A Design Pattern Dictionary

Version 2
of 8-Mar-04

by Avner Ben

www
.skilldesign
.com

PDF version (6.5MB!)

A Design Pattern Dictionary

Section 1 - Introduction

Foil 0 - Contents

www.skilldesign.com

The purpose of the design pattern dictionary is to serve as reference point for commonly accepted software design patterns and idioms

#	Page title	Description	History
1	The purpose of the Dictionary	Purpose	11-Dec-03 *02-Jul-06*
2	The "Skill-driven" form	Dictionary entry structure	11-Dec-03 *02-Jul-06*
3	The "Skill-driven" form - 1/4	Dictionary entry structure discussion - part 1 of 4	11-Dec-03 *02-Jul-06*
4	The "Skill-driven" form - 2/4	Dictionary entry structure discussion - part 2 of 4	11-Dec-03 *08-Mar-04*
5	The "Skill-driven" form - 3/4	Dictionary entry structure discussion - part 3 of 4	11-Dec-03 *07-Mar-04*
6	The "Skill-driven" form - 4/4	Dictionary entry structure discussion - part 4 of 4	11-Dec-03 *07-Mar-04*
7	Style	language conventions	11-Dec-03 *07-Mar-04*
8	Design pattern	From the user"s perspective, a design pattern is a way of ordering objects in space that suggests functionality	11-Dec-03 *07-Mar-04*
9	Sources	Sources cited in the dictionary	11-Dec-03 *09-Mar-04*

Print this section.

A Design Pattern Dictionary

Section 1 - Introduction

Foil 1 - The purpose of the Dictionary

The ***purpose*** of the design pattern dictionary
is to serve as ***reference*** point
for commonly ***accepted*** software design patterns and idioms,
for readers who are already ***familiar*** with the subject*.

The ***majority*** of these design patterns and idioms
are featured in the ***course*** "Design Patterns in Practice"
by Avner Ben
(***foils*** at the Skill-Driven Design site),
for use as reference by past ***students*** .
Some of these patterns are C++ specific.

Comments, corrections and contributions
to make it a more general resource
are ***welcome***.

**Sorry, this is not a design pattern textbook!*

A Design Pattern Dictionary

Section 1 - Introduction

Foil 2 - The "Skill-driven" form

www.skilldesign.com

Dictionary entries consist of the following items:

1. Context:	One or more programmatic requirements that invite the pattern. (See discussion on separate page)
2. Challenge:	One or more discrete programming-level challenges that the designer must face, to satisfy the above requirements. (See discussion on separate page)
3. Skill:	"SkillTree" pseudocode of the Solution. The purpose of this item is to save trivial code samples, using generic notation. (See discussion on separate page)
4. Participants:	Detailed list of static design entities that are refined(or introduced) by the pattern. These may be classes, interfaces, class templates, methods and data. (See discussion on separate page)
5. Signature:	The very " *pattern"* - the unique arrangement of program items by which the solution is recognized (suggesting the problem). (See discussion on separate page)
6. Used patterns and idioms:	A list of other patterns and idioms in this dictionary, which are included in the current pattern language. (See discussion on next page)
7. Variations:	A list of patterns, possibly in this dictionary, that extend the current pattern. (See discussion on separate page)
8. Used by:	A list of pattern languages in this dictionary that may involve the current pattern or idiom. (See discussion on separate page)
9. Known issues:	Important issues to be considered - limitations, side effects - when applying this pattern. (See discussion on separate page)
10. Sources:	Familiar books, publications or class libraries that have inspired the pattern as specified here. (See discussion on separate page)
11. Scope:	The object oriented platforms (languages) to which the pattern applies. (See discussion on separate page)

www
.skilldesign
.com

A Design Pattern Dictionary

Section 1 - Introduction

Foil 3 - The "Skill-driven" form - 1/4

List of dictionary items (#1-2 of 11):

1. Context: One or more programmatic requirements that invite this pattern, (e.g. in the context of the State pattern we find "*the state of one object must reflect the current state of another* "). In many occasions the context stands for a phenomenon that has a Common term (*e.g. the context above summs up the well known "controlled data redundancy"*). This term. where available, is listed up front, quoted and followed by colon *("Controlled data redundancy": the state of one object must reflect the current state of another).*

2. Challenge: One or more discrete programming-level challenges that the designer must face, to satisfy the above requirements. Our insistence of separation of challenge from context (which is not observed by most design pattern textbooks) stresses the distinction between business requirement and implementation dependent solution. *(E.g. In the context of the Prototype pattern we find the business requirement for "Selecting objects by visual example". The challenge "To copy the object without knowing its type" results from the attempt to satisfy this requirement in a programming language that binds object creation to knowledge of its type.)* It is not part of the necessity!

