

DEVELOPMENT OF INFORMATION SYSTEMS

Lecture 7

Object-Relational Structural Patterns II.

Inheritance mapping

General patterns

Review of the last lecture

Object-Relational Structural Patterns

These patterns deal with the structural differences between in memory objects and database tables/rows.

- Identity Field
 - Foreign Key Mapping
 - Association Table Mapping
 - Dependent Mapping
 - Embedded Value
 - Serialized LOB
- 

Review of the last lecture - Identity Field

Saves a database ID field in an object to maintain identity between an in-memory object and a database row.

- Meaningful key
- Meaningless key
- Simple key
- Compound key
- Table unique key
- Database unique key
- Getting a New Key
 - Database auto-generate
 - GUID
 - Generate your own (SQL, Key Table)

Review of the last lecture - Foreign Key Mapping

Maps an association between objects to a foreign key reference between tables.

- Uses Identity Field
- Simple one to one association between classes is replaced by the Foreign key in DB.
- Association one to many is replaced Foreign key in the reverse direction.
- Association many to many use pattern Association table mapping.
- Update strategy
 - Delete and insert.
 - Add and backpoint to owner.
 - Diff and merge of the changes.

Review of the last lecture - Association Table Mapping

Saves an association as a table with foreign keys to the tables that are linked by the association.

- Table has only the foreign key IDs for the two tables that are linked together
- It has one row for each pair of associated objects.
- The link table has no corresponding in-memory object. As a result it has no ID.
- Its primary key is the compound of the two primary keys of the tables that are associated.

Review of the last lecture - Dependent Mapping

Has one class perform the database mapping for a child class. The basic idea behind Dependent Mapping is that one class (the **dependent**) relies upon some other class (the **owner**) for its database persistence.

- A dependent may itself be the owner of another dependent.
- In this case the owner of the first dependent is also responsible for the persistence of the second dependent.
- You can have a whole hierarchy of dependents controlled by a single primary owner.
- Using Dependent Mapping complicates tracking whether the owner has changed.

Review of the last lecture - Embedded Value

Maps an object into several fields of another object's table.

- When the owning object is loaded or saved, the dependent objects are loaded and saved at the same time.
- The dependent classes won't have their own persistence methods since all persistence is done by the owner.
- It is special case of Dependent Mapping design pattern, where the value is a single dependent object.
- All Value Objects (date ranges, money) should be persisted as Embedded Value, since you would never want a table for them there.

Review of the last lecture - Serialized LOB

Saves a graph of objects by serializing them into a single large object (LOB), which it stores in a database field.

Serialization

- Binary form (BLOB)
- Textual form (CLOB)

Common problems

- Identity of object problem
- Duplicate data problem
- Outside references problem

Realization:

- Save the graph applying the serialization into a memory buffer and saving that buffer to the particular field in DB.

Object-Relational Structural Patterns II.

Inheritance mappings

- Single Table Inheritance
 - Class Table Inheritance
 - Concrete Table Inheritance
 - Inheritance Mappers
- 
- A series of three parallel white diagonal lines in the bottom right corner of the slide, extending from the middle of the right edge towards the bottom left.

Inheritance in OOP

Inheritance is one of the most important aspects of Object Oriented Programming (OOP).

The key to understanding Inheritance is that it provides code **re-usability**. In place of writing the same code, again and again, we can simply inherit the properties of one class into the other.

OOP is all about real-world objects and inheritance is a way of representing real-world relationships.

Car, bus, bike – all of these come under a broader category called **Vehicle**. That means they've inherited the properties of class vehicles i.e all are used for transportation.

We can represent this relationship in code with the help of inheritance.

Inheritance in OOP

More formally:

Inheritance is the procedure in which one class inherits the attributes and methods of another class. The class whose properties and methods are inherited is known as the **Parent** class. And the class that inherits the properties from the parent class is the **Child** class.

The important thing is, along with the inherited properties and methods, a child class can have its own properties and methods.

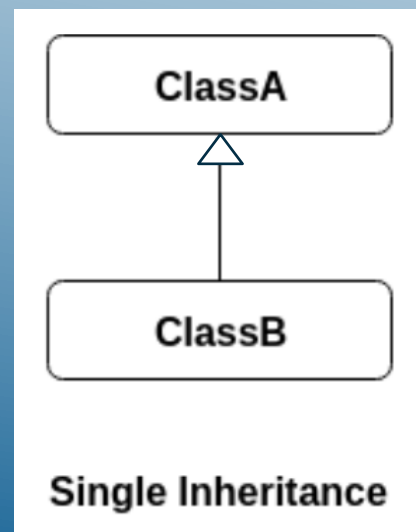
Types of inheritance:

- Single Inheritance
- Multiple Inheritance
- Multi-level Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

Inheritance patterns types

Single inheritance

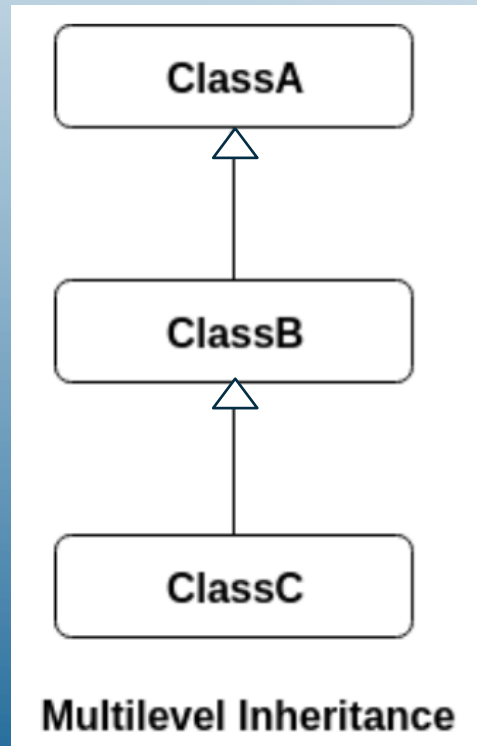
Single inheritance can inherit properties and behavior from at most **one parent class**. It allows a derived/subclass to inherit the properties and behavior of a base class that enable the **code reusability** as well as we can add new features to the existing code. The single inheritance makes the code less repetitive.



Inheritance patterns types

Multilevel inheritance

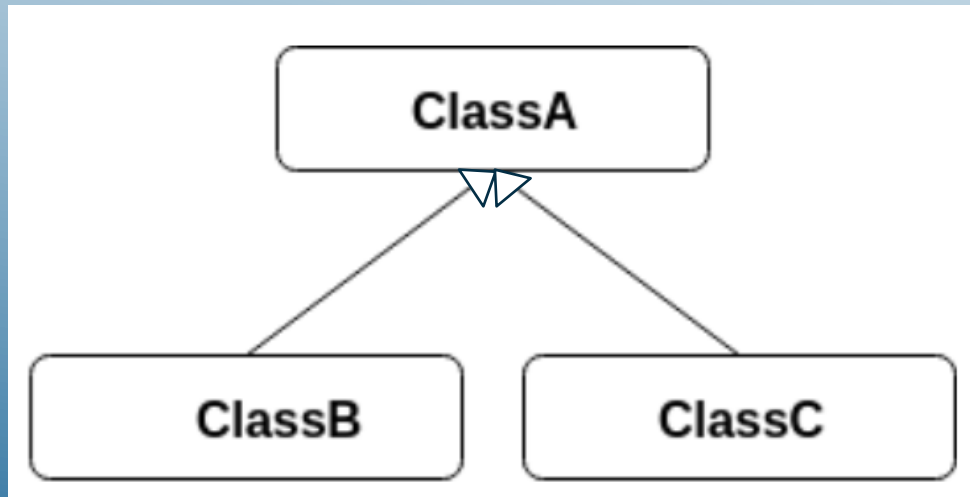
When a derived class is derived from another derived class, then this type of inheritance is known as **multilevel inheritance**. Thus, a multilevel inheritance has more than one parent class. It is similar to the **relation** between Grandfather, Father, and Child.



Inheritance patterns types

Mierarchical inheritance

When more than one subclass is inherited from a single base class, then this type of inheritance is known as **hierarchical inheritance**. Here, all features which are common in sub-classes are included in the base class.

