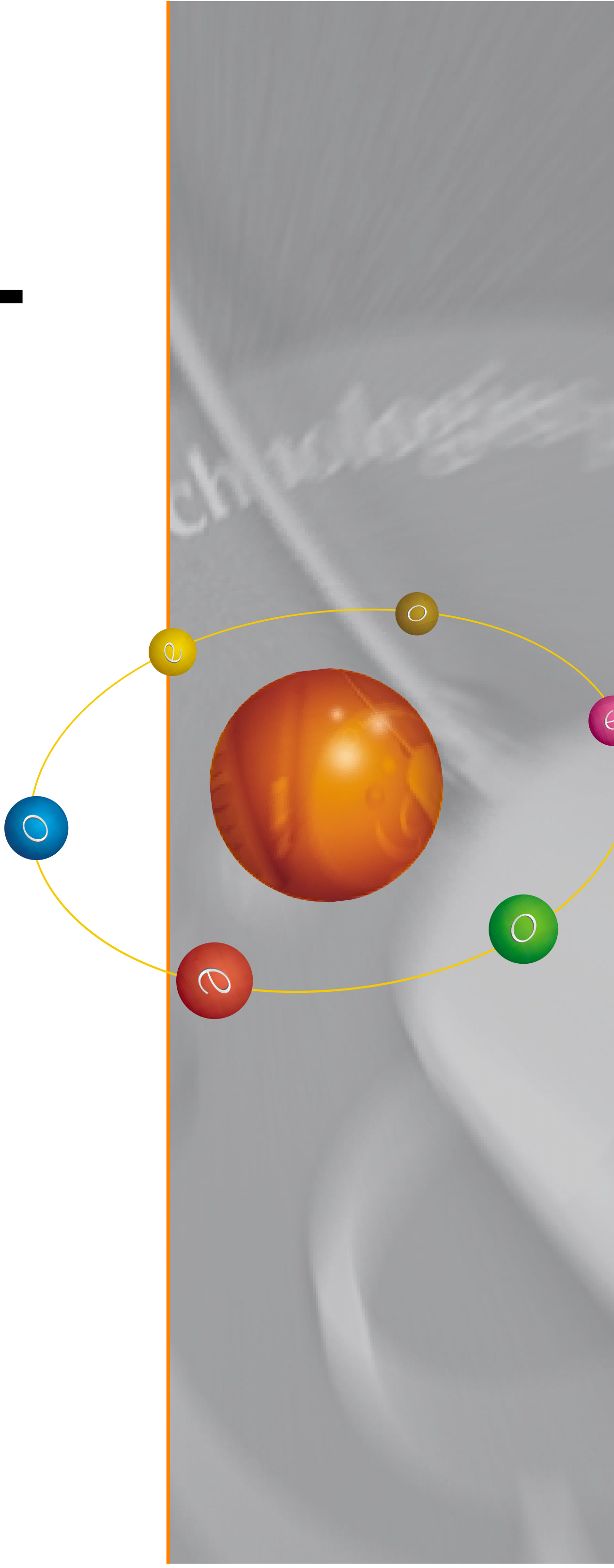
Objecteering/UML

Objecteering/Design Patterns
for C++/Java User Guide

Version 5.2.2



Objecteering Software

www.objecteering.com

Taking object development one step further

Objecteering/UML version 5.2.2 - CODOBJ 001/001

Contents

Chapter 1: Overview

Overview of the Objecteering/Design Patterns for C++/Java module

Introduction

Welcome to the *Objecteering/Design Patterns for C++/Java* user guide!

The *Objecteering/Design Patterns for C++* and *Objecteering/Design Patterns for Java* modules work in conjunction with the *Objecteering/C++* and *Objecteering/Java* code generation modules.

The *Objecteering/Design Patterns for C++* and *Objecteering/Design Patterns for Java* modules are intended for designers who are familiar with object oriented design. They are made up of a group of "*design patterns*", each of which takes advantage of automated assistance. The same patterns are provided for C++ and Java. However, the related code generation modules also generate code specific to the language in question.

The *Objecteering/Design Patterns for C++* and *Objecteering/Design Patterns for Java* modules are available in the "Professional" and "Enterprise" editions of Objecteering/UML.

General characteristics

The *Objecteering/Design Patterns for C++* and *Objecteering/Design Patterns for Java* modules allow you to modify the model, in order to automatically apply the design models described by Erich Gamma & Al. These design models describe the structure and collaboration of a limited number of classes, so as to provide a solution to a design problem which appears in a given context. Most of the time, the problems dealt with are aimed at improving the adaptability and flexibility of an underlying model.

In real terms, the application of a Design Pattern to a model is translated by:

- the creation of new classes, and new associations and attributes where necessary
- the creation of new operations
- the implementation of operational code (C++ code and/or Java code) which transposes the described collaboration

When and why to use Design Patterns ?

Design Patterns are aimed at improving the adaptability of the model. This allows you, amongst other things, to guarantee the present and future extensibility of your model.

Extensibility of functions, such as:

- ♦ the subsequent addition of new functions independent of present functions (Command pattern and Chain of Responsibility command)
- ♦ the adaptation of a sub-processing operation within an algorithm (Template Method pattern)
- ♦ the possibility of modifying the algorithms which apply to elements of the Visitor pattern

Extensibility of structures, such as:

- ♦ the grouping of elements in a recursive composition tree (Composite pattern)
- ♦ the possibility of connecting hierarchies of separated classes without interfering in any of them (Adapter pattern)
- ♦ the delegation of the creation or grouping of instances (Prototype and Singleton patterns).

Design Patterns are, in fact, recurrent design elements, independent of application domains. As such, they are used during development, in the design phase, to modify the analysis model. The application of Design Patterns to an analysis model happens over two stages:

1 - *Choice of pattern to be applied*: This results from the analysis of motivations and constraints (sometimes called forces) regarding the pattern which offers an adequate solution. These motivations and constraints must be organized in order of priority. Finally, for each candidate pattern, a parallel must be established between the consequences of the application of the pattern and the initial motivations and constraints. An incompatible consequence with a priority constraint is a possible reason for the rejection of a candidate pattern.

2 - *Implementation of the pattern*: The model is modified in order to integrate the selected pattern.

The *Objecteering/Design Patterns for C++* and *Objecteering/Design Patterns for Java* modules allow you to automate this second phase. By doing this, they eliminate the risk of errors inherent to the manual implementation of the pattern, and allow the designer to concentrate on the important aspect of the use of Design Patterns, in other words, the selection of the correct pattern.

Design Patterns provide an opportunity to communicate design choices, through a common language. The documentation associated with each pattern serves as a heritage common to the developers' community, thus avoiding the need to re-describe these recurring design elements in detail each time. For further information, please refer to chapter of this user guide, the "*Bibliography*". Most of the works cited in the bibliography are Design Pattern catalogues. Some of these patterns are grouped into Pattern Languages, in other words, a complementary family of models, which covers a particular area. These catalogues organize design models into 3 categories:

1 - *Idioms*: Recurring constructions which are based on the specificities of a programming language. As such, they cannot be transposed between languages.

2 - *Design Patterns*: Characteristic constructions independent of programming languages. Design Patterns formalize the occasional structural choices which can be implemented independently of the realization environment. Idiomatic implementations can be proposed for Patterns.

3 - *Frameworks*: These express the general form of an application or a sub-system, by forming the infrastructure of the interaction. All or a part of a sub-system based on a framework must do this, by deriving certain particular classes of this framework, and by redefining the methods destined to be used. The logic of the interaction of classes must not clash with that of the framework, since this is what guides execution logic.

Structure of the Objecteering/Design Patterns for C++/Java module

Introduction

In this user guide, an individual section is dedicated to each design pattern.

Explanations are provided with regard to an example project, which can be imported by creating a new UML modeling project, and by running the "*Design Patterns for .../Import Design Patterns Samples Project*" command on the UML model root.

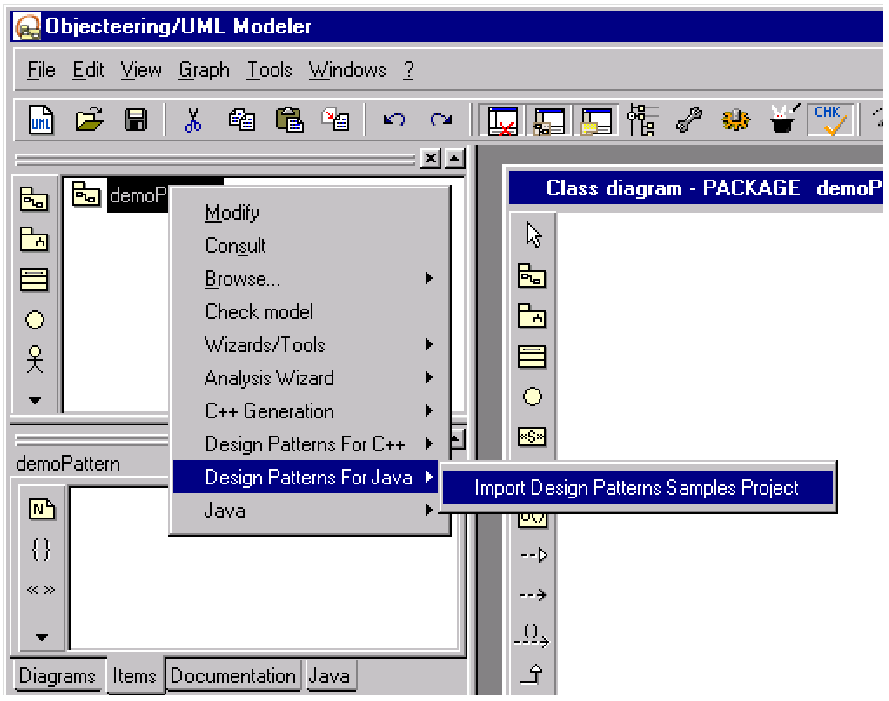


Figure 1-1. The "*Import Design Patterns Samples Project*" command

For each of the patterns, a different package is created. At the outset, each of the packages contains the "*before generation*" structure of the pattern. You may then apply the pattern, by following the documentation, in order to see what has been generated.

Important!: All the vocabulary and the examples provided conform to the corresponding vocabulary and the example (or to one of the examples) described in the work from which the pattern has been extracted. To understand the vocabulary used, and to have further details on these examples, please refer to these specific works.

Command menu labels are prefixed with the abbreviation of the name of the work from which the pattern has been extracted. The abbreviations used are:

- *GOF*: acronym for the *Gang Of Four*, a nickname given to the group of four people who wrote the first "*Design Patterns*" work and from whom the vast majority of the patterns in this module are taken.
- *POSA*: acronym for *Pattern Oriented Software Architecture* from which the "*Counted Pointer*" idiom is taken.

General principle

The examples project contains a package of examples per design pattern which can be used for the First Steps. You should simply follow the instructions given for each design pattern.

Activation of a design pattern consists of designating classes or packages to which the pattern is to be applied, and of choosing a menu which corresponds to the desired design pattern. The model will then be automatically transformed according to the design pattern used.

For the "*Proxy*" and "*Adapter*" design patterns, the model has to be modified after activation of the pattern, so that the example produces an executable. On the package representing the design pattern, a "*comment*" note contains the modifications to be made to the model.

Generation and compilation

Each package runs an executable.

In C++, it is necessary to:

- ♦ define the name of the type interpretation package in C++ module configuration, with "*ObjecteeringTypes*" as its value
- ♦ compile the package with the "*libO*" library, found in the <OBJING_PATH/gencxx/lib directory, and the includes found in the <OBJING_PATH/gencxx/include directory

In C++, launch the following command in an MS-DOS console:

```
<PackageName>
```

In Java, launch the following command:

```
java -classpath <generation directory >;<jdk directory
>\lib\classes.zip <Name of Package>.<PrincipalClassName>
```

Patterns available

The ... pattern	is used to ...
Singleton	manage a unique instance and to make it transparent.
State	adapt the behavior of an object according to its internal state.
Prototype	create an instance, by cloning a "*prototype*" instance.
Memento	externalize the values of an object. This pattern is frequently used for the "*undo/redo*" services.
Visitor	apply a group of operations to a hierarchy of classes
Proxy	define access by proxy to another object.
Adapter (two generation modes)	adapt a class' interface to that of a target class.
Counted Pointer	manage referenced objects (specific to C++).

Chapter 2: Using the Objecteering/Design Patterns modules

Presentation

In order to use the *Objecteering/Design Patterns for C++* and *Objecteering/Design Patterns for Java* modules, the following steps must be carried out:

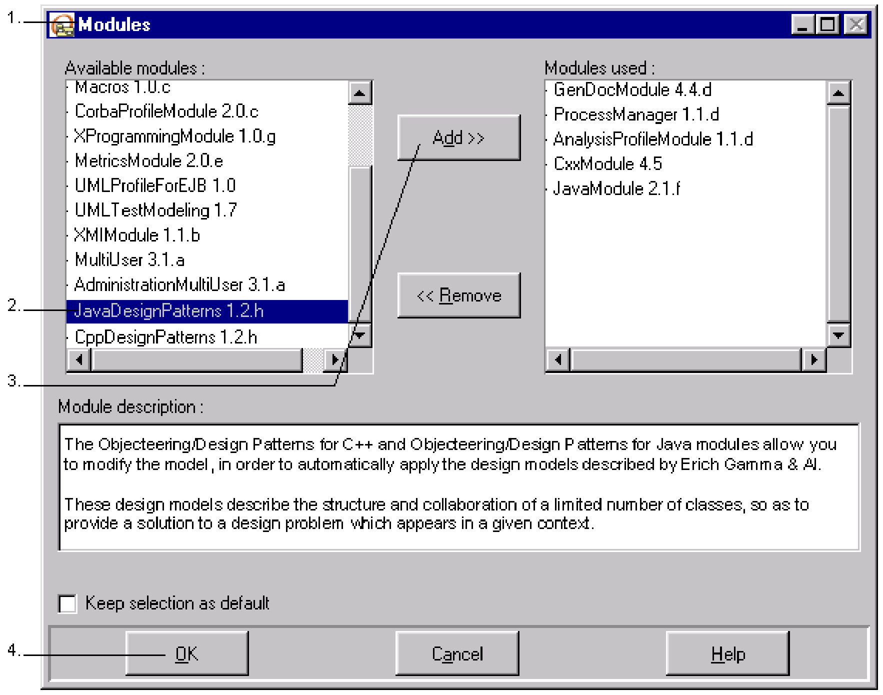
- ♦ the selection of the module at user UML modeling project level
- ♦ the entry of information during configuration of the module

For further details on the selection of modules, please refer to the "*Selecting modules in a UML modeling project*" section of the current chapter of this user guide.

Selecting modules in a UML modeling project

To use one of the two *Design Patterns* modules (or the two modules), *Objecteering/Design Patterns for C++* or *Objecteering/Design Patterns for Java*, they must be selected in your UML modeling project, in order to have the pattern generation menus and commands at your disposal.

This selection is made by transferring the Design Patterns module(s) chosen from the left-hand list to the right hand list of the "*Modules*" dialog box (as shown in Figure 2-1):



The Objecteering/Design Patterns for C++ and Objecteering/Design Patterns for Java modules allow you to modify the model, in order to automatically apply the design models described by Erich Gamma & Al.

These design models describe the structure and collaboration of a limited number of classes, so as to provide a solution to a design problem which appears in a given context.

Figure 2-1. Selection of the *Design Patterns for Java* module

Steps:

1 - Click on the "*UML modeling project modules*" icon or the "*Tools/Modules...*" menu to open the "*Modules*" window.

2 - Click on the "*JavaDesignPatterns*" or "*CppDesignPatterns*" module in the left-hand list.

