

Medii de proiectare și programare

2024-2025

Curs 8

Continut

- Remote Procedure Call
 - Protobuf (cont.)
 - Exemplu Mini-chat
 - gRPC
 - Thrift
- ORM

Pachete

- Opțional se poate specifica pachetul corespunzător tipurilor de mesaje definite într-un fișier .proto pentru a evita coliziuni de nume.

```
package foo.bar;  
message Open { ... }
```

- Numele pachetului poate fi utilizat ulterior la definirea tipului câmpurilor:

```
message Foo {  
    ...  
    foo.bar.Open open = 1;  
    ...  
}
```

- Modul în care specificarea unui pachet afectează codul generat depinde de limbaj:
 - În Java, pachetul este folosit ca și pachet Java, dacă nu se specifică explicit un alt pachet folosind opțiunea `option java_package` în fișierul .proto.
 - În C#, pachetul este folosit ca și spațiu de nume după transformarea la PascalCase, dacă nu se specifică explicit un alt pachet folosind opțiunea `csharp_namespace` în fișierul .proto. (Exemplu, câmpul Open va face parte din spațiul de nume Foo.Bar).

Definirea serviciilor RPC

- Se pot defini servicii RPC într-un fișier `.proto`, iar compilatorul va genera codul corespunzător interfeței și stub-urilor (proxies) în limbajul ales.

```
service SearchService {  
    rpc Search (SearchRequest) returns (SearchResponse);  
}
```

- gRPC - tehnologie open-source cross-platform RPC dezvoltată de Google care folosește protocol buffers și permite generarea codului RPC direct din fișierul `.proto` dacă se folosește un plugin adițional pentru compilatorul protoc.
- Există și alte tehnologii RPC bazate pe Protocol Buffers care pot fi utilizate pentru RPC

Alte opțiuni

- Într-un fișier .proto se pot declara opțiuni individuale care nu modifică semnificația tipurilor de mesaje definite în fișier, dar influențează modul în care este generat codul sursă corespunzător unui anumit limbaj.
- Lista completă a opțiunilor disponibile: [google/protobuf/descriptor.proto](https://github.com/google/protobuf/blob/master/src/descriptor.proto).
- Cele mai folosite opțiuni:
 - **java_package** (nivel fișier): Numele pachetului care va fi folosit la generarea claselor Java. Dacă nu se specifică explicit, numele implicit al pachetului va fi cel specificat folosind opțiunea package.
`option java_package = "com.example.foo";`
 - **java_outer_classname** (nivel fișier): Numele clasei folosite ca și clasă exterioară pentru tipurile mesajelor. Dacă nu se specifică explicit opțiunea **java_outer_classname**, numele clasei se va construi prin transformarea numelui fișierului .proto la camel-case (**foo_bar.proto** va deveni **FooBar.java**).
`option java_outer_classname = "FooBar";`

Alte opțiuni

- `java_multiple_files` (nivel fișier): Determină generarea tipurilor mesajelor, a enumerărilor și a serviciilor la nivel de pachet (nu într-o clasă exterioară) ***în fișiere separate***.

```
option java_multiple_files = true;
```

- Opțiuni C#

```
option (google.protobuf.csharp_file_options).namespace =  
"MyCompany.MyProject";
```

```
option (google.protobuf.csharp_file_options).umbrella_classname =  
"ProjectProtos";
```

Generarea claselor Java/C#

- Generarea claselor Java, C#, Python, C++, etc. se face folosind compilatorul **protoc** asupra fișierului .proto.

```
protoc --proto_path=IMPORT_PATH --java_out=DST_DIR path/to/  
file(s).proto
```

```
protoc --proto_path=IMPORT_PATH --csharp_out=DST_DIR path/to/  
file(s).proto
```

- **IMPORT_PATH** specifică directorul în care compilatorul va căuta fișierele .proto când încearcă să rezolve opțiunile *import*.
 - Dacă nu este specificat, se folosește directorul curent.
 - Se poate folosi opțiunea **-I=IMPORT_PATH** ca și o formă prescurtată a opțiunii **--proto_path**.

Clase generate (Java)

- Pentru fiecare tip de **mesaj *M*** definit într-un fișier `.proto`, compilatorul va genera **clasa *M***.
- Fiecare clasa are un Builder asociat prin intermediul căruia se vor crea instanțe ale clasei respective.
- Clasa *M* și Builder-ul asociat au metode de tip *get* generate automat de compilatorul *protoc*. Clasa Builder are și metode de tip *set* care permit setarea valorilor.
- Clasele corespunzătoare mesajelor generate de compilator sunt ***immutable***.
- După ce un obiect a fost construit, el nu mai poate fi modificat.
- Pentru a construi un mesaj, întâi se construiește builder-ul, se setează valorile câmpurilor folosind builder-ul, iar apoi se apelează metoda `build()` pentru a obține mesajul.
- Fiecare metodă din builder returnează un builder (obiectul *this*) care permite legarea mai multor apeluri de metode `set` într-o singură linie de cod.

Clase generate

- Fiecare mesaj și builder conțin metode care permit verificarea și transformarea mesajului:
 - `toByteArray() : byte[]` serializează mesajul și returnează un șir de octeți conținând mesajul serializat.
 - `parseFrom(byte[] data) : Person` parsează un mesaj dintr-un șir de octeți.
 - `writeTo(OutputStream output)` serializează mesajul și îl scrie într-un `OutputStream`.
 - `writeDelimitedTo(OutputStream output)` serializează mesajul și scrie *dimensiunea* lui și *mesajul* serializat într-un `OutputStream`.
 - `parseFrom(InputStream input) : Person` citește și parsează un mesaj dintr-un `InputStream`.
 - `parseDelimitedFrom(InputStream input) : Person` citește și parsează un mesaj dintr-un `InputStream` (mesajul a fost *scris folosind writeDelimitedTo*).