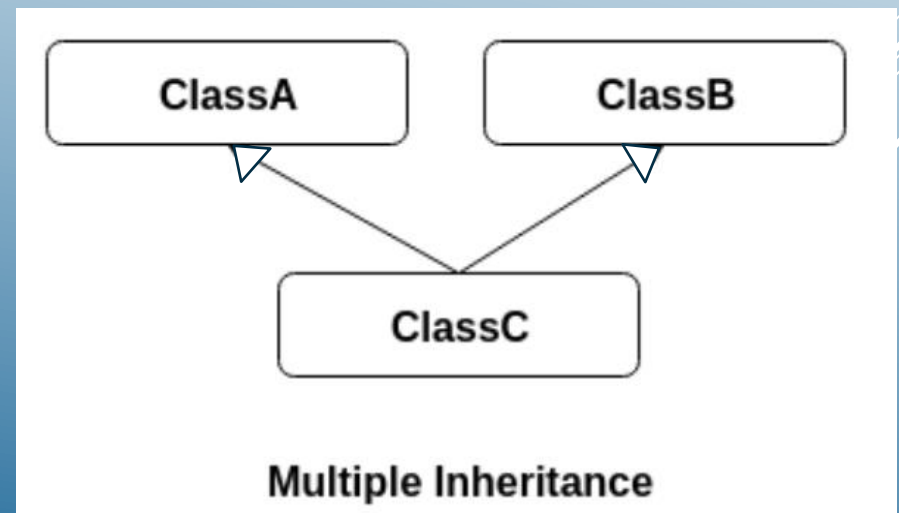


Inheritance patterns types

Multiple inheritance

When an object or class inherits the **characteristics** and **features** from more than one parent class, then this type of inheritance is known as **multiple inheritance**. Thus, a multiple inheritance acquires the properties from more than one parent class. (Support C++, Lisp, Perl, Python)

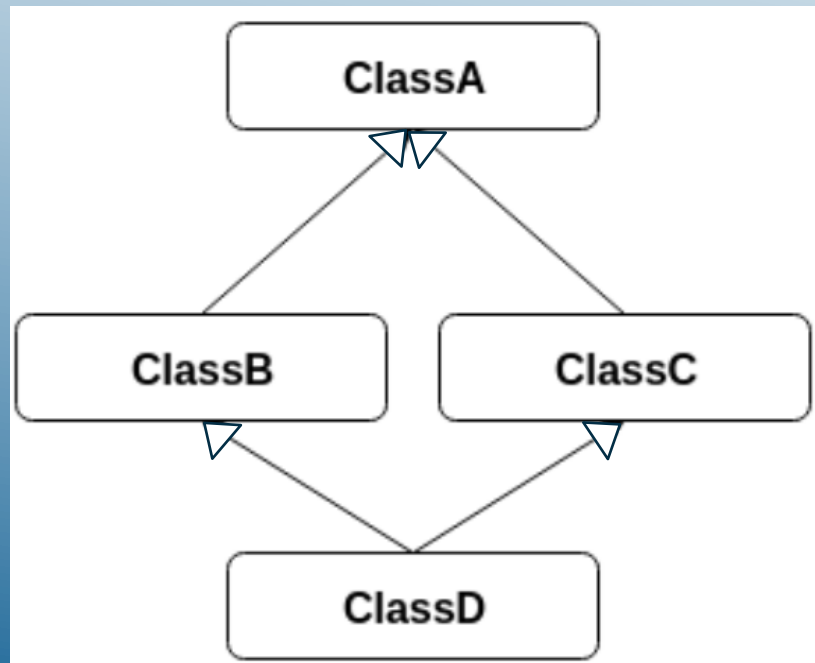
Not supported in many languages
TypeScript, C#, JAVA



Inheritance patterns types

Hybrid inheritance

When a class inherits the characteristics and features from more than one **form of inheritance**, then this type of inheritance is known as **Hybrid inheritance**. In other words, it is a **combination** of multilevel and multiple inheritance. We can implement it by combining more than one type of inheritance.



Inheritance

Why we use it?

- Change of behaviour of class
- Extension of functionality

How it is work?

- Change of data (change state of object)
- Change behaviour (methods have different functionality)

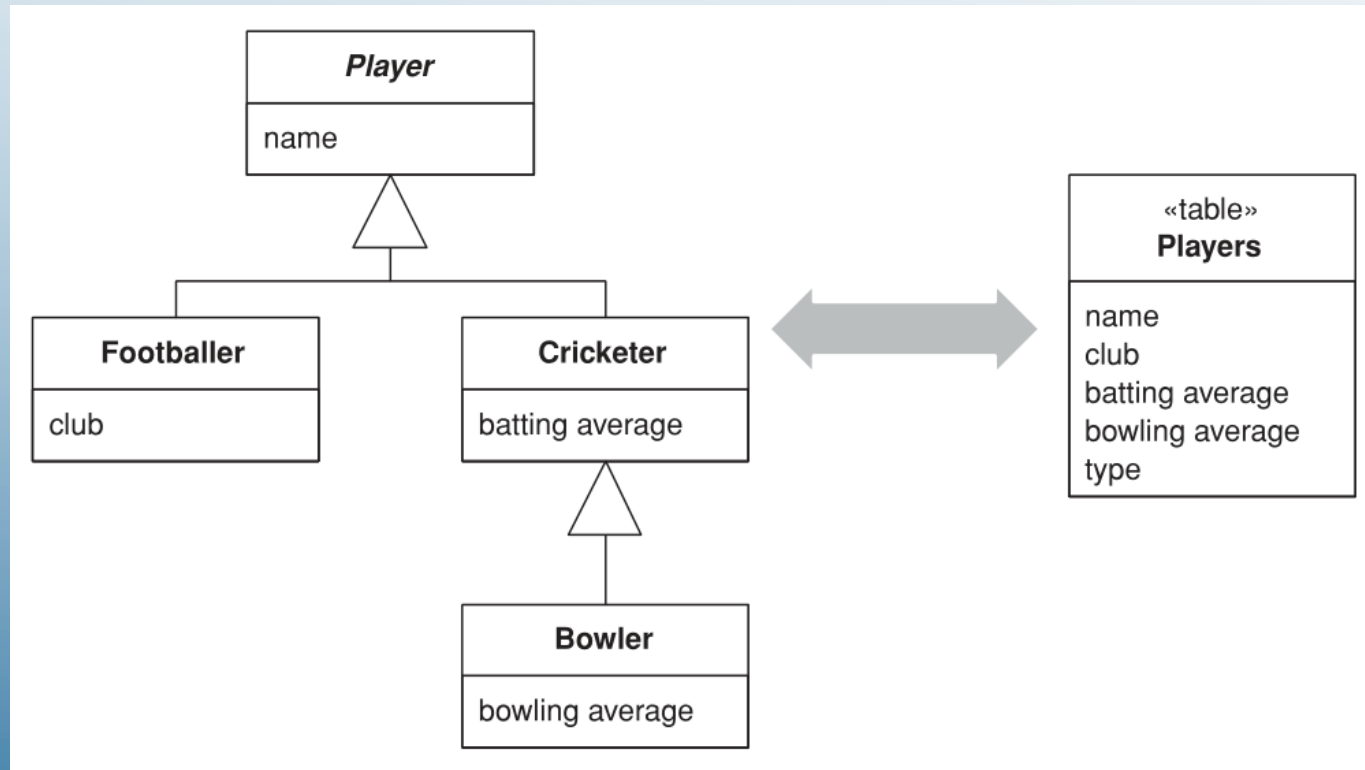
Object Oriented Programming

Four basic concepts:

- **Inheritance** - mechanism of basing an object or class upon another object
- **Abstraction** - “shows” only essential attributes and “hides” unnecessary information.
- **Polymorphism** - ability of an object to take on many forms.
- **Emcapsulation** - process of wrapping the data and the code, that operate on the data into a single entity.

Single Table Inheritance

Represents an inheritance hierarchy of classes as a single table that has columns for all the fields of the various classes.



Single Table Inheritance maps all fields of all classes of an inheritance structure into a single table.

Single Table Inheritance – how it works?