3 - Click on the "*Add*" button. The selected module is then transferred into the right-hand "*Modules used*" list.

4 - Click on "*OK*" to confirm.

Configuring the module

Presentation

The configuration of the chosen module is an essential operation, which must be carried out for all UML modeling projects.

The "Edit configuration" dialog box

The "*Edit configuration*" dialog box is opened by clicking on the 🔧 ("*Modify module parameter configuration*") icon, or through the "*Tools/modify configuration...*" menu. The two modules (*Objecteering/Design Patterns for C++* and *Objecteering/Design Patterns for Java*) are configured separately.

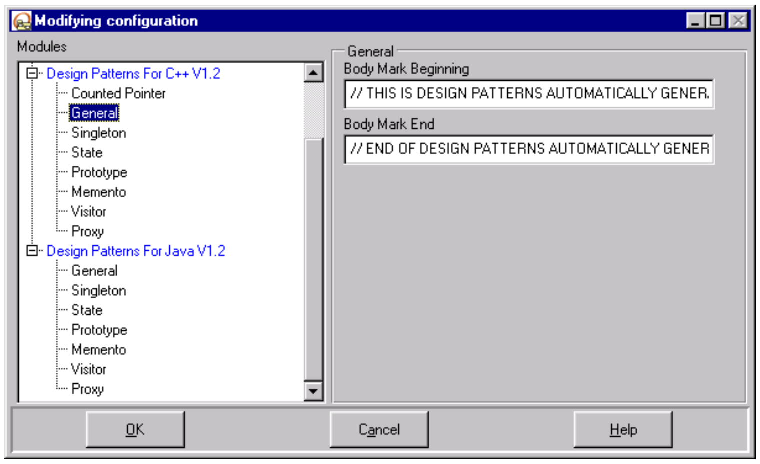
To proceed with module configuration, select the name of the module (either *Design Patterns for C++* or *Design Patterns for Java*) in the configuration module tree. You will see that each module has a set of sub-options:

- A "*General*" sub-option
- Seven sub-options (six for Java) which correspond to each of the patterns available in the module. Their related parameters will be explained in the different sections dealing with each of the patterns in detail. These seven sub-options are as follows:
  - the *Singleton* pattern
  - the *State* pattern
  - the *Prototype* pattern
  - the *Memento* pattern
  - the *Visitor* pattern
  - the *Proxy* pattern
  - the *Counted Pointer* pattern (C++ only)

Note: There is no specific configuration for the *Adapter* pattern.

The "*General*" sub-option contains general module parameters, used to determine:

- the code generation module name, used in conjunction with the Design Patterns
- the body mark beginning and the body mark end used in code generation



General

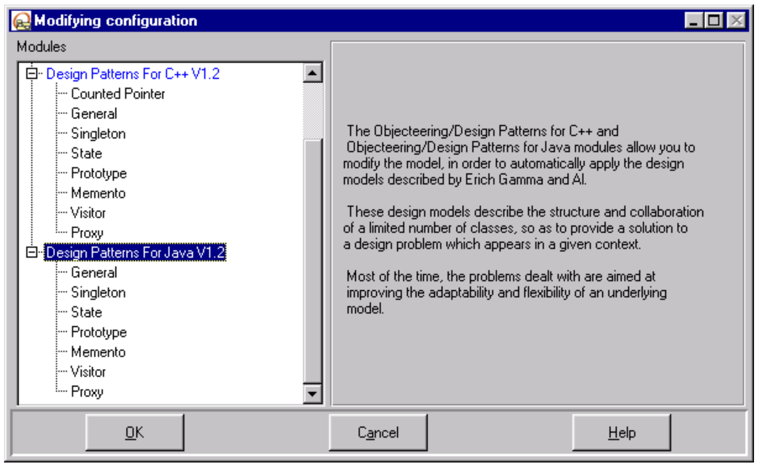
Body Mark Beginning

// THIS IS DESIGN PATTERNS AUTOMATICALLY GENER.

Body Mark End

// END OF DESIGN PATTERNS AUTOMATICALLY GENER

Figure 2-2. Configuring the *Design Patterns for Java* module



The Objecteering/Design Patterns for C++ and Objecteering/Design Patterns for Java modules allow you to modify the model, in order to automatically apply the design models described by Erich Gamma and Al.

These design models describe the structure and collaboration of a limited number of classes, so as to provide a solution to a design problem which appears in a given context.

Most of the time, the problems dealt with are aimed at improving the adaptability and flexibility of an underlying model.

Figure 2-3. Configuring the *Design Patterns for Java* module

The "General" sub-option: Body mark beginning parameter

This parameter corresponds to the flag inserted at the beginning of text generated by the *Objecteering/Design Patterns* modules. This flag is inserted at the beginning of typed text, which is associated with different module elements. The presence of this flag, as well as that of the body mark end flag, allows you to mix code generated by the module and code written by yourself within the same text zone, without running the risk of seeing the code written by yourself cut during subsequent pattern re-application.

This flag must correspond to a commentary and must be sufficiently "recognizable" (in other words, it must not clash with other commentaries).

The "General" sub-option: Body mark end parameter

This parameter corresponds to the flag inserted at the end of generated code, in order to implement a design pattern. It represents the *Design Patterns* body mark end flag, and is the opposite of the parameter described in the previous paragraph. You may add your own code after it.

This flag must correspond to a commentary and must be sufficiently "recognizable" (in other words, it must not clash with other commentaries, in particular with the body mark beginning flags).

Chapter 3: Details of different patterns

The Singleton pattern

Presentation

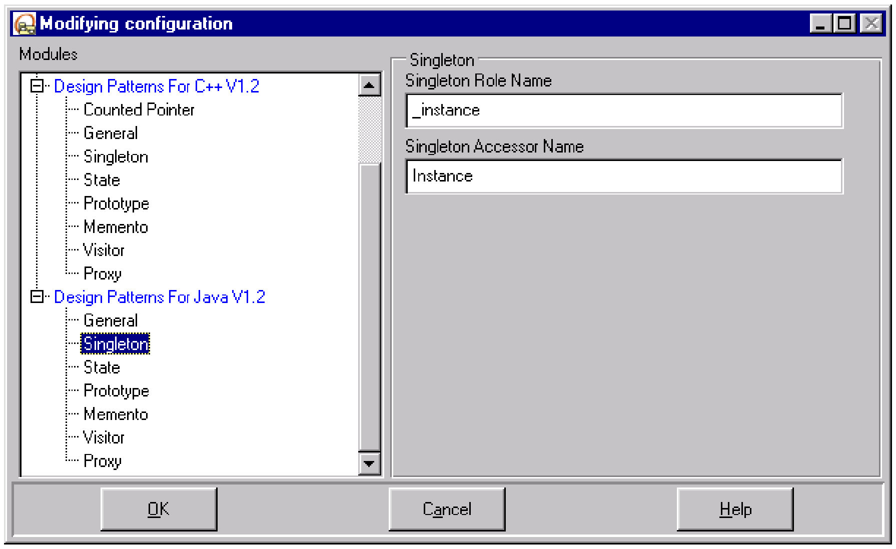
The *Singleton* pattern guarantees the uniqueness of an instance of a given class, through a single access point. The creation and initialization of this instance occurs late, during the initial accessing of this instance.

Motivations for choice

♦ You wish to create a unique instance class, to which access is both simple and secure. You require, amongst other things, a level of security superior to that of the simple declaration of a local variable.

♦ You wish to ask for details of the creation of an instance. The instance must be created as late as possible, so as to economize on cost, if the singleton is never referred to. This instantiation must be carried out in a transparent fashion.

Singleton configuration parameters



Singleton

Singleton Role Name: _instance

Singleton Accessor Name: Instance

Figure 3-1. Configuration parameters for the *Singleton* pattern

- "*Singleton role name*": The unique instance of the Singleton is accessible via a private class relation of 0..1 multiplicity which is generated. Via this parameter, the name of the relation's role, which allows access to this unique instance, is defined. Modify this parameter if, for example, you have to make your code conform to certain naming rules. The configuration of this role name may only be accessed through parameterization. No entry dialog box allowing this name to be modified is proposed at the moment of generation.
- "*Singleton accessor name*": This parameter allows you to define the name which will be given to a generated class operation, which gives access to the unique instance. Modify this parameter if you have to make the naming respect certain rules. As for the previous parameter, the configuration of this name may only be carried out through parameterization.

Singleton structure before pattern application

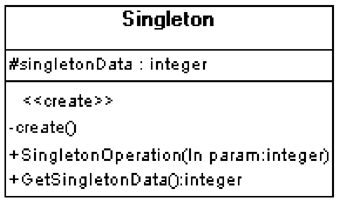

Singleton
#singletonData : integer
<<create>>
-create()
+SingletonOperation(In param:integer)
+GetSingletonData():integer

Figure 3-2. The "*Singleton*" class before pattern application

- One single class (the class on which generation will take place) is necessary.
- This class must not be abstract.
- This class can have sub-classes, but generation will in no way take these into account.
- This class must have a default constructor, either an implicit one (the class has no constructor) or an explicit one (the class has one or several constructors, one of which must be the default constructor). The initial visibility of this constructor is not important; it will be modified later, in order to become "private".
- This class can have components, but must NOT be the component of another class.
- This class must not already have a operation of the same signature as the instance access operation (for example, *Instance()*). If the opposite is true, this operation must correspond to the criteria expected by the pattern: Class operation which returns (after instantiation, if necessary) the unique instance.
- The class must not already have an association with a role name which corresponds to the role name parameterized for the reflexive association of the Singleton. If the opposite is true, this association must correspond to expected criteria: reflexive class association of 0..1 multiplicity, with private visibility.

Singleton pattern operating mode

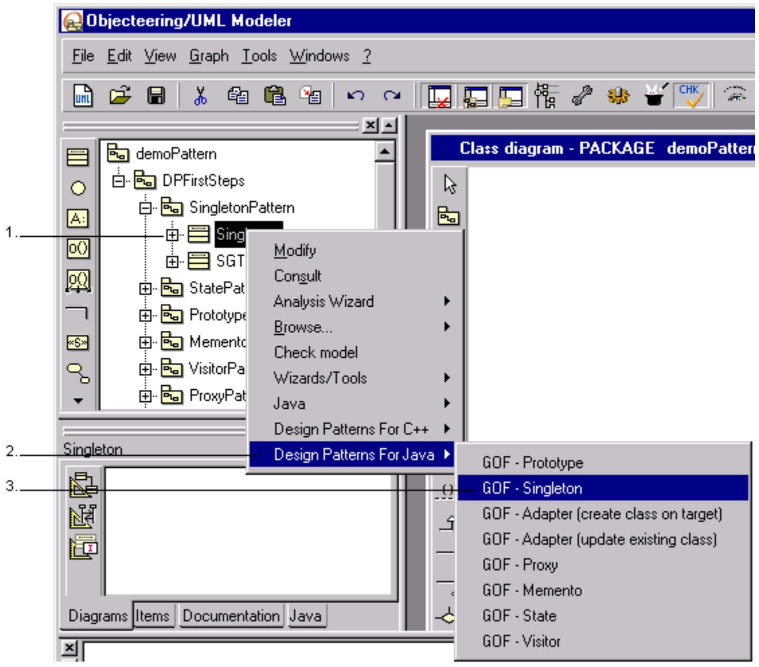


Figure 3-3. Selection of the *GOF - Singleton* pattern for the "*Singleton*" class

Steps:

1 - In the explorer, select the class which is to become a *Singleton* and right-click to open the context menu.

2 - Select the "*Design Patterns for C++*" or "*Design Patterns for Java*" menu item.

3 - Select the "*GOF - Singleton*" pattern.

Note: A message in the console informs you that processing is complete. The class is then modified, as shown below (Figure 3-4).

Singleton structure after pattern application

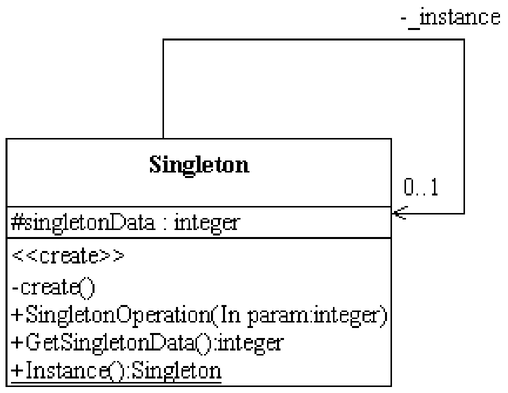


Figure 3-4. Modification of the "*Singleton*" class

- All the class' constructors have now changed to private visibility.
- A private class relation 0..1 from the class towards itself is generated. The role name used to access this relation corresponds to the name specified during module parameterization.
- A public class operation is generated to allow access to the Singleton's unique instance. The name of this access operation is fixed during parameterization. The code of this operation is also generated, and refers to the class' default constructor.

Consequences - Advantages

Access to the Singleton is precisely controlled within the instance access operation. It is, therefore, easy to allow exclusive access to this unique instance between several threads of the same application by applying, for example, the *Double Check Locking Pattern*. The necessary code can be added before and after the code marks generated by the *Objecteering/Design Patterns* module, without it being necessary to modify the generated code.

A supplementary degree of upgradeability is given to the application. If the original Singleton is derived, it is easy to modify the access operation in order to instantiate and send back an instance of this sub-class. This may be carried out in a transparent manner towards the Singleton's clients. However, this necessitates slight modifications to the code generated by the *Objecteering/Design Patterns* module, in order to refer to the instantiation of the sub-class. If the Singleton is a manager, and this manager then evolves into a distributed manager, the access operation can be modified, in order to send back a proxy on the distributed object. Once again, this is carried out in a transparent manner for the client.

The best naming space management is provided. The global naming space is not cluttered with global variables.

Consequences - Drawbacks

Arguments cannot be passed to the Singleton's instantiation. This is a direct consequence of the fact that the Singleton is instantiated in a transparent manner.

There is no explicit destruction of the Singleton. It is simply destroyed at the end of execution.

The State pattern

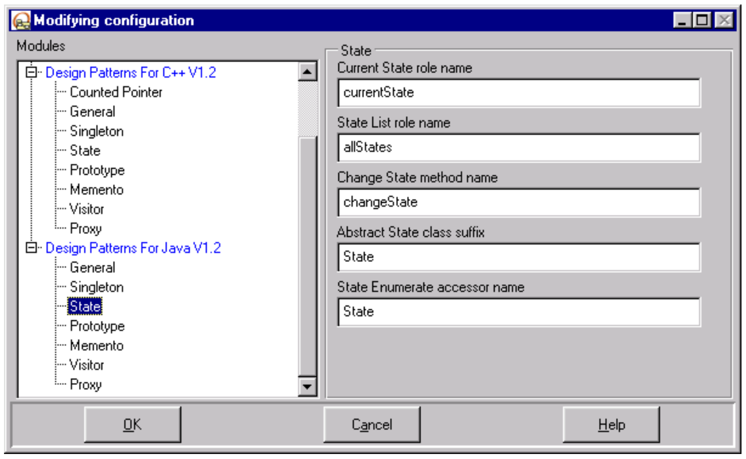
Presentation

The *State* pattern allows an object to adapt itself to the behavior of certain of its operations, according to its internal state. These states, as well as the transitions between these states, are determined by the corresponding control state chart.

Motivations for choice

- You wish to modify the behavior of all or some of the services proposed by an object during execution. The behavior chosen for this or these service(s) depends directly on the control state chart.
- You wish to integrate an operation control based on the verification of the control state chart, which allows you to ensure the applicability of operations, according to the current state.

