Medii de proiectare și programare

2023-2024
Curs 8

Conținut

- Remote Procedure Call
  - Protobuf (exemplu Mini-chat)
  - gRPC
  - Thrift
- ORM
  - Hibernate

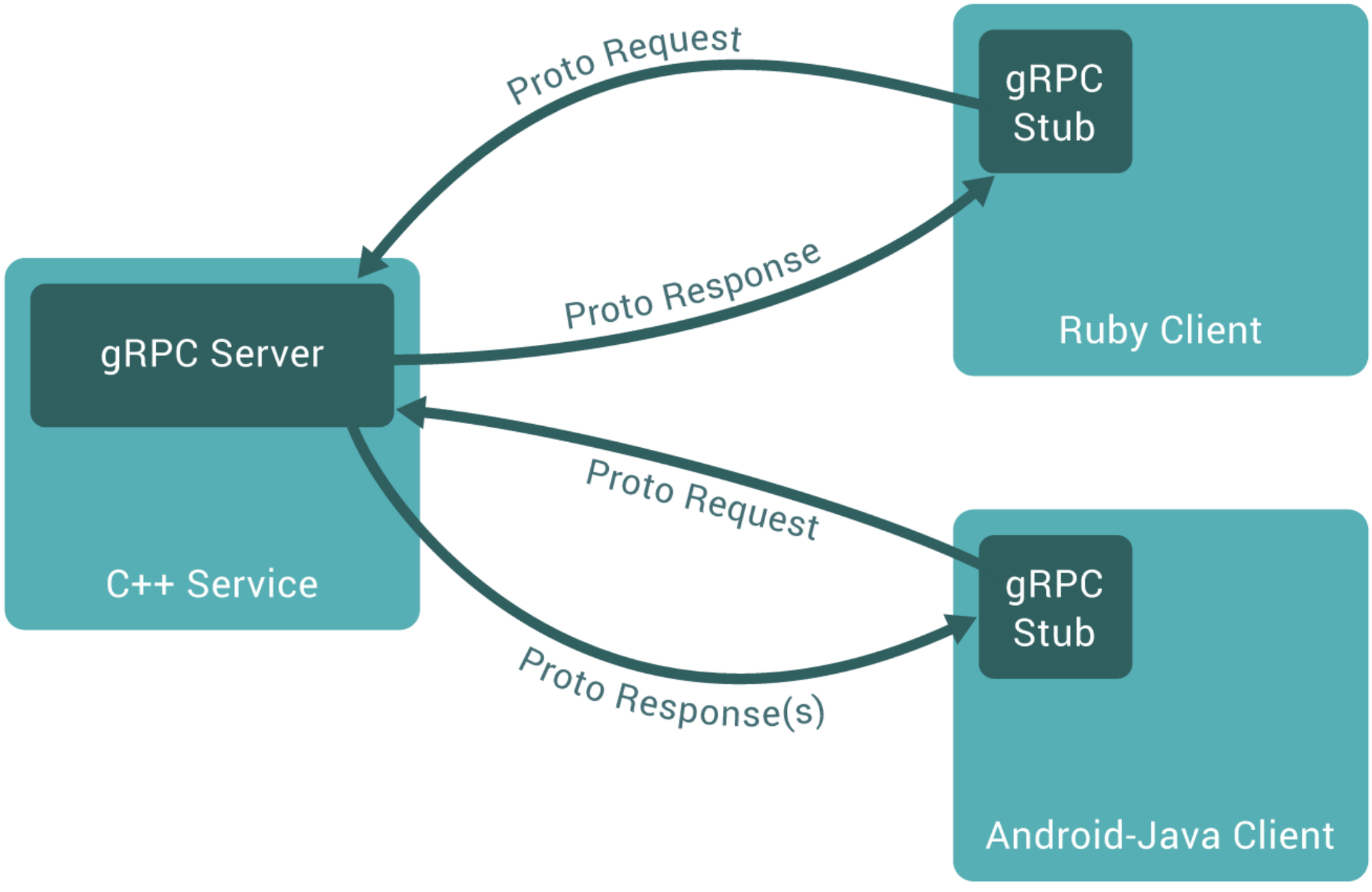
Protocol Buffers

- Exemplu MiniChat

gRPC

- Cu ***gRPC*** o aplicație client poate apela metode la distanță ca si cum ar fi metode ale unui obiect local.
- Facilitează crearea aplicațiilor distribuite și a serviciilor.
- gRPC folosește idea definirii unui serviciu (interfețe) care specifică metodele ce pot fi apelate la distanță, împreuna 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