   Many of these challenges default to one of the following three categories:

   - Challenge to the very object-oriented paradigm. (*E.g. the Visitor Pattern challenges the object-oriented premise that each operation has exactly one receiver object.*) These patterns appeal to the technically-minded and have inspired much technical fireworks. They are best eliminated by the availability of languages that bypass the entire problem.
   - Challenge to the paradigm of particular object-oriented languages. (*E.g. the Singleton pattern challenges the lack of support for real modularity by most current object-oriented languages.*) These are uninteresting patterns solving trivial problems by the application of routine job and are best eliminated by the availability of suitable languages. Naturally, (and sadly!) these are the best known and most cited of all patterns.
   - Challenge to impose the object-oriented paradigm over the very problem-domain (*e.g., the Composite and decorator patterns challenges to rearrange both composition/aggregation and classification aspects of the entity/relationship model, to allow substutability for the purpose of navigation, resulting in an entity/relationsip model that is likely to be contra-intuitive for those familiar with the domain but uninitiated in object-oriented design.*) These are the profound of all patterns and are, naturally, either missunderstood or ignored.

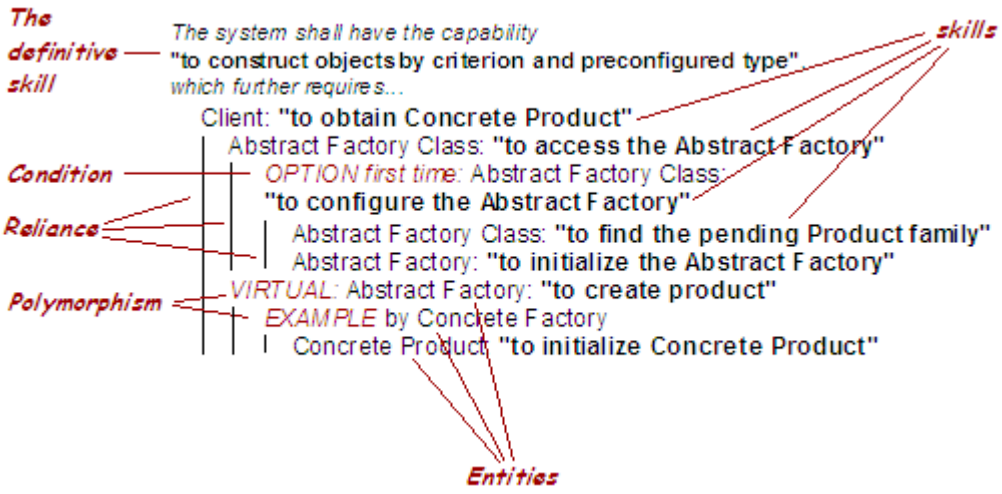
Foil 4 - The "Skill-driven" form - 2/4

List of dictionary items (#3 of 11):

3. Skill: The purpose of this item is to prevent the need for code samples, *using the "SkillTree" notation.*

A ***skill*** is a functional requirement of the software product that is visible in the code. In the design, one or more skills gather to define the role of one business (or technical) entity. In the code, a skill maps to either a method or a code block.

Each design pattern or idiom in this dictionary is usually defined by a single requirement-level skill. This skill is of descriptive nature and is not necessarily traceable in code in one piece. Bellow it are indented skills that map to code directly as methods, blocks of code or statements. *(e.g. "To switch among alternative implementations of an entire functionality" is the definitive skill of a finite state machine. "To shift to the next state" is a program-level skill, required by the above).*



The indentation of skill below skill represents***reliance***. The skill above *relies upon* the skills below. The skill above further *requires* the skills below. *(E.g. "to attach to Resource" relies upon "to increase ownership" - the first requirement is not satisfied without the second skill.)*

Reliance is translated to code as statement nesting, block nesting or "message passing" (function call), as applies. *In the rare cases where a skill is assigned to a class, rather than an object (as in "instance() by Singleton Class"), it is to be implemented as a "static method".*

Following the condition, a reliance is preceeded by the name of the ***entity*** blessed with the skill.

Reliance may be restricted by an explicit ***condition***. *Selection* and *iteration* have obvious code mapping. *"**Option**"* maps to code as "if", "else", "case", etc. Mutually eclusive options are indented below the *"**alternatives**"* title. *"**Many**"* maps to code as "while", "foreach", etc. The condition, where not obvious, may be followed by explanatory text *(e.g. "many item in group:", "Option first time:").*

Use-case driven design is expressed by subliminal reliance variants which, being requirement-minded, have no mechanical translation to code primitives. An "***asynchronous***" reliance would handle an event required by the skill above it, but is not explicitly *controlled* by it. (e.g. "to end-initialize registered threads" requires both "to register thread at the application" and Many: "to end initialize thread". However, the registration is not controlled from this point in program flow - it is an asynchronous *requirement*!

