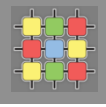
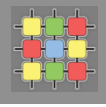


Architecture of software systems

Course 14: Serialization, externalization, NIO interface,
Intelligent systems, Multi-agent systems

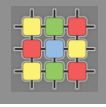
David Šišlák
david.sislak@fel.cvut.cz

Serialization



- usefull for
  - persisting object graphs – all members to disk or database
  - network transmission
  - other – e.g. compute object signature
- key classes:
  - java.io.Serializable (no method definitions, only marker)
  - ObjectInputStream
  - ObjectOutputStream
- **produce special binary stream**
  - serialization uses reflection of all non-static members except **transient**
  - class definition is not saved !!!
  - store field names
- constructor and members can be also private, sub-classes requires protected/public constructors

Serialization

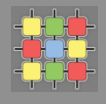


- all subclasses are automatically Serializable
- non-serializable class can be made serializable in any sub-type
  - but there has to be accessible no-arg constructor in them
  - data from parent are not automatically serialized !
- identification of non-serializable object when traversing a graph causes NotSerializableException

```
ObjectOutputStream out = new ObjectOutputStream(
        new FileOutputStream("test.dat"));
out.writeObject(serializableObject);
out.close();
```

```
ObjectInputStream in = new ObjectInputStream(
        new FileInputStream("test.dat"));
serializableObject = in.readObject();
in.close();
```

Serialization



- each Seraializable class has

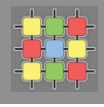
```
private static final long serialVersionUID = 710635817258052445 6L;
```

  - generated based on class name, modifiers, interfaces, methods, etc.
- BEWARE of **changes of class definitions**
  - InvalidClassException – different serialVersionUID !
- define own serialVersionUID using **serialver** tool

- define serialization fields – can be used for evolving objects
  - only **non-transient** and **non-static** are serialized
  - serialPersistentFields (ObjectStreamField[])
    - suitable for compatibility with old versions

```
private final static ObjectStreamField[] serialPersistentFields = {
  new ObjectStreamField("numberPrimitive", Integer.TYPE),
  new ObjectStreamField("doubleObject", Double.class),
  new ObjectStreamField("myObject", Test.class)
};
```

Serialization



- **special handling of classes** (exact method signature)
  - additional information
  - initialization of non-serialized fields
  - solve incompatibility of versions

private void **writeObject**(ObjectOutputStream out) throws IOException

- can call out.**defaultWriteObject** (default nebo serialPersistentFields)

private void **readObject**(ObjectInputStream in) throws IOException

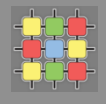
- can call in.**defaultReadObject** (default nebo serialPersistentFields)

private void **readObjectNoData**() throws ObjectStreamException

- given class is not listed as a superclass of deserialized object
- receiver's version extends classes that are not extended by the sender's version

- *anyway serialization continue with superclass serialization automatically*

Serialization



» use **alternative objects**

ANY-MODIFIER Object **writeReplace**() throws ObjectStreamException

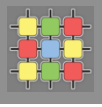
- serialize different object than this

ANY-MODIFIER Object **readResolve**() throws ObjectStreamException

- after deserialization the object is replaced

```
public class Singleton implements Serializable {
        ...

        protected Object readResolve() {
                return getInstance();
        }
}
```

Externalizable



- faster than Serialization
- usually produce shorter binary stream
- control object graph traversal, but what about repeating objects?
- but you loose flexibility, add more bugs, class object is usually longer

- *Externalization doesn't continue with superclass serialization automatically!*
- **requires public no-arg constructor**

  public void **writeExternal**(ObjectOutput out) throws IOException

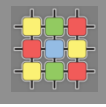
  public void **readExternal**(ObjectInput in) throws IOException

```
ObjectOutputStream out = new ObjectOutputStream(
        new FileOutputStream("test.dat"));
out.writeObject(externalizableObject);
out.close();
```

VS.