Proto Request
gRPC Stub
Ruby Client
Proto Response
gRPC Server
C++ Service
Proto Request
gRPC Stub
Android-Java Client
Proto Response(s)

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 singura 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 durata 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 durata 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/

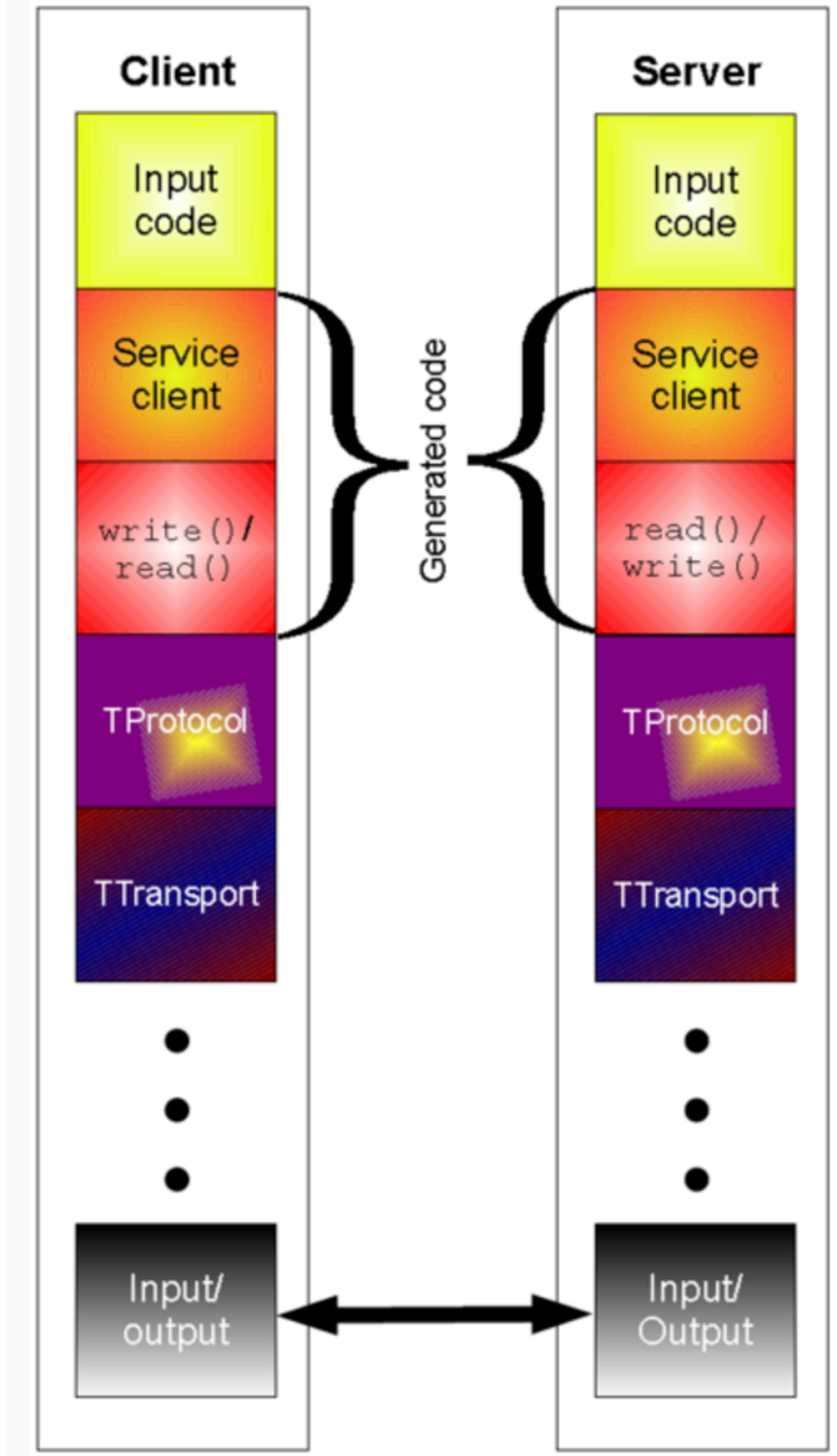
  https://github.com/google/protobuf

- 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.

