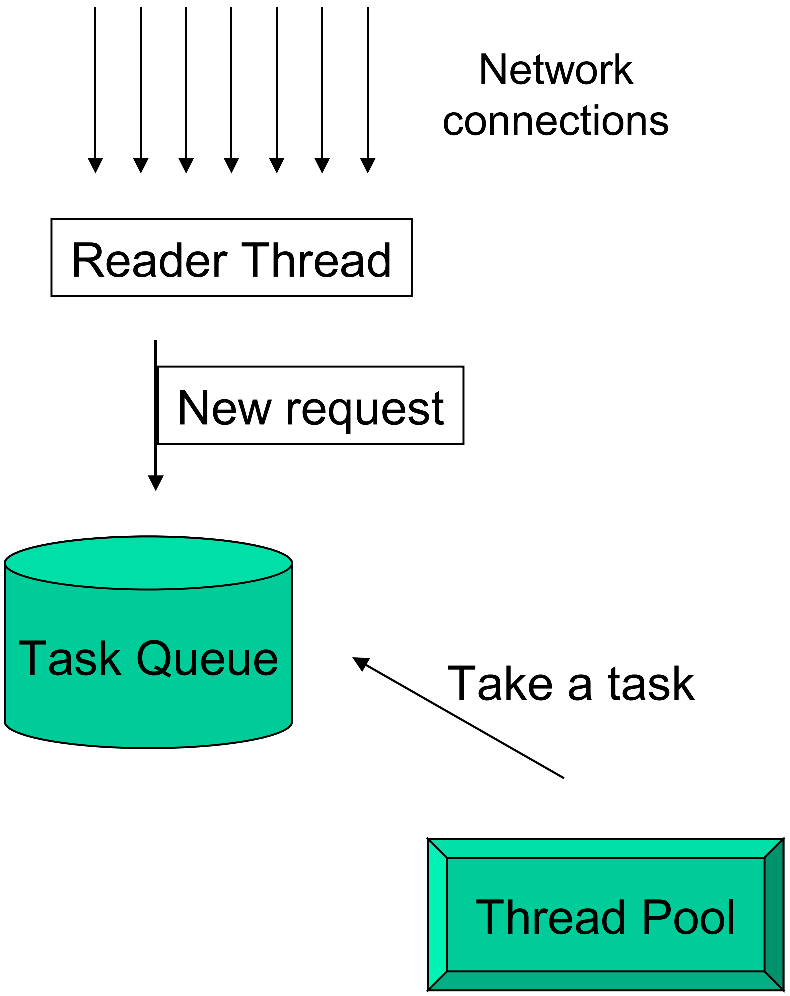
```
class DirectExecutor
        implements Executor {
  public void execute(Runnable r) {
     r.run();
  }
}
```

```
class ThreadPerTaskExecutor
              implements Executor {
  public void execute(Runnable r) {
     new Thread(r).start();
  }
}
```

CERN School of Computing

Multithreaded server

- **Pool of threads**
- **Each new request is taken by an available thread**
- **A possible solution:**
  - One thread reads request from the network (UNIX *select* call)
  - Creates a request and passes it to a working thread
- **Problems:**
  - Passing request between threads requires frequent context switching and synchronization
- **Can we do it better?**