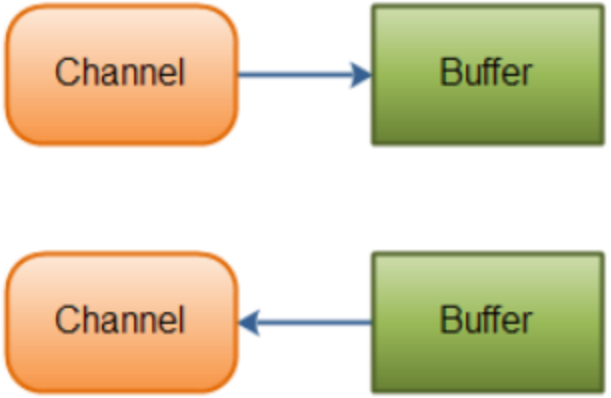
```
ObjectOutputStream out = new ObjectOutputStream(
        new FileOutputStream("test.dat"));
externalizableObject.writeExternal(out);
out.close();
```

JAVA networking – NIO

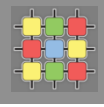


- NIO – new IO – implemented in java.nio starting from Java 1.4
- API for
  - scalable I/O – **asynchronous I/O** requests and **polling**
  - high-speed **block-oriented** binary and character I/O working – including mapping files to the memory, using channels and selectors
  - regular expressions
  - charset conversion
  - improved files system interface

