Object-Oriented Programming

Coupling

Michael L. Collard, Ph.D.

Department of Computer Science, The University of Akron

Coupling

Degree of interdependence between software modules; a measure of how closely connected two routines or modules are; the strength of the relationships between modules.

- External measure of the relationships of a class to other classes

Coupling

- Dependency between elements
- Degree or reliance between elements
- Increasing cohesion may lead to more coupling (Why?)
- Entropy of systems and designs increases coupling
- Coupling needs to be limited and controlled

Why is Coupling Important?

- Perhaps the most important characteristics of a system
- Affects development path
- Affects how we partition the system for testing
- Affects how much reuse is possible
- Significant effect on the complexity of a system

Coupling Goal

- Minimize external interaction (extramural) with other elements (*coupling*)

- For classes: Minimize and reduce relationships with other classes

States of Coupling

- *tightly coupled* High degree of coupling

  Very difficult to develop/test/maintain/use

- *loosely coupled* Low degree of coupling

  Much easier to develop/test/maintain/use

- *decoupled* Classes with zero to minimal coupling

  Very easy to develop/test/maintain/use

  Common refactoring activity

Types of Coupling

- Message Coupling (very best)
- Data Coupling (best)
- Stamp Coupling
- Control Coupling
- Common Coupling
- Content Coupling (worst)

Content Coupling

- A module directly references the content of another module:

  One module *p* modifies a statement of another module *q*

  One module *p* references or alters the local data of another module *q*

  One module *p* branches into another module *q*
- Content-coupled modules are inextricably interlinked

  Change to module *q* requires a change to module *p*, including recompilation

  Reusing module *p* requires using module *q*
- Exposing data members via public access is a form of this

Common Coupling

- Using global variables, i.e., global coupling
- All modules have read/write access to a piece of global data
- Functions exchange data using global data instead of arguments

Common Coupling Example

```
// NOTE: Not a field in a class
int numStudents = 0;
```

```
void registerStudent() {
    // ...
    ++numStudents;
}
```

```
void deregisterStudent() {
    // ...
    --numStudents;
}
```

Common Coupling (cont)

- A situation where a single function has write access and all other functions have only read access is not *common coupling*

- To determine why a variable has a particular *state* (value), you have to examine all functions

- Side effects, so all the code in a function needs to be examined

- Function is exposed to more data than is needed

Control Coupling

```
void drawing(std::string_view command) {
    if (command == "drawCircle")
        drawCircle()
    else
        drawRectangle()
}
```

- Client passes a flag or command that explicitly controls what the called code is doing
- Independent reuse is not possible
- Client should pass data and leave control path decisions private to a module

Stamp Coupling

```
swap(v[], i, j);

calcSalary(employee);
```

- One module passes more data than needed to another module
- Often involves records (structs) with lots of fields
- Entire record passed, but only a few fields are used
- Efficiency considerations?

Data Coupling

```
swap(v[i], v[j]);

calcSalary(employee.PayInfo);
```

- One module only passed the data needed by the other module
- Only the required data is passed from one module to another
- All arguments are homogeneous (i.e., all the same type) data items

  simple data type

  complex data type, but the client code uses all parts
- Allows for comprehension, reuse, maintenance, security, etc.

Message Coupling

- Client code passes parameters via a non-private data format
- Most flexible, since any language or tool can generate the data
- Possible to store or cache requests
- All arguments are homogeneous (i.e., all the same type) data items

  simple data type

  complex data type, with all parts used
- Allows for comprehension, reuse, maintenance, security, …
- Potential performance overhead (?)

	Tightly Coupled	Loosely Coupled
Changes	Change in one module leads to changes in other modules	Allows independent changes to modules
Assembly	Difficult to assemble separately	Easier to assemble separately
Test	Difficult to test individual parts	Easily test individual parts
Reuse	Difficult to reuse individual parts	Easily reuse individual parts
Additional Features	Difficult to add feature/use of the system	Easier to add feature/use of the system

Coupling Observations

- If there is no way to separate modules so that they are loosely coupled, consider combining or packaging them together

- Subclass Coupling - Between derived and base class

Advantages of Tightly-Coupled Modules

- Only uses the data available and can ignore other parts of the data

- Decoupling leads to abstractions, which can make programming more difficult than direct use