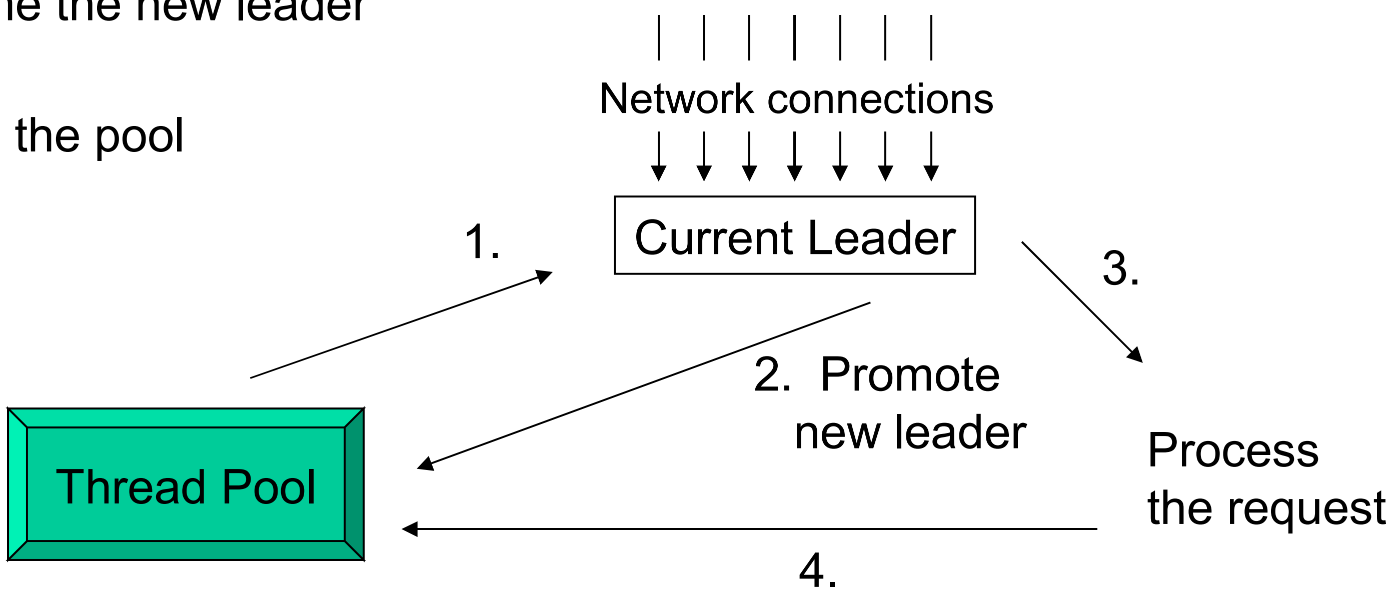


CERN School of Computing

Leader-Followers pattern

- **Threads take turns in reading request**
- **After reading a request:**
  - Wake up another thread to read the next request
  - Process the request
  - If there is no leader
    - Become the new leader
  - Else
    - wait in the pool

```
loop {
  ReadARequest();
  ThreadsPool.Notify();
  ProcessTheRequest();
  if (there is another leader)
     ThreadsPool.Wait();
}
```



CERN School of Computing

Global values

- **The C errno value**
  - Global for the whole application
  - Is set after a system function’s call has failed
- **When there are threads the value might get overwritten in another thread**
- **Possible solution: protect the value with a lock**
  - System function will acquire a lock when an error appears
  - User application will read the value and release the lock
- **Very bad a solution**
  - It is difficult to change system functions
  - A programmer may forget to release the lock – deadlock

Such a problem appears in other legacy systems as well.

CERN School of Computing

Thread-specific storage

- **Introduce a global access point for such a global variable**
- **But keep separate value that variable per thread**
- **Values can be associated with threads**
  - Java: Thread class:
    - public static Thread currentThread()
    - When subclassed values can be kept in the Thread object
    - ThreadLocal object – a value, different for each thread
  - C (pthread):
    - pthread_t pthread_self(void);
    - Gives the thread's ID. ID's can be mapped to objects.
    - pthread also has functions for associating values with keys for a thread

```
#define errno (*(__errno()))
```

```
int *__errno() {
  // find the pointer p to thread-specific errno
  return p;
}
```