Exemplu mini-chat (proto2)

```
syntax="proto2";
package chat.protocol;
option java_package = "chat.protocol.protobuf";
option java_outer_classname = "ChatProtobufs";
message User{
    required string id=1;
    optional string passwd=2;
}
message Message{
    required string receiverId=1;
    required string senderId=2;
    required string text=3;
}
message ChatRequest {
    enum Type { Login = 1; Logout = 2; SendMessage = 3; GetLoggedFriends=4 ;}
    // Identifies which request is filled in.
    required Type type = 1;
    // One of the following will be filled in, depending on the type.
    optional User user = 2;
    optional Message message = 3;
}
message ChatResponse{
    enum Type { Ok = 1; Error = 2; GetLoggedFriends=3; FriendLoggedIn = 4; FriendLoggedOut=5; NewMessage=6; }
    // Identifies which request is filled in.
    required Type type = 1;
    // One of the following will be filled in, depending on the type.
    optional string error = 2;
    repeated User friends=3;
    optional User user=4 ;
    optional Message message = 5;
}
```

Exemplu mini-chat (proto3)

```
syntax="proto3";
package chat.protocol;
option java_package = "chat.network.protobufprotocol";
option java_outer_classname = "ChatProtobufs";
message User{
    string id=1;
    string passwd=2;
}
message Message{
    string receiverId=1;
    string senderId=2;
    string text=3;
}
message ChatRequest {
    enum Type {Unkown=0; Login = 1; Logout = 2; SendMessage = 3; GetLoggedFriends=4 ;}
    // Identifies which request is filled in.
    Type type = 1;
    // One of the following will be filled in, depending on the type.
    oneof payload{
        User user = 2;
        Message message = 3;
    }
}
message ChatResponse{
    enum Type { Unknown=0; Ok = 1; Error = 2; GetLoggedFriends=3; FriendLoggedIn = 4; FriendLoggedOut=5; NewMessage=6;}
    // Identifies which request is filled in.
    Type type = 1;
    // One of the following will be filled in, depending on the type.
    string error = 2;
    repeated User friends=3;
    User user=4 ;
    Message message = 5;
}
```

Bibliografie

- Protocol Buffers

<https://developers.google.com/protocol-buffers/>

<https://github.com/google/protobuf>

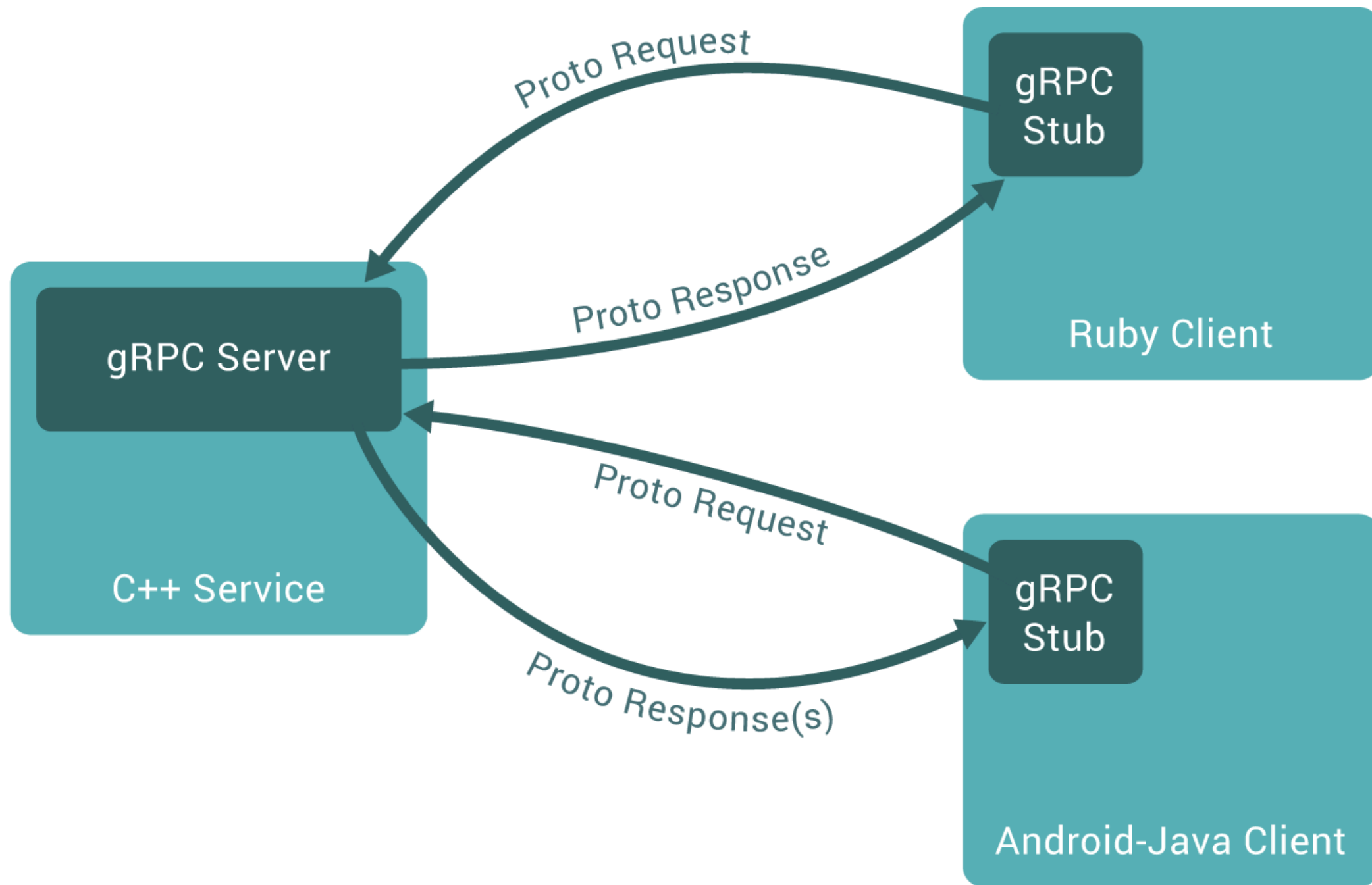
Protocol Buffers

- Exem plu MiniChat

gRPC

- Cu **gRPC** o aplicație client poate apela metode la distanță ca și cum ar fi metode ale unui obiect local.
- Facilitează crearea aplicațiilor distribuite și a serviciilor.
- gRPC folosește ideea definirii unui serviciu (interfețe) care specifică metodele ce pot fi apelate la distanță, împreună cu parametrii și tipul returnat al acestora.
- În aplicația server există un obiect remote care implementează interfața, iar serverul gRPC gestionează cererile clienților.
- Clientul (aplicația client) are un stub (proxy) care oferă aceleași metode ca și obiectul remote.
- Clienții și serverele gRPC pot fi scrise și pot rula în diferite limbaje de programare: Java, C#, C++, Python, Go, etc.