- some functions are dependent on the underlying OS
- object **Channel**

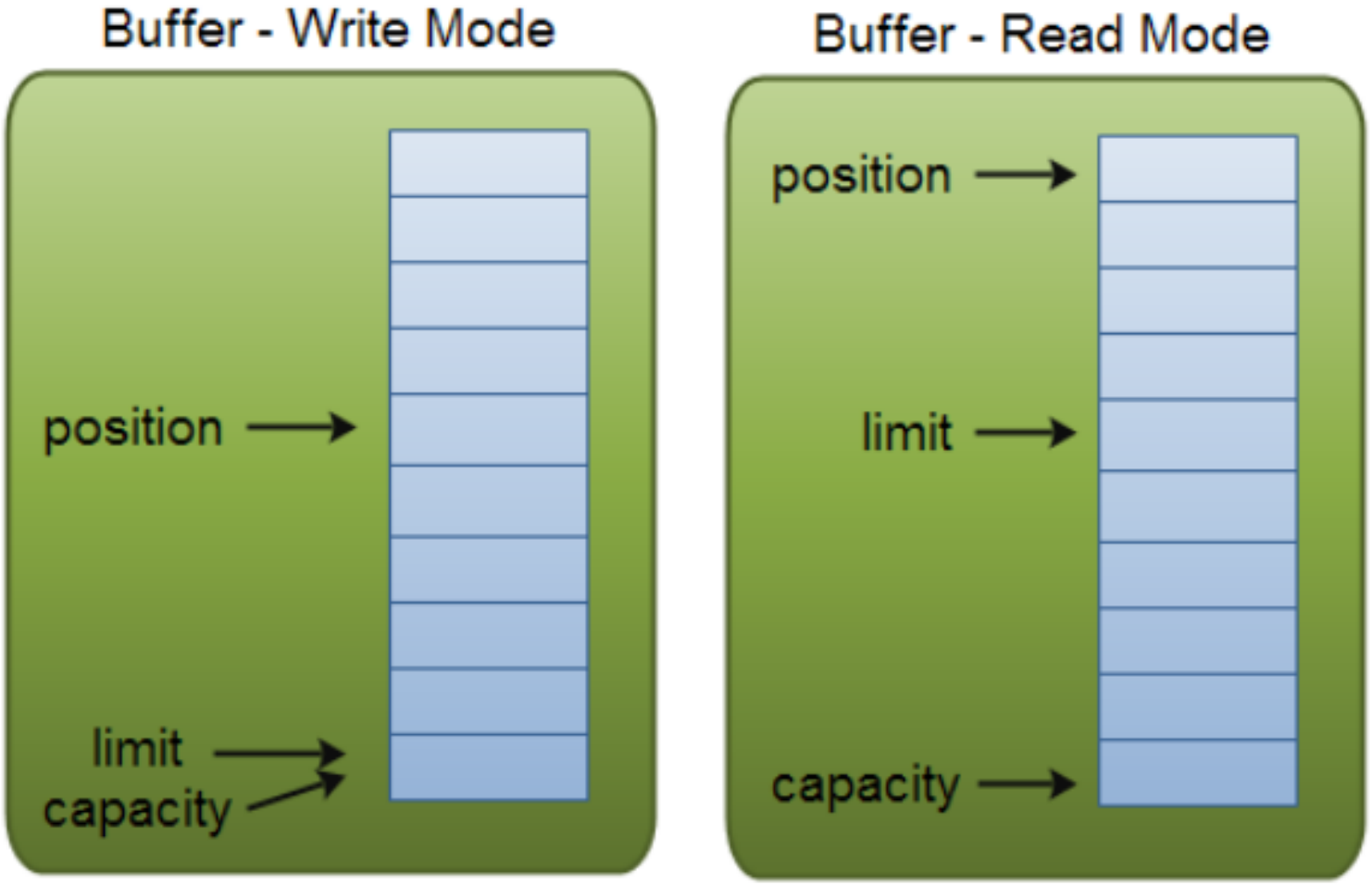


JAVA networking – NIO - Buffer



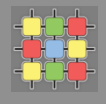
- java.nio.Buffer – **direct buffer** is ***stored outside of heap***
  - linear, **finite sequence** of elements of a specific primitive type
    - ByteBuffer, CharBuffer, DoubleBuffer, FloatBuffer, IntBuffer, LongBuffer, ShortBuffer, MappedByteBuffer {FileChannel.map(…)}
  - not thread safe, **multi mode** for the same buffer (*read, write*)
  - key properties – 0 <= mark <= position <= limit <= capacity
    - capacity – numbers of elements, never changing !
    - limit – index of the first element that should not be read or written
    - position – index of the next element to be read or written
    - mark – index to which its position is set after reset()
  - **initial content is undefined** – *not nulled* as it can be non-heap located
  - **clear**() – position=0, limit=capacity => *ready for channel read (**put**)*
  - **flip**() – limit=position, position=0 => *ready for channel write (**get**)*
  - **rewind**() – limit unchanged, position=0 => *ready for re-reading*
  - **mark**() – mark = position
  - **reset**() – position=mark

JAVA networking – NIO - Buffer



- write mode – channel.**read**(buf); buf.**put**(…);
- read mode – channel.**write**(buf); … buf.**get**();

JAVA networking – NIO - Buffer



- java.nio.Buffer
  - **isReadOnly**() – can be read-only
  - **hasArray**() – is backed by an accessible array (array()), **non-native** only
  - **equals**(), **compareTo**() – compare remainder sequence

  - can be allocated to physical memory – ***direct OS operation over it*** !

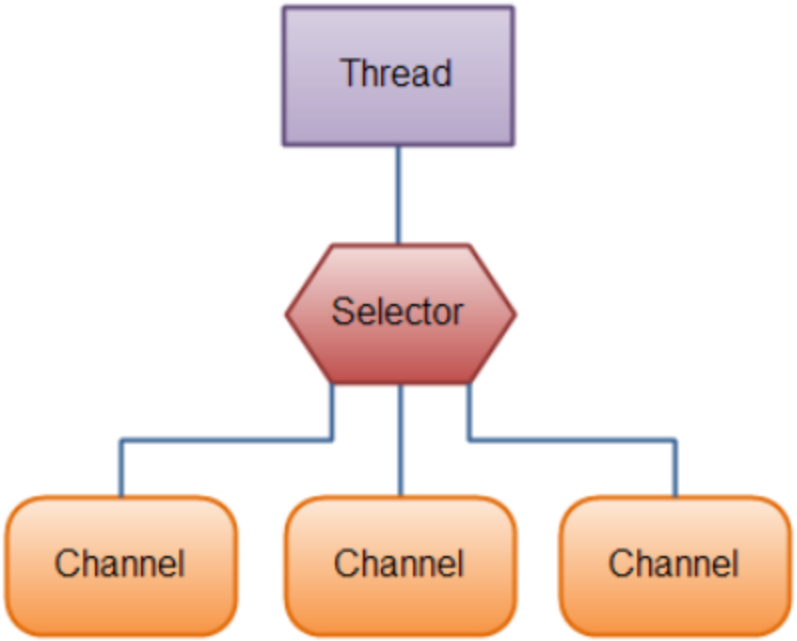
    ByteBuffer ByteBuffer.**allocateDirect**(int capacity)