- In this mapping scheme we have one table that contains all the data for all the classes in the inheritance hierarchy.
- Each class stores the data that's relevant to it in one table row.
- Any columns in the database that aren't relevant are left empty.
- When loading an object into memory you need to know which class to instantiate. For this you have a field in the table that indicates which class should be used. This can be the name of the class or a code field.
- In loading data you read the code first to figure out which subclass to instantiate. On saving the data the code needs be written out by the superclass in the hierarchy.

Single Table Inheritance – when to use it?

Single Table Inheritance is one of the options for mapping the fields in an inheritance hierarchy to a relational database.

Pros:

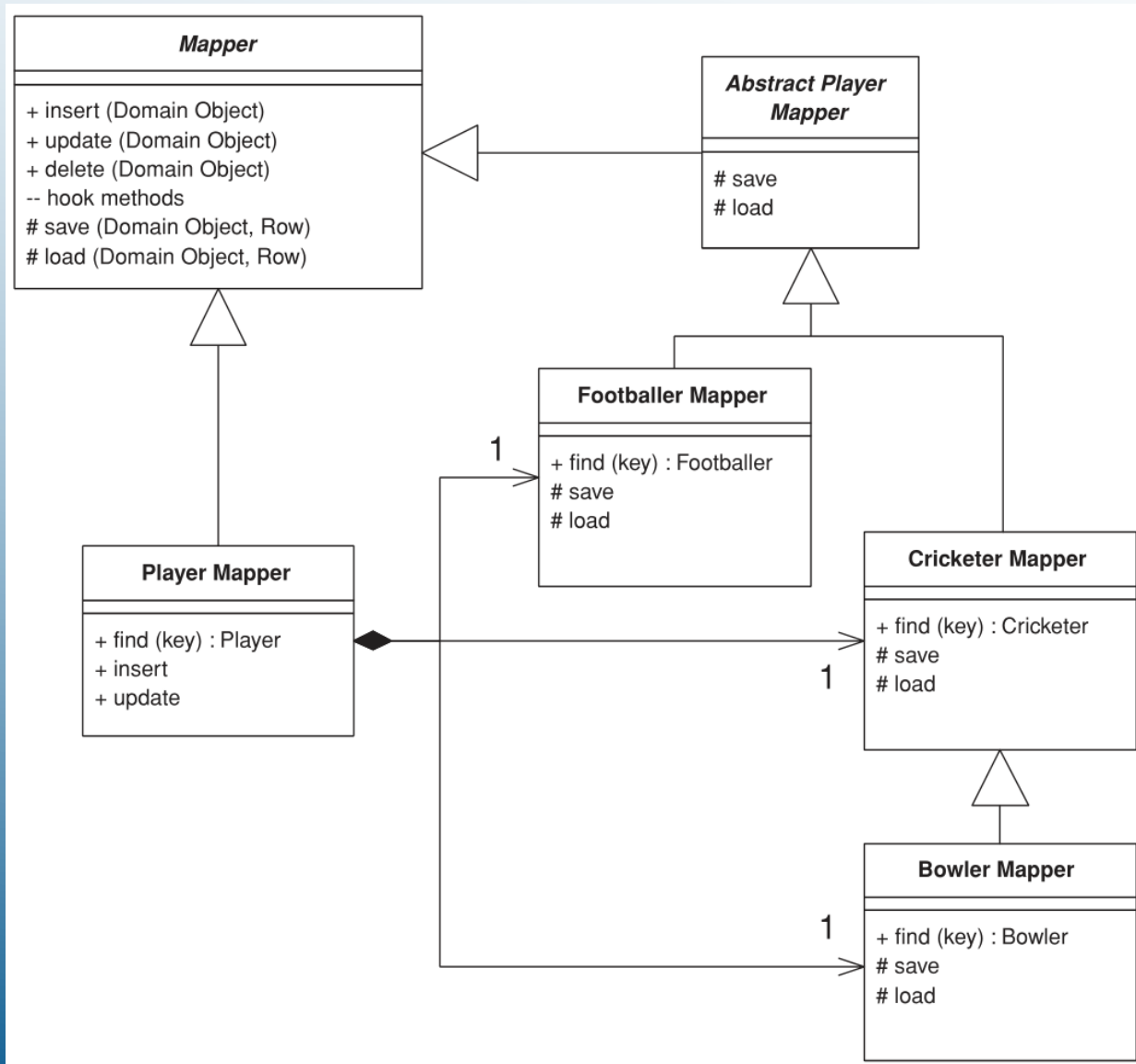
- There's only a single table to worry about on the database.
- There are no joins in retrieving data.
- Any refactoring that pushes fields up or down the hierarchy doesn't require you to change the database.

Cons:

- Fields are sometimes relevant and sometimes not.
- Wasted space in the database.
- The single table may be too large, performance problem.
- Field names problem

Single Table Inheritance – implementation

The generic class diagram of **Inheritance Mappers**



Single Table Inheritance – implementation

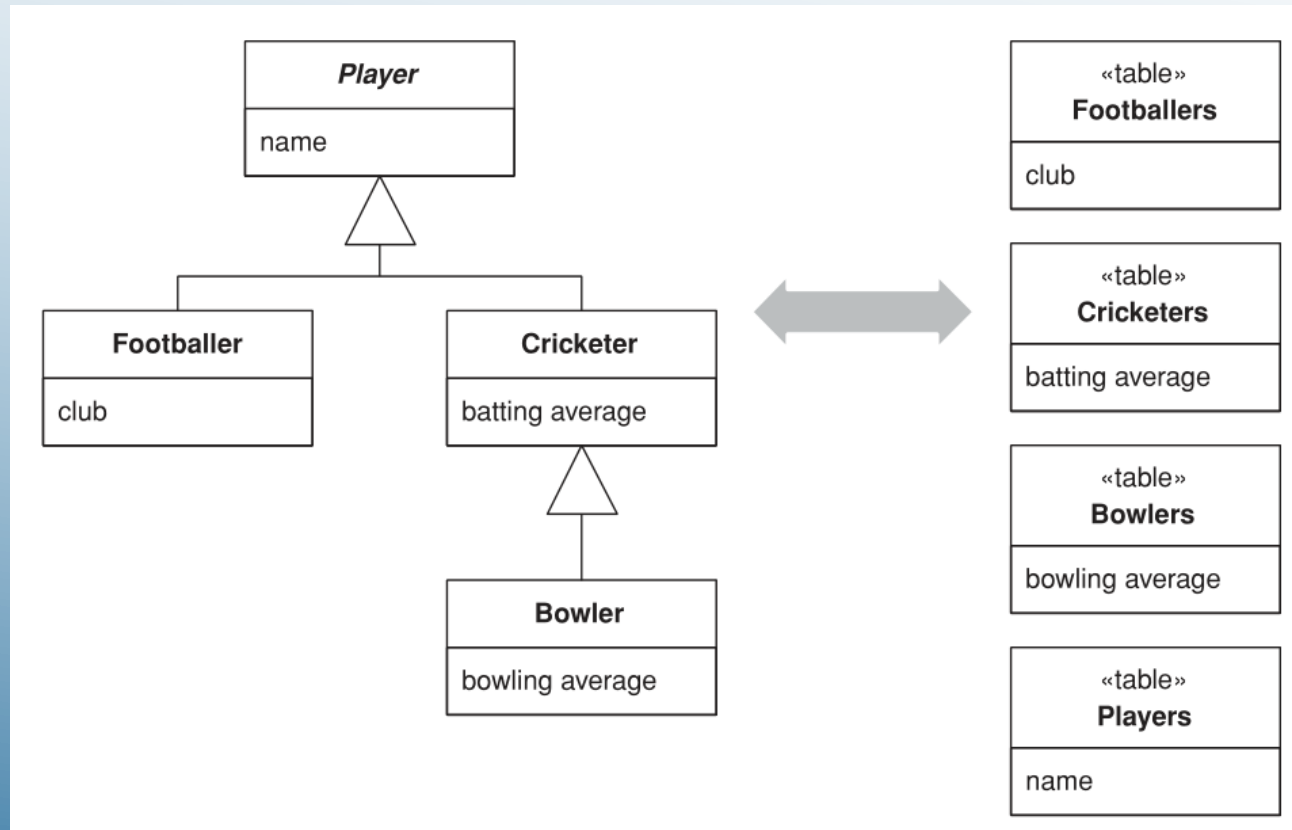
- Each mapper needs to be linked to a data table in a data set.
- This link can be made generically in the mapper superclass. The gateway's data property is a data set that can be loaded by a query.
- Since there is only one table, this can be defined by the abstract player mapper.
- Each class needs a type code to help the mapper code figure out what kind of player it's dealing with.
- The type code is defined on the superclass and implemented in the subclasses.

