

Medii de proiectare și programare

2021-2022

Curs 5

Conținut curs 5

- Configurare Spring (cont.)
- Aplicații client-server
 - Șablonul Proxy
 - Data transfer object
- Networking și threading în Java
 - Exemplu Mini-Chat

Configurare automată folosind Java

- Configurarea automată (implicită) este o metodă prin care containerul Spring descoperă automat bean-rile care trebuie create, dependențele dintre acestea și încearcă să creeze aceste bean-uri.
- `@ComponentScan` specifică containerului opțiunea de configurare automată

```
package pizzeria.config;  
  
@Configuration  
@ComponentScan  
public class PizzeriaAutowireConfig {  
  
}
```

- Implicit, containerul va încerca să descopere bean-urile începând cu pachetul clasei de configurare.

```
@ComponentScan("pizzeria")  
@ComponentScan(basePackages={"pizzeria","cofetarie"})  
@ComponentScan(basePackageClasses={C.class, D.class})
```

Configurare automată folosind Java

- Declararea bean-urilor: @Component

@Component

```
public class DefaultPizzaValidator implements Validator<Pizza> {  
    public void validate(Pizza pizza) {  
        //...  
    }  
}
```

- Implicit, id-ul bean-ului este numele clasei cu prima literă transformată în literă mică.
- Bean cu id explicit:

@Component("pizzaVal")

```
public class DefaultPizzaValidator implements Validator<Pizza> {  
    public void validate(Pizza pizza) {  
        //...  
    }  
}
```

Configurare automată folosind Java

- Marcarea dependențelor: @Autowired
- Constructori, atribute, metode (set, etc.)

@Component

```
public class PizzaInMemoryRepository implements PizzaRepository {  
    private Validator<Pizza> valid;  
    @Autowired  
    public PizzaInMemoryRepository(Validator<Pizza> valid) { ... }  
    ...  
}
```

@Component

```
public class PizzaInMemoryRepository implements PizzaRepository {  
    private Validator<Pizza> valid;  
    public PizzaInMemoryRepository() { ... }  
    @Autowired  
    public void setValidator(Validator<Pizza> val) {...}  
}
```

Configurare automată folosind Java

- Marcarea dependențelor: @Autowired

@Component

```
public class PizzaInMemoryRepository implements PizzaRepository {  
    private Validator<Pizza> validator;  
    public PizzaInMemoryRepository() { ... }  
    @Autowired(required=false)  
    public void setValidator(Validator<Pizza> val) {...}  
}
```

- Dacă nu există nici un bean care să satisfacă dependență, proprietatea va rămâne neinițializată.

```
public void save(Pizza p) {  
    if (validator!=null) {  
        //...  
    }  
}
```

Configurare automată folosind Java

- @Component, @Autowired: adnotări specifice frameworkului Spring
- Dependența codului de frameworkul Spring
- @Named, @Inject : adnotări din specificația *Java Dependency Injection*
- Pachetul **javax.inject**

```
import javax.inject.Inject;  
import javax.inject.Named;
```

```
@Named
```

```
public class PizzaRepositoryMock implements PizzaRepository {  
    private Validator<Pizza> valid;  
    @Inject  
    public PizzaRepositoryMock(Validator<Pizza> val){ ... }  
    //...  
}
```

- În majoritatea cazurilor sunt interschimbabile.

Configurare automată folosind Java

- @Scope: specificarea scopului (*implicit singleton*, prototype, request, session)
- @Component, @Bean

@Component

@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)

```
public class ABean { ... }
```

@Bean

@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)

```
public ABean abean() {  
    return new ABean();  
}
```


Crearea containerului Spring Java Autowire

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.AnnotationConfigApplicationContext;

public class StartApp{
    public static void main(String[] args){
        ApplicationContext context=new
        AnnotationConfigApplicationContext(PizzeriaAutowireConfig.class);
        //obtinerea referintei catre un bean din container
        PizzaService repo= factory.getBean(PizzaService.class);
    }
}
```

Configurare automată folosind Java

- Ambiguități.

`@Autowired`