State configuration parameters



State

Current State role name: currentState

State List role name: allStates

Change State method name: changeState

Abstract State class suffix: State

State Enumerate accessor name: State

Figure 3-5. Configuration parameters for the *State* pattern

- "*Current state role name*": Role name allocated to the 1..1 association, which points to the object and which represents the current state of the object.
- "*State list role name*": Role name of the 1..n relation which records all the state objects representing all the potential states of the class, as they are stipulated by the control state chart.
- "*Change state operation name*": Operation name which allows you to change the current state of the object. This operation, generated by the *Objecteering/Design Patterns* module, is referred to by operations which cause transitions between states. As a parameter, it takes a whole, which must be a value of the enumerate representing all the possible states of a class.
- "*Abstract state class suffix*": Suffix which determines the name of the base class from the generated state classes (see the "*Structure after pattern application*" paragraph below). This base class takes the name of the context class + suffix. Classes which represent concrete states take the name of the context class + state name (as it is specified in the control state chart).
- "*State enumerate accessor name*": Accessor which allows you to find out the current state.

State structure before pattern application

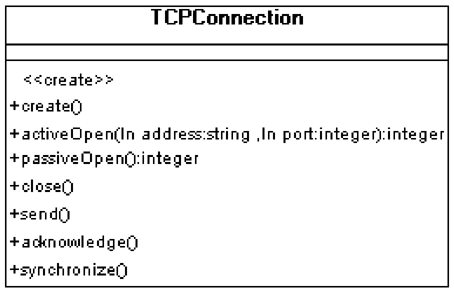


Figure 3-6. The "*TCPConnection*" class before pattern application

- One single class (the class on which generation is to take place) is necessary. This class cannot be named "*State*", because if it is named "*State*", the code generated will not compile.
- One single control state chart must be defined on the class.

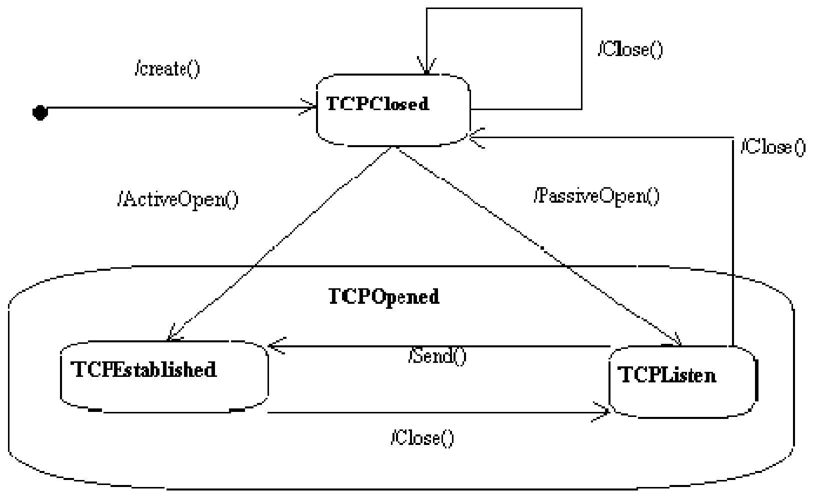


Figure 3-7. State chart of the "*TCPConnection*" class

- The operations whose implementation or applicability depends on the current state must figure as control state chart transitions.
- The state chart must contain one single transition resulting from the initial state (non-named). This transition must be carried out by the "*create*" operation.
- The state chart must not be redefined in a sub-class.
- Transitions which originate from or arrive in an abstract state are not authorized.

For a given origin state, an operation can only make the state chart transit to one single state. In other words, the state chart below is forbidden, since the "*Close ()*" operation leaves the "*TCPEstablished*" state twice, and heads towards two different states.

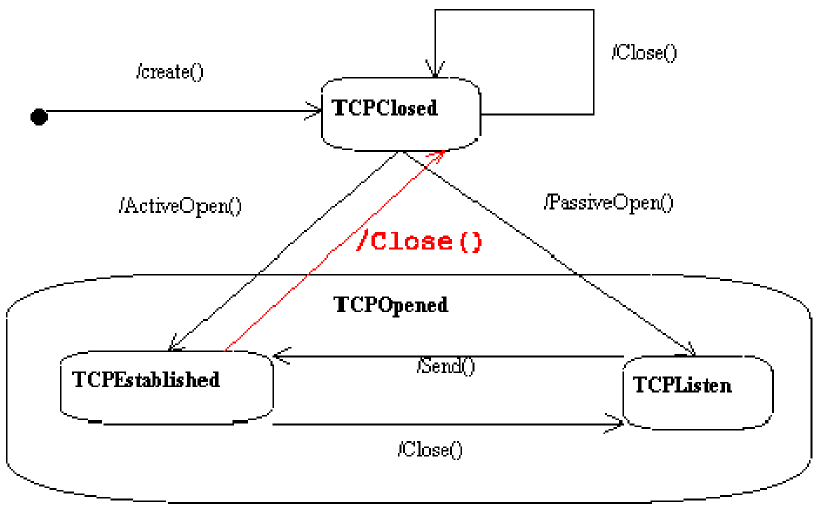


Figure 3-8. Incorrect state chart

State pattern operating mode

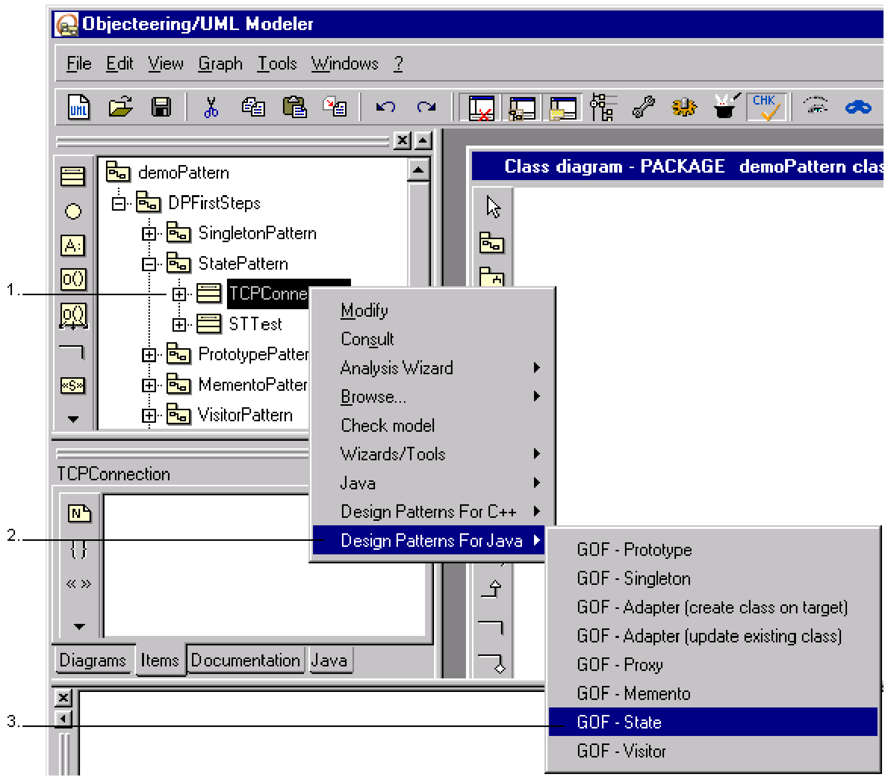


Figure 3-9. Application of the *State* pattern to the "*TCPConnection*" class

Steps:

1 - In the explorer, select the class on which the *State* pattern is to be applied and right-click to open the context menu.

2 - Select the "*Design Patterns for C++*" or "*Design Patterns for Java*" menu item.

3 - Select the "*GOF - State*" pattern.

State structure after pattern application

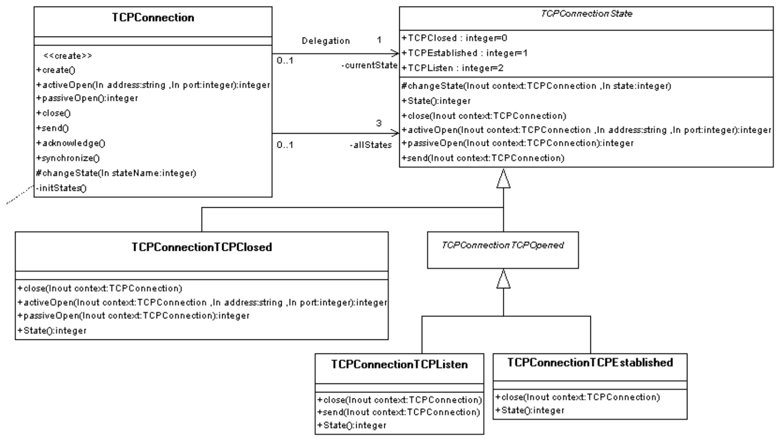

TCPConnection
<<create>>
+create()
+activeOpen(In address:string ,In port:integer):integer
+passiveOpen():integer
+close()
+send()
+acknowledge()
+synchronize()
#changeState(In stateName:integer)
-initStates()
Delegation
1
0..1
-currentState
3
0..1
-allStates
TCPConnection State
+TCPClosed : integer=0
+TCPEstablished : integer=1
+TCPListen : integer=2
#changeState(Inout context:TCPConnection ,In state:integer)
+State():integer
+close(Inout context:TCPConnection)
+activeOpen(Inout context:TCPConnection ,In address:string ,In port:integer):integer
+passiveOpen(Inout context:TCPConnection):integer
+send(Inout context:TCPConnection)
TCPConnectionTCPClosed
+close(Inout context:TCPConnection)
+activeOpen(Inout context:TCPConnection ,In address:string ,In port:integer):integer
+passiveOpen(Inout context:TCPConnection):integer
+State():integer
TCPConnection TCPOpened
TCPConnectionTCPListen
+close(Inout context:TCPConnection)
+send(Inout context:TCPConnection)
+State():integer
TCPConnectionTCPEstablished
+close(Inout context:TCPConnection)
+State():integer

Figure 3-10. Modification of the "*TCPConnection*" class

The Context class (here "*TCPConnection*") on which the pattern has been applied is transformed as follows:

- ♦ Addition of a private visibility 1..n association, in which the instances of all the possible states of the class will be recorded. The role name of this association can be parameterized.
- ♦ Addition of a private visibility 1..1 association, which references a state object representing the current state of the Context object. The role name of this association can be parameterized at module level.
- ♦ Addition of an "*initStates*" operation, which instantiates all the possible states of the Context object, and which records these state objects in the association which lists all the states.
- ♦ Addition of a state change operation, whose name is configured at module level.
- ♦ For operations involved in transitions between states, the C++ or Java code which redirects this call towards the corresponding operation of the "*State*" class is generated. If code already exists in these operations, it is copied into the concrete state classes.

An "*abstract state*" class is created. This class bears the name of the Context class, to which a suffix is added. This name can be configured at module level. If, before pattern application, there already exists an association bearing the name defined by the "*Current State Role Name*" parameter, the class linked by this association serves as base class for the concrete states.

♦ This class defines the operations which correspond to (and which have the same name as) the Context class' operations implicated in the transitions between states. By default, all these operations provoke a constantly false pre-condition lifting, since they should be abstract.

Note: These operations are not generated as abstract, since they are not all redefined in all the sub-classes.

♦ This class defines whole type class constants. These constants bear the same names as the states defined in the control state chart, and are attributed values ranging from 0 to n-1 (number of state charts less one).

♦ It defines a state change operation, whose name can be configured at module level, and whose role is to refer, in turn, to the state change on the Context class (the corresponding code is generated).

If the abstract states have been defined, they figure in the form of abstract intermediary state classes.

Concrete state classes, which are derived from the abstract state class, are created. These classes bear the name of the Context class suffixed with the state names described in the control state chart of the Context class. In this way, as many concrete state classes are created, as there are states in the control state chart.

♦ Each concrete state class redefines the operations for which it is the starting state of the transition.

♦ These operations' code is initialized with the correct state change. The user must add the applicable code before this code.

Consequences - Advantages

Transitions between states become explicit.

Control structures (tests or switch-cases), destined to determine which processing must be applied according to the internal state of the object, are removed from the body of operations. These control structures, which depend on the current state of the object, are often repeated identically on several operations. The application of the *State* pattern reduces the complexity of these operations.

Operational control is managed and consistency with the control state chart maintained. It follows the modifications made to this control state chart during re-application of the pattern. Invariants specific to each state can be specified after pattern application.

The classes which correspond to destroyed or renamed states and the operations which correspond to destroyed transitions are not deleted, in order to avoid the untimely loss of code. The destruction of these classes and/or these operations is left up to the user.

In the same way, if the interface (name or parameters) of operations which correspond to transitions is modified after pattern application, then the operations previously created are not destroyed. To reapply this pattern after one of these modifications, you should:

- ♦ copy the operations' code from the abstract class into the Context class
- ♦ destroy all the operations generated in the abstract class and all the concrete classes

Consequences - Drawbacks

The number of classes and the structural complexity increases.

Efficiency is slightly reduced, following the introduction of a supplementary indirection level.

The Prototype pattern

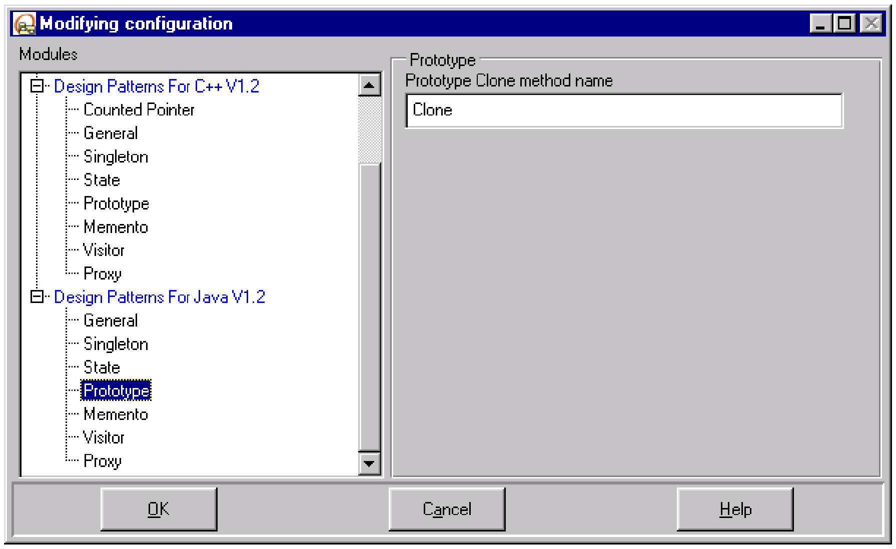
Presentation

The *Prototype* pattern creates new instances, by cloning an existing instance called prototype. The object obtained is initialized in exactly the same way as the current state of the prototype. The *Prototype* pattern is applied to a class hierarchy, and in this way, the duplication of any sub-classes may be referred to, simply by manipulating them at their most abstract level, without having to worry about their real type.

Motivations for choice

- ♦ You wish to be able to vary the products to be instantiated and/or the initial values of these products at execution. This is made possible by adding, withdrawing and modifying prototypes, where necessary.
- ♦ The quantity of different products which can be instantiated, and their possible initial states, is very large. You wish to avoid reproducing an instance fabrication structure which is parallel to the hierarchy of the products, and which mimics it, in order to preempt future maintenance problems.

Prototype configuration parameters



Prototype Clone method name: Clone

Figure 3-11. Configuration parameters for the *Prototype* pattern

- "*Prototype clone operation name*": This parameter allows you to attribute a particular name to the operation which will duplicate the object. The choice of name for this operation can only be made via this parameterization, and may not be modified during pattern application.