Loading an Object from the Database

- Each concrete mapper class has a find method to get an object from the data. This calls generic behavior to find an object.
- We load the data into the new object with a series of load methods, one on each class in the hierarchy.
- We can also load a player through the player mapper. It needs to read the data and use the type code to determine which concrete mapper to use.

Class Table Inheritance

Represents an inheritance hierarchy of classes with one table for each class.



You want database structures that map clearly to the objects and allow links anywhere in the inheritance structure.

Class Table Inheritance – how it works?

The straightforward thing about Class Table Inheritance is that it has one table per class in the domain model.

- The fields in the domain class map directly to fields in the corresponding tables.

How to link the corresponding rows of the database tables?

- **Primary key** - use a common primary key value so that, rows correspond to the same domain object. (unique PK across tables)
- **Foreign keys** - let each table have its own primary keys and use foreign keys into the superclass table to tie the rows together.

Class Table Inheritance – when to use it?

This pattern is one of the three basic patterns for class inheritance mapping. The choice of this pattern is based on the preferences of the implementor and the pros and cons.

Pros:

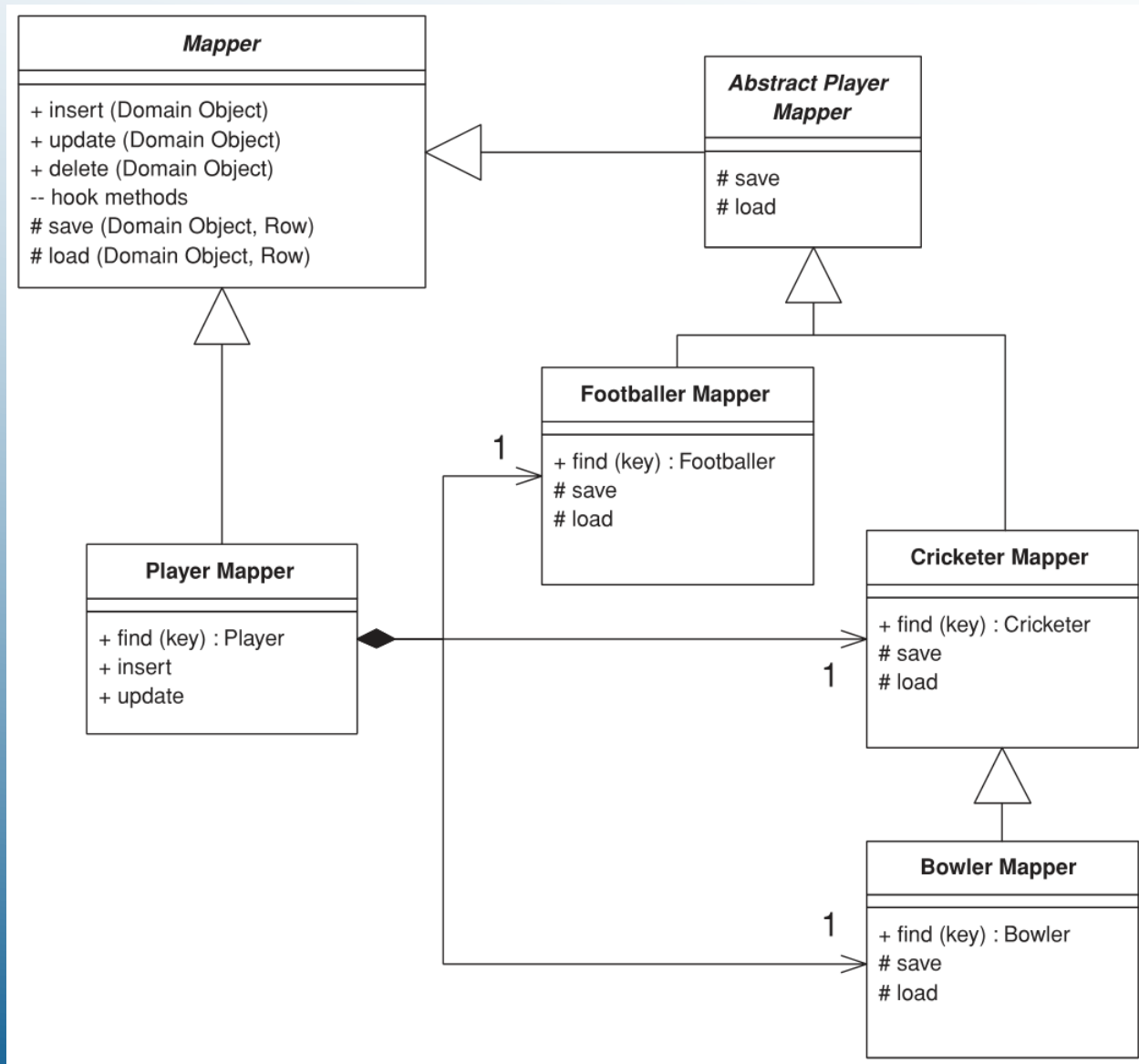
- All columns are relevant for every row so tables are easier to understand and don't waste space.
- The relationship between the domain model and the database is very straightforward.

Cons:

- You need to touch multiple tables to load an object, which means a join or multiple queries and sewing in memory.
- Any refactoring of fields up or down the hierarchy causes database changes.
- The supertype tables may become a bottleneck because they have to be accessed frequently.
- The high normalization may make it hard to understand for ad hoc queries.