```
public void setDessert(Dessert dessert) {  
    this.dessert = dessert;  
}
```

`@Component`

```
public class Cake implements Dessert { ... }
```

`@Component`

```
public class Cookies implements Dessert { ... }
```

`@Component`

```
public class IceCream implements Dessert { ... }
```

- Care bean satisface dependența?

NoUniqueBeanDefinitionException: nested exception is

org.springframework.beans.factory.NoUniqueBeanDefinitionException:

No qualifying bean of type [com.desserteater.Dessert] is defined:

expected single matching bean but found 3: cake,cookies,iceCream

Configurare automată folosind Java

- Ambiguități - Soluția 1 - @Primary.

@Autowired

```
public void setDessert(Dessert dessert) {  
    this.dessert = dessert;  
}
```

@Component

@Primary

```
public class Cake implements Dessert { ... }
```

@Component

```
public class Cookies implements Dessert { ... }
```

@Component

```
public class IceCream implements Dessert { ... }
```

- Soluția 2 - @Qualifier
- Soluția 3 - Annotare proprie

Configurare Spring folosind Autowire

- Exemplu
 - TaskSpringAutowire

Configurare XML vs. JavaConfig vs Autowire

XML	JavaConfig	Autowire
<ul style="list-style-type: none">• Nu necesită modificarea codului sursă• Nu necesită recompilarea când apar modificări• Necesită învățarea unui nou limbaj (XML)• Se poate folosi când nu avem acces la tot codul sursă al aplicației• Nu se pot verifica tipurile bean-urilor și dependențele la compilare• Nu apar ambiguități	<ul style="list-style-type: none">• Necesită recompilare când apar modificări• Nu necesită învățarea unui nou limbaj• Se verifică static tipurile• Se poate folosi când nu avem acces la tot codul sursă al aplicației	<ul style="list-style-type: none">• Necesită recompilare când apar modificări• Nu necesită învățarea unui nou limbaj• Se verifică static tipurile• NU se poate folosi când nu avem acces la tot codul sursă al aplicației• Dependența codului sursă de Spring

Referințe Spring

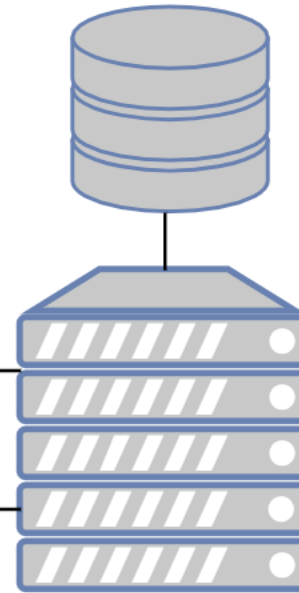
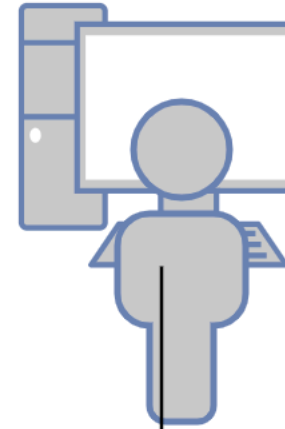