Image from wikipedia.org

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 valoarea 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 si 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.
- Compilatorul 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.
- Exemplu: Text transformer

Bibliografie

- Apache Thrift

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

  https://thrift.apache.org/

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>>

Order

- dateOrdered: Date
- dateFulfilled: Date
- federalTax: Currency
- stateTax: Currency
- localTax: Currency
- subtotalBeforeTax: Currency

+ cancel()
+ getTaxes(): Vector
+ getTotal(): Currency
+ ship()
- calculateTaxes()
- scheduleShipment()

{ordered} 1..*

OrderItem

- numberOrdered: int

+ getTotal(): Currency

<<Physical Data Model>>

Order

OrderId: INT24 <<PK>>
DateOrdered: Date
DateFulfilled: Date
Tax: Float
SubtotalBeforeTax: Float
ShipToContactID: INT24 <<FK>>
BillToContactID: INT24 <<FK>>
LastUpdate: TimeStamp

{ordered} 1..*

OrderItem

OrderId: INT24 <<PK>> <<FK>>
ItemSequence: INT24 <<PK>>
ItemNo: INT24 <<FK>>
NumberOrdered: INT24
LastUpdate: TimeStamp

Diferențe

- Sunt mai multe atribute pentru **`tax`** în modelul orientat obiect - doar o singură coloană în schema relațională. Cele trei atribute 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

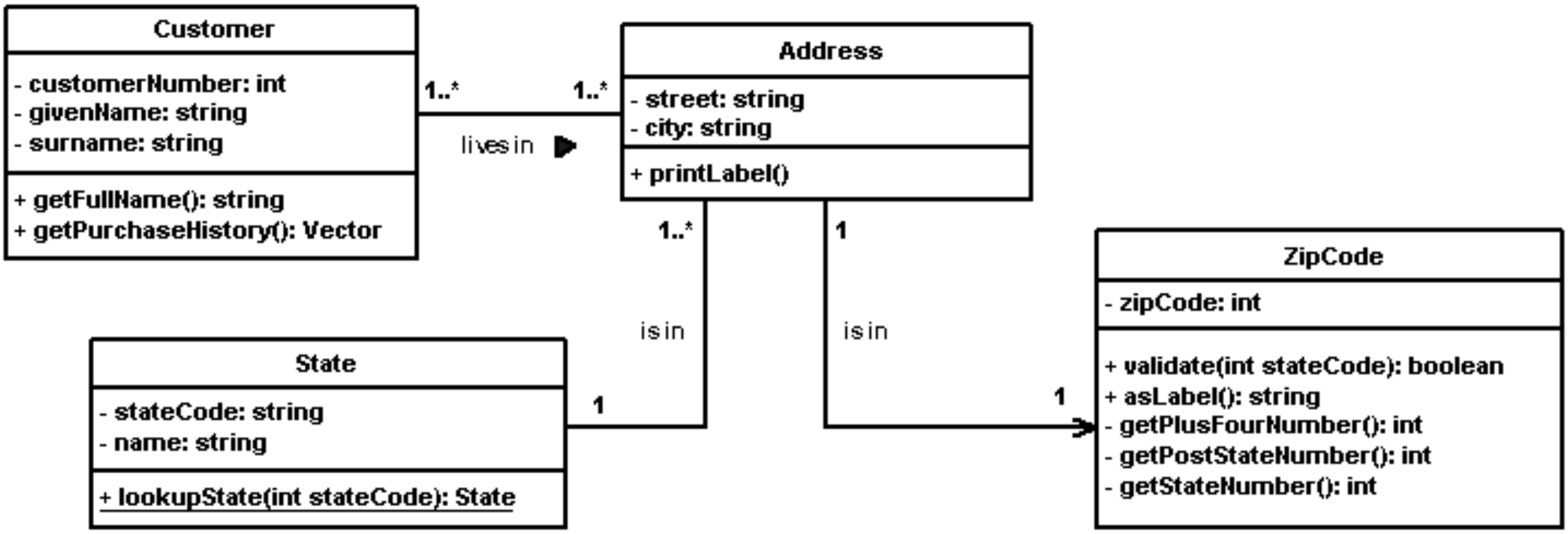
- Metadata păstrează informații despre date. Maparea metadatelor descrie modul în care metadatele reprezentând proprietățile sunt mapate la metadatele corespunzând tabelelor.