Class Table Inheritance – implementation

The generic class diagram of **Inheritance Mappers**

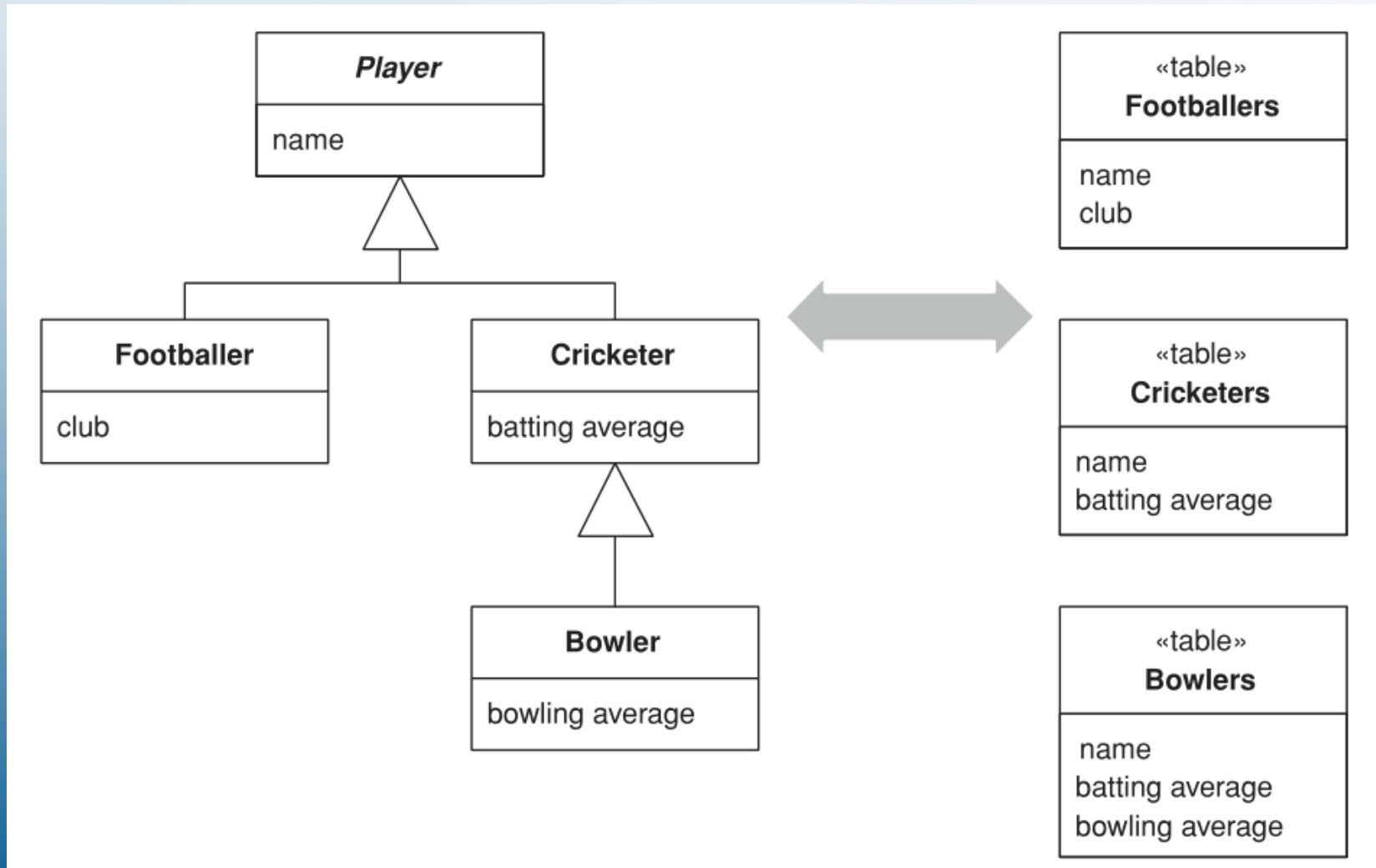


Class Table Inheritance – implementation

- Each class needs to define the table that holds its data and a type code for it.
- Unlike the other inheritance approaches, this one doesn't have a overridden table name because we have to have the table name for this class even when the instance is an instance of the subclass.
- Loading an Object If you've been reading the other mappings, you know the first step is the find method on the concrete mappers.
- The abstract find method looks for a row matching the key and, if successful, creates a domain object and calls the load method on it.
- There's one load method for each class which loads the data defined by that class.
- The player mapper determines which kind of player it has to find and then delegates the correct concrete mapper.

Concrete Table Inheritance

Represents an inheritance hierarchy of classes with one table per concrete class in the hierarchy.



Concrete Table Inheritance – how it works?

Concrete Table Inheritance uses one database table for each concrete class in the hierarchy.

- Each **table contains columns** for the **concrete class** and **all its ancestors**, so any fields in a superclass are duplicated across the tables of the subclasses.
- The key thing is to ensure that keys are unique not just to a table but to all the tables from a hierarchy.
- Often you need a key allocation system that keeps track of key usage across tables; also, you can't rely on the database's primary key uniqueness mechanism.
- For compound keys you can use a special key object as your ID field. This key uses both the primary key of the table and the table name to determine uniqueness.

Concrete Table Inheritance – when to use it?

When figuring out how to map inheritance, Concrete Table Inheritance, Class Table Inheritance, and Single Table Inheritance are the alternatives.

Pros:

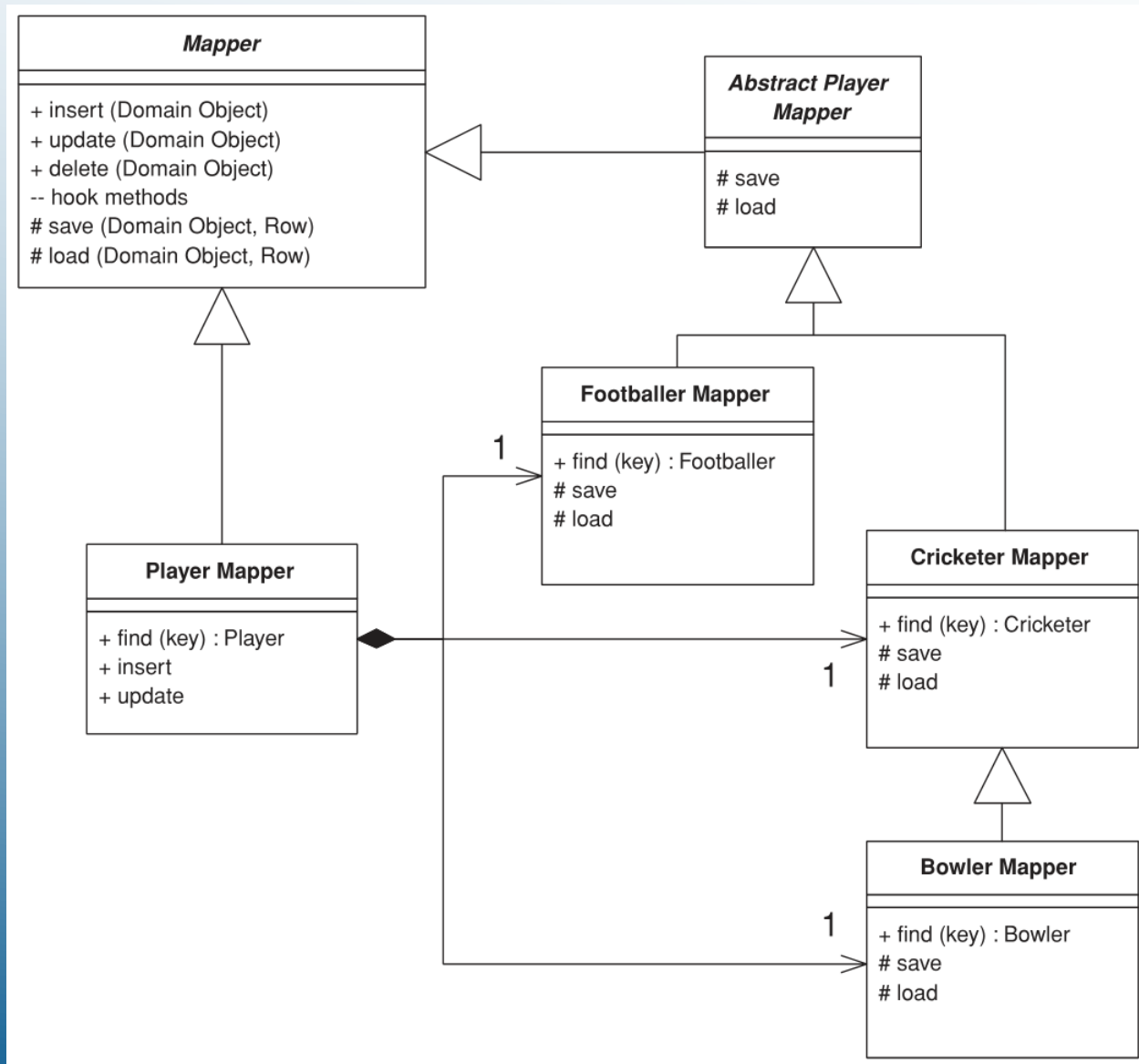
- Each table is self-contained and has no irrelevant fields.
- There are no joins to do when reading the data from the concrete mappers.
- Each table is accessed only when that class is accessed, which can spread the access load.

Cons:

- Primary keys can be difficult to handle.
- You can't enforce database relationships to abstract classes.
- If the fields on the domain classes are pushed up or down the hierarchy, you have to alter the table definitions.
- If a superclass field changes, you need to change each table.
- A find on the superclass forces you to check all the tables, which leads to multiple database accesses (or a weird join).

Concrete Table Inheritance – implementation

The generic class diagram of **Inheritance Mappers**



Concrete Table Inheritance – implementation

- Each mapper is linked to the database table that's the source of the data. A data set holds the data table.
- The gateway class holds the data set within its data property. The data can be loaded up by supplying suitable queries.
- Each concrete mapper needs to define the name of the table that holds its data.
- The player mapper has fields for each concrete mapper.

Loading an Object from the Database

- Each concrete mapper class has a find method that returns an object given a key value.
- The abstract behavior on the superclass finds the right database row for the ID, creates a new domain object of the correct type, and uses the load method to load it up.
- The actual loading of data from the database is done by the load method, or rather by several load methods: one each for the mapper class and for all its superclasses.

