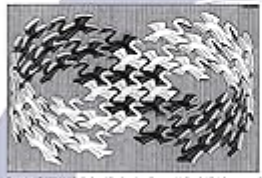
Design Patterns

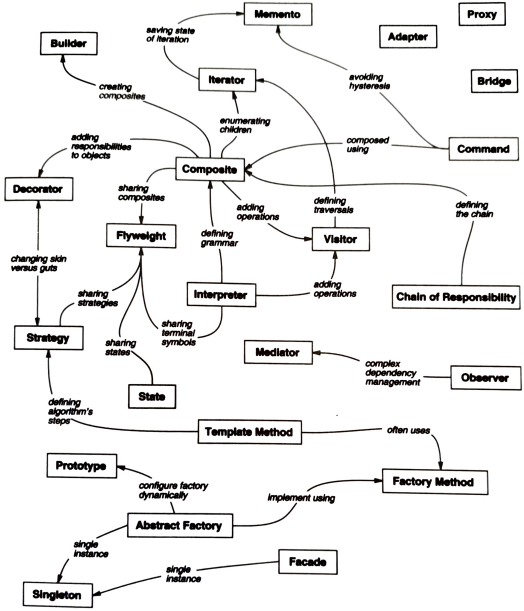
Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

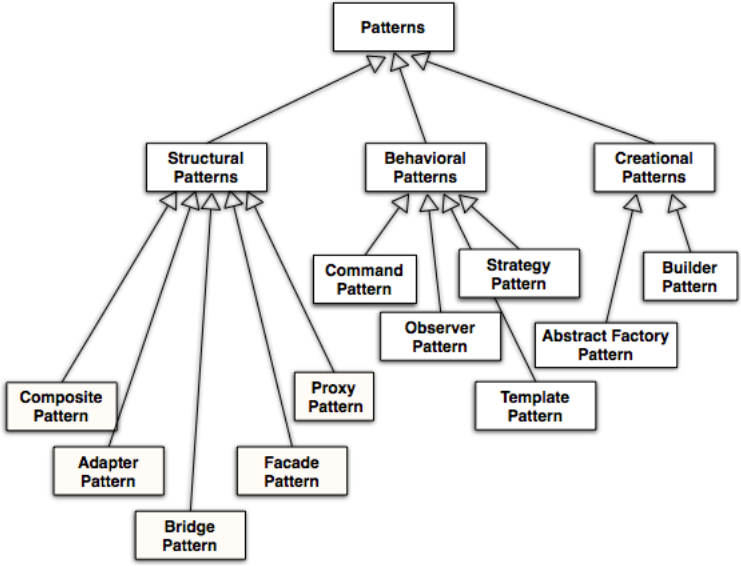


Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES



Design Pattern Relationships



Bernd Bruegge. (I can't find this in the book.)

fix the position of individual buildings on the site, within the complex, one by one, according to the nature of the site, the trees, the sun: this is one of the most important moments in the language;

104. SITE REPAIR
105. SOUTH FACING OUTDOORS
106. POSITIVE OUTDOOR SPACE
107. WINGS OF LIGHT
108. CONNECTED BUILDINGS
109. LONG THIN HOUSE

within the buildings' wings, lay out the entrances, the gardens, courtyards, roofs, and terraces: shape both the volume of the buildings and the volume of the space between the buildings at the same time—remembering that indoor space and outdoor space, yin and yang, must always get their shape together;

110. MAIN ENTRANCE
111. HALF-HIDDEN GARDEN
112. ENTRANCE TRANSITION
113. CAR CONNECTION
114. HIERARCHY OF OPEN SPACE
115. COURTYARDS WHICH LIVE
116. CASCADE OF ROOFS
117. SHELTERING ROOF
118. ROOF GARDEN

when the major parts of buildings and the outdoor areas have been given their rough shape, it is the right time to give more detailed attention to the paths and squares between the buildings;