Polymorphic *requirements* are expressed by ***virtual*** reliance, usually followed by some typical ***examples***. Where the first would become a virtual function call in a base class reference, the latters are the implementations in some typical derived classes. Like asynchronous reliances, example reliances are a use-case-driven feature with no direct mapping to code.

A Design Pattern Dictionary

Section 1 - Introduction

Foil 5 - The "Skill-driven" form - 3/4

List of dictionary items (#4-5 of 11):

4. Participants: A detailed list of static design elements that the pattern involves. These are normally object classes and rarely, the very class objects. One expects to find, in a good design, entities that either rerpresent the problem domain ("business enitities") or encapsualte technical solutions. Design patterns are often guidleines for the latter. The definition of each participant stresses *functionality* - what it does, its role, its responsibility. In addition, participant definition may repeat significant static information, (from the *signature* item). *(E.g. in signature we find "...contains the heterogeneous collection" - an association. In participants we find "...features the generic access sub-interface for that collection" - functionality.)*

5. Signature: The imprint of the pattern in the code, the very *pattern* . Most design patterns possess a strong *structural* signature, conveniently represented by an entity/relationship diagram ("class diagram" in UML). Control-flow is reduced to navigation in the model, inspired, where not obvious, by the SkillTree pseudocode item above. Only a few patterns (e.g. Prototype and Template method) feature pure procedural signature that does not require additional classes or even member data.
This dictionary uses text to render the class diagram, by means of capitalized entity names and verbs. (The real class diagrams are abundant elsewhere). E.g. the text " *FSM contains States, and references the current among them*" renders the following class diagram: " *1:n composite aggregation between FSM and State, 1:1 shared aggregation between FSM and State, an and-constraint on these two associations*".
The following *verbs* bear special significance:

- "*Contains*" - composite aggregation, containment by value. *(Class diagram: Solid diamond at source of association).* Association target in the plural form denotes 1:n multiplicity, *e.g. "Invoker contains Commands"*.
- "*References*" - shared aggregation, containment by reference. *(Class diagram: No/hollow diamond at source of association).* Association source in the plural that "*share*" - reference n:1, *e.g. "Resource Proxies share a Resource Implementation*".
- "*...and represents*" (in addition to *contains* or *references)* - the source also functions as "proxy" for the target. *(This behavioral aspect has no significant class diagram mapping. A "stereotype" may be handy.)*
- "*...by*" (in addition to *contains/ references)* - qualified association. *(Class diagram: Named rectangle at association source). (E.g. "Factory references Creator Plugs by unique retrieval key".)*
- "*is concrete...*" - inheritance. *(Class diagram: Hollow triangle at target of association).* The base-class is abstract (possibly pure interface), unless stated otherwise. "Concrete" entities are always "kind of" the respective entities.
- "*Parameterized over*" - genericity, class template. *(Class diagram: Generic-argument-named dotted rectangle, at upper right corner of the class-template-named rectangle).*
- Any other association implied by use of a verb between two entities *(e.g. "creates", "notifies")* is a "uses" association *(Class diagram: Named dotted arrow).* The use of the plural form on this occasion *(e.g. "factory creates products")* does not suggest multiplicity. (i.e. we do not know how many products are emitted at each request or at all.)
- Other elements (besides classes and associations) are data and methods added to existing classes. These additional design elements have little - or no - class diagrams significance.

www
.skilldesign
.com

A Design Pattern Dictionary

Section 1 - Introduction

Foil 6 - The "Skill-driven" form - 4/4

List of dictionary items (#6-11 of 11):

6. Used patterns and idioms:	A list of other patterns and idioms in this dictionary, which are included in the current pattern language.
7. Variations:	A list of patterns, possibly in this dictionary, that extend the current pattern.
8. Used by:	A list of pattern languages in this dictionary that may involve the current pattern or idiom. Also, applications that typically apply the pattern.
9. Known issues:	Important issues to be considered when applying the current pattern, such as problems that may arise from applying the pattern under unsuitable conditions *(e.g. using double dispatch in an unstable hierarchy)* and enhancements worth exploring *(e.g. multithreading precautions)*.
10. Sources:	Familiar books, publications or class libraries that have inspired the pattern as specified here, with little or no significant change. (See list on separate page).
11. Scope:	The object oriented languages to which the pattern applies, characterized by the availability of particular language facilities or the lack of them: e.g. strong typing, automatic garbage collection, genericity, class-objects, global variables, multi-methods, etc.