gRPC



gRPC - Definirea unui serviciu

- Implicit, gRPC folosește Protocol Buffers ca și limbaj de definire a tipurilor de mesaje și a serviciilor (IDL).
- Se pot folosi alte IDL-uri.

```
service HelloService {  
    rpc SayHello (HelloRequest) returns (HelloResponse);  
}
```

```
message HelloRequest {  
    string greeting = 1;  
}
```

```
message HelloResponse {  
    string reply = 1;  
}
```


gRPC - Definirea unui serviciu

- gRPC permite specificarea a 4 tipuri de apeluri de metode la distanță:
 - *unare* - clientul trimite o singură cerere la server și primește un singur răspuns de la server (apeluri de funcții normale).

```
rpc SayHello (HelloRequest) returns (HelloResponse) {}
```

- *server streaming* - clientul trimite o cerere la server și primește un *stream* cu ajutorul căruia poate citi o secvență de răspunsuri de la server. Clientul citește răspunsurile până când nu mai sunt mesaje pe stream.

```
rpc LotsOfReplies (HelloRequest) returns (stream HelloResponse) {}
```

- *client streaming* - clientul scrie o secvență de mesaje și le trimite la server folosind streamul furnizat. După ce clientul a terminat scrierea mesajelor, așteaptă ca serverul să le citească și să trimită un singur răspuns.

```
rpc LotsOfGreetings (stream HelloRequest) returns (HelloResponse) {}
```

gRPC - Definirea unui serviciu

- gRPC permite specificarea a 4 tipuri de apeluri de metode la distanță:
 - *Bidirectional streaming* - atât serverul cât și clientul trimit o secvență de mesaje folosind un stream bidirecțional (read-write). Streamurile (read-write) funcționează independent; clienții și serverul pot citi și scrie în ordinea dorită de ei.

Exemple:

-serverul poate aștepta până primește toate mesajele de la client înainte de a trimite răspunsurile;

-serverul poate citi un mesaj, trimite răspunsul, sau altă combinație de read-write.

Ordinea mesajelor din fiecare stream se păstrează.

```
rpc BidiHello(stream HelloRequest) returns (stream HelloResponse) {}
```

gRPC - Definirea unui serviciu

- Pornind de la definiția unui serviciu dintr-un fișier `.proto`, gRPC furnizează **plugin-uri** pentru compilatorul protoc care permit generarea codului corespunzător clienților și serverului.
- Clienții și serverul gRPC folosesc acest API pentru implementarea funcționalității dorite:
 - În aplicația server, serverul implementează metodele declarate de serviciu, și le face disponibile clienților folosind un server gRPC. Frameworkul gRPC decodifică cererile, execută metodele remote și codifică răspunsurile.
 - În aplicația client, clientul are un obiect local (stub, proxy) care implementează aceleași metode ca și serviciul.
 - Clientul poate apela aceste metode ca și cum ar fi apeluri locale, folosind parametrii corespunzători.

gRPC - Timeout

- gRPC permite clienților să specifice cât timp sunt dispuși să aștepte ca un apel la distanță să își încheie execuția. Dacă durată specificată a fost depășită, apelul la distanță se va încheia cu eroarea `DEADLINE_EXCEEDED`.
- Pe server se poate verifica dacă un anumit apel la distanță a depășit durată sau cât timp mai are la dispoziție.
- Atât clientul cât și serverul pot anula execuția unui apel la distanță în orice moment. Anularea duce la oprirea imediată a execuției apelului la distanță.
- Anularea nu este o operație de tip “undo”: modificările făcute înainte de anulare nu vor fi anulate și ele.

gRPC - Terminare, canale de comunicare

- Terminarea unui apel - clientul și serverul determină independent succesul execuției unui apel la distanță, iar concluziile lor pot să difere.
 - Exemplu 1: un apel RPC se poate termina cu succes pe server (“I have sent all my responses!”), dar poate eșua pe client (“The responses arrived after my deadline!”).
 - Exemplu 2: serverul poate decide terminarea înainte ca clientul să fi terminat de trimis toate cererile.
- Canal de comunicare gRPC - furnizează o conexiune la un server gRPC specificând adresa și portul și se folosește la instanțierea unui proxy (client stub). Clienții pot specifica parametrii pentru a modifica comportamentul implicit (setarea opțiunii de compresie a mesajelor, etc).
- Un canal are asociată o stare (dacă este conectat, dacă este idle, etc.)

gRPC - Exemplu

- O aplicație simplă client/server:
 - Obiectul remote are o metodă care transformă un text în text cu litere mari și adaugă data și ora la care a fost primit textul.
 - Clientul obține o referință la obiectul remote, apelează metoda și tipărește rezultatul primit.