Prototype structure before pattern application

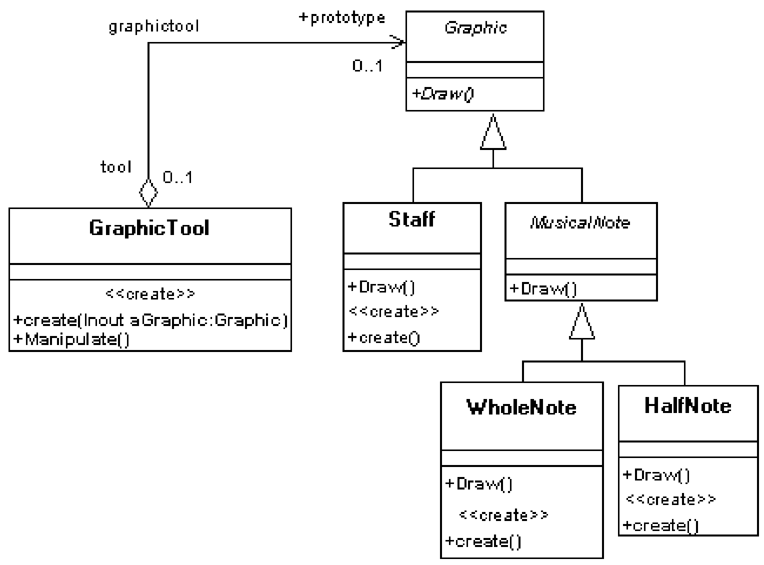


Figure 3-12. Classes before pattern application

- A hierarchy of classes is available.
- In C++, you must check that the copied semantics of these classes is correct.
- In Java, adequate copy constructors have to be created.

Prototype pattern operating mode

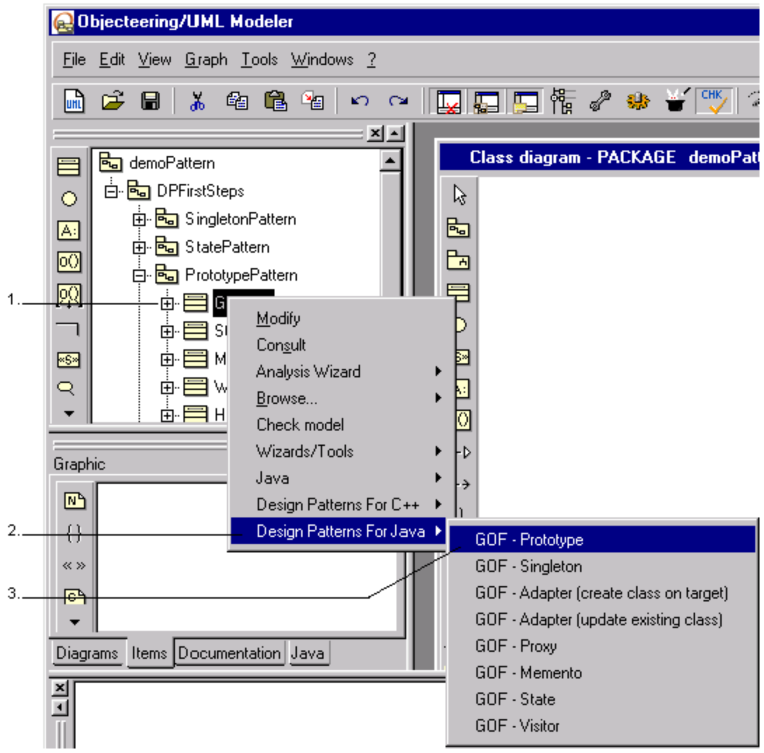


Figure 3-13. Application of the *Prototype* pattern to a root class

Steps:

1 - In the explorer, select the class on which the *Prototype* pattern is to be applied and right-click to open the context menu.

2 - Select the "*Design Patterns for C++*" or "*Design Patterns for Java*" menu item.

3 - Select the "*GOF - Prototype*" pattern.

Prototype structure after pattern application

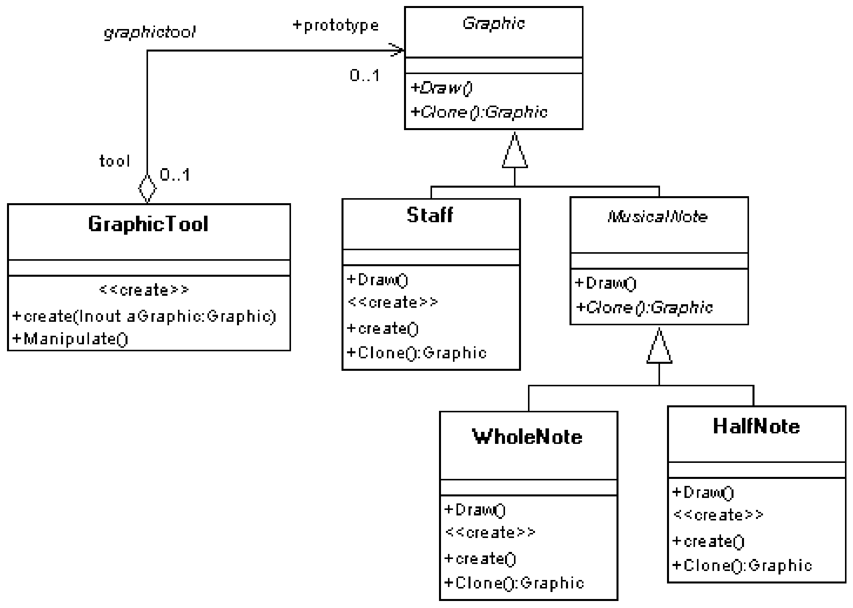


Figure 3-14. Modification of classes

An object duplication operation, bearing the name specified during module configuration, is created in the class to which the pattern has been applied. This operation is redefined on all the sub-classes of the class to which the pattern has been applied.

- ♦ On abstract classes, the duplication operation is defined as abstract.
- ♦ On non-abstract classes, the code associated with this duplication operation refers to the constructor by copy of the class, in C++. In Java, the copy constructor not being taken into account by the compiler, the implementation of the duplication operation is left to the developer’s discretion. It must be written for all classes.

Consequences - Advantages

It is now possible to vary the products created at execution, simply by recording or detaching the adequate prototypes. In the same way, the modification of these products' initial values is carried out by modifying the values which correspond to the prototypes.

For products created which are derived from an abstract Prototype class, it is possible to subsequently expand the range of these products, by creating new classes, which are derived from the abstract Prototype class, without client modification.

It is becoming possible to define new objects, by modifying the values of an object, instead of defining new classes. In as much as the Prototype pattern allows the initialization of newly created objects with values which can be dynamically configured, it is possible, in certain cases, to replace the definition of new sub-classes by the definition of new state values in a single class. Thus, the proliferation of classes is reduced, where this aspect appears crucial. In the same way, instead of new values, we can define a prototype as being made up of sub-parts, i.e., as being an aggregate. The operation which ensures the cloning must, in this case, simply realize a "deep copy", that is to say, also guarantee the copying of all the sub-parts.

Consequences - Drawbacks

The constructor by copy must be implemented for each sub-class of the Prototype, implementation which can be non trivial.

In as much as the products to be created are dynamically determined, product creation dependencies do not structurally appear.

In C++, it is possible to annotate a generalization as being private or protected. These two annotations transform the generalization into an implementation generalization. Sub-classes defined in this way are not polymorphs of the base class. The application of the *Prototype* pattern does not take into account these annotations, and produces a code which is accepted by the compiler. In any case, the behavior of such a structure will not conform to the *Prototype* pattern.

The Memento pattern

Presentation

The *Memento* pattern externalizes the values of an object, called Originator, in a Memento object. This Memento object is created by the Originator, and initialized with the current values of the former. When you wish to restore the Originator object to its original state, you need simply re-read the values retained by the Memento. The Memento is a non-dynamic object, whose only objective is to store data.

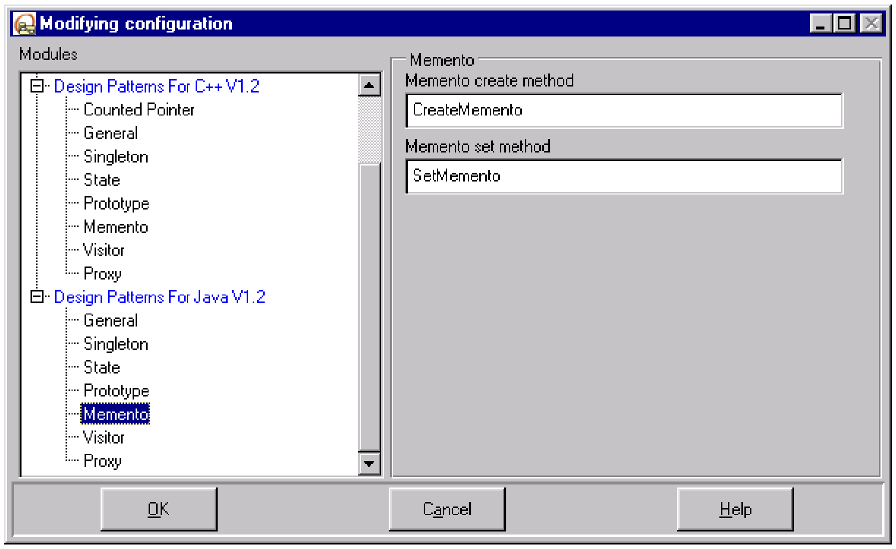
Motivations for choice

- The *Memento* pattern allows the externalization of an object's state, without violating its encapsulation.
- The *Memento* pattern allows you to keep values associated with an object before modification in the memory, in order to be able to subsequently restore the object to its original state. This can be useful when you have to provide such functions as transactions, or an "*undo/redo*" mechanism.

Read-write accessors

In order for the Memento pattern to be applied, accessors must be in read-write mode, which means that the "*ReadWrite*" radio button in the C++ and/or Java tabs of the properties editor for the attribute must be checked.

Memento configuration parameters



Memento

Memento create method: CreateMemento

Memento set method: SetMemento

Figure 3-15. Configuration parameters for the *Memento* pattern

- "*Memento create operation*": Operation name to be generated on the Originator, in order to ensure the instantiation of a Memento.
- "*Memento set operation*": Operation name to be generated on the Originator, in order to read the Memento.

Memento structure before pattern application

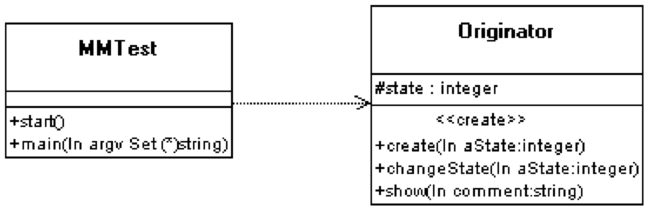

MMTest
+start()
+main(In argv Set (*)string)
Originator
#state : integer
<<create>>
+create(In aState:integer)
+changeState(In aState:integer)
+show(In comment:string)

Figure 3-16. The "*Originator*" class before pattern application

An "*Originator*" class to which the *Memento* pattern will be applied.

- This class has one or several attributes, whose values you wish to memorize for subsequent restorations.

Memento pattern operating mode

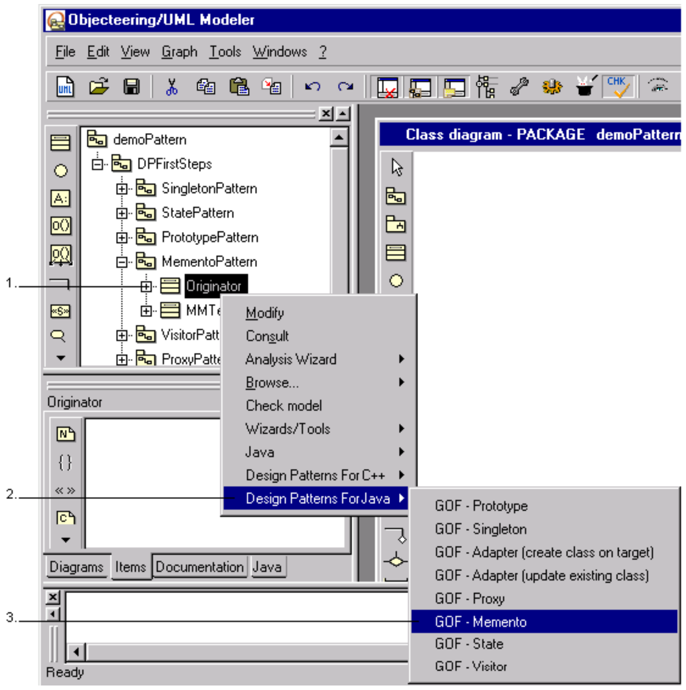


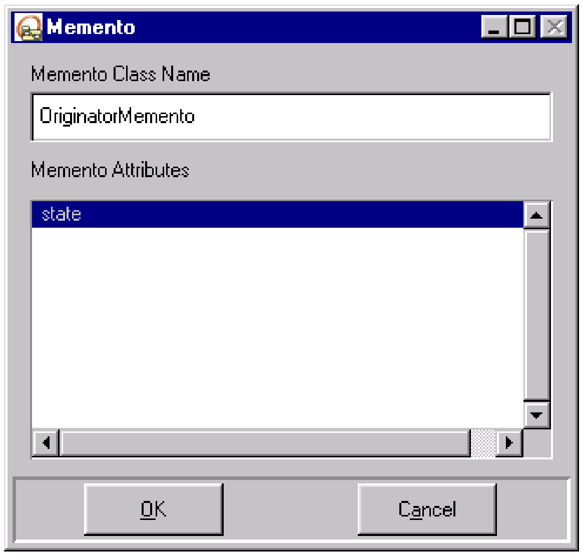
Figure 3-17. Application of the *Memento* pattern to a class

Steps:

1 - In the explorer, select the class which will play the role of Originator, and right-click to open the context menu.

2 - Select the "*Design Patterns for C++*" or "*Design Patterns for Java*" menu item.

3 - Select the "*GOF - Memento*" pattern. A dialog box (as shown in Figure 3-18) then appears.



Memento Class Name

OriginatorMemento

Memento Attributes

state

Figure 3-18. First dialog box for the application of the *Memento* pattern

- The first entry field, "*Memento class name*", allows you to choose a name for the Memento class.
- Its default value is made up of the Originator class name suffixed with the Memento. This class will generally be created by the Design Patterns generator. In this case, its name must not clash with an already existing class name.
- If you wish to assign the Memento role to an existing class, be sure that the class is accessible from the Originator class (same package or package accessible from the Originator package), then enter the name of this class in the "*Memento class name*" field.
- The "*Memento attributes*" list presents the list of all public attributes, both protected and private, of all base classes. In this list, you may make multiple selections of attributes which you wish to see figure in the Memento. If you wish to go further, it is imperative that you select at least one attribute before the generation of the *Memento* pattern.
- If you click on the "*Cancel*" key, generation is interrupted.

- If you click on "*OK*", the dialog box disappears and the values entered are checked. A second dialog box then appears (as shown in Figure 3-19).

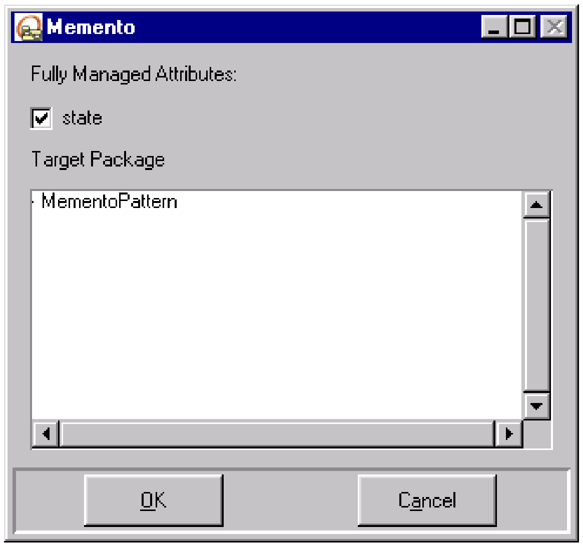


Figure 3-19. Second dialog box for the application of the *Memento* pattern

All the boxes to be checked under the "*Fully managed attributes*" label allow you to specify, attribute by attribute, for each of the attributes that you have chosen above, the elements which must be taken completely into account by the generator. When you check this option for an attribute:

- A corresponding parameter is added to the *Memento* constructor. The code which corresponds to the initialization of the attribute to which the parameter refers is generated in the constructor.
- The code which corresponds to the re-reading of the value which is taken into account is also generated in the "*SetMemento*" operation.

