MSO
Template Method Pattern
Factory Method Pattern

Wouter Swierstra & Hans Philippi

October 30, 2018

Two more patterns

- Template Method Pattern
- Factory Method Pattern

We will see them working in an integrated way

Template Method case study: Databases

Encapsulate variation!

```
prepareQuery(query1);
...
if (database == ORACLE)
  conn = formatOracleConnect();
else
  conn = formatSQLServerConnect();
if (database == ORACLE)
  res = formatOracleSelect(conn, query1);
else
  res = formatSQLServerSelect(conn, query1);
processResults(res);
```

Case study: Databases

Suppose we need to write software that generates SQL queries on different databases (both Oracle and SQL Server)

In principle the queries have the same structure:

1. Format the CONNECT command
2. Send the CONNECT command to the database
3. Format the SELECT command
4. Send the SELECT command to the database
5. Return the answer from the database

. . . but there is slight variation in the formatting procedures for the different database backends

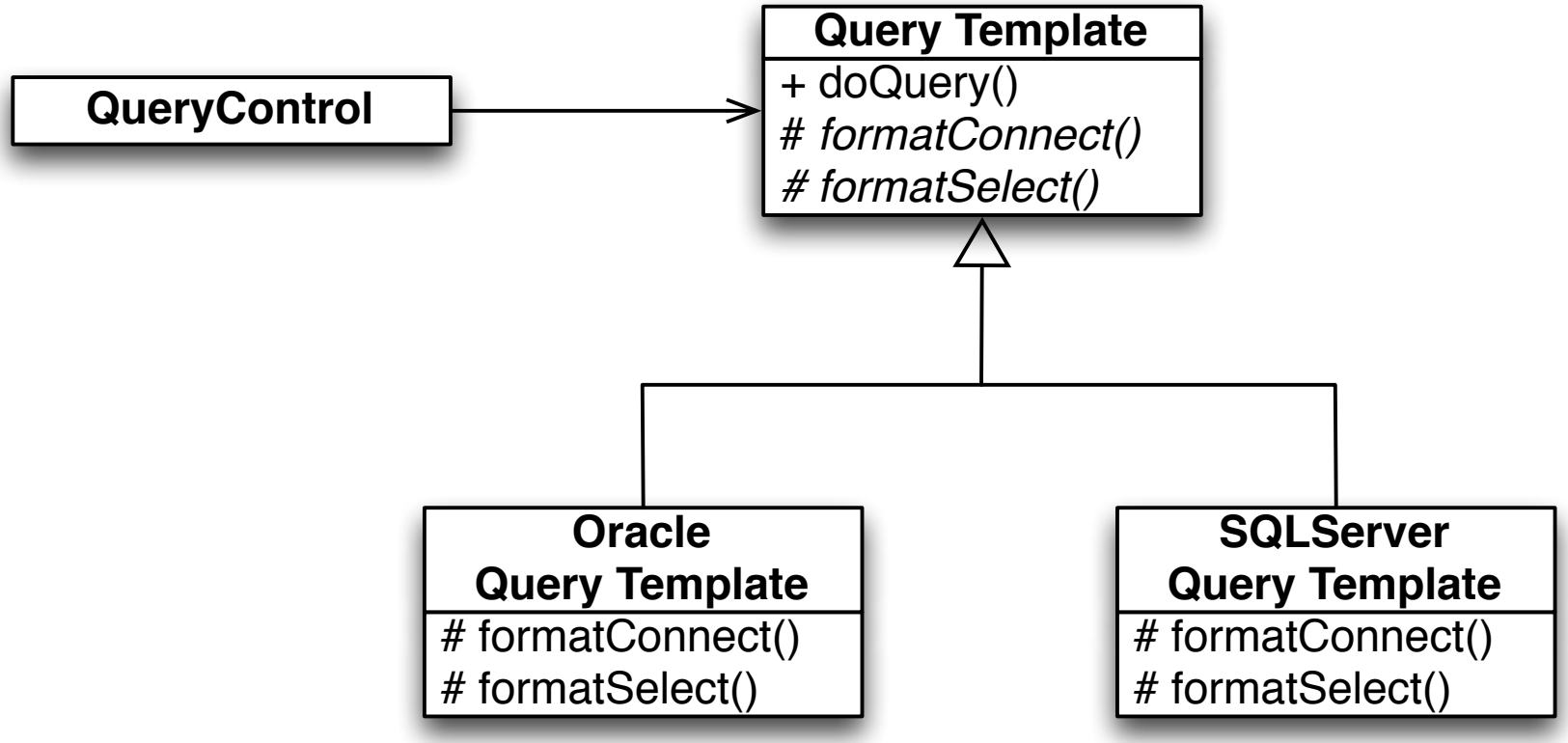
Template Method: Intent

The Gang of Four define the intent of the Template Method Pattern as:

> *Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Redefine the steps in an algorithm without changing the algorithm's structure.*

This sounds similar to our database problem ...