  - typical usage

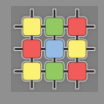
    1. Write data into the Buffer
    2. Call `buffer.flip()`
    3. Read data out of the Buffer
    4. Call `buffer.clear()` or `buffer.compact()`

Note: **compact**() – bytes between position and limit are copied to the beginning of the buffer, buffer ready for write

JAVA networking – NIO – Channel, Selector



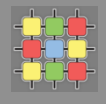
- **one thread** works with **multiple channels** at the same time
- Channel – cover UDP+TCP network IO, file IO
  - FileChannel – from Input/OutputStream or RandomAccessFile
  - DatagramChannel
  - MulticastChannel (since 1.7)
  - SocketChannel
  - ServerSocketChannel



- **Channel**
  - read/write at the same time (streams are only one-way)
  - always read/write from/to a buffer
  - channel.**transferFrom**(int pos, int count, Channel source), **transferTo** …

```
RandomAccessFile aFile = new RandomAccessFile("data/nio-data.txt", "r");
FileChannel inChannel = aFile.getChannel();

ByteBuffer buf = ByteBuffer.allocateDirect(48);

int bytesRead = inChannel.read(buf);
while (bytesRead != -1) {

  System.out.println("Read " + bytesRead);
  buf.flip();

  while(buf.hasRemaining()){
      System.out.print((char) buf.get());
  }

  buf.clear();
  bytesRead = inChannel.read(buf);
}
aFile.close();
```

JAVA networking – NIO – Selector



- **Selector**
  - Selector Selector.**open**();
  - only channels in **non-blocking** mode can be registered

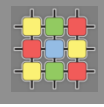
    channel.**configureBlocking**(false);

    SelectionKey channel.**register**(selector, SelectionKey.OP_READ);
  - FileChannel doesn't support non-blocking mode !

- **SelectionKey** – events you can listen for (can be combined together)
  - OP_CONNECT
  - OP_ACCEPT
  - OP_READ
  - OP_WRITE

- events are **filled automatically by channel** which is ready with operation

JAVA networking – NIO – Selector



- **SelectionKey** – returned from register method
  - interest set – your configured ops
  - ready set – which ops are ready, sk.isReadable(), sk.isWritable(), …
  - the channel
  - selector
  - optional attached object – sk.**attach**(Object obj); Object sk.**attachment**()

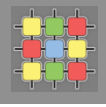
    SelectionKey channel.**register**(selector, ops, attachmentObj);

- Selector with registered one or more channels
  - int **select**() – blocks until at least one channel is ready
  - int **select**(long timeout) – with timeout milliseconds
  - int **selectNow**() – doesn't block at all, returns immediately

  - return the number of channels which are ready, need remove processed

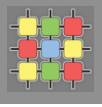
    Set<SelectionKey> selector.**selectedKeys**();

JAVA networking – NIO – Selector



```
Set<SelectionKey> selectedKeys = selector.selectedKeys();

Iterator<SelectionKey> keyIterator = selectedKeys.iterator();

while(keyIterator.hasNext()) {

    SelectionKey key = keyIterator.next();

    if(key.isAcceptable()) {
        // a connection was accepted by a ServerSocketChannel.

    } else if (key.isConnectable()) {
        // a connection was established with a remote server.

    } else if (key.isReadable()) {
        // a channel is ready for reading

    } else if (key.isWritable()) {
        // a channel is ready for writing
    }

    keyIterator.remove();
}
```