Bibliografie

- Protocol Buffers

<https://developers.google.com/protocol-buffers/>

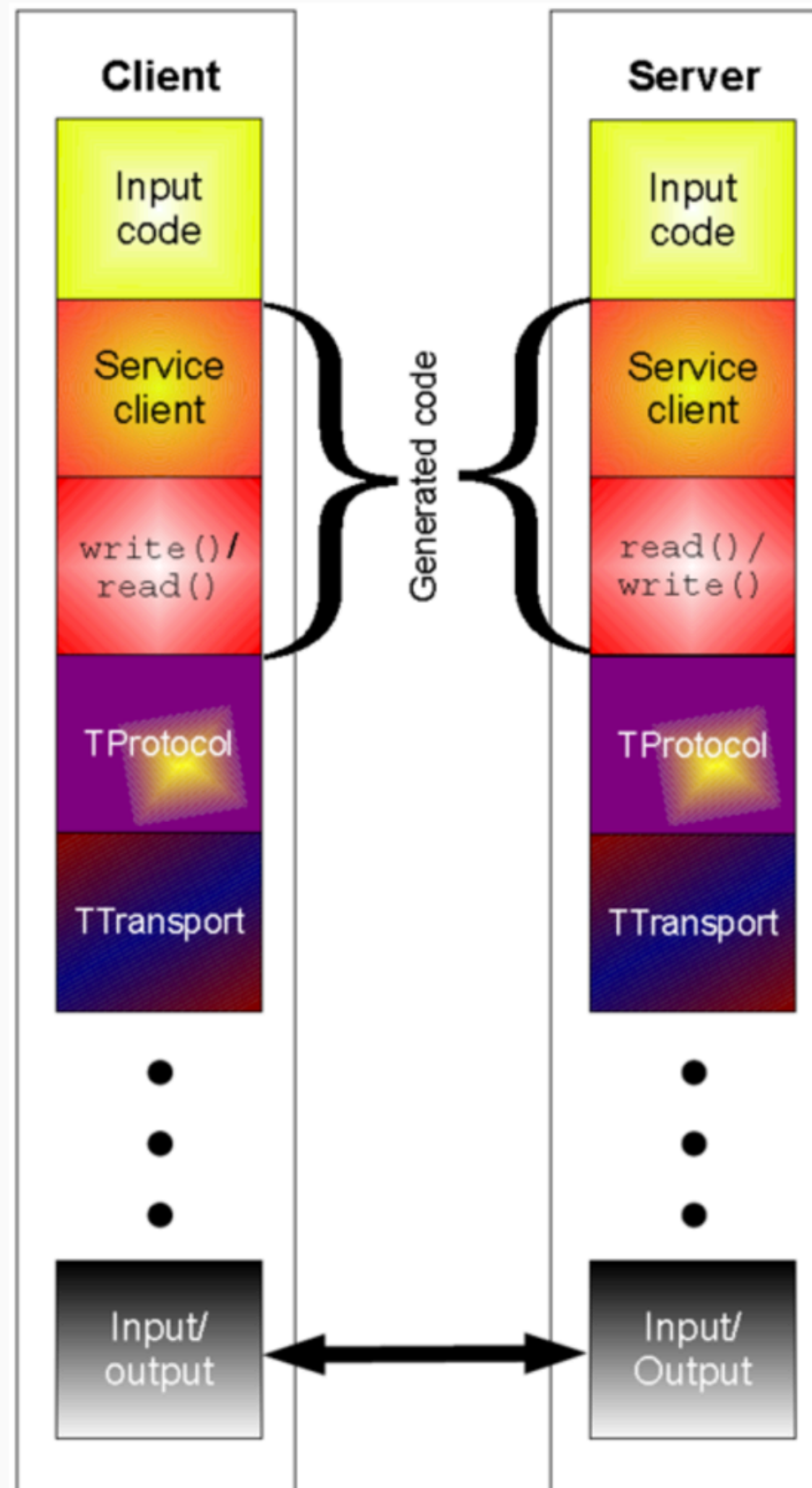
<https://github.com/google/protobuf>

- gRPC

<http://www.grpc.io/>

Apache Thrift

- **Thrift** este o tehnologie cross-platform pentru RPC.
- Thrift oferă separare clară între nivelul transport, serializarea datelor și logica aplicației.
- Thrift a fost dezvoltat inițial de Facebook, acum este un proiect open-source găzduit de Apache.
- Apache Thrift conține un set de unelte de generare de cod care permit utilizatorilor să dezvolte clienți și servere RPC doar prin definirea tipurilor de date necesare și a interfețelor serviciilor într-un fișier IDL **.thrift**.
- Folosind acest fișier ca și date de intrare, se generează automat cod sursă pentru clienți și servere RPC în diferite limbaje de programare care permit aplicațiilor să comunice indiferent de limbajul folosit pentru scrierea clientului/serverului.
- Thrift suportă diferite limbaje de programare: C++, Java, C#, Python, PHP, Ruby.



Thrift Architecture.

Apache Thrift

- Componente cheie:

- tipuri de date
- transport
- protocol
- versioning
- processors

- Tipuri de date Thrift

- Sistemul de tipuri de date Thrift nu introduce tipuri speciale dinamice sau obiecte wrapper.
- Programatorul nu trebuie să scrie cod pentru serializarea tipurilor de date sau pentru transportul datelor.
- Conține tipuri de bază, structuri, containere.
- Tipuri de bază:
 - *bool* - valoarea booleană: true sau false
 - *byte* - a signed byte
 - *i16* - întreg cu semn pe 16-biți
 - *i32* - întreg cu semn pe 32-biți
 - *i64* - întreg cu semn pe 64-biți
 - *double* - real pe 64-biți
 - *string*
 - *binary* - șir de octeți folosit pentru reprezentarea blob-urilor.