Concrete Table Inheritance – implementation

- Each mapper is linked to the database table that's the source of the data. A data set holds the data table.
- The gateway class holds the data set within its data property. The data can be loaded up by supplying suitable queries.
- Each concrete mapper needs to define the name of the table that holds its data.
- The player mapper has fields for each concrete mapper.

Loading an Object from the Database


- Each concrete mapper class has a find method that returns an object given a key value.
- The abstract behavior on the superclass finds the right database row for the ID, creates a new domain object of the correct type, and uses the load method to load it up.
- The actual loading of data from the database is done by the load method, or rather by several load methods: one each for the mapper class and for all its superclasses.

Using of Inheritance patterns

We choose pattern by the criterions:

- Storage efficiency
- Performance
- Database querying

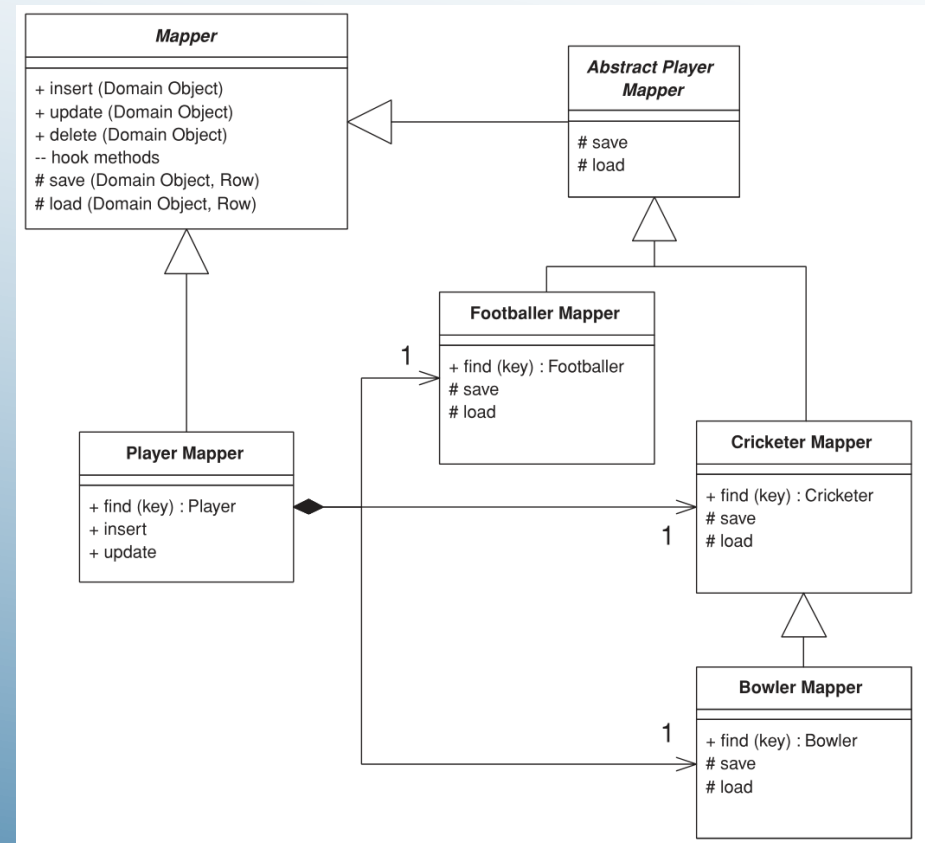
They can also be combined (e.g. different patterns can be used for different levels of the inheritance hierarchy)



Inheritance mappers

A structure to organize database mappers that handle inheritance hierarchies the same for all inheritance patterns.

- A general structure to organize database mappers that handle inheritance hierarchies.
- There is a need to minimize the amount of code needed to save and load the data to the database.
- Although the details of this behavior vary with the inheritance mapping scheme the general structure works the same for all of them.



You want to provide both abstract and concrete mapping behavior that allows you to save or load a superclass or a subclass.

Inheritance mappers – how it works

- You organize the mappers with a hierarchy.
- Each domain class has a mapper that saves and loads the data for that domain class.
- Concrete mappers know how to map the concrete objects in the hierarchy.

In case of you need mappers for the abstract classes.

- It can be implemented with mappers that are actually outside of the basic hierarchy but delegate to the appropriate concrete mappers.

Inheritance mappers – how it works


- The concrete mappers are the mappers for footballer, cricketer, and bowler.
- Their basic behavior includes the find, insert, update, and delete operations.
- The find methods are declared on the concrete subclasses because they will return a concrete class.
- The basic behavior of the find method is to find the appropriate row in the database, instantiate an object of the correct type (a decision that's made by the subclass), and then load the object with data from the database.
- The load method is implemented by each mapper in the hierarchy which loads the behavior for its corresponding domain object.

Inheritance mappers – when to use it?


This general scheme makes sense for any inheritance based database mapping.

The alternatives involve such things as duplicating superclass mapping code among the concrete mappers and folding the player's interface into the abstract player mapper class.

The former is a heinous crime, and the latter is possible but leads to a player mapper class that's messy and confusing. On the whole, then, it's hard to think of a good alternative to this pattern.

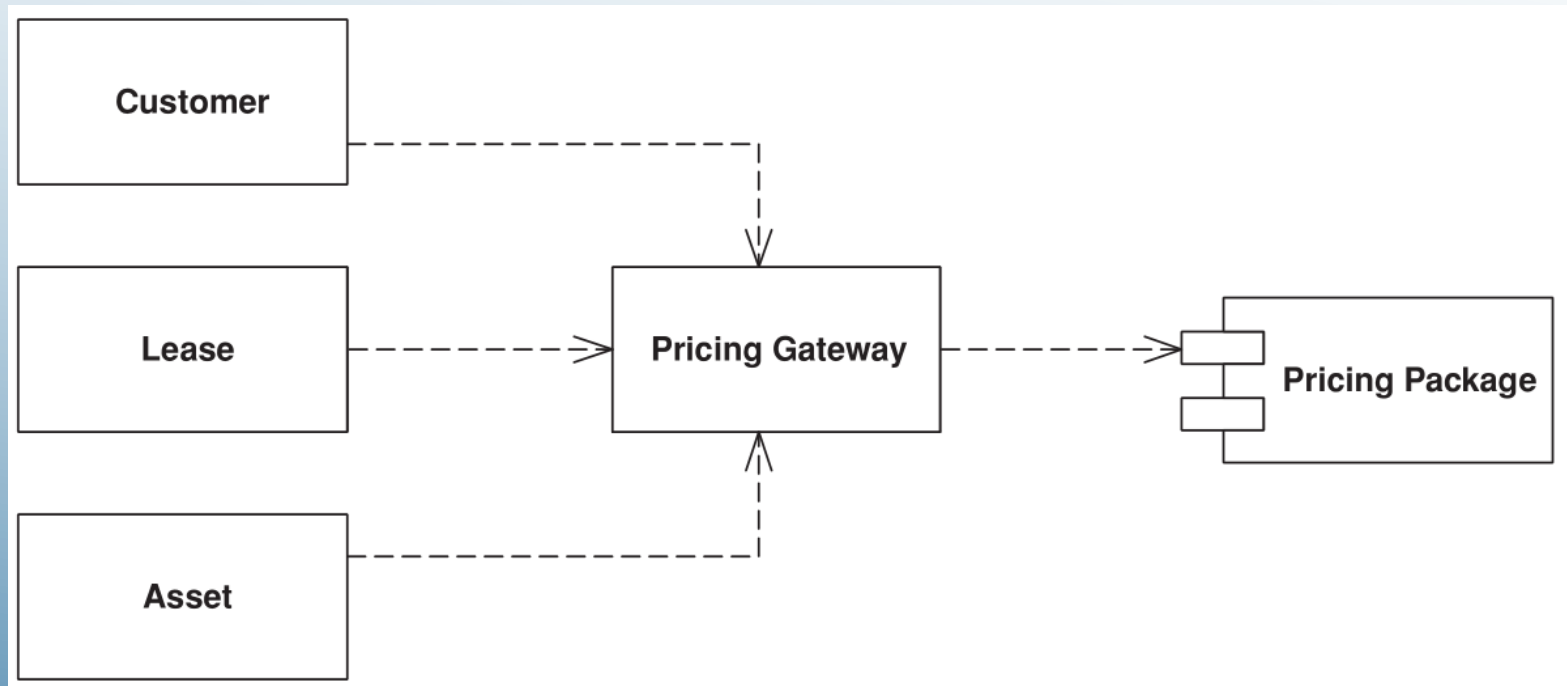
Three white lines of varying lengths and slopes are positioned in the bottom right corner of the slide, extending from the right edge towards the bottom left.