Template Method



Where the magic happens

The QueryTemplate class defines the following method:

```
public doQuery(String query)
{
   ...
   conn = formatConnect();
   ...
   res = formatSelect(conn, query);
   processResults(res);
}
```

An *abstract* class (QueryTemplate) defines a public, concrete method (doQuery), that itself calls abstract, protected methods (formatSelect and formatConnect)

Code-based CVA

Template Methods are a great way to clean up 'duplicated' methods, or fixing code that has been copy-pasted and edited

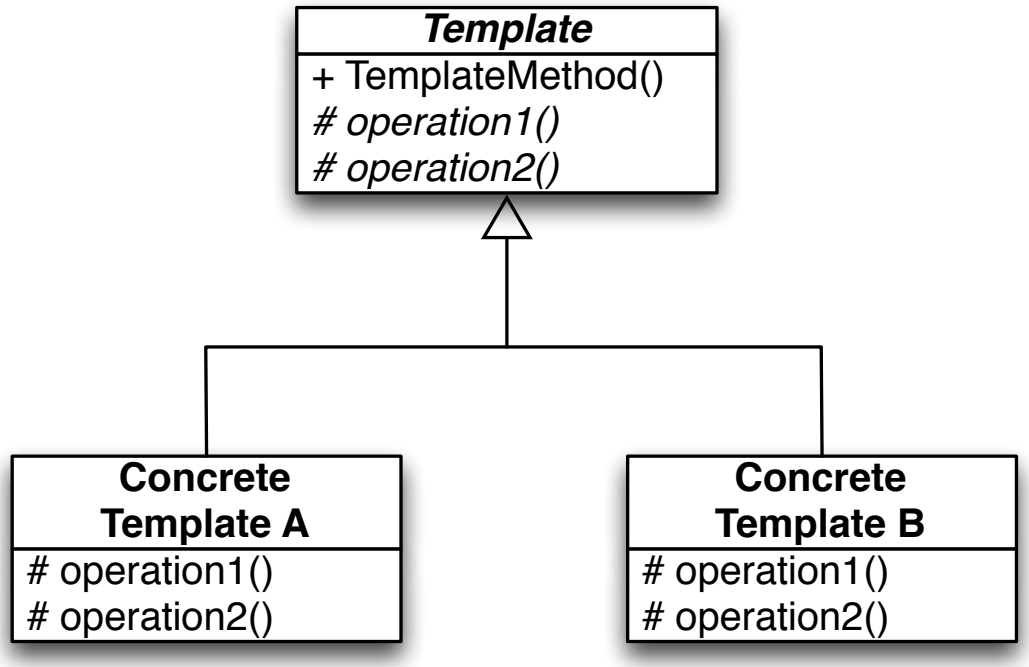
1. Look for the common parts – these go in the abstract class
2. Look for the variation – these go in the concrete instances

Template method I

- **Intent:** Define the skeleton of an algorithm, deferring some steps to subclasses, making it possible to redefine some of the steps in the algorithm, without changing the algorithm's structure.
- **Problem:** There is a procedure or set of steps that is consistent at one level of detail, but individual steps may have different implementations.
- **Solution:** Allows for a definition of substeps that vary while maintaining a consistent basic process.
- **Participants:** The Template Method defines an abstract class with a concrete method, that uses abstract methods that need to be overridden.