Tipuri Thrift -Structuri

- O structură Thrift definește un obiect comun tuturor limbajelor ce poate fi transmis prin rețea.
- O structură este echivalentul unei clase dintr-un limbaj orientat obiect.
- Structura conține o mulțime de câmpuri care au un tip definit și un identificator unic asociat.
- Sintaxa asemănătoare structurii C.
- Câmpurile pot fi adnotate cu un identificator numeric unic (valoare întreagă) și o valoare implicită (opțional). Identificatorii trebuie să fie unici în cadrul structurii.
- Dacă identificatorii câmpurilor sunt omiși, li se vor atribui automat valori.

```
struct Example {  
    1:i32 number=10,  
    2:i64 bigNumber,  
    3:double decimals,  
    4:string name="thrifty"  
}
```

Tipuri Thrift -Containere

- Containerele Thrift sunt containere cu tip care se vor mapa la cele mai folosite containere din limbajele de programare corespunzătoare.
- Sunt parametrizate folosind stilul Java Generics/C++ template.
- Trei tipuri de containere disponibile:
 - **list<type>** : o listă de elemente.
 - Se mapează la STL **vector**, Java **ArrayList**, sau orice alt tablou din limbajele scripting. Poate conține duplicate.
 - **set<type>** : O mulțime neordonată de elemente.
 - Se mapează la STL **set**, Java **HashSet**, **set** în Python, sau dicționare native în PHP/Ruby.
 - **map<type1 , type2>** :Un dicționar cu chei unice
 - Se mapează la STL **map**, Java **HashMap**, PHP associative array, sau Python/Ruby dictionary.
- Elementele din container pot fi de orice tip valid Thrift, inclusiv alte containere sau structuri.
- În codul generat corespunzător limbajului de programare dorit, fiecare definiție va conține două metode, **read** și **write**, folosite pentru serializarea și transportul obiectelor folosind Thrift TProtocol.

Thrift - Excepții

- Excepțiile Thrift sunt sintactic și funcțional echivalente cu structurile Thrift doar că sunt declarate folosind `exception` în loc de `struct`.
- Clasele generate vor moșteni din clasele de bază corespunzătoare excepțiilor în limbajul de programare folosit, pentru a se putea folosi în mod transparent cu mecanismul de tratare a excepțiilor din limbajul respectiv.

Servicii Thrift

- **Serviciile Thrift** se definesc folosind tipurile Thrift.
- Definirea unui serviciu este echivalentă semantic cu **definirea unei interfețe** într-un limbaj de programare orientat pe obiecte.
- **Compilerul Thrift** va genera stub-uri client și server complet funcționale care implementează interfața.
- **Lista parametrilor și lista excepțiilor** sunt implementate ca și structuri Thrift.

```
service <name> {  
    <returntype> <name>(<arguments>) [throws (<exceptions>)]  
    ...  
}
```

```
service StringCache {  
    void set(1:i32 key, 2:string value),  
    string get(1:i32 key) throws (1:KeyNotFound knf),  
    void delete(1:i32 key)  
}
```

Nivelul transport Thrift

- Nivelul transport este folosit de codul generat automat pentru a ușura transmiterea datelor între clienți și server.
- Thrift se folosește de obicei peste TCP/IP ca și nivel de bază pentru comunicare.
- Codul generat Thrift trebuie să știe doar cum să scrie și cum să citească datele.
- Originea sau destinația datelor sunt irelevante: poate fi socket, memorie partajată sau un fișier pe discul local.
- Interfața Thrift **TTransport** conține metodele:
 - **open** deschiderea transportului
 - **close** închiderea transportului
 - **isOpen** verifică dacă transportul este deschis
 - **read** citește date
 - **write** scrie date
 - **flush** forțează scrierea datelor păstrate în zona tampon.

Nivelul transport Thrift

- Interfață **TServerTransport** folosită pentru crearea și acceptarea obiectelor transport:
 - **open** deschidere
 - **listen** așteaptă conexiuni de la clienți
 - **accept** returnează un nou obiect transport (când s-a conectat un client nou)
 - **close** închidere
- Interfața transport este proiectată pentru implementare ușoară în orice limbaj de programare.
- Se pot defini noi mecanisme de transport:
 - Clasa **TSocket** este implementată în toate limbajele de programare suportate. Oferă o modalitate simplă de comunicare cu un socket TCP/IP.
 - Clasa **TFileTransport** este o abstractizare a unui stream ce reprezintă un fișier de pe discul local. Poate fi folosită pentru a salva cererile clienților într-un fișier de pe disc.

Thrift - Protocol

- Thrift cere respectarea unei anumite structuri a mesajelor când sunt transmise prin rețea, dar **nu știe de protocolul efectiv folosit pentru serializarea/codificarea acestora.**
- Nu contează dacă datele sunt serializate folosind XML, human-readable ASCII (stringuri) sau octeți, dacă mecanismul suportă o mulțime de operații prestabilite care permit citirea și scrierea datelor.
- Interfața Thrift Protocol suportă:
 - *trimiterea mesajelor bidirectional,*
 - *codificarea tipurilor de bază, a structurilor și a containerelor.*
- **Are o implementare care folosește un protocol binar.**
- **Scrie toate datele într-un format binar:**
 - **Tipurile întregi** sunt **convertite într-un format rețea independent de limbaj.**
 - **Stringurile** au **adăugate la început lungimea lor în număr de octeți.**
 - **Mesajele și antetul câmpurilor** sunt **scrise folosind construcții de serializare a datelor întregi.**
 - **Numele câmpurilor nu sunt serializate,** **identificatorii asociați** sunt **suficienți pentru reconstruirea datelor.**

Thrift - Protocol

```
writeMessageBegin(name, type, seq)
writeMessageEnd()
writeStructBegin(name)
writeStructEnd()
writeFieldBegin(name, type, id)
writeFieldEnd()
writeMapBegin(ktype, vtype, size)
writeMapEnd()
...
name, type, seq = readMessageBegin();
readMessageEnd()
name = readStructBegin()
readStructEnd()
name, type, id = readFieldBegin()
readFieldEnd()
k, v, size = readMapBegin()
readMapEnd()
...
```

Thrift - Versioning

- Thrift este **robust** la schimbări de versiuni și de definiție a datelor.
- **Versioning** este implementat folosind identificatorii asociați câmpurilor:
 - Antetul unui câmp dintr-o structură Thrift este codificat folosind identificatorul unic.
 - Combinația (identificator câmp, tip câmp) este folosită pentru a identifica unic un câmp din structură.
- Limbajul IDL Thrift permite atribuirea automată de identificatori pentru câmpuri, dar se recomandă definirea explicită a acestora.
- Dacă la parsare/decodificare/deserializare se întâlnește un câmp necunoscut acesta este **ignorat** și **distrus**.
- Dacă un câmp care ar trebui să existe nu apare, programatorul poate fi notificat de lipsa acestuia (folosind structura **isset** definită în interiorul fiecărei obiect).
- Obiectul **isset** din interiorul unei structuri Thrift poate fi interogat pentru a determina existența unui anumit câmp. De fiecare dată când se primește o instanță a unei structuri, programatorul ar trebui să verifice existența unui câmp înainte de folosirea lui.