- In general, tightly-coupled modules can be more efficient

- but typically at a high cost of flexibility

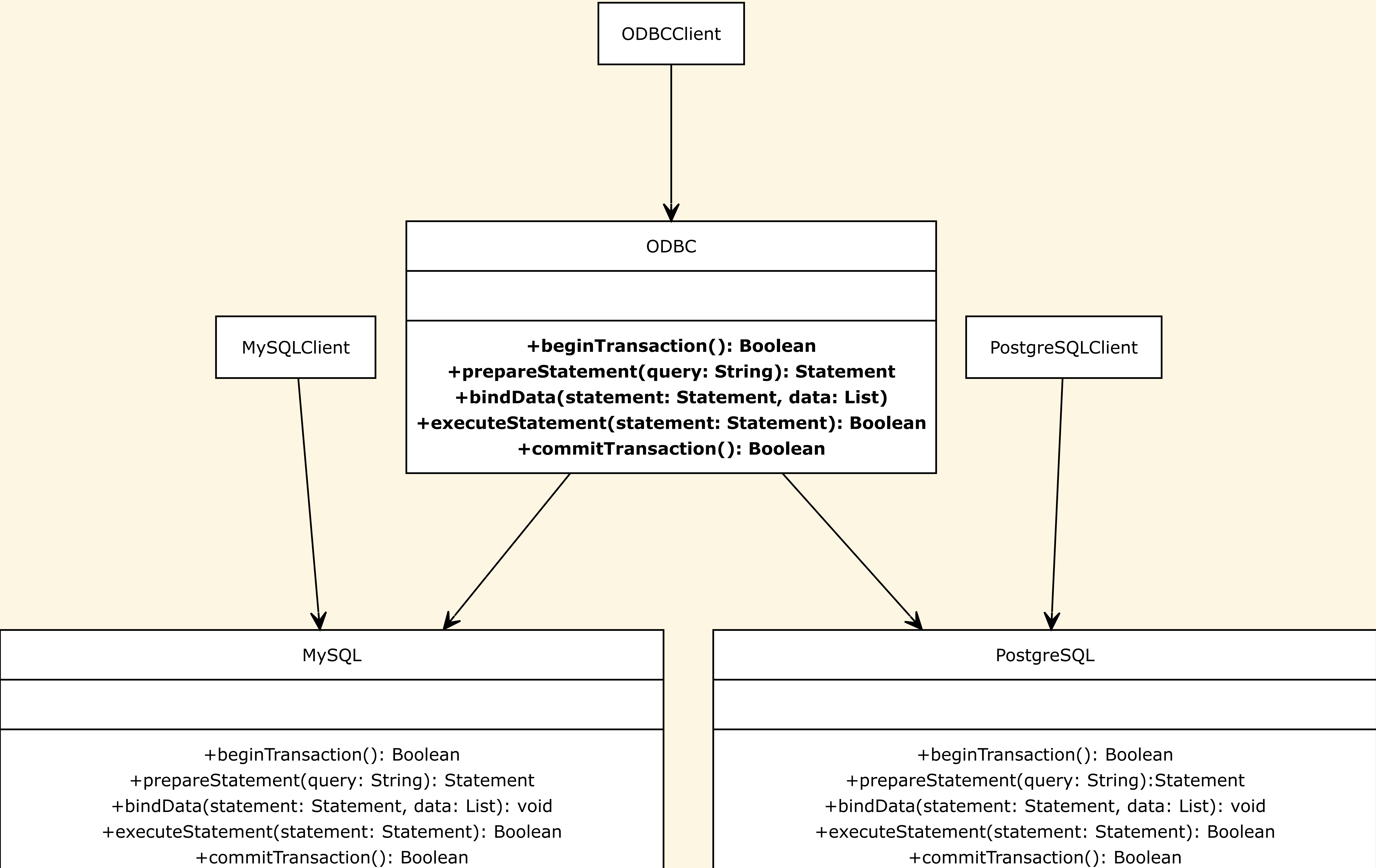
Advantages of Tightly-Coupled Modules Examples

- Obtain only the data that is needed. E.g., parsing source code for a specific query

