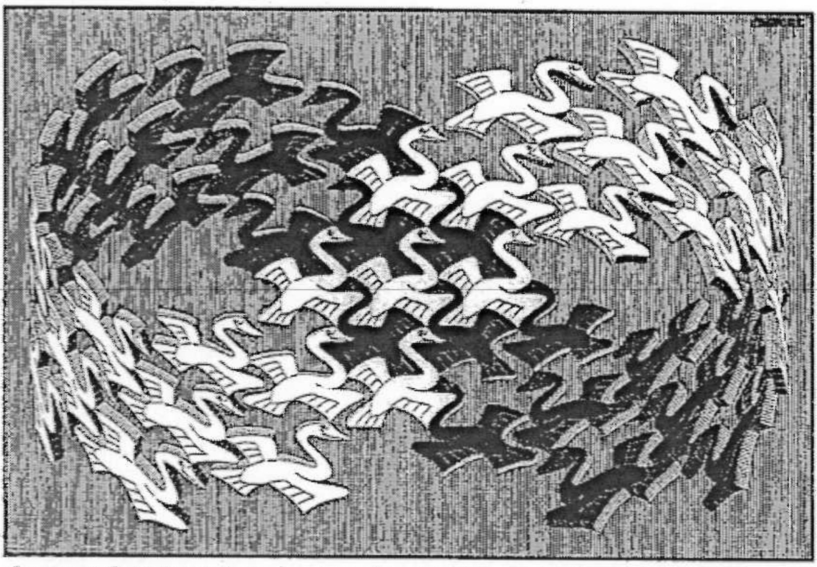
Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Foreword by Grady Booch

SOFTWARE Development PRODUCTIVITY AWARD 1994

Contents

Preface xi

Foreword xiii

Guide to Readers xv

1 Introduction 1

1.1 What Is a Design Pattern? 2
1.2 Design Patterns in Smalltalk MVC 4
1.3 Describing Design Patterns 6
1.4 The Catalog of Design Patterns 8
1.5 Organizing the Catalog 9
1.6 How Design Patterns Solve Design Problems 11
1.7 How to Select a Design Pattern 28
1.8 How to Use a Design Pattern 29

2 A Case Study: Designing a Document Editor 33

2.1 Design Problems 33
2.2 Document Structure 35
2.3 Formatting 40
2.4 Embellishing the User Interface 43
2.5 Supporting Multiple Look-and-Feel Standards 47
2.6 Supporting Multiple Window Systems 51
2.7 User Operations 58
2.8 Spelling Checking and Hyphenation 64

2.9 Summary 76

Design Pattern Catalog 79

3 Creational Patterns **81**

Abstract Factory 87

Builder 97

Factory Method 107

Prototype 117

Singleton 127

Discussion of Creational Patterns 135

4 Structural Patterns **137**

Adapter 139

Bridge 151

Composite 163

Decorator 175

Facade 185

Flyweight 195

Proxy 207

Discussion of Structural Patterns 219

5 Behavioral Patterns **221**

Chain of Responsibility 223

Command 233

Interpreter 243

Iterator 257

Mediator 273

Memento 283

Observer 293

State 305

Strategy 315

Template Method 325
Visitor 331

Discussion of Behavioral Patterns 345

6 Conclusion **351**
6.1 What to Expect from Design Patterns 351
6.2 A Brief History 355
6.3 The Pattern Community 356
6.4 An Invitation 358
6.5 A Parting Thought 358

A Glossary **359**

B Guide to Notation **363**
B.1 Class Diagram 363
B.2 Object Diagram 364
B.3 Interaction Diagram 366

C Foundation Classes **369**
C.1 List 369
C.2 Iterator 372
C.3 ListIterator 372
C.4 Point 373
C.5 Rect 374

Bibliography **375**

Index **383**

1.3 Describing Design Patterns

How do we describe design patterns? Graphical notations, while important and useful, aren't sufficient. They simply capture the end product of the design process as relationships between classes and objects. To reuse the design, we must also record the decisions, alternatives, and trade-offs that led to it. Concrete examples are important too, because they help you see the design in action.

We describe design patterns using a consistent format. Each pattern is divided into sections according to the following template. The template lends a uniform structure to the information, making design patterns easier to learn, compare, and use.

Pattern Name and Classification

The pattern's name conveys the essence of the pattern succinctly. A good name is vital, because it will become part of your design vocabulary. The pattern's classification reflects the scheme we introduce in Section 1.5.

Intent

A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

Also Known As

Other well-known names for the pattern, if any.

Motivation

A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help you understand the more abstract description of the pattern that follows.

Applicability

What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations?

Structure

A graphical representation of the classes in the pattern using a notation based on the Object Modeling Technique (OMT) [RBP+91]. We also use interaction diagrams [JCJO92, Boo94] to illustrate sequences of requests and collaborations between objects. Appendix B describes these notations in detail.

Participants

The classes and/or objects participating in the design pattern and their responsibilities.

Collaborations

How the participants collaborate to carry out their responsibilities.