- Documentația frameworkului Spring
<http://docs.spring.io/spring-framework/docs/current/spring-framework-reference/html/index.html>
- Craig Walls, *Spring in Action*, Fourth Edition, Ed. Manning, 2015 (sau versiuni mai noi: fifth edition, sixth edition)

Aplicații client-server

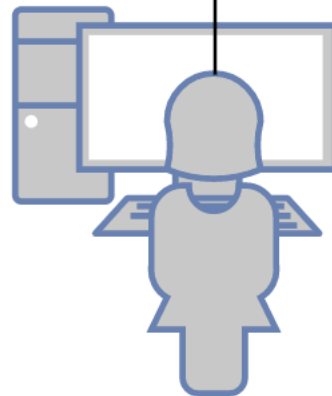
Utilizator 3



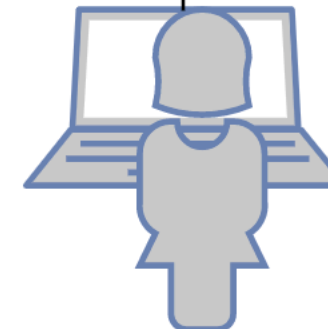
Utilizator 4



Server



Utilizator 1



Utilizator 2

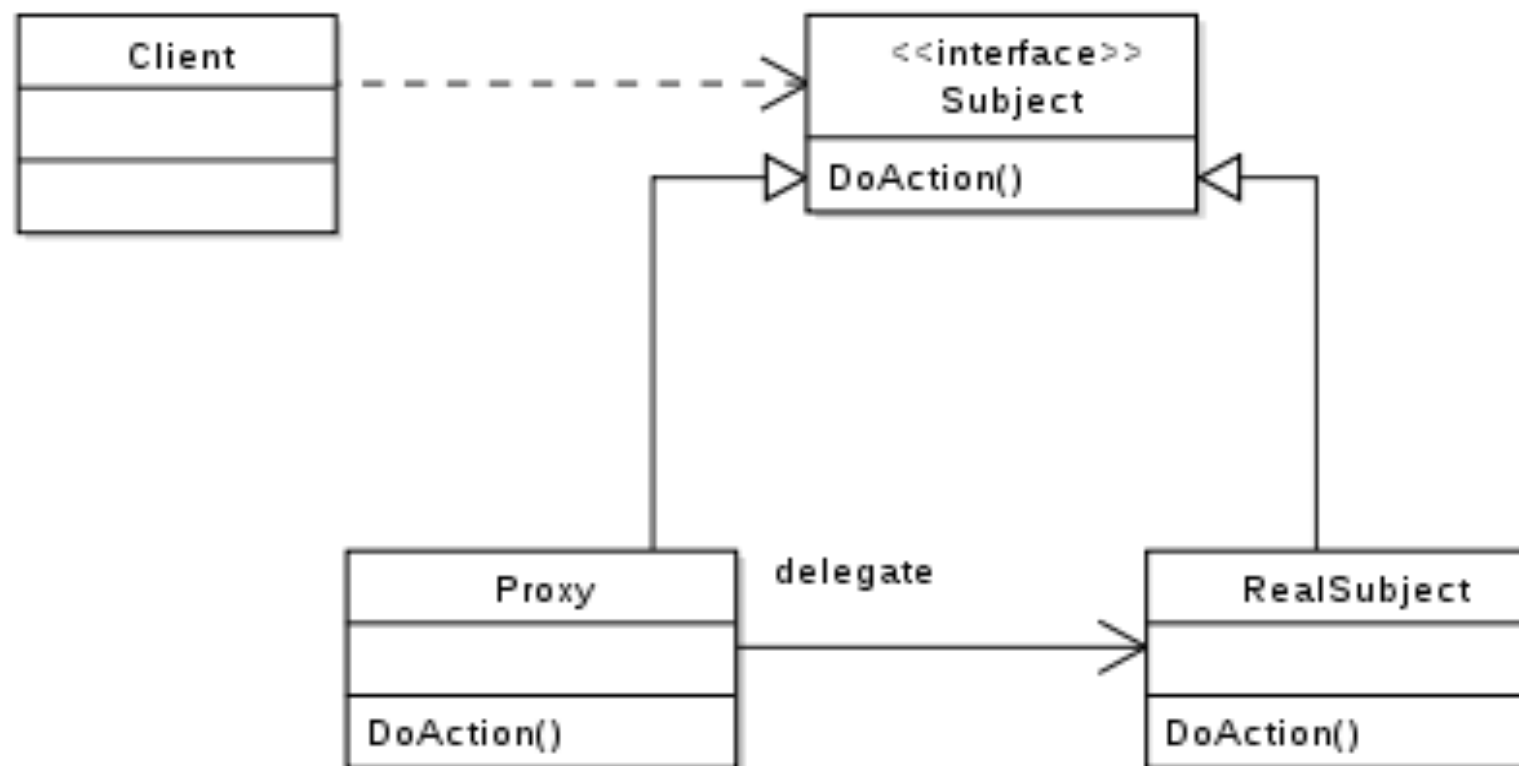
Mini-chat

- Proiectați și implementați o aplicație client-server pentru un mini-chat având următoarele funcționalități:
 - *Login*. După autentificarea cu succes, o nouă fereastră se deschide în care sunt afișați toți prietenii *online* ai utilizatorului și o listă cu mesajele trimise/primate de utilizator. De asemenea, toți prietenii online văd în lista lor că utilizatorul este *online*.
 - *Trimiterea unui mesaj*. Un utilizator poate trimite un mesaj text unui prieten care este online. După trimiterea mesajului, prietenul vede automat mesajul în fereastra lui.