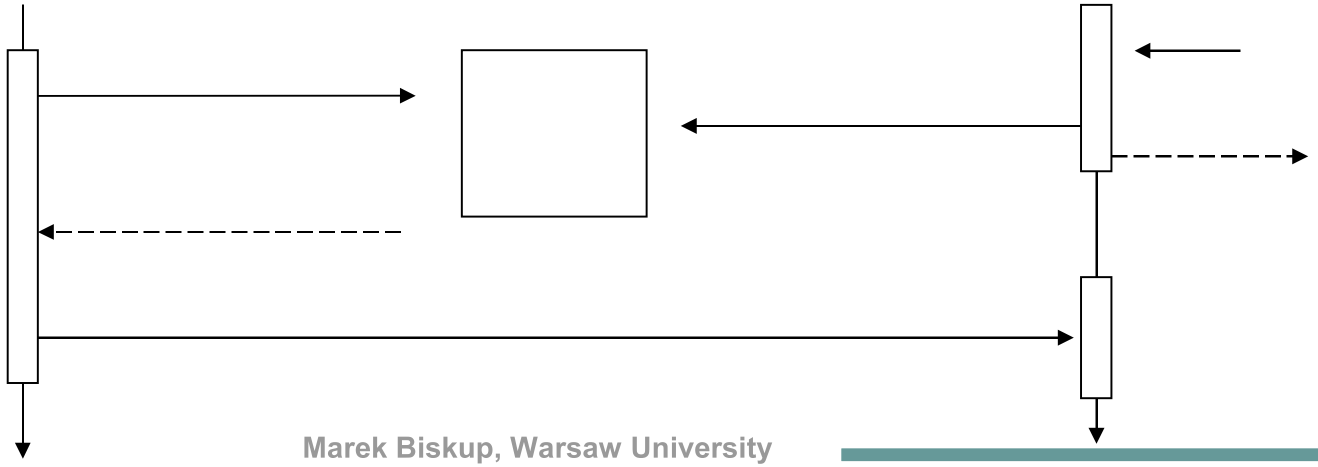
CERN School of Computing

Future Object

- **Some operations have to be processes asynchronously**
  - Using a Thread Pool
  - Via the operating system (async. IO)
- **Future Object encapsulates**
  - The asynchronous operation's code
  - Completion (called after the operation has been finished)
- **All code at the same place**
- **Example: web server serving a client**
  - Asynchronous operation: reading file from disk
  - Completion: Invoke another async. operation – send the file.
    - Another completion: e.g. close the connection

Future Object cont.

- **At some point one has to use the result from the asynchronous IO.**
- **How to check if the task is finished?**
  - call isDone() method of the Future
  - call get() method (waits for the result)
  - Use a queue of finished Futures
- **Queue**
  - The future at the end of its asynchronous operation puts its identifier into a queue
  - The main task waits on the queue for finished tasks

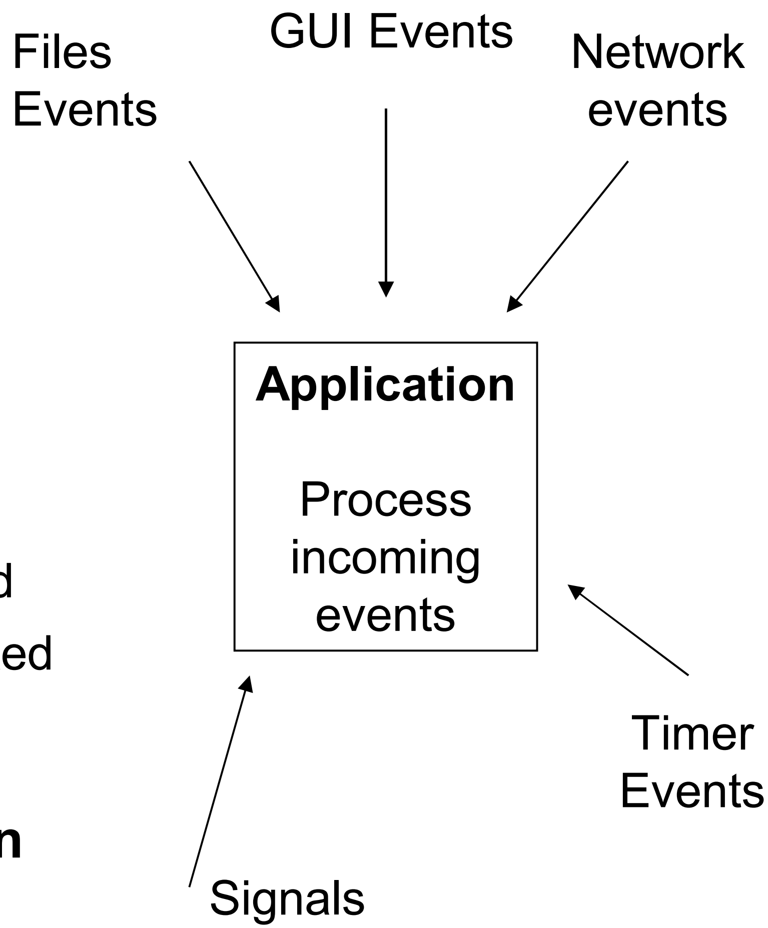


Futures in Java

- **Interface Future<V>:**
  - **boolean** cancel()
  - V get()
  - **boolean** isDone()
  - **boolean** isCancelled()
  - Additional methods may implement the completion
- **Class FutureTask<V>; additional methods:**
  - run() – the async. operation itself
  - done() – called when the state turns to *done*. May be overridden e.g. to notify the main thread
  - Additional methods for completion may be added when subclassed
- **FutureTask may be executed using Executors (e.g. ThreadPoolExecutor)**