Thrift - Implementare RPC

- Instanțe a clasei `TProcessor` sunt folosite pentru a trata cererile RPC:

```
interface TProcessor {  
    bool process(TProtocol in, TProtocol out) throws TException  
}
```
- Pentru fiecare serviciu dintr-un fișier `.thrift` se generează următoarele:
 - o interfață `ServiceIf` corespunzătoare serviciului
 - clasa `ServiceClient` implementează `ServiceIf`
`TProtocol in`
`TProtocol out`
 - clasa `ServiceProcessor` : `TProcessor`
`ServiceIf handler`
 - clasa `ServiceHandler` implementează `ServiceIf`
 - `TServer(TProcessor processor,`
`TServerTransport transport,`
`TTransportFactory tfactory,`
`TProtocolFactory pfactory)`
`serve()`

Thrift - Implementare RPC

- Serverul încapsulează logica corespunzătoare conexiunii, threadurilor, etc, iar obiectul de tip TProcessor se ocupă de apelul metodelor la distanță.
- Programatorul trebuie să scrie doar codul din fișierul .thrift și implementarea corespunzătoare serviciilor (ServiceHandler)
- Există mai multe implementări posibile pentru TServer:
 - TSimpleServer: server secvențial
 - TThreadedServer: server concurent (se creează câte un thread pentru fiecare conexiune)
 - TThreadPoolServer: server concurent care folosește un container de threaduri.
-

Bibliografie

- Apache Thrift

<http://thrift-tutorial.readthedocs.io/en/latest/index.html>

<https://thrift.apache.org/>

- Exem plu: Text transformer

Object/Relational Mapping (ORM)

- *Object-relational mapping* (ORM, O/RM sau O/R mapping) este o tehnică de programare pentru convertirea informațiilor/tipurilor dintr-un sistem orientat-obiect într-o bază de date relațională.
- Principiul mapării obiect-relație/înregistrare este de a delega altor instrumente managementul persistenței și de a lucra doar cu entitățile din domeniu, nu cu structurile dintr-o bază de date relațională.
- Instrumentele de mapare obiect-relație stabilesc o legătură bidirecțională între o bază de date relațională și obiectele din sistem, pe baza unei configurații și execută interogări SQL la baza de date (interogări construite dinamic).

Terminologie

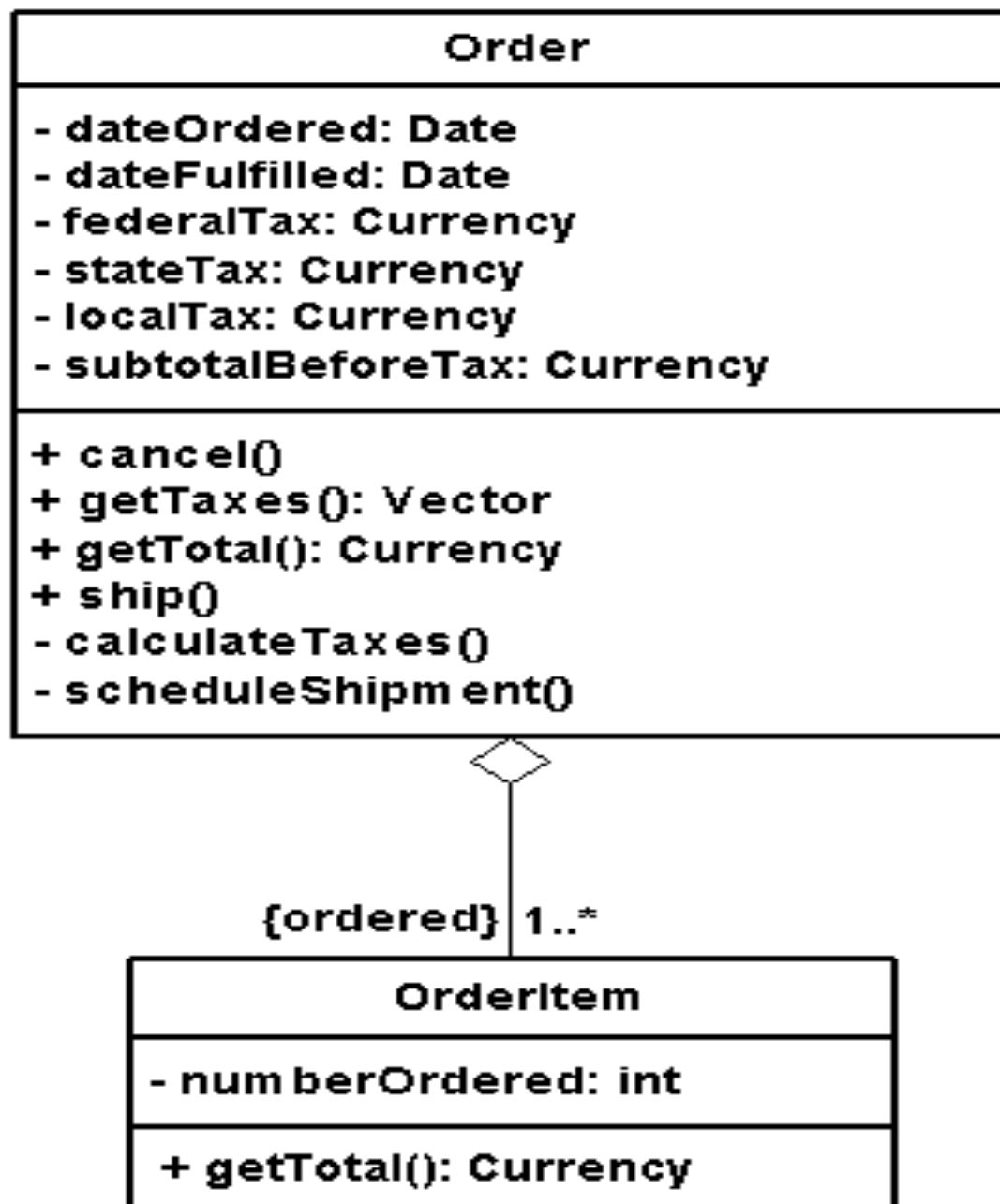
- *Mapare*. Determinarea modului în care obiectele și relațiile dintre ele vor fi păstrate într-un mediu de stocare permanent (ex. bază de date relațională, fișiere XML).
- *Proprietate*. O proprietate care poate avea asociat un atribut `firstName:string` sau o metodă prin care se determină valoarea `getTotal()`.
- *Maparea proprietății*. O mapare care descrie cum va fi stocată valoarea proprietății.
- *Maparea relațiilor*. O mapare care descrie cum vor fi persistate relațiile dintre unul sau mai multe obiecte (asociere, agregare, moștenire).