JAVA networking – NIO – Selector, SocketChannel

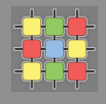


- **Selector** (cont.)
  - **wakeUp**() – different thread can "wake up" thread blocked in **select**()
  - **close**() – invalidates selector, channels are not closed

- **SocketChannel**
  - can be configured as non-blocking before connecting
  - SocketChannel socket.**getChannel**();
  - SocketChannel SocketChannel.**open**();
  - sch.**connect**(…)

  - **write**(…) and **read**(…) may return without having written/read anything for non-blocking channel !

JAVA networking – NIO

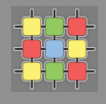


- **ServerSocketChannel**
  - can be configured as non-blocking
  - can be created directly using **open**() or from **ServerSocket**
  - **accept**() – returns **SocketChannel** in the same mode

- **DatagramChannel**
  - can be configured as non-blocking
  - can be created directly using **open**() or from **DatagramSocket**
  - **receive**(…), **send**(…)

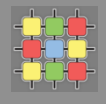
- **FileChannel**
  - **cannot be non-blocking** !
  - support – **direct buffers, mapped files**, locking

Intelligent systems



» intelligent system integrates concepts of **artificial intelligence** such as:
  - knowledge reprepresentation and expert systems
  - advanced search methods
  - mathematical reasoning methods
  - **nature inspired computing**: artificial neural networks and genetic algorithms
  - **agent-based computing**: distributed artificial intelligence and multi-agent systems

Introduction to agents #1

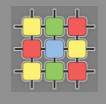


» **autonomous agents and multi-agent systems** (also referred to as agent-based computing):

- a specific sub-field of computer science and artificial intelligence,
- investigates the concepts of autonomous decision making, communication and coordination, distributed planning and distributed learning but also game-theoretic aspects of competitive behavior or logical formalization of higher level knowledge structures representing interaction attitude of actors in multi-actor environment.

A multi-agent system is a decentralized computational (software) system, often distributed (or at least open to distribution across hardware platforms) whose behavior is defined and implemented by means of complex, peer-to-peer interaction among autonomous, rational and deliberative units – agents.

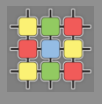
Introduction to agents #2



An **agent** is an encapsulated computational (or physical, even human) system, that is ***situated in*** some ***environment***, and that is capable of flexible, ***autonomous behavior*** in order to meet its design objective (Wooldridge, 2000). The agent can exists on its own but often is a component of a ***multi-agent system***.

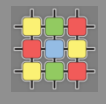
Agent technology provides a set of ***tools***, **algorithms** and **methodologies** for development of distributed, asynchronous intelligent software applications that leverage the above listed theories.

Key properties of an autonomous intelligent agent



- **Autonomy** – the agent is accountable for execution of its own actions and is not controlled from outside. Often the agent's reasoning mechanism that selects the action to be executed is unknown from outside of the agent (unlike e.g. objects).
- **Reactivity** – the agent is able react qucikly to the events in the environment and to the requests from other agents, it is able to reconsider its activity according to the change of the environment in timely fashion. Often the longest reasoning cycle of an agent needs to perform faster than the fastest change in the environment (calculative rationality).
- **Intentionality** – the agents is able to maintain its long term intention encoded by the agent's designer and is capable to consider both the long term intentions and immediate reactive inputs when selecting the next action.
- **Social capability** – the agent is able to interact, collaborate, form teams but also to perform different levels of reasoning about the other agents.

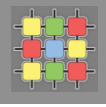
Agent design levels



Agents design levels:

- **organization-level**: related to the agent communities as a whole (organizational structure, trust, norms, obligations, self-organization, etc.);
- **interaction-level**: concern communication among agents (languages, interaction protocols, negotiations, resource allocation mechanisms);
- **agent-level**: concern individual agents (agent architecture, reasoning, learning, local processing of social knowledge).