If the box is not checked for an attribute, it is added to the memento's attributes and its accessors are generated. However, the code corresponding to the Originator's and the Memento constructor's "*SetMemento*" operation is not generated and should be carried out by the developer. This allows you to handle cases where you wish to implement an incremental save.

- ♦ The "*Target package*" list allows you to specify a particular package to welcome the Memento class which will be created. This package is chosen from amongst the packages accessible from the originator package, which is also included. If you have specified an existing class name, be sure to select the package where it is located. If this is not done, generation will fail.

Memento structure after pattern application

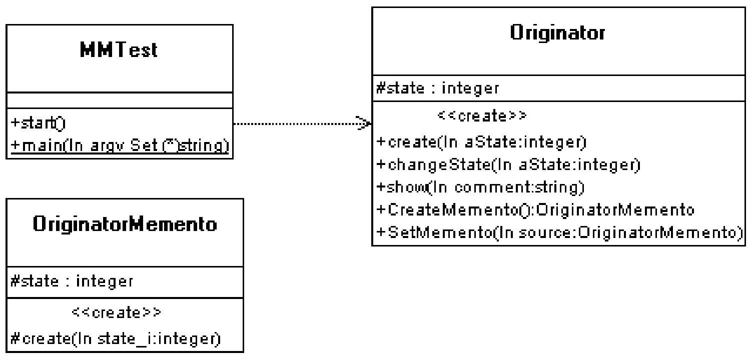

MMTest
+start()
+main(In argv Set (*)string)
Originator
#state : integer
<<create>>
+create(In aState:integer)
+changeState(In aState:integer)
+show(In comment:string)
+CreateMemento():OriginatorMemento
+SetMemento(In source:OriginatorMemento)
OriginatorMemento
#state : integer
<<create>>
#create(In state_i:integer)

Figure 3-20. Modifications after pattern application

Creation of a "*Memento*" class which ensures the memorization of attributes selected on the "*Originator*" class:

- ♦ The Memento attributes are identical to those of the Originator selected in the HMI of the pattern generation.
- ♦ The Originator attributes marked as "fully managed" also make up a part of the Memento constructor signature. This guarantees that for these attributes, the Memento cannot be created without the initialization of the above attributes.

Creation of the "*SetMemento*" operation on the Originator. This operation takes a Memento in parameter, and uses the values which are contained therein, in order to update itself. The C++ code necessary to the re-reading of attributes marked as "fully managed" is generated.

In C++, the Memento attributes change to protected and the Originator is declared the "friend" of the Memento.

Consequences - Advantages

The encapsulation of the Originator is preserved. You do not run the risk of creating dependencies built on the internal data structure of the Originator by exposing them.

A slim and intelligent interface on the Originator: only two services are necessary for externalization. Access to the Memento interface, through which data may be retrieved, is limited to the Originator.

Consequences - Drawbacks

Significant data copying costs may be entailed. In such cases, it is necessary to envisage the implementation of a variant, which will call an incremental update.

Note: In Java, the encapsulation rupture is shifted from the Originator to the Memento. The former remains vulnerable, since it exposes its accessors and mutators to public visibility.

The Visitor pattern

Presentation

The *Visitor* pattern allows you to define a group of operations, organized into a hierarchy of classes called Visitors, on a group of classes called "*Elements*", also organized into a hierarchy. These operations are applied to elements using a "double dispatching" mechanism, or a calls reflection, which allows you to precisely adapt the realization of each of the visitors' hierarchy operations to each of the elements' hierarchy classes.

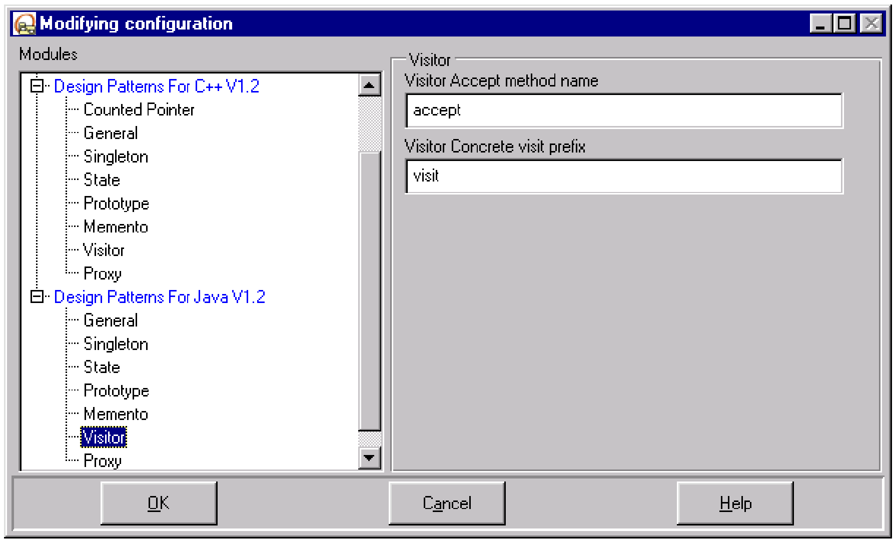
Motivations for choice

♦ You wish to apply a group of operations to a class structure. The concrete classes on which the operation is applied differ in their interface and/or by the way in which this operation must be applied.

♦ You wish to separate a group of operations, which is applied to a class structure, from the classes to which they are applied.

♦ The group of operations applied to the Elements is likely to evolve, and you would like this to happen, without questioning the "*Elements*" classes.

Visitor configuration parameters



Visitor

Visitor Accept method name: accept

Visitor Concrete visit prefix: visit

Figure 3-21. Configuration parameters for the *Visitor* pattern

- "*Visitor accept operation name*": Name of the dispatching operation which must be generated on the visited elements structure.
- "*Visitor concrete visit prefix*": Prefixed with the name of the second dispatching operations to be generated on a class hierarchy Visitor, which applies its operations to the elements structure. The definitive name of these operations is prefix + "*Element*" concrete class name, on which the visit is operated.

Visitor structure before pattern application

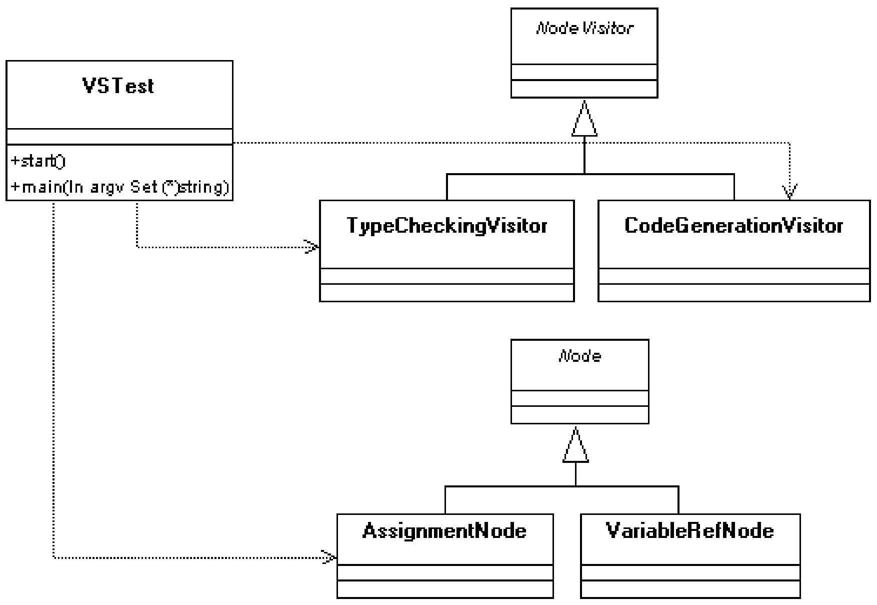


Figure 3-22. Structure before pattern application

- ♦ A hierarchy of classes called "*Elements*" (here, the hierarchy in "*Node*"), to which you wish to apply the operations.
- ♦ A hierarchy of classes called "*Visitors*" (here, the hierarchy in "*NodeVisitor*") which organizes a class hierarchy of operations which will apply to "*Elements*".
- ♦ No prior relationship between the two hierarchies is necessary.

Visitor pattern operating mode

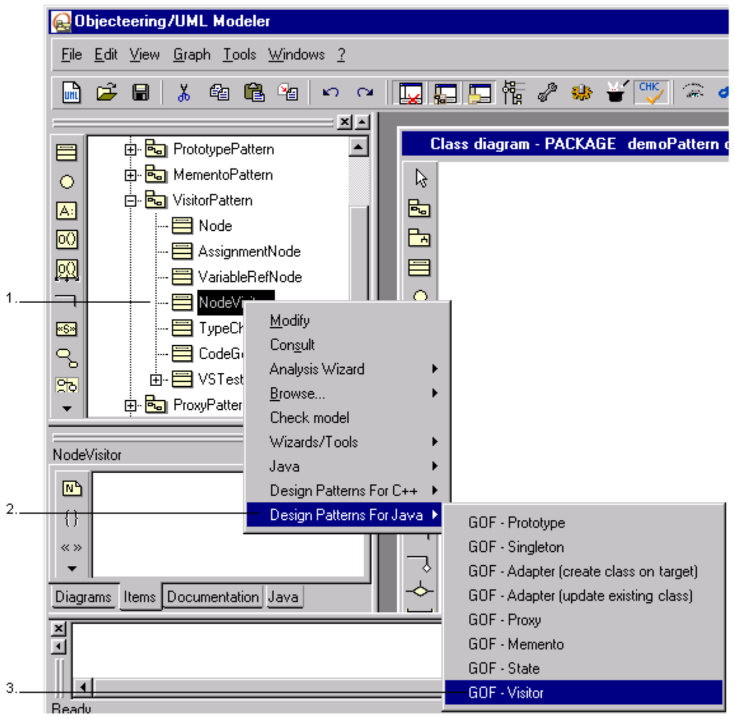


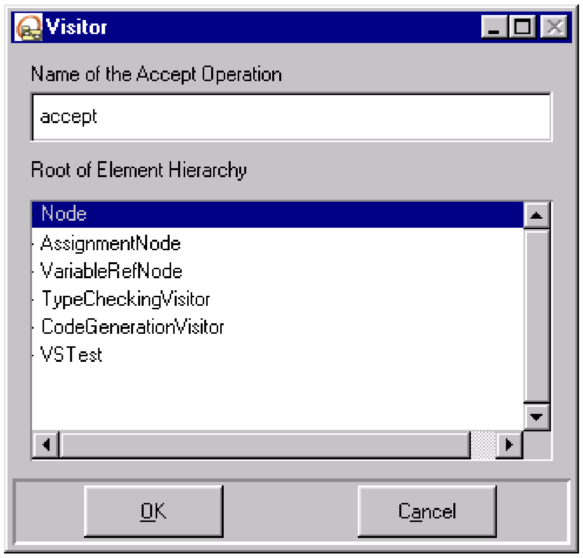
Figure 3-23. Application of the *Visitor* pattern

Steps:

1 - In the explorer, select the class to which the *Visitor* pattern is to be applied, and right-click to open the context menu.

2 - Select the "*Design Patterns for C++*" or "*Design Patterns for Java*" menu item.

3 - Select the "*GOF - Visitor*" menu item. The following dialog box (shown in Figure 3-24) then appears.


Name of the Accept Operation

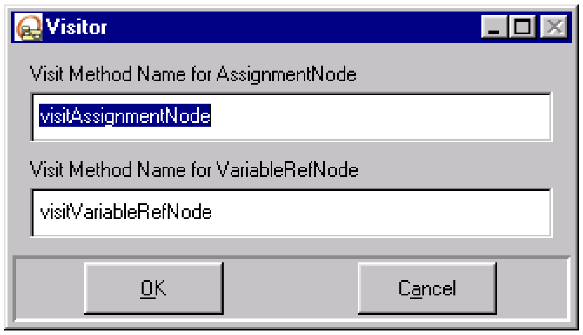
accept

Root of Element Hierarchy

Node
AssignmentNode
VariableRefNode
TypeCheckingVisitor
CodeGenerationVisitor
VSTest

Figure 3-24. The parameters of the *Visitor* pattern

- The first entry field allows you to give a name to the acceptance operation which will be added to the classes of the "*Elements*" hierarchy. By default, the name which figures in this field is the name of the default operation set during parameterization.
- "*Root of Element Hierarchy*": This list proposes all the classes accessible from the selected visitor root. You must select one class only, which will constitute the root class of the "*Elements*" hierarchy.
- Once this entry and this selection have been carried out, and once the group has been confirmed, the following dialog box will appear (as shown in Figure 3-25).



Visit Method Name for AssignmentNode: visitAssignmentNode

Visit Method Name for VariableRefNode: visitVariableRefNode

Figure 3-25. Parameters of the *Visitor* pattern

- "*Visit operation name for xxx*": For each of the classes which make up the hierarchy of the "*Elements*" classes, an entry field proposes a name for the operations which will be generated on the Visitors hierarchy. The Visitors group will support all these operations.

The code specific to the acceptance operations situated on the "*Elements*" hierarchy is entirely undertaken and generated by the *Objecteering/Design Patterns for C++* and *Objecteering/Design Patterns for Java* modules. It calls the visitor operations which are specific to them.

The code specific to visitor operations must be written by the user.

Visitor structure after pattern application

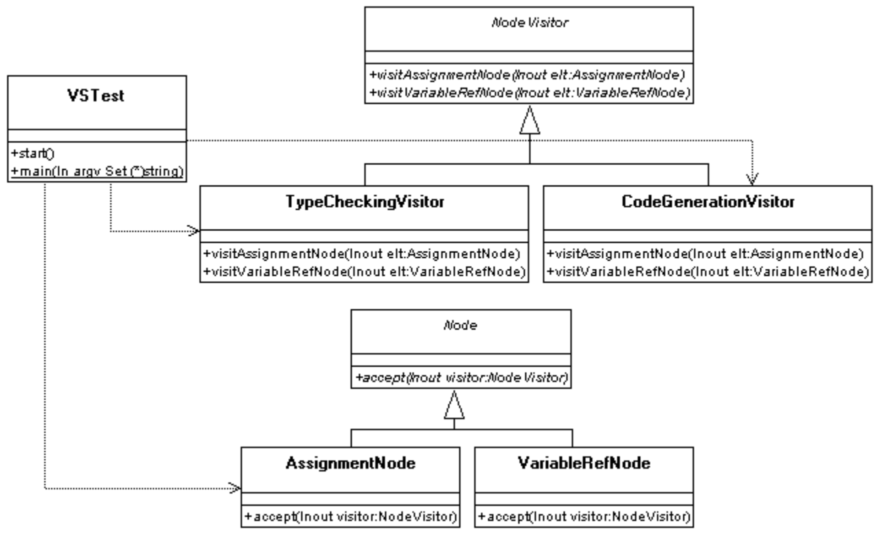

NodeVisitor
+visitAssignmentNode(Inout elt:AssignmentNode)
+visitVariableRefNode(Inout elt:VariableRefNode)
VSTest
+start()
+main(In argv Set (*)string)
TypeCheckingVisitor
+visitAssignmentNode(Inout elt:AssignmentNode)
+visitVariableRefNode(Inout elt:VariableRefNode)
CodeGenerationVisitor
+visitAssignmentNode(Inout elt:AssignmentNode)
+visitVariableRefNode(Inout elt:VariableRefNode)
Node
+accept(Inout visitor:NodeVisitor)
AssignmentNode
+accept(Inout visitor:NodeVisitor)
VariableRefNode
+accept(Inout visitor:NodeVisitor)

Figure 3-26. Structure after pattern application

- A visitor acceptance operation is generated on each of the element classes in the hierarchy.
- A visit operation specific to each of the element hierarchy classes is generated on each of the Visitors classes.
- The code, which allows each of the acceptance operations of each of the elements to call the visit operation which corresponds to this element, is generated.