Mapări simple

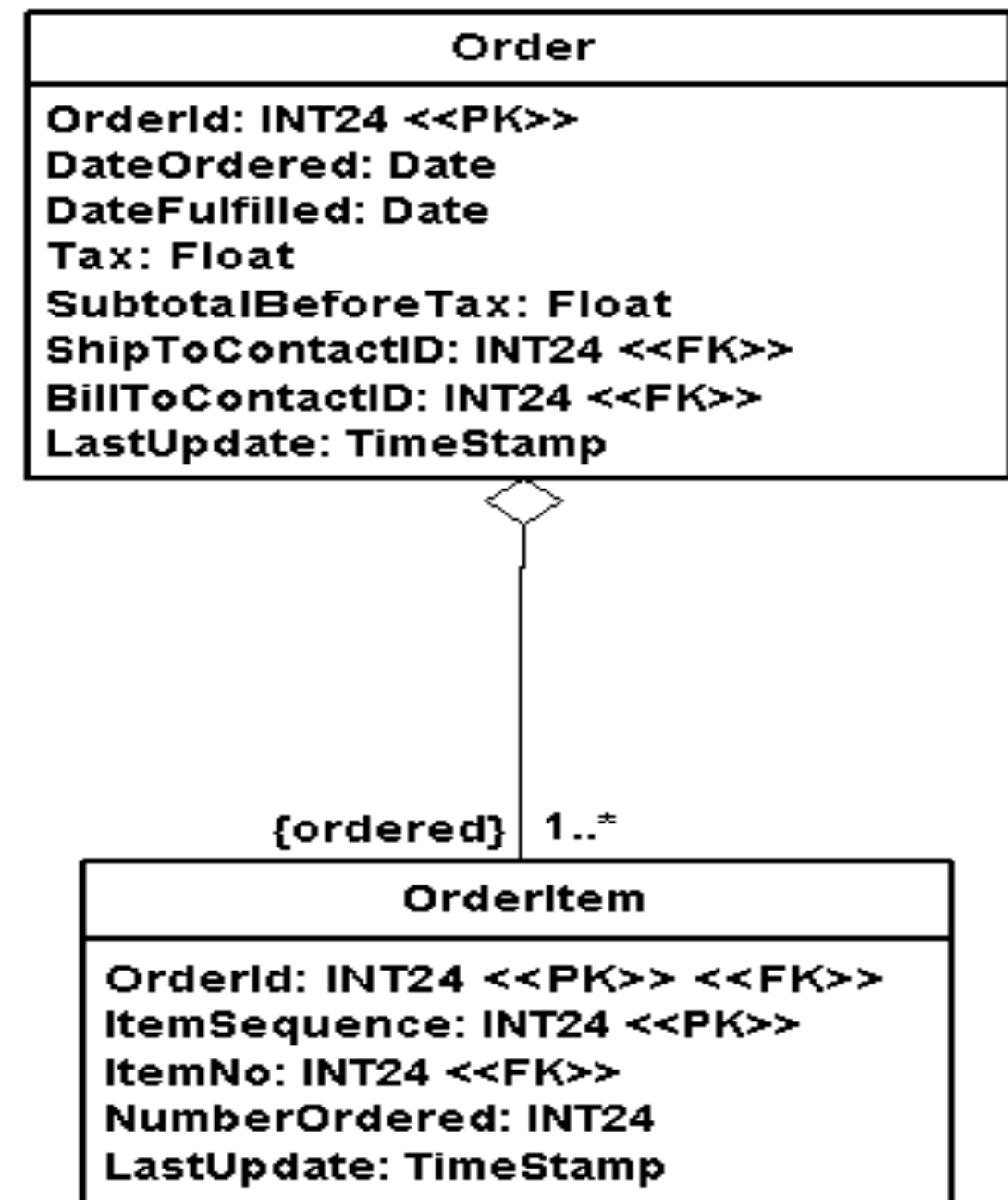
- O clasă este mapată într-o tabelă.
- Exceptând modele foarte simple, maparea unu-la-unu este rară.
- Cea mai simplă mapare este maparea unei proprietăți asociate unui singur atribut la o singură coloană din tabela corespunzătoare.
- Este și mai simplu dacă ambele (atributul și coloana) au aceleași tipuri de bază.
 - ambele sunt stringuri/date,
 - atributul este un string, coloana e de tip char(varchar),
 - atributul este un număr, iar coloana este float.