Template method II

- **Consequences:** Templates provide a good platform for code reuse. They are also helpful in ensuring the required steps are implemented.
- **Implementation:**



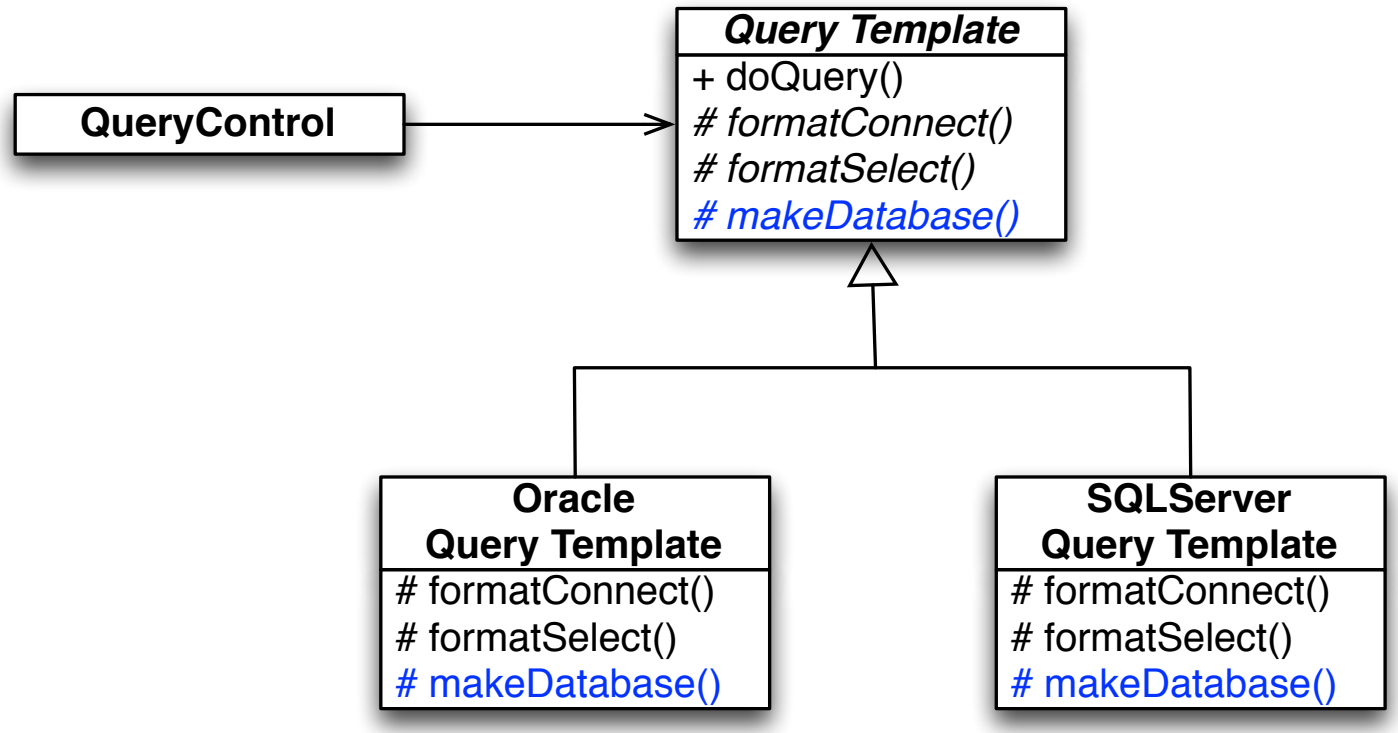
There we go again: creation?

- In our case study, the Template Method pattern showed how to handle variation in database communication
- An abstract QueryTemplate object hides implementation details
- But who creates the associated database object?

Assigning responsibility

If the database creation varies between the two Template subclasses (for Oracle or SQL Server databases), the *subclasses* should be responsible

The diagram


QueryControl
Query Template
+ doQuery()
# formatConnect()
# formatSelect()
# makeDatabase()
Oracle
Query Template
formatConnect()
formatSelect()
makeDatabase()
SQLServer
Query Template
formatConnect()
formatSelect()
makeDatabase()

What does this achieve?