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

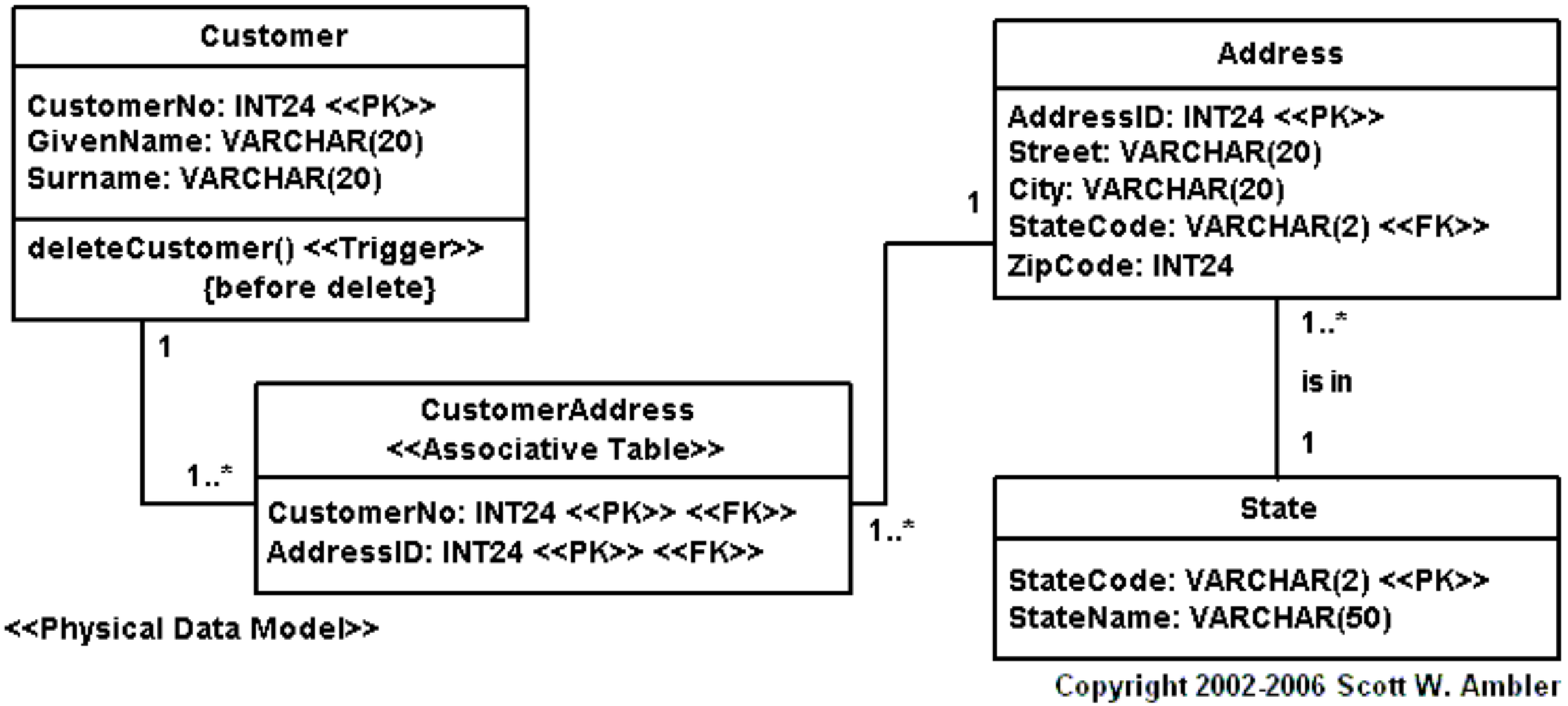
ORM Impedance Mismatch

- 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



ORM Impedance Mismatch



<<Physical Data Model>>

Strategii pentru Impedance Mismatch

- Maparea moștenirii
- Maparea relațiilor dintre obiecte
- Maparea proprietăților statice