Consequences - Advantages

The addition of new operations is straightforward. In as much as the elementary services necessary to the realization of operations to come already exist, new operations may be added simply by adding new Visitor classes, without modifying the "*Element*" classes. Nevertheless, if the "*Elements*" classes are numerous, each new operation will have to support just as many realization operations. It is, therefore, recommended that you reapply the pattern via the *Objecteering/Design Patterns* module even in this case, in order to systematize the implementation of these operations and to avoid realization errors.

Consequences - Drawbacks

The addition of new "*Element*" classes necessitates the modification of all the Visitor classes. This classic negative consequence of the *Visitor* design pattern is widely offset by the possibility of reapplying the pattern. In any case, do not forget that the coding of new operations generated in this way on "*Visitor*" classes is left up to you.

The Proxy pattern

Presentation

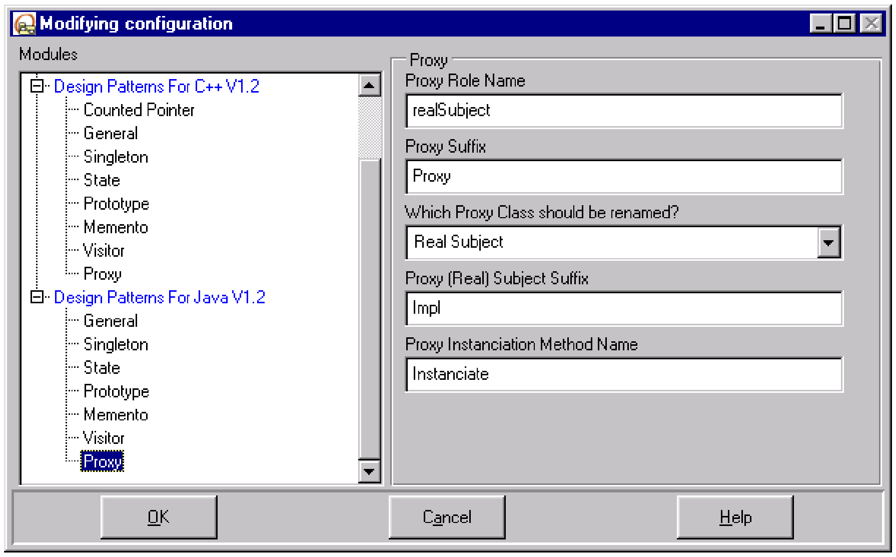
The *Proxy* Pattern defines access by proxy towards another object. A class called "*Proxy*" presents the same interface as a "*RealSubject*" class. In this way, a client object of "*RealSubject*", which wishes to refer to its services, can identically address a Proxy instance. The Proxy redirects calls destined for it towards the "*RealSubject*", but offers services, according to its nature, which may be:

- Reference counting, in order to free the "*RealSubject*" as soon as possible (smart reference/smart pointer).
- Locking, in order to ensure synchronization between several threads or several processes. In this case, proxies are situated in separate execution spaces, which implement a mechanism, which in turn allows operation calls (towards "*RealSubject*") beyond these boundaries (smart reference/lock).
- Calling "*RealSubject*" services between machines on the same network in the case of distributed objects (remote proxy).
- To allow the tardy instantiation of a "*RealSubject*", where this is greedy in terms of resources (virtual proxy).
- To carry out security checks before accessing the "*RealSubject*" (proxy protection).

Motivations for choice

The motivations which can lead to the implementation of a proxy are described in the presentation above. In general, proxies allow access to a service, by inserting an "added value" into the service, whose nature depends on the nature of the proxy: remote access, locking, security.

Proxy configuration parameters



Proxy

Proxy Role Name: realSubject

Proxy Suffix: Proxy

Which Proxy Class should be renamed?: Real Subject

Proxy (Real) Subject Suffix: Impl

Proxy Instanciation Method Name: Instanciate

Figure 3-27. Configuration parameters for the *Proxy* pattern

"*Proxy role name*": Role name of the association linking the Proxy and its "*RealSubject*".

"*Proxy suffix*": The suffix specified here, added to the name of the class to which the pattern is applied, allows you to propose a default class name for the Proxy.

"*Which proxy class should be renamed?*": The possible choices are "*Real Subject*" and "*Subject*".

- ♦ If "*Real Subject*" is chosen, it is the class on which the Pattern, which is renamed, is applied. The base class (interface) of the implementation class then takes the name which was previously attributed to it. By doing this, the previous operational references to "*RealSubject*" will become references to "*Subject*", since these are based on the name. Structural references are also modified and follow the name change. Thus, the Proxy and the level of abstraction procured by the "*Subject*" are set up at the same time. However, this often necessitates some manual adjustments in certain parts of the code.
- ♦ If "*Subject*" is chosen, it is the base class of *RealSubject* and of the Proxy, which takes a newly created name. This choice allows you to retain the previous function after application of the pattern. The use of the Proxy can be put in place later, but only manually.

"*Proxy (Real) subject suffix*": This suffix, added to the class name, allows you to propose a default name which may be used either for the *Subject* or for the *Real Subject*, according to the choice of parameter implemented for "*Which proxy class should be renamed?*".

"*Proxy instantiation operation name*": This allows you to parameterize the class operation name of "*Subject*", which will instantiate the proxy and send it back in the form of a "*Subject*", which allows you to mask the use (or possibly the non-use according to a context defined by the developer) proxies from the client.

Proxy structure before pattern application

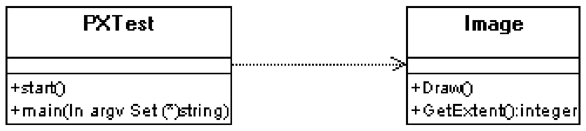


Figure 3-28. Structure before pattern application

- The proxy pattern is applied to a class which then constitutes the "*RealSubject*". No other class or association is necessary.

Proxy pattern operating mode

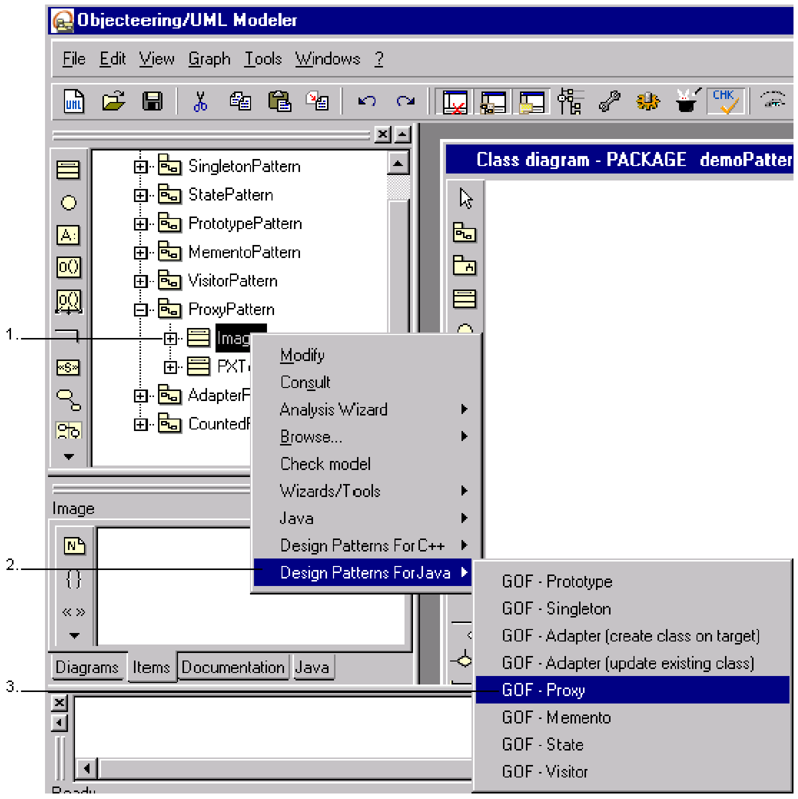


Figure 3-29. Application of the *Proxy* pattern to a class

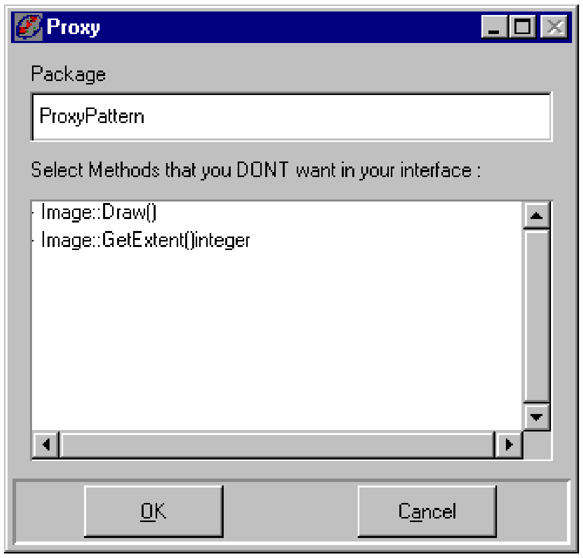
Steps:

1 - In the explorer, select the class for which you wish to generate a proxy and right-click to display the context menu.

2 - Select the "*Design Patterns for C++*" or "*Design Patterns for Java*" menu item.

3 - Select the "*GOF - Proxy*" menu item.

The following dialog box is then displayed (as shown in Figure 3-30).



ProxyPattern

Image::Draw()
Image::GetExtent()integer

Figure 3-30. Parameters of the *Proxy* pattern

- The "*Package*" entry field allows you to specify in which package "*Subject*" and "*Proxy*" are to be generated. By default, this is the package which contains "*RealSubject*".
- The "*Select operations that you DON'T want in your interface*" list allows you to specify the "*Real Subject*" operation which do not have to be redirected from the Proxy, but instead will remain local to "*Real Subject*". This list only presents the "*Real Subject*" public operations, which are the only ones concerned by a definition on the Proxy. This parameter allows the creation of proxies, which only offer their clients a part of the "*RealSubject*" interface (re-encapsulation).

Proxy structure after pattern application

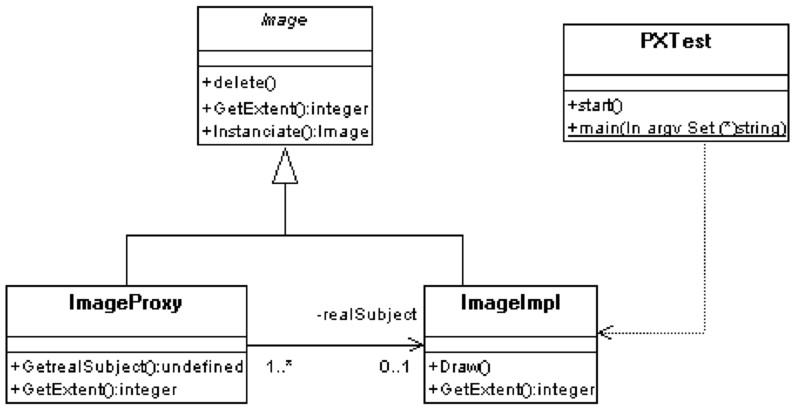


Figure 3-31. Structure after pattern application

A "*Subject*" class has been created. According to the option chosen, either it bears the name previously attributed to "*RealSubject*" or a new name.

- ♦ This class is abstract and defines a virtual destroyer.
- ♦ This class bears the definitions of the "*RealSubject*" public operations, which have not been previously excluded. These operations are defined as being abstract.

A proxy class has been created. This class is derived from the abstract "*Subject*" class, and has an association towards the "*RealSubject*" class (the class on which pattern generation has been carried out). This class redefines the entirety of the operations present on "*Subject*", and the corresponding implementation consists of a redirection of these calls towards the "*RealSubject*" via the association which links these two classes.

The "*Real Subject*" class is modified in order to specialize the "*Subject*" class.

- According to the configuration of the parameters, the name of this class is either changed or left unchanged.
- Non-excluded public operations are modified, in order that they become operations redefined from those presented on the "*Subject*".

All use of the "*RealSubject*" class (associations, operational use), with the exception of parameter types, are redirected to the "*Subject*" class.

A class instantiation operation (by default "*Instantiate()*") is created on the "*Subject*" class. It sends back a "*Subject*" and serves as the manufacturer (please see the *Factory* pattern) for the "*Subject*" class. Its implementation by default instantiates a proxy and sends it back.

A "*Get<NameAssociation>*" operation is created on the "*Proxy*" class, in order to define the proxy type on the "*RealSubject*" class. The code of this operation is left up to the user.

By default, the returned parameter type is "*<ClassName RealSubject>*" in C++, and "*< ClassName RealSubject>*" in Java. This type may be modified by the user.

Consequences - Advantages

Indirect access to the "*Real Subject*" through the Proxy occurs in a transparent manner, without it being necessary to introduce code modifications, in order to take this indirect nature into account.

New services, which will come in between the "*Real Subject*" and the Proxy (distant access, unlocking, protection), may be implemented, after the implementation of the "*Real Subject*" itself and the client who is using it.

Consequences - Drawbacks

An extra level of indirection is introduced, as well as an increase in structural complexity, due to the introduction of new classes.

The Adapter pattern

Presentation

The *Adapter* pattern is used to convert the interface of an "*Adapter*" class towards the interface of a "*Target*" class, with these two classes having no relation. This allows the "*Target*" class to use the services of an existing class, in order to realize one or several functions.

There exist two types of adapter:

- The class adapter is obtained by simultaneously deriving a class from the "*Target*" and "*Adapted*" classes.
- The object adapter is derived from the "*Target*" class and is associated with an "*Adapted*" class object.

Motivations for choice

- You wish to put in place a reusable class, but one which is based on diverse existing or future classes, which have been developed by third parties, and which do not necessarily have an interface which is compatible with our reusable class.
- You wish to unify the use of diverse non-related classes within a structure, which presents a unique manipulation interface.
- You wish to mask the interface of a service behind a proprietary interface, in order to limit the impact of future evolution of this service's interface.

Adapter structure before applying the pattern

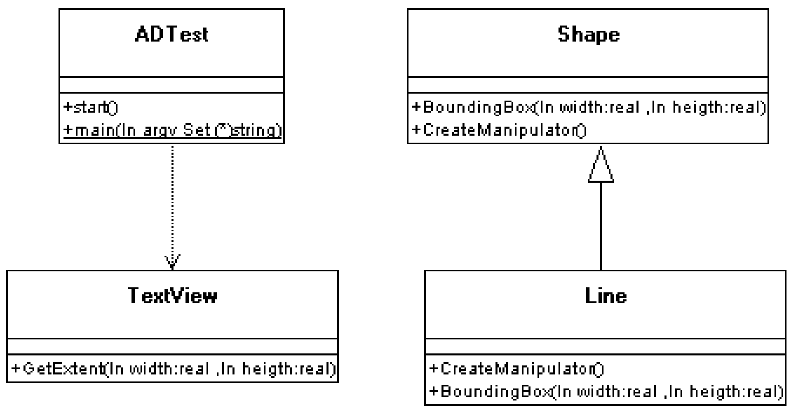

ADTest
+start()
+main(In argv Set (*)string)
Shape
+BoundingBox(In width:real ,In heigth:real)
+CreateManipulator()
TextView
+GetExtent(In width:real ,In heigth:real)
Line
+CreateManipulator()
+BoundingBox(In width:real ,In heigth:real)

Figure 3-32. Structure before pattern application

The *Adapter* pattern can be generated in two ways:

1 - Generation from the "*Target*" class (represented here by the "*Shape*" class). This is the "*Create class on target*" mode:

- ♦ In this scenario, the class which plays the role of Adapter is generated, whether a class adapter or an object adapter is created . The preliminary structure, then, consists of 2 classes:
    - ♦ A "*Target*" class, which will be the class from which the desired services will be referred to by clients.
    - ♦ An "*Adapted*" class, which represents the realization of the service, whose interface does not correspond to the interface of the "*Target*" class. This class has no link to the previous one.