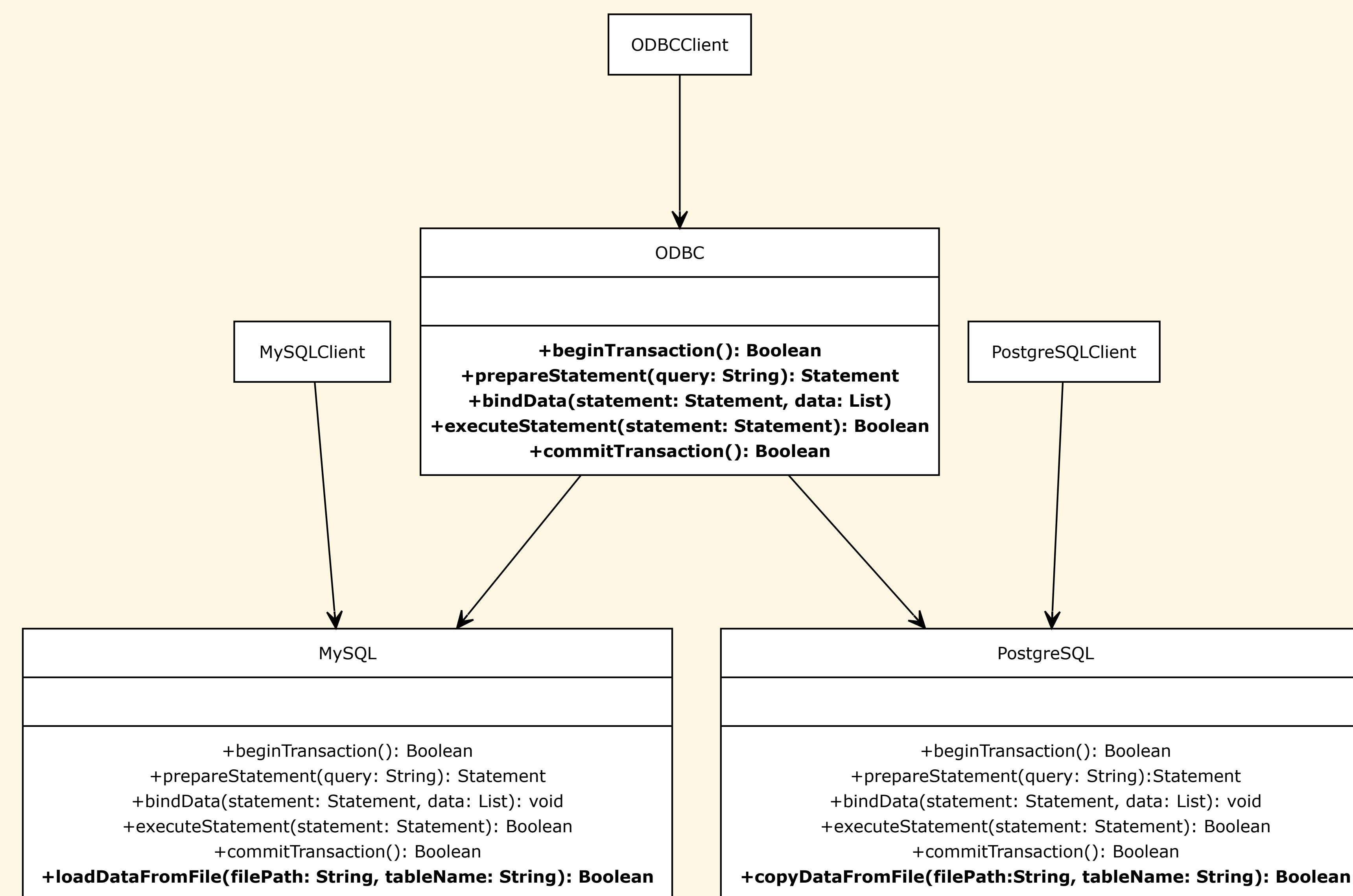
- Directly use the data with no need to translate to/from a data format

- Layers that are not needed can be skipped. E.g., ISO 7-level network model vs. TCP/IP

Comparison: Client using MySQL vs ODBC (Overview)