 - *Logout*. Toți prietenii online ai utilizatorului văd în lista lor că utilizatorul nu mai este *online*.

Exemplu Java

Șablonul Proxy

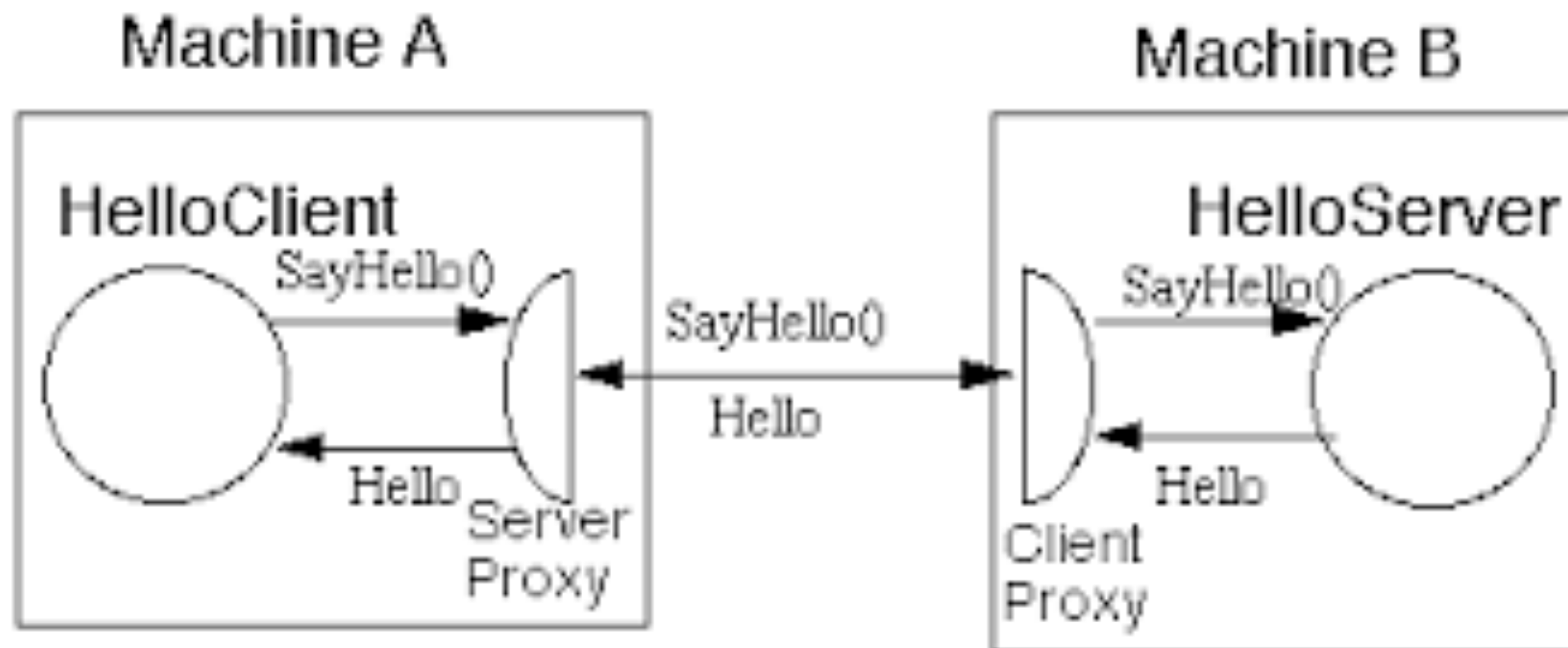
- Asigură pentru un obiect existent, un surogat sau un înlocuitor în scopul controlării accesului la acesta.
- Înlocuitorul poate fi:
 - Proxy la distanță (eng. *remote proxy*) - obiect în alt spațiu de adresă,
 - Proxy virtual (eng. *virtual proxy*) - un obiect mare din memorie,
 - Proxy de protecție - controlează accesul la obiectul original,
 - etc.



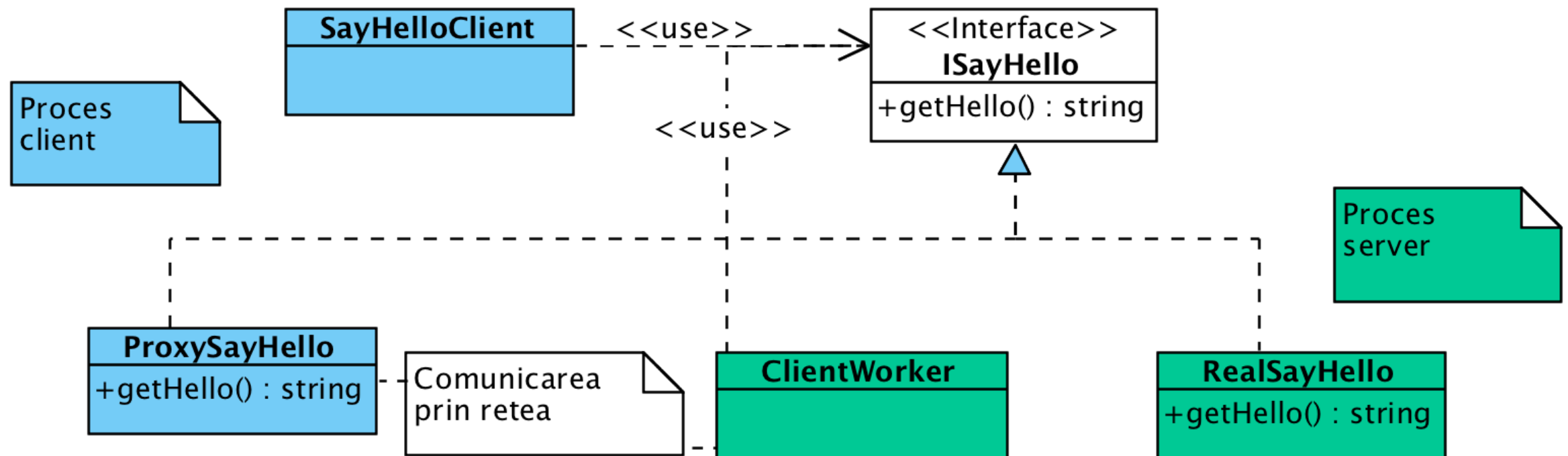
Șablonul *Remote Proxy*

- Remote Proxy oferă un înlocuitor local pentru un obiect aflat în alt spațiu de adresă/memorie.
- Înlocuitorul este responsabil cu codificarea unei cereri, a parametrilor și trimiterea lor către obiectul real aflat într-un spațiu de adresă diferit.
- Clientul cererii crede că comunică cu obiectul real, dar este un proxy între ei.
- Proxy-ul transformă cererile clientului în cereri la distanță, obține rezultatul cererii și îl transmite clientului.