Similar concepts



objects - computational entity with its ***encapsulated state***, ability to perform methods on the state and communicating with the other objects via message passing

» lesser **degree of autonomy** - possibility to have a public method

» joint goal is set-up at the **design-time**

» multi-agent systems are inherently **multi-threaded**

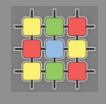
expert systems - the most important technology of the 1980's

» expert systems are **disembodied** from the environment

» expert systems are not capable of **reactive** and **proactive behavior**

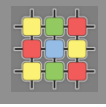
» expert systems are not equipped with the **social ability**

Agent technology domains



- **manufacturing**: planning highly complex production, control of dynamic, unpredictable, unstable processes, diagnostics, repair, reconfiguration/ replanning.
- **virtual enterprises**: forming business alliances, forming long-term/short-term deals, managing supply chains.
- **internet agents**: mainly for intelligent shopping and auctioning, information retrieval and searching, remote access to information and remote system control.
- **transport**: intelligent car, public transport, logistic and material handling, but also peace-keeping missions, military maneuvers, etc.
- **collective robotics operations**: cooperation and autonomy in the group of robotic entities (UAS, ground vehicles, unattended sensors), replacement of teleoperation with autonomous decision making
- **utility networks**: energy distribution networks, mobile operators networks, cable provider networks - simulation and predication of alarm situations, prevention to black-out and overload, intrusion detection.

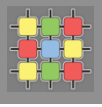
Agent architectures #1



» **Reactive agents** are agents that contain no symbolic knowledge representation (ie: no state, no representation of the environment, no representation of the other agents, ...). Their behaviour is defined by a set of perception-action rules.

$$\text{rules} \times \text{percept} \rightarrow \text{action}$$

Agent architectures #2

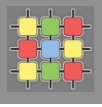


The classical approach to building agents is to view them as a particular type of knowledge-based system, and bring all the associated methodologies of such systems to bear. We define a **deliberative agent** architecture to be one that:

» contains an explicitly represented, **symbolic model of the world**;

» makes decisions (e.g. about what actions to perform) e.g. via **symbolic reasoning**.

Agent architectures #3



Belief-desire-intention (BDI) model is framework for reasoning about formal abstract models of mental states (based on Theory of Practical Reasoning).

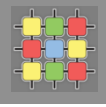
» contains representations (as objects, data structures, or whatever) of:

- **beliefs**, which constitute its knowledge of the state of its environment (and perhaps also some internal state),
- **desires**, which determine its motivation what it is trying to bring about, maintain, find out, etc.,
- **intentions**, which capture its decisions about how to act in order to fulfill its desires (committed desires)

» **intention** is something between the agents' **state of mind** (belief) and the immediate action to be performed

» unlike **desire/goal** an intention may be seen as agents immediate **commitment** to implementing an action.

Interaction schemes



From the point of view of interaction among agents we distinguish:

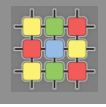
» **Open systems**

- Interaction among various types of agents from different developers
- Common understanding of messages is necessary – specification of message structure
- May act as self-interested and try to harm others/whole system
  - security issues
- Strong emphasis on interoperability

» **Closed systems**

- Interact with a predefined set of agents known to the developer in advance
- Proprietary data formats can be used
- Interoperability can be sacrificed for the sake of performance optimizations