2 - Generation from the "*Adapter*" class. This is the "*Update existing class*" mode. This option does not produce any new classes, and allows you either to use an existing class, such as "*Adapter*", or simply to re-apply the pattern to an "*Adapter*" class.

The "Create class on target" operating mode

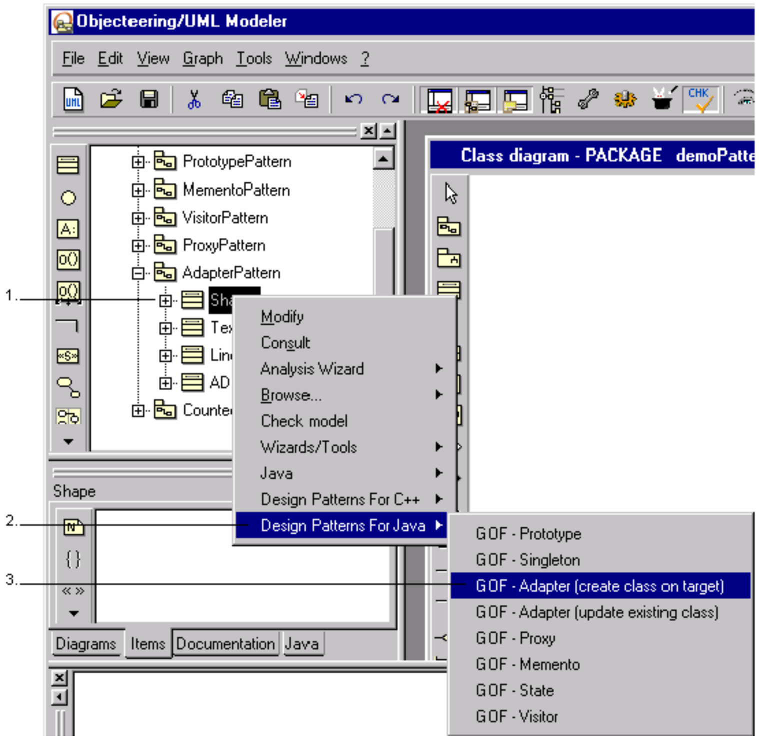


Figure 3-33. Application of the *Adapter* pattern to a class

Steps:

1 - In the explorer, select the class which will play the role of *Target*, and right-click to open the context menu.

2 - Select the "*Design Patterns for C++*" or "*Design Patterns for Java*" menu item.

3 - Select the "*GOF - Adapter (create class on target)*" menu item.

For the *Objecteering/Design patterns for C++* module, the following dialog box (Figure 3-34) then appears:

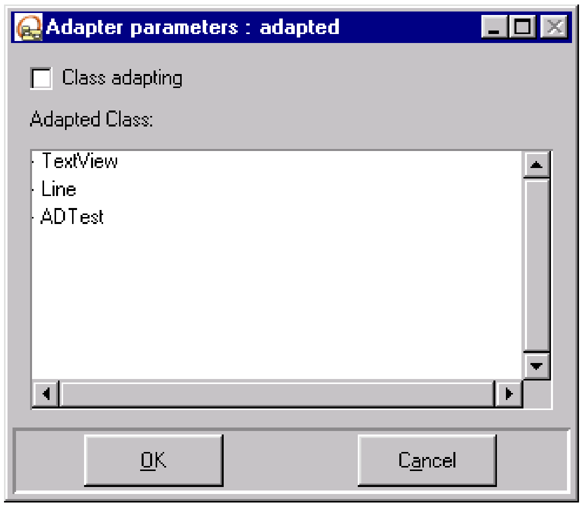


Figure 3-34. Parameters of the *Adapter* pattern

- If the "*Class adapting*" box is checked, you may refer to the creation of a class adapter. Where it is not checked, an object adapter is put in place.
- The "*Adapted class*" list allows you to select the "*Adapted*" class.

For the *Objecteering/Design Patterns for Java* module, the following dialog box (Figure 3-35) then appears:

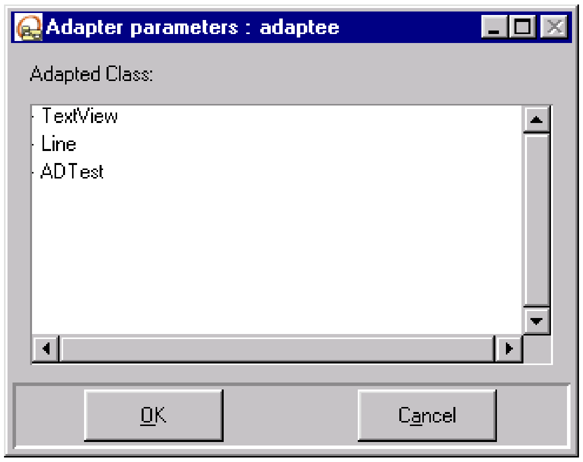


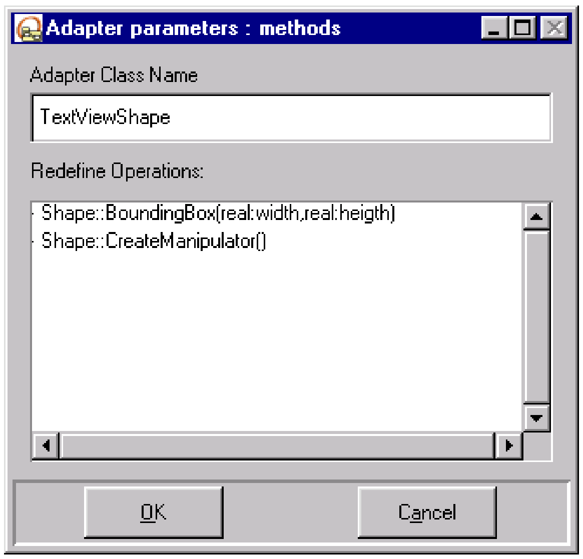
Figure 3-35. Parameters of the *Adapter* pattern

This dialog box is identical to the previous one, except for the box to be checked, which allows you to refer to the creation of a class adapter. Only the object adapter is possible, since the class adapter is based on multiple generalization, a function which is not available in Java.

Let us now look at the operating mode specific to each of these two variants.

Operating mode for the implementation of a class adapter

When the "*Class adapting*" option has been selected (*Objecteering/Design Patterns for C++* module only), the following dialog box (Figure 3-36) appears:



TextViewShape

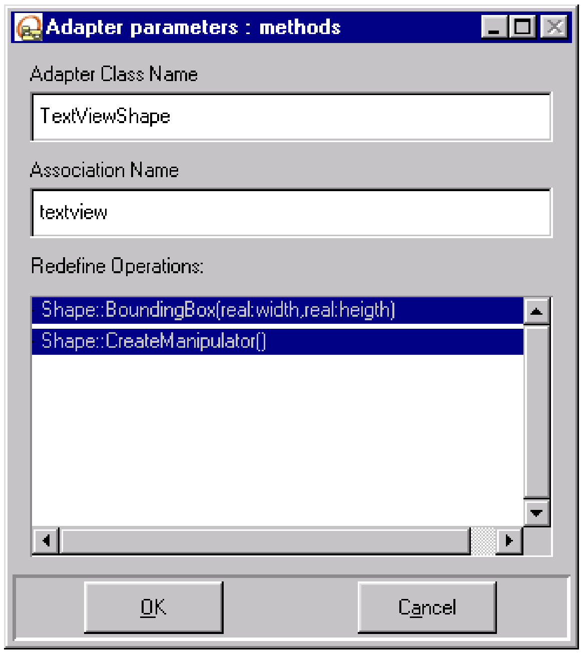
Shape::BoundingBox(real:width,real:heigth)
Shape::CreateManipulator()

Figure 3-36. Parameters of the *Adapter* pattern

- The "*Adapter class name*" field: This allows you to give a name to the "*Adapter*" class, which will be created. Its default value is made of the concatenation of the names of the "*Adapted*" and "*Target*" classes.
- The "*Redefines operation*" list: This allows you to select the services of the "*Target*" class to be redefined in the "*Adapter*". These services are chosen from the list of the "*Target*" class public operations.

Operating mode for implementing an object adapter

Where the "*Class adapting*" option has been unchecked for the *Objecteering/Design Patterns for C++* module, and in all cases where the *Objecteering/Design Patterns for Java* module is concerned, the following dialog box appears:



Adapter Class Name

TextViewShape

Association Name

textview

Redefine Operations:

Shape::BoundingBox(real:width,real:heigth)

Shape::CreateManipulator()

Figure 3-37. Parameters of the *Adapter* pattern

- The "*Adapter class name*" entry field allows you to attribute a name to the "*Adapter*" class which will be created. Its default value is made up of the concatenation of the names of the "*Adapted*" and "*Target*" classes.
- The "*Association name*" entry field allows you to specify the role name of the protected visibility association which will be created between the "*Adapter*" and the "*Adapted*". Its default value is the "*Adapted*" class name in lower case.
- The "*Redefine operations*" list allows you to select the services of the "*Target*" class which must be redefined within the "*Adapter*" class. These services are chosen from the list of the "*Target*" class public operations.

The "Update existing class" operating mode

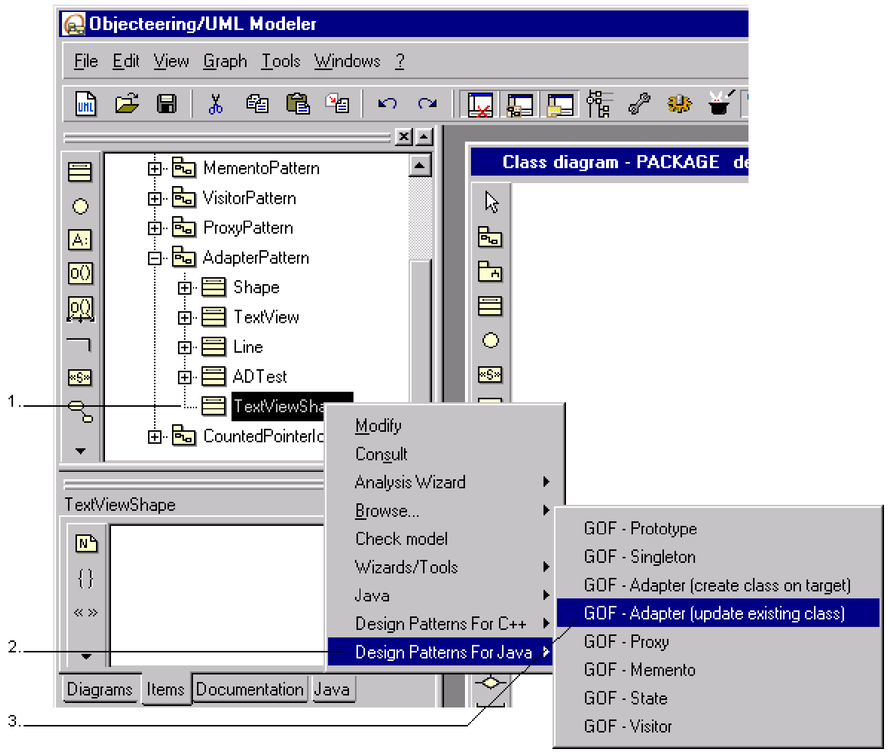


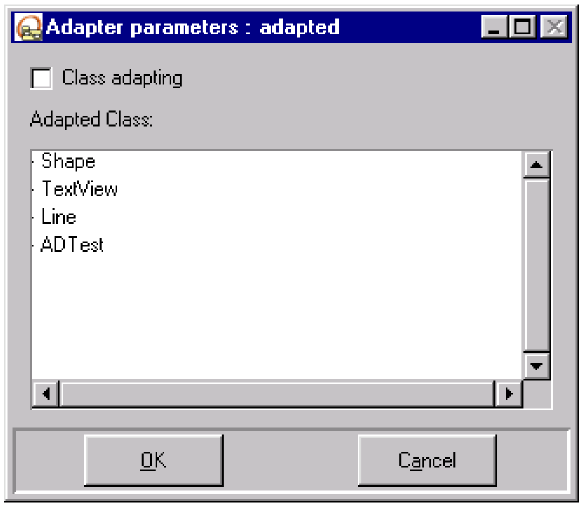
Figure 3-38. Application of the *Adapter* pattern to a class

Steps:

1 - In the explorer, select the class which will play the role of the "*Adapter*" (or which is already playing that role), and right-click to open the context menu.

2 - Select the "*Design Patterns for C++*" or "*Design Patterns for Java*" sub-menu and then select the "*GOF - Adapter (update existing class)*" item.

For the *Objecteering/Design Patterns for C++* module, the following dialog box then appears:


Adapter parameters : adapted

☐ Class adapting

Adapted Class:

- Shape
- TextView
- Line
- ADTest

Figure 2-39. Parameters of the *Adapter* pattern

This dialog box has previously been described in this section.

The "*Target*" class, which serves as base class to the "*Adapter*", must now be indicated. This is carried out via the following dialog box.

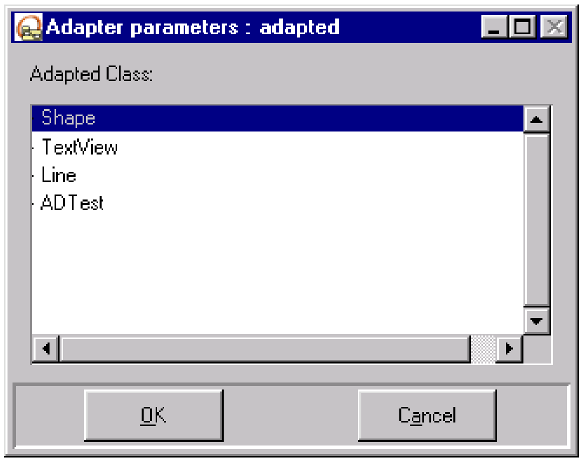
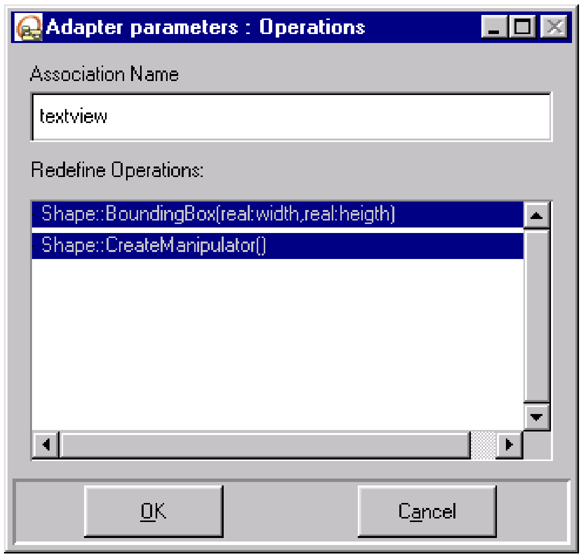


Figure 3-40. Parameters of the *Adapter* pattern

The list presented in this dialog box is that of the classes accessible from the "*Adapter*" class. The "*Target*" class must be one of them. If this is not the case, then a use link from the package in which the "*Adapter*" class is found towards the package where the desired "*Target*" class is found must be missing.