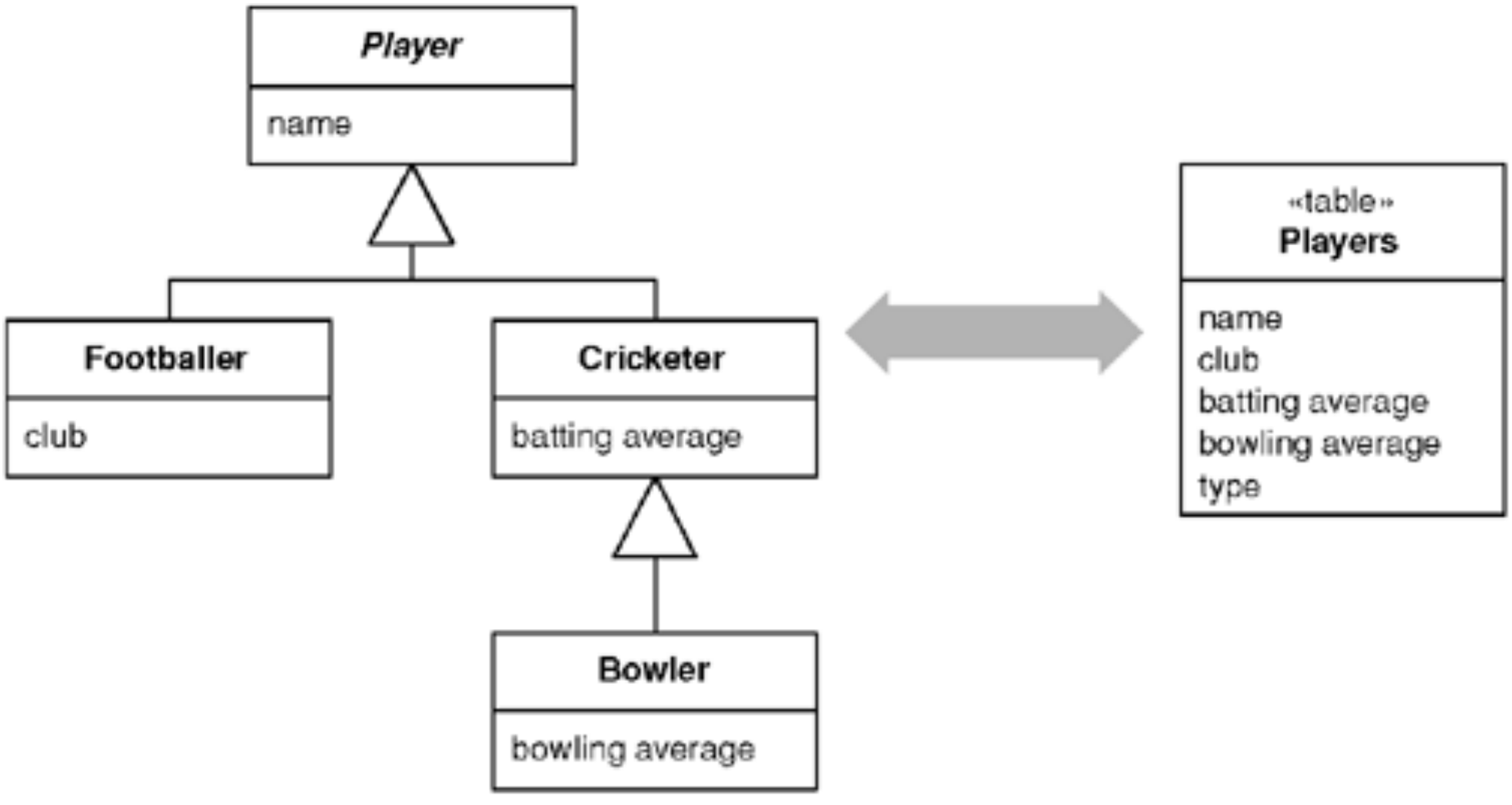
Maparea moștenirii

- Bazele de date relaționale nu suportă moștenirea.
- Programatorul trebuie să mapeze moștenirea dintre entitățile din modelul orientat obiect într-o bază de date relațională.
- Tehnici:
  - Maparea ierarhiei de clase într-o singură tabelă.
  - Maparea fiecărei clase concrete în tabela ei.
  - Maparea fiecărei clase într-o tabelă.
  - Maparea claselor într-o structura de tabele generică.

Moștenirea - O singură tabelă

- Reprezentarea unei ierarhii de clase (moștenire) ***ca și o singură tabelă*** cu coloane pentru toate atributele din toate clasele din ierarhie.
- Fiecare clasă păstrează informațiile relevante pentru ea într-o înregistrare din tabelă. Coloanele care nu sunt relevante rămân goale.
- Când se încarcă un obiect din tabelă, instrumentul ORM trebuie să știe ce clasă să instanțieze.
- În tabelă se adaugă o coloană care indică ce clasă ar trebui instanțiată (numele clasei sau un cod):
  - Codul trebuie interpretat în codul sursă pentru a putea face maparea cu clasa corespunzătoare.
  - Numele clasei poate fi folosit direct pentru instanțiere (folosind reflecție).