General patterns

- They are the basis for a set of more specific patterns.
 - They are a generalization of a group of patterns with common behavior.
 - They describe behaviour at a higher level of design abstraction.
- 

General patterns - Gateway

An object that encapsulates access to an external system or resource. (Table data gateway)



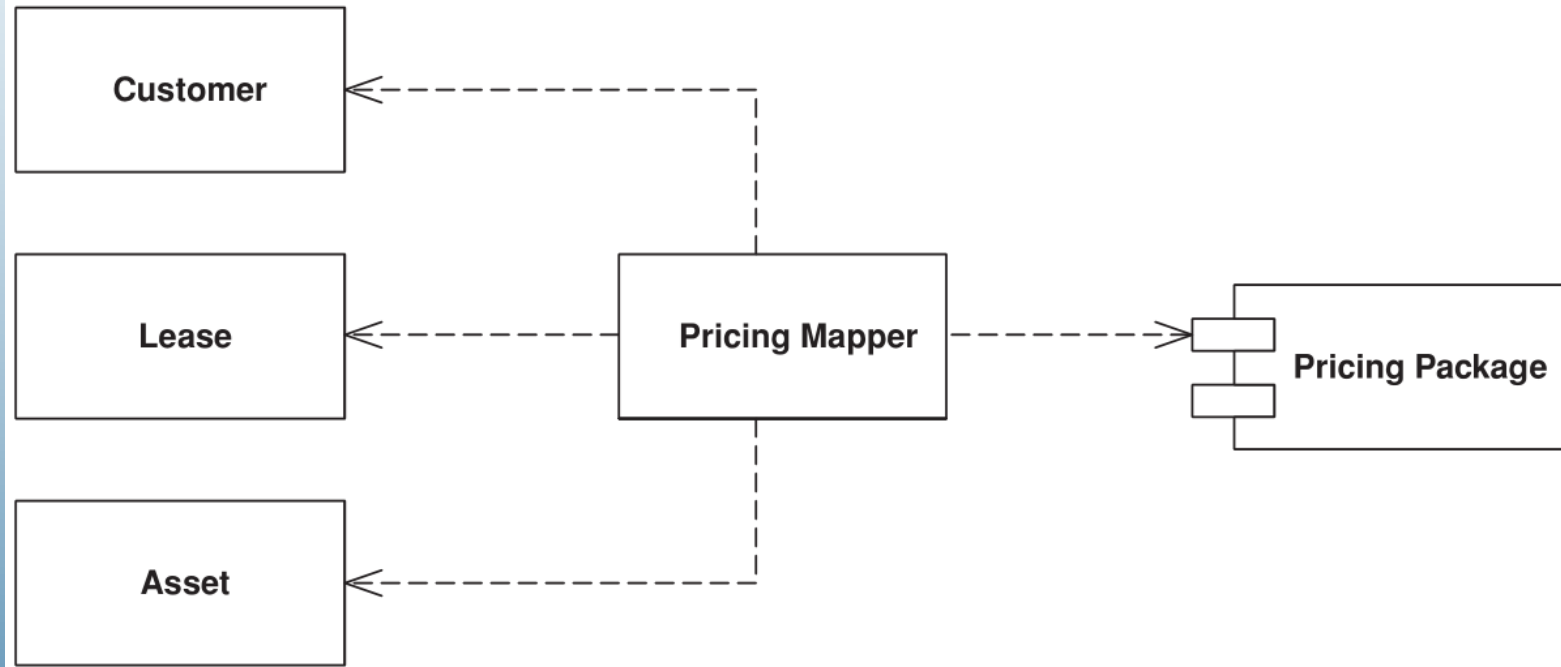
Wrap all the special API code into a class whose interface looks like a regular object. Other objects access the resource through this Gateway, which translates the simple method calls into the appropriate specialized API.

Gateway - how it works?

- In reality this is a very simple wrapper pattern. Take the external resource.
- Create a simple API for your usage and use the Gateway to translate to the external source.
- Keep a Gateway as simple as you can. Focus on the essential roles of adapting the external service and providing a good point for stubbing.
- The Gateway should be as minimal as possible and yet able to handle these tasks.
- Any more complex logic should be in the Gateway's clients.

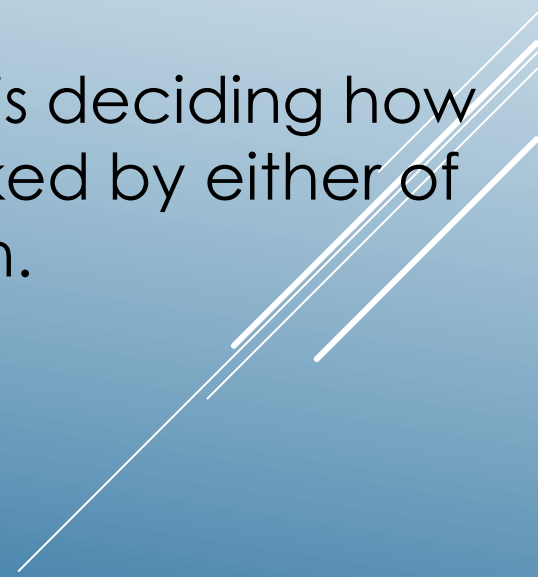
General patterns - Mapper

An object that sets up a communication between two independent objects. (Data mapper)




Sometimes you need to set up communications between two subsystems that still need to stay ignorant of each other. This may be because you can't modify them or you can but you don't want to create dependencies between the two or even between them and the isolating element.

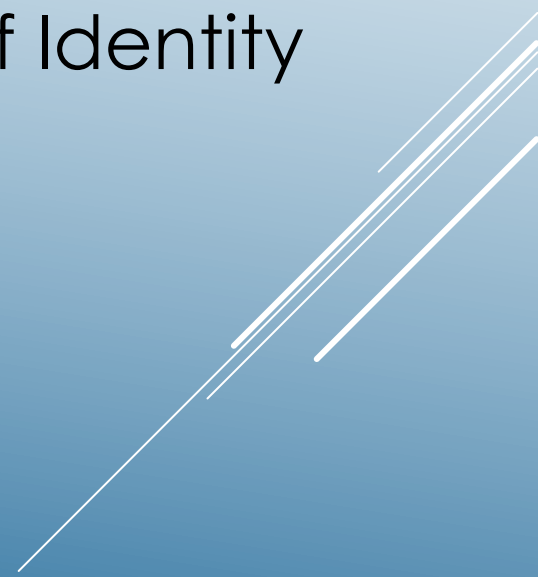
Mapper - how it works?

- A mapper is an insulating layer between subsystems.
 - It controls the details of the communication between them without either subsystem being aware of it.
 - A mapper often shuffles data from one layer to another. Once activated for this shuffling, it's fairly easy to see how it works.
 - The complicated part of using a mapper is deciding how to invoke it, since it can't be directly invoked by either of the subsystems that it's mapping between.
- 
- Three parallel white diagonal lines are located in the bottom right corner of the slide, extending from the middle of the right edge towards the bottom left.