FIPA specifications #1



Foundation for Intelligent Physical Agents - FIPA

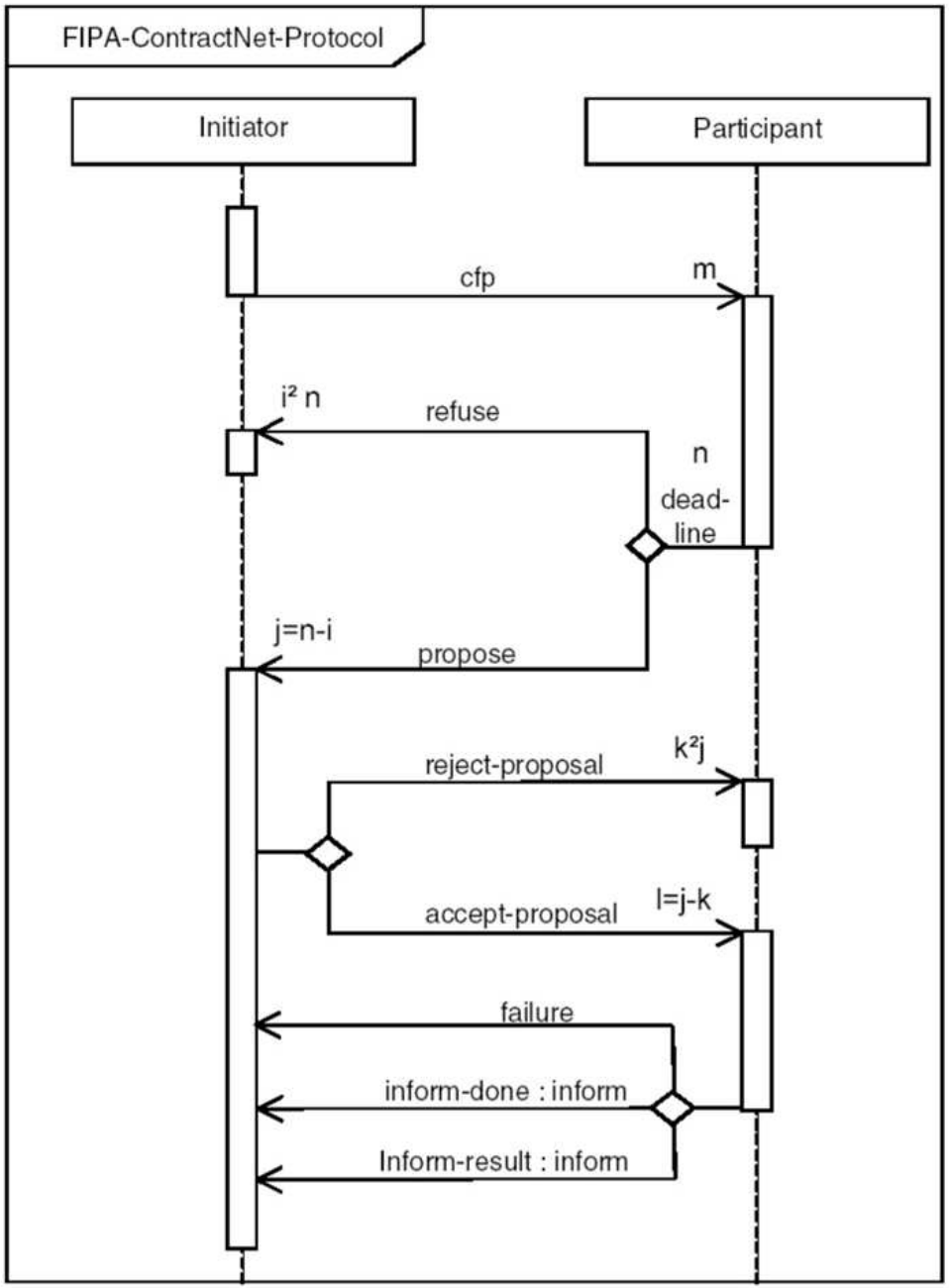
» Founded to create specification that will ***ensure interoperability among agents***

» Complete set of ***specifications*** from different categories:

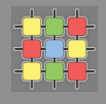
- agent communication
- agent transport
- agent management
- abstract architecture and applications

» Most significant for agent interoperability is agent communication and transport

Agent communication language



Mobile agents #1



Mobile agents are characterized by code mobility

» Mobile agents travel to places, where they perform tasks on behalf of a user

» Reason why to travel - insufficient computational power, unreliable communication, remote data source

» Security issues

» Mobility vs. cloning, stand-in agents

Types of agent mobility

» **Strong mobility** - mobility of code, data and execution state, transparent for the computational process, requires support from the OS and execution environment

» **Weak mobility** - only the code and data are transferred, intentional mobility