Moștenirea - O singură tabelă



Moștenirea - O singură tabelă

```
Footballer fb=new Footballer("A A", "ABC")
Cricketer cr=new Cricketer("C C", 23);
Bowler bw=new Bowler("B B", 21, 47);
Footballer fb2=new Footballer("D D", "BGD");
Bowler bw2=new Bowler("H H", 12, 23);
```

Players

PK	Name	Club	BattlingAvg	BowlingAvg	Type
1	A A	ABC			Footballer
2	C C		23		Cricketer
3	B B		21	47	Bowler
4	D D	BGD			Footballer
5	H H		12	23	Bowler

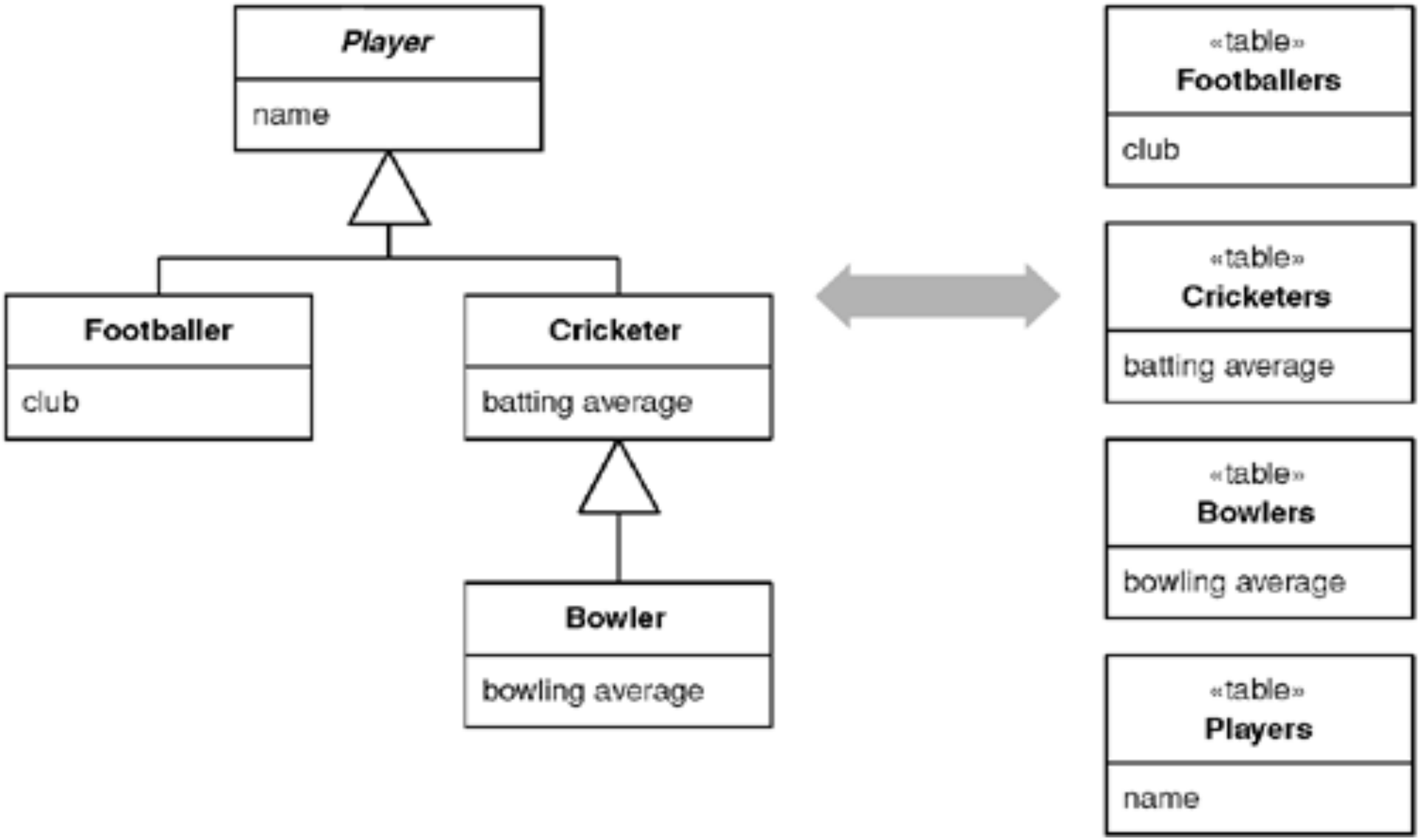
Moștenirea - O singură tabelă

- Avantaje:
  - Există doar o singură tabelă în baza de date.
  - Nu e nevoie de operații join pentru regăsirea informației.
  - Orice refactorizare care mută atributele în ierarhie nu necesită modificarea bazei de date.
- Dezavantaje:
  - Nu toate câmpurile din tabelă sunt relevante (depinde de tipul clasei). Este confuz pentru cei care folosesc tabelele direct (fără instrumentul ORM).
  - Coloanele folosite doar de subclase duc la spațiu nefolosit (multe coloane goale).
  - Tabela poate deveni prea mare, cu mulți indecși și blocări dese ale tabelei. Poate afecta performanța.
  - Există un singur spațiu de nume pentru câmpuri, dezvoltatorul trebuie să se asigure că se vor folosi nume diferite în tabelă.
    - Adăugarea numelui clasei (prefix, postfix) poate ajuta. (ex. NumeClasă_NumeProprietate)