Şablonul Remote Proxy



Șablonul Remote Proxy

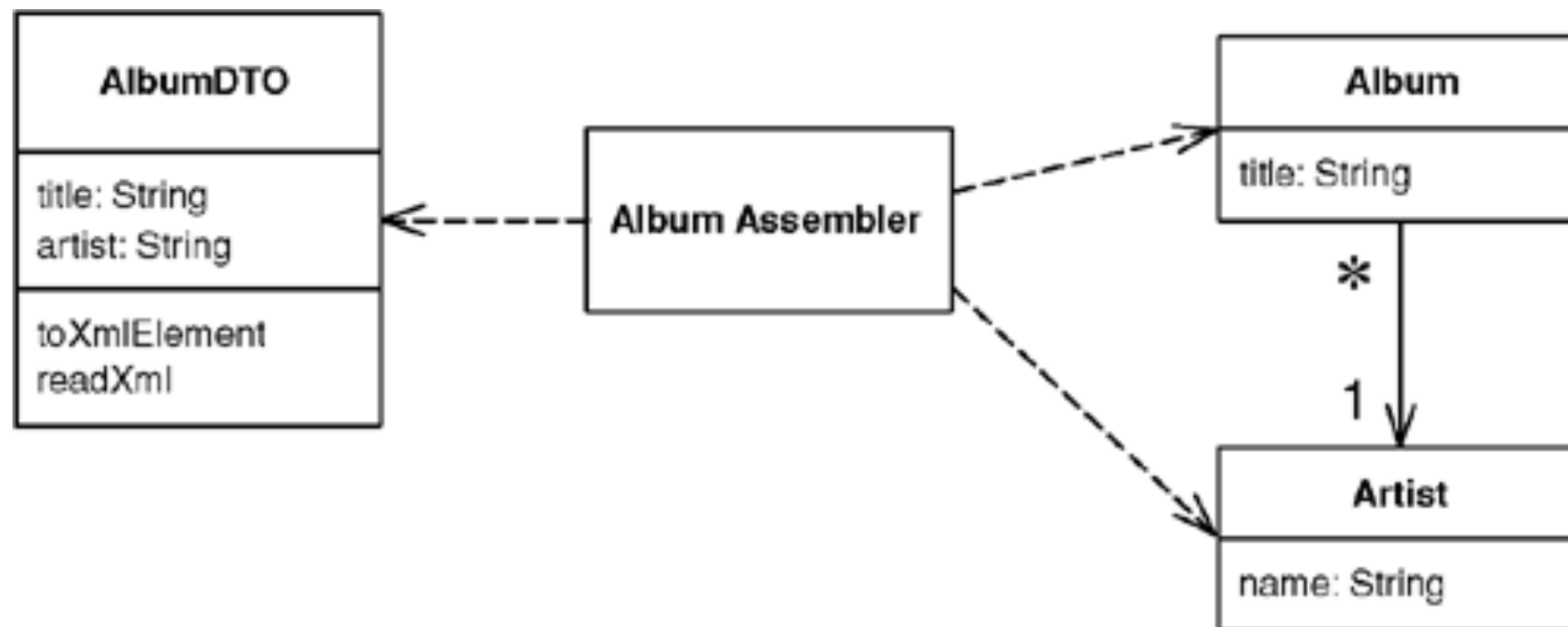


Șablonul Data Transfer Object

- Un DTO este un obiect care conține informația ce trebuie transmisă între unul sau mai multe procese pentru a reduce numărul de apeluri (dintre procese).
- Fiecare apel de metodă remote este costisitor, de aceea numărul de apeluri ar trebui redus și mai multă informație ar trebui transmisă la un apel.
- O soluție posibilă este de a folosi mai mulți parametri:
 - dificil de programat
 - în unele cazuri nu este posibil (ex., în Java o metodă poate returna o singură valoare).
- Soluția: crearea unui DTO (*Data Transfer Object*) care păstrează toată informația necesară unui apel. De obicei obiectul este serializabil (binar, XML, etc.) pentru a putea fi transmis prin rețea.
- Un alt obiect este responsabil cu conversia datelor din model într-un DTO și invers.

Șablonul Data Transfer Object

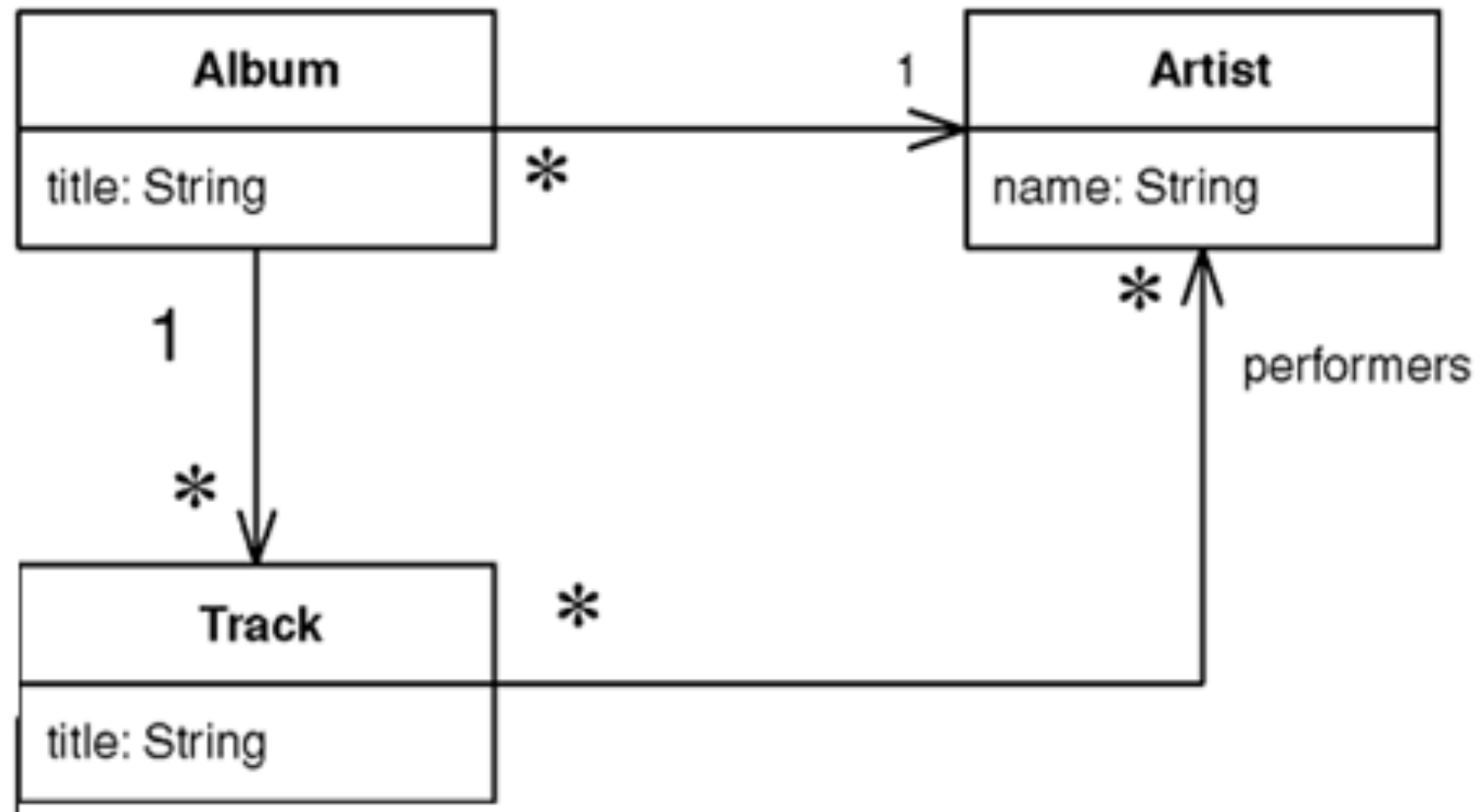
- Un DTO conține, de obicei, multe attribute și metode de tip get/set pentru acestea.
- Când un obiect remote are nevoie de date, cere DTO-ul corespunzător. DTO poate să conțină mai multă informație decât este necesar la acel apel, dar ar trebui să conțină toată informația de care va avea nevoie obiectul remote o perioadă.
- Un DTO conține de obicei informație provenind de la mai multe obiecte din model.



Șablonul Data Transfer Object

- Un DTO ar trebui folosit ori de câte ori este necesară transmiterea mai multor date între două procese într-un singur apel de metodă.
- Alternative:
 - De a folosi metode de tip set/get cu mai mulți parametri transmiși prin referință.
 - Multe limbaje (ex. Java) permit returnarea unei singure valori.
 - Alternativa poate fi folosită pentru actualizări (metode de tip set), dar nu poate fi folosită pentru a obține date (metode de tip get).
 - Folosirea unei reprezentări sub formă de string.
 - Totul va fi cuplat cu reprezentarea sub formă de string (poate fi costisitoare).

Data Transfer Object - Exemplu



Data Transfer Object - Exemplu