Mapări simple

<<Class Model>>



<<Physical Data Model>>



Diferențe

- Sunt mai multe attribute pentru **tax** în modelul orientat obiect - doar o singură coloană în schema relațională. Cele trei attribute din clasa **Order** ar trebui însumate și rezultatul păstrat în tabelă.
- În schema relațională apar chei, în modelul orientat obiect nu există chei. Înregistrările din tabela relațională sunt identificate în mod unic de cheia primară, iar relațiile dintre tabele sunt păstrate folosind chei străine.
- Relațiile dintre obiecte sunt păstrate prin referințe, nu prin chei străine. Pentru a putea persista obiectele și relațiilor dintre ele, obiectele trebuie să știe de valoarea cheilor păstrate în baza de date pentru a le putea identifica. Informația adițională este numită “*shadow information*”.
- Sunt folosite diferite tipuri în modelul orientat obiect și în schema relațională:
 - atributul **subTotalBeforeTax** din clasa **Order** este de tip **Currency**
 - coloana **SubTotalBeforeTax** din tabela **Order** este de tip float.
 - Pentru a implementa maparea trebuie să putem converti între cele două reprezentări fără a pierde informații.

Shadow Information

- *Shadow information* sunt orice informații pe care obiectele trebuie să le păstreze (pe lângă informațiile normale) pentru a putea fi persistate.
- Include:
 - *Cheia primară*: în special când cheia primară este o cheie surogat care nu are altă semnificație în domeniu.
 - Informații pentru *controlul concurenței*: timestamps sau incremental counters.
 - Informații pentru *păstrarea versiunii*: *versioning* numbers.
- Exemplu: tabela **Order** are coloana **OrderID** folosită ca și cheie primară și coloana **LastUpdate** folosită pentru controlul concurenței care nu apar în clasa **Order**.

Mapare Metadata

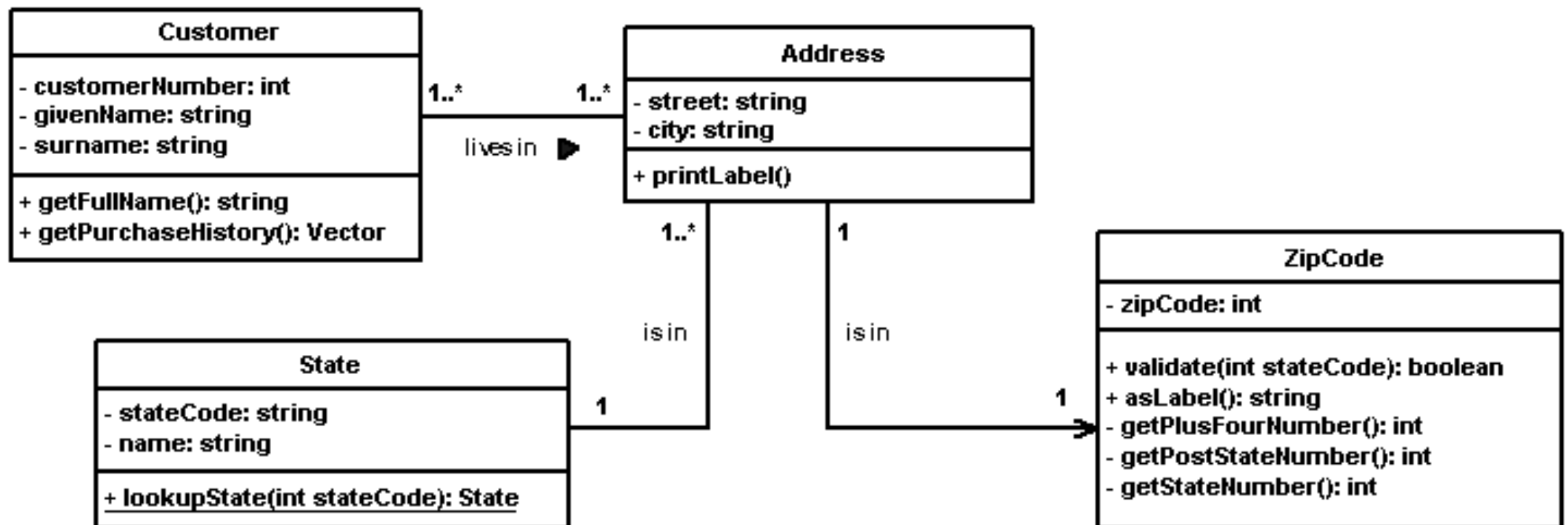
- Metadata păstrează informații despre date. Maparea metadatelor descrie modul în care metadatele reprezentând proprietățile sunt mapate la metadatele corespunzând tabelelor.

Proprietate (OO)	Coloana (BD)
Order.orderID	Order.OrderID
Order.dateOrdered	Order.DateOrdered
Order.dateFulfilled	Order.DateFulfilled
Order.getTotalTax()	Order.Tax
Order.subtotalBeforeTax	Order.SubtotalBeforeTax
Order.shipTo.personID	Order.ShipToContactID
Order.billTo.personID	Order.BillToContactID
Order.lastUpdate	Order.LastUpdate
OrderItem.ordered	OrderItem.OrderID
Order.orderItems.position(orderItem)	OrderItem.ItemSequence
OrderItem.item.number	OrderItem.ItemNo
OrderItem.numberOrdered	OrderItem.NumberOrdered

ORM Impedance Mismatch

- Paradigma orientată obiect promovează dezvoltarea aplicațiilor folosind obiecte care păstrează date, dar conțin și logica aplicației.
- Bazele de date relaționale stochează datele în tabele și manipulează datele folosind proceduri stocate și interogări SQL.
- Diferențele dintre cele două abordări au fost numite: *object-relational impedance mismatch* sau doar *impedance mismatch*.
- Ex. în paradigma orientată obiect obiectele sunt traversate folosind relațiile dintre ele, în paradigma relațională se folosește operația de join.
- Tipurile de date diferite în limbajele orientate obiect și bazele de date relaționale:
 - Java: string și int - Oracle: varchar și smallint.
 - Java: colecții - Oracle: tabele
 - Java: obiecte - Oracle: blobs

ORM Impedance Mismatch



Copyright 2002-2006 Scott W. Ambler

ORM Impedance Mismatch