For the object adapter, all that remains is to specify the role name of the association which links "*Adapter*" and "*Adapted*", and the "*Target*" class operations, which must be redefined. The dialog box below is used to do this.



Association Name

textview

Redefine Operations:

Shape::BoundingBox(real:width,real:heigth)

Shape::CreateManipulator()

Figure 3-41. Parameters of the *Adapter* pattern

Adapter structure after applying the pattern

The final structure differs according to what was chosen to obtain a "*Class adapter*" or an "*Object adapter*".

Structure after application of the Class Adapter pattern

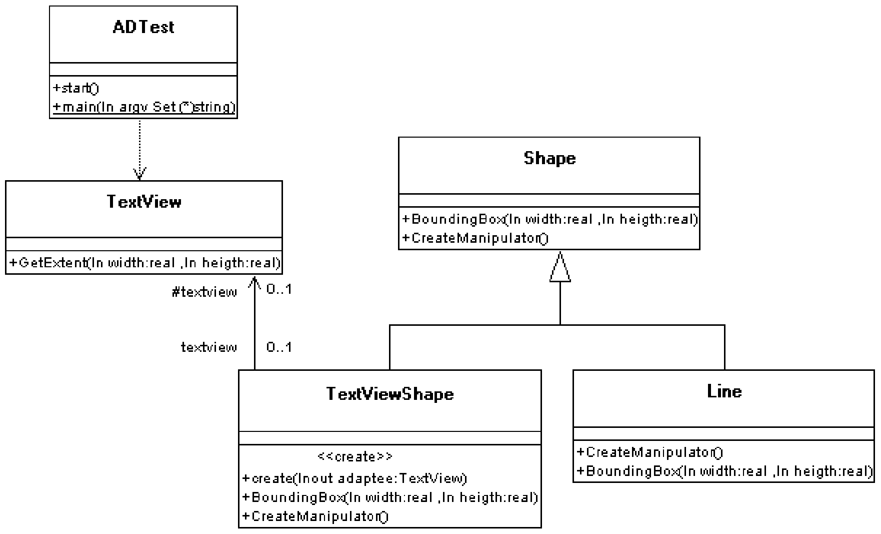

ADTest
+start()
+main(In argv Set (*)string)
TextView
+GetExtent(In width:real ,In heigth:real)
#textview
0..1
textview
0..1
Shape
+BoundingBox(In width:real ,In heigth:real)
+CreateManipulator()
TextViewShape
<<create>>
+create(Inout adaptee:TextView)
+BoundingBox(In width:real ,In heigth:real)
+CreateManipulator()
Line
+CreateManipulator()
+BoundingBox(In width:real ,In heigth:real)

Figure 3-42. Structure after application of the *Class Adapter* pattern

If the class adapter has been generated via the "*GOF - Adapter (create class on target)*" menu item, the "*Adapter*" class (here "*TextShape*") is generated.

Whatever the class *Adapter* generation alternative:

- a generalization between the "*Adapter*" and the "*Target*" class has been implemented.
- a generalization between the "*Adapter*" and the "*Adapted*" class has been implemented.
- the operations redefined on the "*Adapter*" class have been implemented for all the selected operations. The implementation of these operations, in other words the reference to the "*Adapted*", remains up to the user.

Consequences - Advantages

Class adapter: Other than the redefinition of the "*Target*" class "*Request*" operation, the "*Target*" class can also redefine some of the "*Adapted*" class operations, where necessary.

Class adapter: The adaptation is obtained with a good degree of efficiency, since it is realized within one sole object which fuses the interface of the "*Target*" class and the service provided by the "*Adapted*" class, without it being necessary to refer to operations via indirections.

Object adapter: The interface of a class hierarchy can be adapted, without it being necessary to create an adapter for each of them, simply by basing oneself on this hierarchy parent class interface.

Consequences - Drawbacks

Class adapter: The language used must allow the use of multiple generalizations, which excludes Java, and must allow you to specify generalization as an implementation, which is indeed the case in C++ (private or protected generalization). If this is not the case, the "*Adapter*" could be handled as if it were the "*Adapted*" type, which it is not.

Class adapter: The adaptation of a class and all its sub-classes within a sole "*Adapter*" class cannot be factorized, by basing oneself on the parent class interface. You are obliged to create an "*Adapter*" per "*Adapted*" class concerned.

Object adapter: The public interface of the "*Adapted*" must expose services sufficiently to allow adaptation, since the "*Adapter*" does not have access to the protected interface.

The Counted Pointer Idiom

Presentation

The *Counted Pointer* C++ idiom facilitates the management of memory spaces allocated dynamically, by allowing the automatic destruction of these objects when they are no longer referenced.

Instead of directly accessing dynamically allocated objects (called Body), a "*Handle*" class, which allows access to the Body, is introduced.

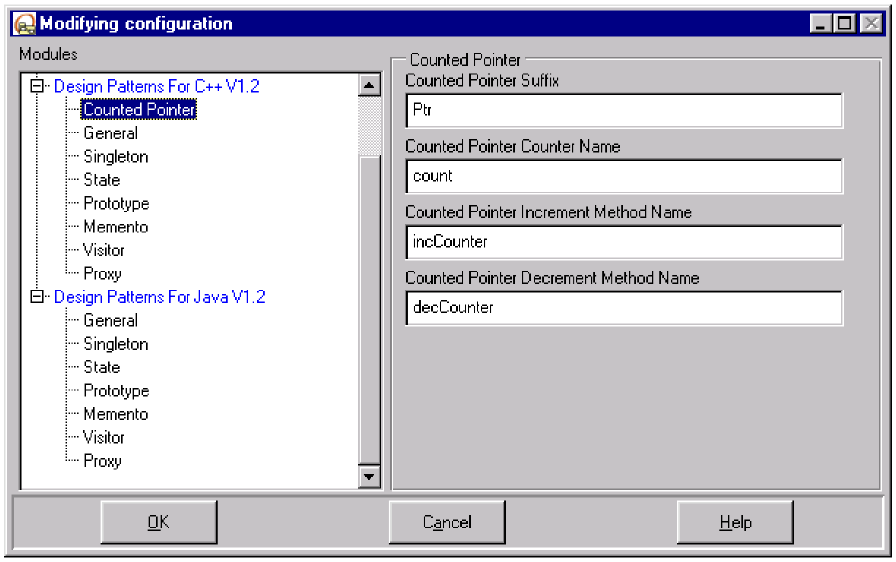
The "*Handle*" class is an intelligent pointer type, and, in this sense, redefines the "*to*" operator, but furthermore refers to reference counting situated in the Body: this counter is incremented when a "*Handle*" instance is created, and decremented when this instance is destroyed. Due to this, the "*Handle*" instances must be allocated on the pile rather than on the heap.

The *Counted Pointer* idiom can be considered as a particular variant of the *Proxy* pattern, although its implementation is completely different.

Motivations for choice

- ♦ You wish to integrate an automatic garbage collector mechanism into a C++ application, in order to ensure the destruction of objects which are no longer referenced.
- ♦ Furthermore, you wish to ensure secure access (in terms of validity of references) to objects allocated on the heap, in order not to undergo problems linked to the use of gross pointers such as non-initialized pointers, or invalid references (destroyed objects).
- ♦ You wish to share objects between different clients. Responsibility for the management of these objects cannot be explicitly and definitively given to one of these clients in particular, so you do not wish any of these clients to manage these objects.

Counted pointer configuration parameters



Counted Pointer Suffix: Ptr

Counted Pointer Counter Name: count

Counted Pointer Increment Method Name: incCounter

Counted Pointer Decrement Method Name: decCounter

Figure 3-43. Configuration parameters for the *Counted Pointer* idiom

- "*Counted pointer suffix*": Allows you to specify a suffix which will be added to the name of the Body class, on which the generation is run, to create the name of the "*Handle*" class.
- "*Counted pointer counter name*": Allows you to specify the name which will be given to the attribute which realizes the reference counting on the Body class. This whole type attribute is generated on the Body class.
- "*Counted pointer increment operation name*": Name of the operation to be generated on the Body class, in order to increment reference counting.
- *Counted pointer decrement operation name*": Name of the operation to be generated on the Body class, in order to decrement reference counting.

Structure before application of the idiom

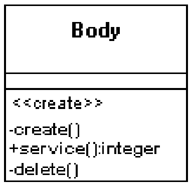


Figure 3-44. Structure before application of the *Counted Pointer* idiom

Only the Body class is necessary before this generation. This class:

- ♦ must be allocated to the heap.
- ♦ must not already possess an attribute which has exactly the same name as that parameterized in the "*Counted pointer counter name*" field, and a whole different type.
- ♦ must not already possess operations whose names correspond to the names of operations specified in the "*Counted pointer increment operation name*" or "*Counted pointer decrement operation name*" fields, if they have not already been generated by this pattern.

Note: There must not already exist a class whose name corresponds to the name of the Body class + the suffix parameter in the "*Counted pointer suffix*" field.

Counted pointer idiom operating mode

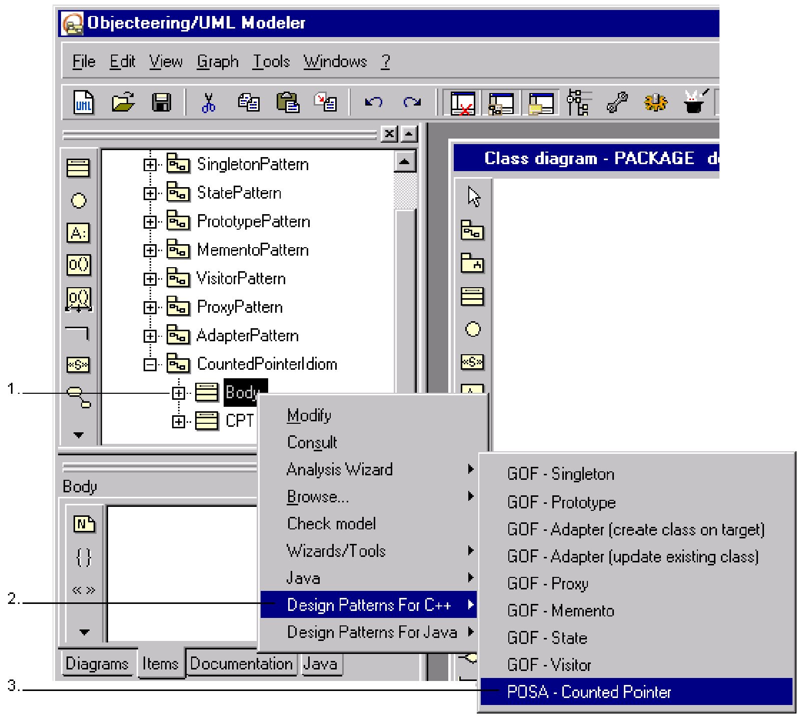


Figure 3-45. Application of the *Counted Pointer* idiom on a pointer class

Steps:

1 - In the explorer, select the class on which you wish to generate a pointer class with reference counting, and right-click to open the context menu.

2 - Select the "*Design Patterns for C++*" menu item.

3 - Select the "*POSA - Counted pointer*" menu item.

Structure after application of the idiom

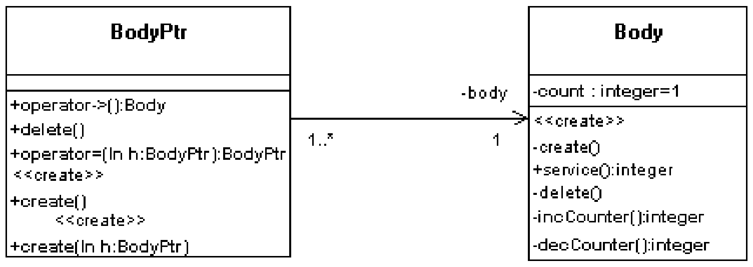


Figure 3-46. Structure after application of the *Counted Pointer* idiom

A "*Handle*" class is generated, whose name is made up of the corresponding Body class, to which the suffix specified in the module parameterization is added. The "*Body*" class is modified as follows:

- Addition of an attribute which allows reference counting
- Addition of two operations to increment and decrement the counter

Consequences - Advantages

There is no longer any need to manage the destruction of objects for which reference management has been installed for this purpose.

This strategy prevents "memory leaks".

Consequences - Drawbacks

Control over the destruction of objects is lost.

The co-existence of two strategies of control over objects' lifetimes is not really possible, since confusion in manipulation will occur sooner or later. It is, therefore, preferable to apply the "*counted pointer*" strategy to all objects allocated on the heap, or to none at all.

Other advantages

To apply this strategy to all the classes of a leaf package, you need simply use the *"Design patterns for C++/POSA - Counted pointer on each class*" command. This command will generate a "*counted pointer*" on each of the package's classes, except for those classes which are themselves "*counted pointers*". These classes are recognized through the suffix which was chosen during module parameterization.

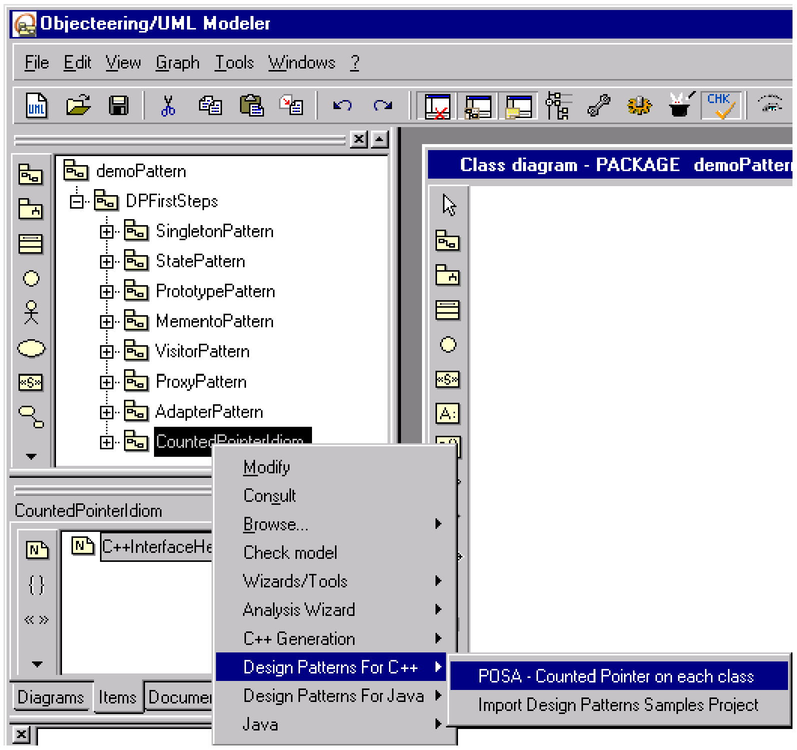


Figure 3-47. Application of the *Counted Pointer on each class* idiom to a package

Chapter 4: Bibliography

Bibliography

- [Gamma+95] *Design Patterns*, Elements of Reusable Object Oriented Software - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissids - Addison Wesley 1995.
- [Buschmann+] *Pattern Oriented Software Application* - Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad & Michael Stal - John Wiley & sons 1996
- [PLoPD2] *Pattern Languages of Program Design vol. 2* - John M. Vlissides, James O. Coplien, Norman L. Kerth edt - Addison Wesley 1996
- [PLoPD3] *Pattern Languages of Program Design vol. 3* - Robert Martin, Dirk Riehle & Frank Bushmann edt. - Addison Wesley 1998.

Index