- The QueryTemplate class does not know which database is created
- It just knows that all concrete derived classes will define a method to create a database object
- The derived classes are responsible for the creation

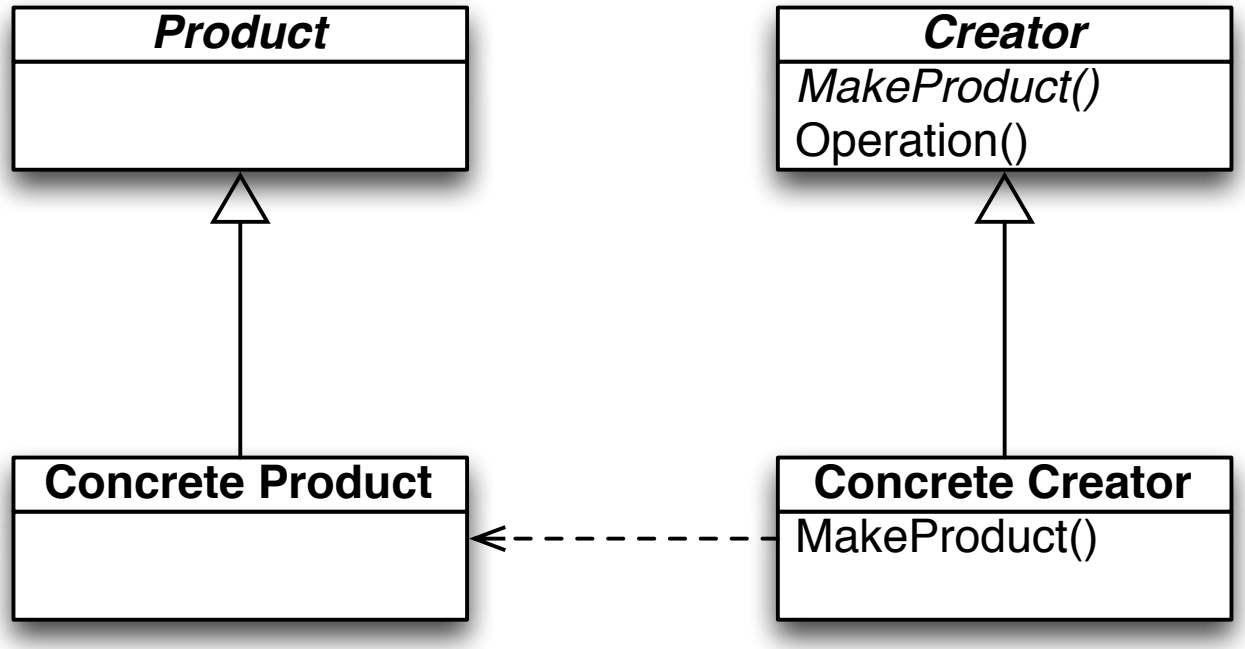
This is called the *Factory Method* pattern, which is used heavily in C# collections libraries

Factory Method: Intent

The Gang of Four define the intent of the Factory Method Pattern:

> *Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.*

Factory Method



Example: (book is vague on this issue)
Creator <=> Query Template
MakeProduct() <=> MakeDatabase()
Operation() <=> logic in creation process

Overview

1. Identifying objects based on responsibilities (CVA, Analysis matrix, noun-verb analysis, GRASP)
2. Deciding how to use these objects (design patterns)
3. Deciding how to manage these objects (factories)

Philosophy behind Factories

- Conceptually similar objects are often used in the same way
- When creating these objects, there is some logic involved that decides to choose one specific concrete class
- The *using* class should not know which specific concrete class it is using – it should program to an interface, not an implementation
- So if the using class cannot choose how to instantiate objects, someone must be responsible - enter Factories

Factories and changing requirements

In practice most new requirements affect *either*

- the users of an object or system (or its internal implementation)
- or the logic controlling the choice of which objects to create

It is much less common that *both* these things need to change

Golden rule of object creation

Try to make an object responsible for using **or** *creating, but never for both*

Epilog

- We have seen a lot of tools and concepts to support analysis and design
- The pattern story is not finished here . . .
- . . . find your own!
- Have an open mind towards new concepts, patterns, ideas . . .
- . . . instead of sticking to what you already know!