Consequences

How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What aspect of system structure does it let you vary independently?

Implementation

What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues?

Sample Code

Code fragments that illustrate how you might implement the pattern in C++ or Smalltalk.

Known Uses

Examples of the pattern found in real systems. We include at least two examples from different domains.

Related Patterns

What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?

Creational Patterns

Abstract Factory (87) Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Builder (97) Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Factory Method (107) Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Prototype (117) Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Singleton (127) Ensure a class only has one instance, and provide a global point of access to it.

Structural Patterns

Adapter (139) Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Bridge (151) Decouple an abstraction from its implementation so that the two can vary independently.

Composite (163) Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Decorator (175) Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Facade (185) Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Flyweight (195) Use sharing to support large numbers of fine-grained objects efficiently.

Proxy (207) Provide a surrogate or placeholder for another object to control access to it.

Behavioral Patterns

Chain of Responsibility (223) Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Command (233) Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Interpreter (243) Given a language, define a represention for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Iterator (257) Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Mediator (273) Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Memento (283) Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

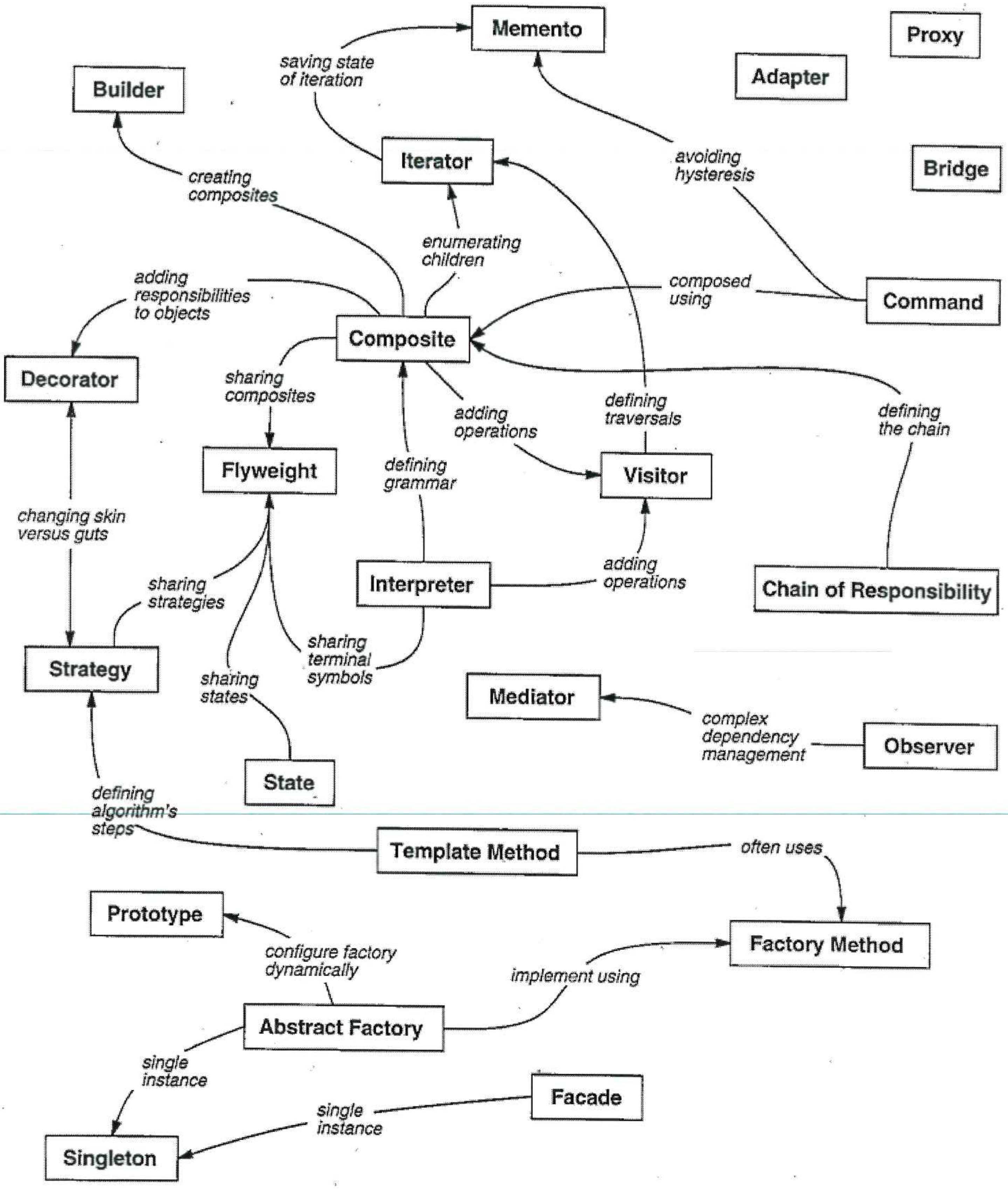
Observer (293) Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

State (305) Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Strategy (315) Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Template Method (325) Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Visitor (331) Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.



Design Pattern Relationships