```
class AlbumAssembler{
    public AlbumDTO writeDTO(Album subject) {
        AlbumDTO result = new AlbumDTO();
        result.setTitle(subject.getTitle());
        result.setArtist(subject.getArtist().getName());
        writeTracks(result, subject);
        return result;
    }
    private void writeTracks(AlbumDTO result, Album subject) {
        List<TrackDTO> newTracks = new ArrayList<TrackDTO>();
        for(Track track: subject.getTracks()){
            TrackDTO newDTO = new TrackDTO();
            newDTO.setTitle(track.getTitle());
            writePerformers(newDTO, track);
            newTracks.add(newDTO);
        }
        result.setTracks(newTracks.toArray(new TrackDTO[newTracks.size()]));
    }
}
```

Data Transfer Object - Exemplu

```
private void writePerformers (TrackDTO dto, Track subject) {  
    List<String> result = new ArrayList<String>();  
    for (Artist artist: subject.getPerformers()) {  
        result.add(artist.getName());  
    }  
    dto.setPerformers(result.toArray(new String[result.size()]));  
}
```

Networking în Java

- `java.net` - pachetul conține clase pentru comunicarea TCP/UDP prin rețea.
- TCP: `Socket` și `ServerSocket`.
- UDP: `DatagramPacket`, `DatagramSocket` și `MulticastSocket`.
- Clasa `InetAddress` reprezintă o adresă IP:
 - `Inet4Address`: pentru adrese IPv4 (32 bits).
 - `Inet6Address`: pentru adrese IPv6 (128 bits).

```
InetAddress localhost=InetAddress.getLocalHost();
```

```
InetAddress googAdr=InetAddress.getByName("www.google.com");
```

- `InetSocketAddress` asociere între o adresă IP și un port:

```
InetSocketAddress(InetAddress addr, int port) ;
```

```
InetSocketAddress(String hostname, int port);
```

Networking in Java

- **ServerSocket** reprezintă clasa corespunzătoare serverului care așteaptă conexiuni TCP.

- Constructori/Metode:

```
public ServerSocket(int port) throws BindException, IOException
```

```
public ServerSocket( ) throws IOException //not bind yet, since Java 1.4
```

```
//binds a server to a port
```

```
public void bind(SocketAddress endpoint) throws IOException
```

```
//blocks and waits for clients
```

```
public Socket accept( ) throws IOException
```

```
//closes the server
```

```
public void close() throws IOException
```

Networking in Java

```
ServerSocket server=null;
try{
    server=new ServerSocket(5555) ;
    while(keepProcessing){
        Socket client=server.accept() ;
        //processing code
    }
}catch(IOException ex){
    //...
}finally{
    if(server!=null){
        try{
            server.close() ;
        }catch(IOException ex){...}
    }
}
```