Tabelă pentru fiecare clasă

- Fiecare clasă din ierarhie are tabela ei.
- Atributele din clasă se mapează direct la coloanele corespunzătoare din tabelă.
- *Problemă*: Cum se leagă înregistrările din tabele?
  - *Soluția A*: folosirea cheii primare atât în tabela corespunzătoare clasei de bază cât și în clasa derivată. Deoarece clasa de bază are câte o înregistrare pentru fiecare înregistrare din clasele derivate, cheia primară va fi unica între toate tabele.
  - *Soluția B*. Fiecare tabelă să aibă cheia primară proprie, și folosirea cheii străine pentru a păstra legătura cu tabela corespunzătoare clasei de bază.
- Provocare: încărcarea/regăsirea informațiilor din mai multe tabele în mod eficient.
  - Operații de join între diferite tabele
  - Operațiile de join între mai mult de 3 sau 4 tabele sunt lente din cauza modului în care bazele de date optimizează operațiile interne.
- Interogările asupra bazei de date sunt dificile.

Tabelă pentru fiecare clasă



Tabelă pentru fiecare clasă

```
Footballer fb=new Footballer("A A", "ABC")
Cricketer cr=new Cricketer("C C", 23);
Bowler bw=new Bowler("B B",21, 47);
Footballer fb2=new Footballer("D D", "BGD");
Bowler bw2=new Bowler("H H",12, 23);
```

Players

PK	Name
1	A A
2	C C
3	B B
4	D D
5	H H

Footballers

PK	Club
1	ABC
4	BGD

Cricketers

PK	BattlingAvg
2	23
3	21
5	12

Bowlers

PK	BowlingAvg
3	47
5	23

Tabelă pentru fiecare clasă

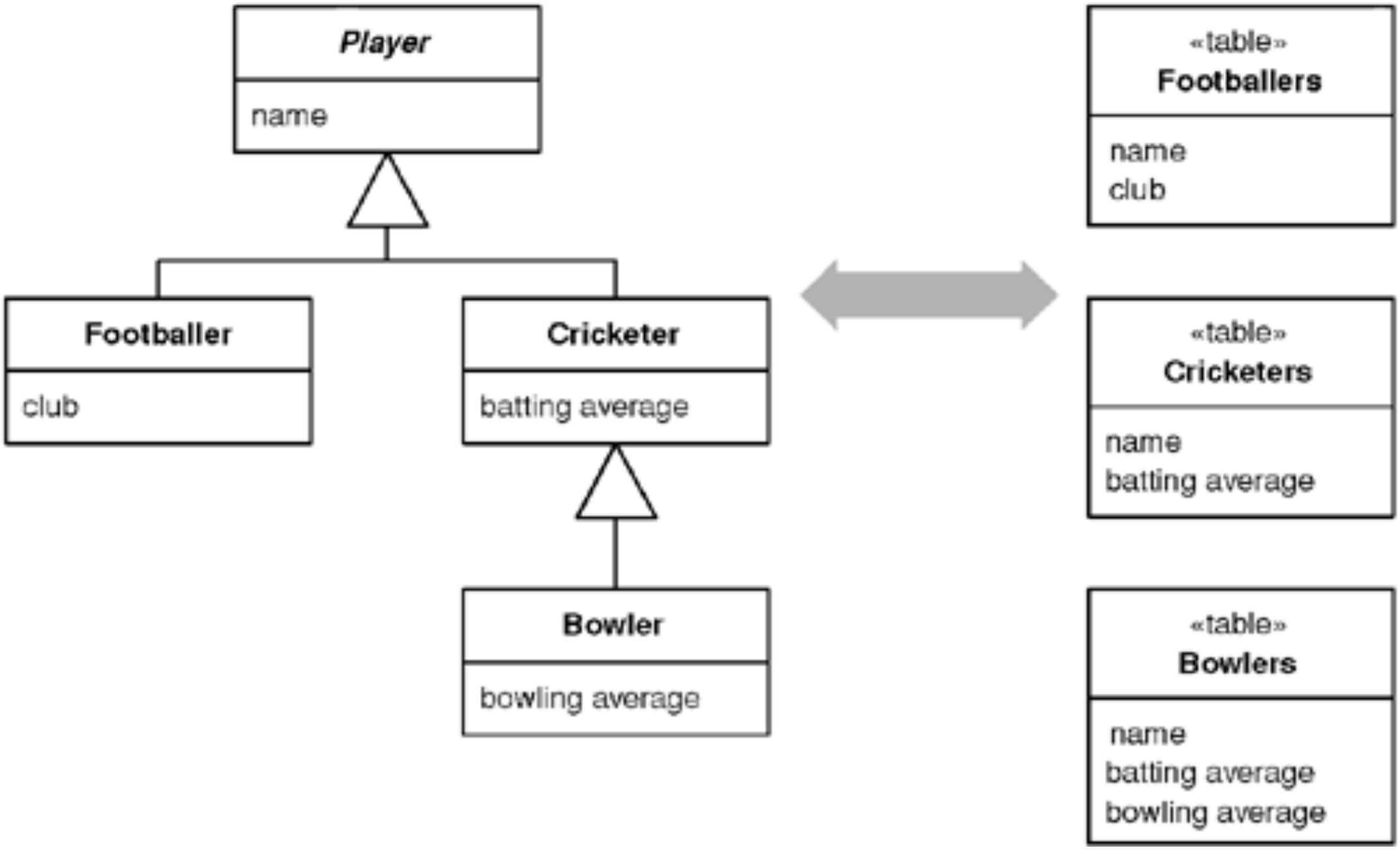
- Avantaje:
  - Toate coloanele sunt relevante pentru fiecare înregistrare, tabelele sunt mai ușor de înțeles și nu se folosește spațiu în mod ineficient.
  - Relația dintre entitățile din modelul orientat obiect și baza de date relațională este ușor de înțeles.