Events

- **External „message"**
  - GUI event
    - The user clicked on a button
    - The user moved the mouse pointer
  - Network event
    - New network connection is coming
    - New data is arriving
    - More data can be sent right now
  - Operating system's events
    - Application is going to be terminated
    - Asynchronous IO has been completed
- **Or internal message, from another thread/process within the application**



CERN School of Computing

Event Handles

- **Each event has an associated Event Handle**
  - An identifier of the event source (network connection, open file, GUI widget, etc.)
  - Usually provided by the O.S
- **Examples:**
  - For a network connection, files: descriptor number
  - For GUI: widgetID
  - Timers: timerID
- **There may be several kinds of events for one handle**
  - GUI widget:
    - MouseClicked
    - MouseMoved
  - For a file descriptor
    - Read event – more data to read
    - Write event – more data to write

CERN School of Computing

Event-driven programming

- **Normal model of programming:**

```
Initialize();
DoComputations();
Terminate();
```

- **Event-driven programming**
  - Events are processed in a loop (so called „Event Loop”)
  - When no events no computations are performed

```
Initialize();
while ((e = GetEvent()) != 0) {
  HandleEvent(e);
}
Terminate();
```

CERN School of Computing

Event Loop

- **The Event Loop is often provided by the O.S.**
- **Event handler – user defined function that handles an event**
  - Has to be registered for a specific event type
  - The O.S. will call back the event handler when a specific event appears
- **Inversed program's logic**
  - No main loop
  - Just event handlers should be written (+ the main initialization function)

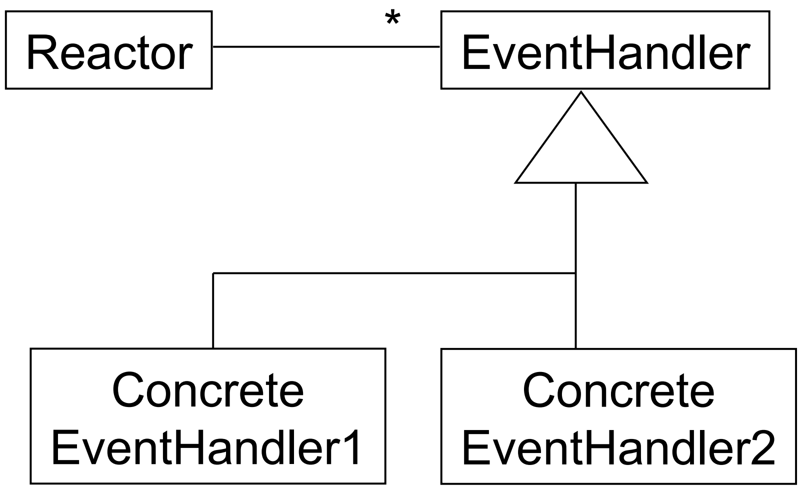
```
void SocketHandler() {
  // read the incomming data
}
void ExitHandler() {
  OS.StopEventLoop();
}
void main() {
  socket s = OpenConnection();
  OS.registerEventHandler(
    EXIT_EVENT, ExitHandler);

  OS.registerEventHander(
    SOCKET_EVENT, s,
    SocketHandler);

  OS.EventLoop();
}
```

callback functions (event handlers)

CERN School of Computing

Reactor pattern

Object-oriented way of implementing the event loop



- **EventHandler – abstract class**
  - Owns a handle
  - HandleEvent() method
- **Concrete Event Handlers inherit from EventHandler class**
- **Reactor**
  - EventLoop()
  - RegisterHandler(handler)
  - RemoveHandler(handler)