General patterns – Layer Supertype

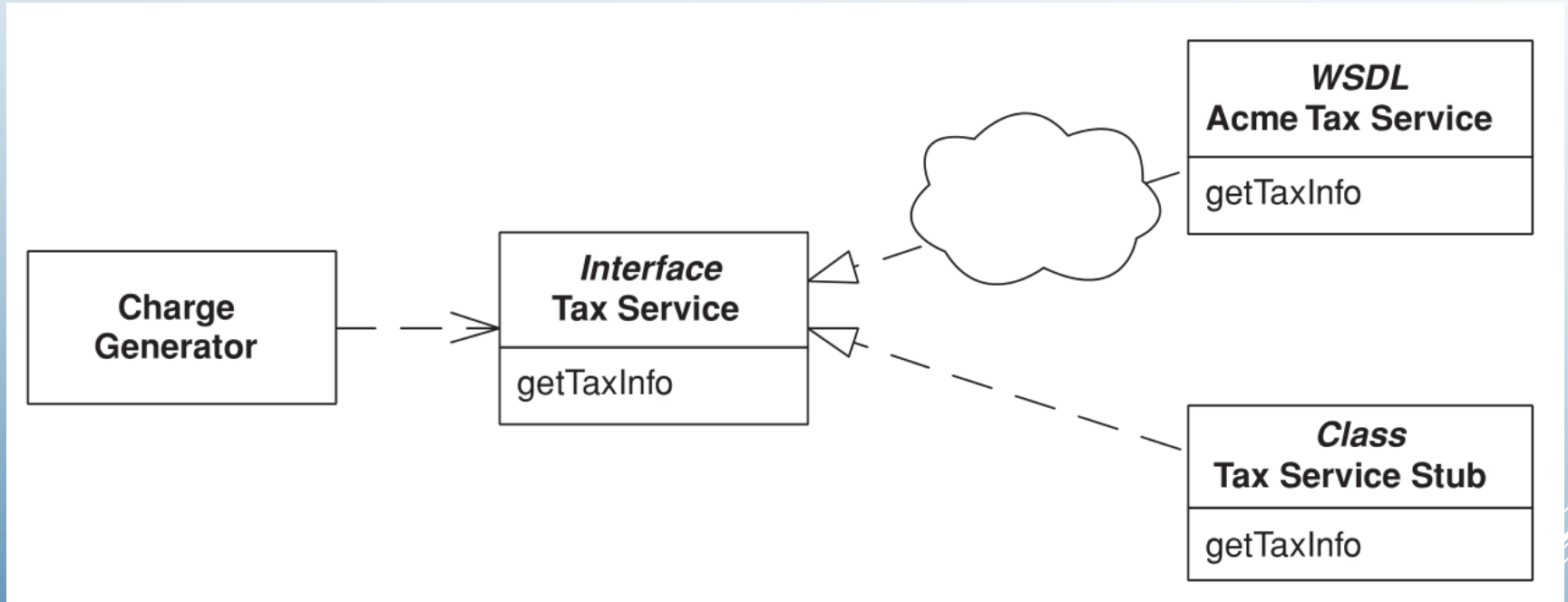
- A type that acts as the supertype for all types in its layer.
 - It's not uncommon for all the objects in a layer to have methods you don't want to have duplicated throughout the system. You can move all of this behavior into a common Layer Supertype.
 - Use Layer Supertype when you have common features from all objects in a layer.
- 

Layer Supertype - examples

- Abstract class for Identity Field pattern.
 - Abstract class for common behavior of objects of the Data pattern Mapper.
 - Domain Object superclass for all the domain objects in a Domain Model - common features, such as the storage and handling of Identity Fields, can go there.
- 
- A series of three parallel white diagonal lines in the bottom right corner of the slide, pointing towards the bottom right.

Service Stub (Mock objects – in XP)

Removes dependence upon problematic services during testing.



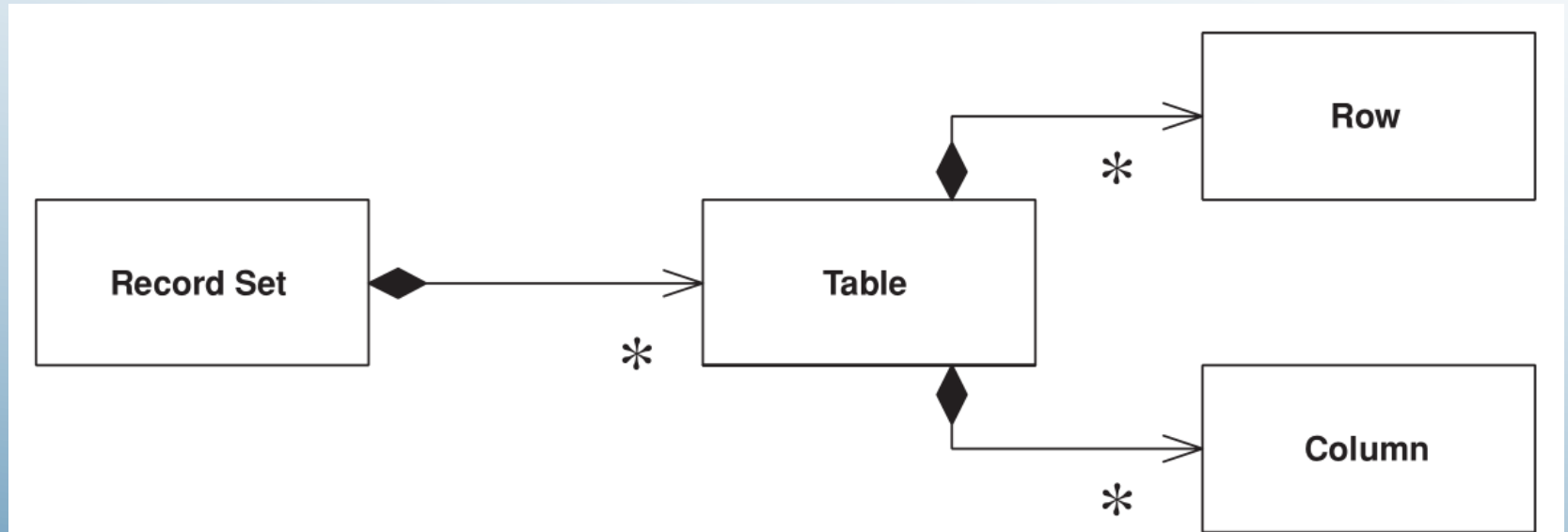
Replacing the service during testing with a Service Stub that runs locally, fast, and in memory improves your development experience.

Service stub - how it works?

- Define access to the service with a Gateway.
- The Gateway should not be a class but rather a Separated Interface so you can have one implementation that calls the real service and at least one that's only a Service Stub.
- The desired implementation of the Gateway should be loaded using a Plugin pattern.
- The key to writing a Service Stub is that you keep it as simple as possible - complexity will defeat your purpose.

Record sets

An in-memory representation of tabular data.



The idea of the Record Set is to give you an in-memory structure that looks exactly like the result of an SQL query but can be generated and manipulated by other parts of the system.

Record Sets – how it works?

- A Record Set is usually something that you won't build yourself, provided by the vendor of the software platform you're working with. (examples include the data set of ADO.NET and the row set of JDBC 2.0.)
- The first essential element of a Record Set is that it looks exactly like the result of a database query.
- You can use the classical two-tier approach of issuing a query and throwing the data directly into a data-aware UI with all the ease that these two-tier tools give you.
- The second essential element is that you can easily build a Record Set yourself or take one that has resulted from a database query and easily manipulate it with domain logic code.

Semestral project -Artifact 6


System architecture

Description of the system architecture - layout and interconnection of the logical and physical layers.

- Diagram of components
- Deployment diagram



Exercise tasks

- Present your current state of semestral work code.
 - Discussion of the domain model focuses on the patterns used and the interaction of objects within the patterns.
 - Continue the implementation of inheritance mapping, layer supertype patterns for the semester project.
- 

Lecture checking questions

- Describe the nature of Single / Class / Concrete Table Inheritance patterns and in what situations they are appropriate to use.
- Write a code fragment that shows that you have used the Single Table Inheritance pattern.
- What is the difference between the Class and Concrete Table Inheritance patterns? Write two code snippets that show that you have used these patterns.
- Think of and describe an example of using the Gateway pattern.
- Think of and describe an example of using the Mapper pattern.
- Devise and describe an example using the Layer Supertype pattern. Write a code fragment using inheritance, from that shows that you have used this pattern.
- Devise and describe an example using the Service Stub (Mock Object) pattern. Write a code fragment that shows that you have used this pattern.