- Dezavantaje:
  - Este necesară folosirea mai multor tabele pentru a încărca un obiect din mediu persistent (operație join sau mai multe interogări și folosirea memoriei).
  - Orice refactorizare (mutarea câmpurilor în ierarhia de clase) cauzează modificarea structurii bazei de date.
  - Tabelele corespunzătoare claselor de bază pot cauza probleme de performanță din cauza accesării dese.
  - Normalizarea poate duce la înțelegerea dificilă a interogărilor ad-hoc.

Tabelă pentru fiecare clasă concretă

- Fiecare clasă concretă (non-abstract) din ierarhie are tabela ei.
- Fiecare tabelă conține coloane pentru toate proprietățile din ierarhie până la ea. ***Atributele din clasa de bază sunt duplicate în tabelele corespunzătoare subclaselor***.
- Este responsabilitatea programatorului de a se asigura că cheile sunt unice nu doar în tabela corespunzătoare clasei dar și între toate tabelele asociate ierarhiei.

Tabelă pentru fiecare clasă concretă



Tabelă pentru fiecare clasă concretă

```
Footballer fb=new Footballer("A A", "ABC")
Cricketer cr=new Cricketer("C C", 23);
Bowler bw=new Bowler("B B",21, 47);
Footballer fb2=new Footballer("D D", "BGD");
Bowler bw2=new Bowler("H H",12, 23);
```

Footballers

PK	Name	Club
1	A A	ABC
4	D D	BGD

Cricketers

PK	Name	BattlingAvg
2	C C	23

Bowlers

PK	Name	BattlingAvg	BowlingAvg
3	B B	21	47
5	H H	12	23

Tabelă pentru fiecare clasă concretă

- Avantaje:
  - Fiecare tabelă păstrează toate informațiile relevante și nu are câmpuri irelevante. Este ușor de înțeles și de alte aplicații care nu folosesc obiecte.
  - Nu este nevoie de operații join pentru citirea datelor.
  - Fiecare tabelă este accesată doar când clasa respectivă este accesată. Performanța este mai bună.
- Dezavantaje:
  - Gestiunea dificilă a cheilor primare.
  - Nu pot fi constrânse relațiile către clasele abstracte.
  - Dacă câmpurile din modelul obiectual sunt mutate în ierarhie, trebuie modificate definițiile tabelelor.
  - Dacă se modifică un câmp dintr-o clasă de bază, trebuie modificate toate tabelele corespunzătoare subclaselor, pentru că aceste câmpuri sunt duplicate.
  - O operație de căutare folosind clasa de bază, necesită căutari în toate tabelele (accesări multiple ale bazei de date sau o operație de join complicată).