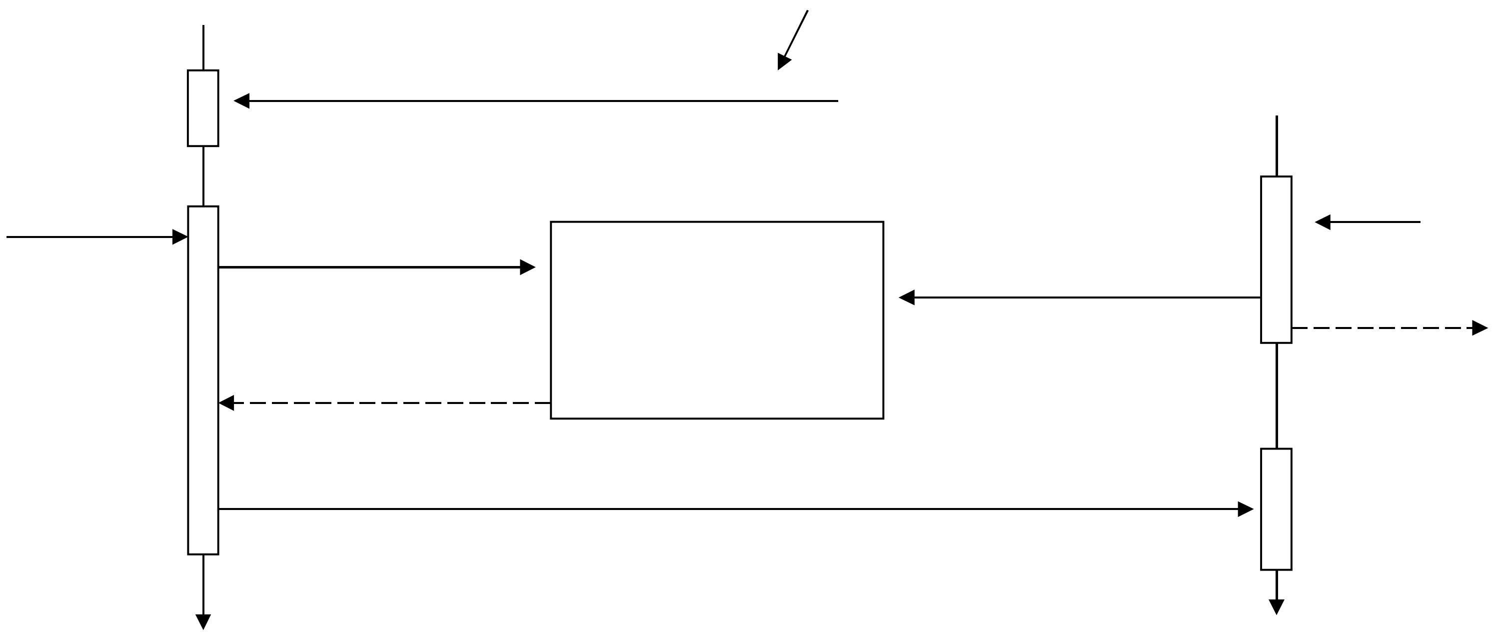
Loop:
- Get handles from all registered handlers
- Wait for an event from one of the handles (using the O.S)
- Call EventHandler->HandleEvent() for the handle returned by the demultiplexer

Java AWT – Event Handling

- **The Event Loop runs in a separate thread**
- **EventHandlers: EventListener interface**
  - Separate interfaces for different events
    - *MouseListener* for mouse events
    - *KeyListener* for keyboard events
- **Registering handlers to widgets:**
  - aWidget.addMouseListener(aMouseListener);
  - aWidget.addKeyListener(aKeyListener);
- **Handling events:**
  - Appropriate method of a listener is called:
    - e.g. mousePressed(), mouseEntered()
  - Each method takes a parameter: EventObject
    - MouseEvent for mouse events (mouse position, button state, etc.)

Handling asynchronous operations

- Reactor Pattern can be used for handling asynchronous operations
- Remember the Future pattern?



Summary

- **Design Patterns are successful solutions to common design problems**
- **There are useful patterns for concurrent programs**
- **Synchronization patterns: Scoped Locking, Thread-Safe Interface, Synchronizing Proxy, Read-Write Lock, Monitor**
- **Concurrency Patterns: Thread Pool, Leaders-Followers, Thread-Specific Storage**
- **Event-Handling Patterns: Future, Reactor, Proactor**

CERN School of Computing

Further Reading and materials used

- **Schmidt, Stal, Rohnert, Buchmann – *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects, Volume 2***
- **Gamma, Helm, Johnson, Vlissides – *Design Patterns: Elements of Reusable Object-Oriented Software***
- **Java standard library**
- **Wikipedia**
- **Contact: mbiskup@mimuw.edu.pl**