Comparison: Client using MySQL vs ODBC

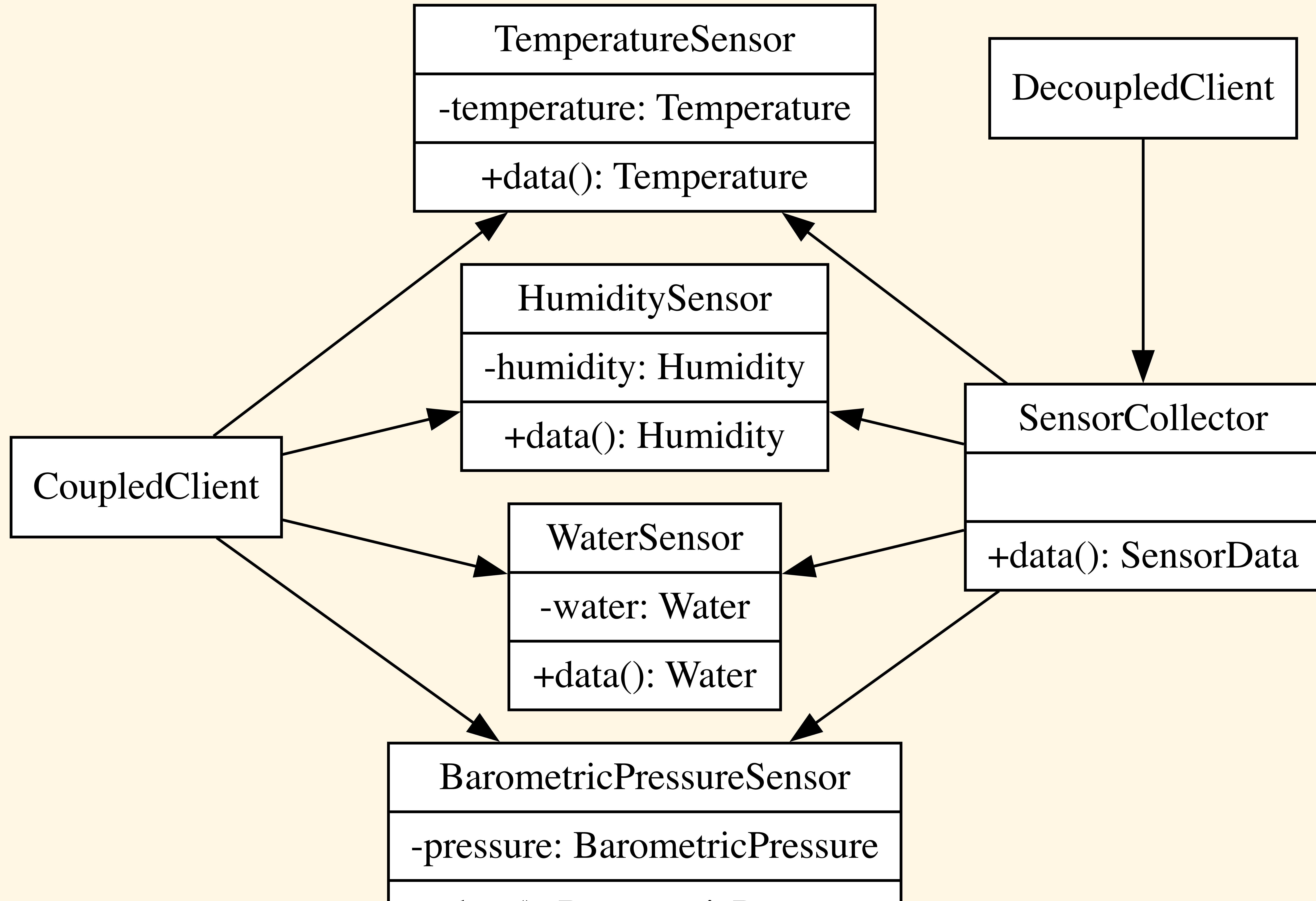


- Clients tightly coupled to MySQL and PostgreSQL can use possibly more efficient features

- However, since they are tightly coupled, they cannot switch to another database

- Clients using ODBC are decoupled from MySQL and PostgreSQL and can switch to an underlying MySQL, PostgreSQL, or even another database

Use Data Directly vs a Common Format



Search: Find the declarations of all variables with the type of `int`

- Assume the srcML format
- Search as you parse
- As you parse the XML, output any variable declarations that have the type `int`
- Ignores anything else
- Same approach as in srcFacts
- As fast as possible
- Tightly coupled to the XML parsing

XPath Query: Find the declarations of all variables with a type of `int` in a program

```
//src:decl_stmt[src:decl/src:type/src:name='int']
```

- Assume the srcML format
- Parse and apply XPath
- Use any XML tool that lets you use XPath queries
- Disadvantage: The entire DOM-like tree must be loaded into memory
- Disadvantage: May be a speed penalty

srcQL Query: Find all the variables with a type of `int` in a program

```
FIND int $V;
```

- Parse and apply the *srcQL* query language
- Part of srcml 1.1.0
- Disadvantage: The entire DOM-like tree must be loaded into memory
- Disadvantage: May be a speed penalty

Search/Query & srcML Coupling