119. ARCADES
120. PATHS AND GOALS
121. PATH SHAPE
122. BUILDING FRONTS
123. PEDESTRIAN DENSITY
124. ACTIVITY POCKETS
125. STAIR SEATS
126. SOMETHING ROUGHLY IN THE MIDDLE

now, with the paths fixed, we come back to the buildings: within the various wings of any one building, work out the fundamental gradients of space, and decide how the movement will connect the spaces in the gradients;

127. INTIMACY GRADIENT
128. INDOOR SUNLIGHT
129. COMMON AREAS AT THE HEART
130. ENTRANCE ROOM
131. THE FLOW THROUGH ROOMS
132. SHORT PASSAGES
133. STAIRCASE AS A STAGE
134. ZEN VIEW
135. TAPESTRY OF LIGHT AND DARK

Alexander et al. *A Pattern Language*. Oxford UP, 1977.

Cohesion. A measurement of the dependencies among classes within a subsystem. "How well a unit of code maps to a logical task of entity" (Barnes and Köllig).

Coupling. A measure of the dependencies between two subsystems. "Classes that are tightly coupled are hard to reuse in isolation, since they depend on each other. Tight coupling leads to monlithic systems, where you can't change or remove a class without understanding and chanigin many other classes" (Gamma et al).

Partitions and layers. Partitions vertically divide a system into several independent subsystems that provide services on the same level of abstraction. Layers divide a system horizontally; a layer is a subsystem that provides services to a higher layer/level of abstraction.

Extensibility. The extent to which new functional requirements can be added easily to the existing system.

Scalability. The extent to which existing components can be multiplied easily in the system.

Reusability. The extent to which a module/system/subsystem can be used by another system without requiring changes in the existing system model (design reuse) or code base (code reuse).

It's also important to understand the difference between class inheritance and interface inheritance (or subtyping). Class inheritance defines an object's implementation in terms of another object's implementation. In short, it's a mechanism for code and representation sharing. In contrast, interface inheritance (or subtyping) describes when an object can be used in place of another.

It's easy to confuse these two concepts, because many languages don't make the distinction explicit.

Gamma et al, pg 17

Because inheritance exposes a subclass to details of its parent's implementation, it's often said that "inheritance breaks encapsulation." The implementation of a subclass becomes so bound up with the implementation of its parent class that any change in the parent's implementation will force the subclass to change.

Gamma et al, pg 19

Liskov substitution principle

> *If for each object o_1 of type S, there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T.*

Barbara Liskov, "Data Abstraction and Hierarchy," *SIGPLAN Notices*, 23.5, May 1988.

> *If a value of type S can be substituted into any context where a value of type T is expected, then S is a subtype of T.*

Program to an interface, not an implementation. *DP, pg 18*

Favor object composition over class inheritance. *DP, pg 20*

Delegation *is a way of making composition as powerful for reuse as inheritance. In delgation, two objects are involved in handling a request: a receiving object delegates operations to its* **delegate**.

The main advanteage of delegation is that it makes it easy to compose behavioiurs at run-time and to change the way they're composed. *DP, pg 20-21*

[I]n the OO world you hear a good deal about "patterns". I wonder if these patterns are not sometimes evidence of [the need to make code transformations the compiler should do]. When I see patterns in my programs, I consider it a sign of trouble. . . Any other regularity in the code is a sign, to me at least, that I'm using abstractions that aren't powerful enough. often that I'm generating by hand the expansions of some macro that I need to write. . . .

Peter Norvig found that 16 of the 23 patterns in *Design Patterns* were "invisible or simpler" in Lisp.

Paul Graham, "Revenge of the Nerds"

Design Patterns in Dylan or Lisp

16 of 23 patterns are either invisible or simpler, due to:

- First-class types (6): `Abstract-Factory, Flyweight, Factory-Method, State, Proxy, Chain-Of-Responsibility`
- First-class functions (4): `Command, Strategy, Template-Method, Visitor`
- Macros (2): `Interpreter, Iterator`
- Method Combination (2): `Mediator, Observer`
- Multimethods (1): `Builder`
- Modules (1): `Facade`