A Design Pattern Dictionary

Section 1 - Introduction

Foil 7 - Style

Idiom: The word "Idiom" is used here (after Coplien) for items whose definition matches that of a pattern, but which do not merit the honorable title of "pattern" *(e.g. Reference Counting idiom)*.

Resource: The term "Resource" *(as in "To prevent redundant creation of global resources")* is always used here in the meaning of *encapsulated* resource, in the object-oriented sense - "an object that represents the (problem-domain) resource (in the solution domain)". Therefore, such skills as "to create a resource" may (or may not) mean creating a physical resource outside the computer, but will always mean encapsulating the latter's state and behavior inside an object that proceeds to spend its entire life inside the computer.

Type: By "type" *(as in "Context is parameterized over Policy type")* is meant here "user-defined type", as in an OOP class (rather than an interface detached from implementation, as is sometimes to be found in scientific texts). A "type family" is an inheritance hierarchy. Type and type families are polymorphic, unless stated otherwise. *(E.g. in "to copy an object without knowing its type", the object in question is obviously polymorphic.)* Base-classes are abstract (possibly pure interface), unless otherwise implied. *"Any"* of a type family is some subclass.

Behavior: "Behavior" and "functionality" always apply *to an objec*t (understood from context) and never to a global procedure. *(E.g. in "To reconfigure discrete functionality" - the functionality is of the Context object.)* "Discrete" functionality means a single method or part of its logic.

Ability: "The capability for..." is an abstract skill characteristic of a superclass. "Responsible for..." is how the skill is implemented at a subclass.

Concrete: A " Concrete" entity is a subclass of the respective entity suggested by its name (after GOF).

Capitalization: References to participant entity names are capitalized and sometimes italicized.

A Design Pattern Dictionary

Section 1 - Introduction

Foil 8 - Design pattern

www.skilldesign.com

Design pattern of architecture:

From a reform theory of building architecture (Alexander). *Main highlights:*

- From the user's point of view, a design pattern is a way of ordering objects in space that suggests functionality.*
- From the architext's point of view, a design pattern is a structured solution to a common problem (featuring only the core of the solution and a working example).
- Design patterns are to be found interwoven. Buildings should form a "living pattern language", guiding the visitor to useful areas in the obvious way.
- In this and in additional works, Alexander features a comprehensive catalogue of "design patterns" in the default "Alexandrian" form.
- A design pattern is a form of prose consisting of passages that normally describe a common context, a recurring problem in this context, and the core of the solution, often by example.
- A design pattern is not a formula. Repeated a million times, its instances will never be exactly the same.
- It all has to do with "The Quality Without a Name".

Design pattern of software:

Originated in attempts at the early 1990's to formulate common practices found in object-oriented programming, regardless of language, adapting Alexander's terminology.

- The representative work is GOF *(see sources)* - by Gamma, Vlissides, Helm and Johnson, 1995, in conjunction with pioneering work by Coplien, Schmidt and others.
- Mastery of software design patterns has become the recognized structured way of traversing the gap between top-level object-oriented design and object-oriented programming.

---

* *Alexander never phrased it this way - this is my simplified interpretation.*

Code	Name	Author	Publisher	Year
Alexander	"The Timeless Way of Building"	Christopher Alexander	Oxford University Press	1979
Alexandrescu	"Modern C++ Design"	Andrei Alexandrescu	Addison Wesley	2001
Coplien	Articles at "C++ Report magazine"	James O. Coplien		1998-9
C++	The C++ programming language class library			
GOF	"Design Patterns: Elements of Reusable Software"	Gama, Vlissidess, Helm & Johnson	Addison Wesley	1995
Java	The Java programming language class library			
Lauder	"Pluggable Factory in Practice," C++ Report Magazine	Anthony Lauder		Oct 1999
Martin	"Agile software development"	Robert C. Martin *et al*		2003
Meyers	"Effective C++"	Scott Meyers	Addison Wesley	1997
Stroustrup	"The C++ Programming Language, 3rd Ed."	Bjarne Stroustrup	Addison Wesley	1999
Vlissidess	"Pattern Hatching" - *articles at* "C++ Report" magazine	John Vlissidess		1996-9

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 0 - Contents

www.skilldesign.com

The Design-Pattern Dictionary contains structured summaries of some 30 design patterns and idioms, commonly used in object-oriented programming

	State	"To switch among alternative implementations of an entire functionality"
	Startup Registration idiom	"To let the default object population be registered *spontaneously*"
	Strategy	"To reconfigure discrete functionality"
T	Template Method	"To outline and guarantee the execution of a generic process"
V	Visitor	"To assign discrete functionality by object type"

Print this section.

www .skilldesign .com

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 1 - Abstract Factory

Context:
Send feedback

1. Creating objects by two criteria: Base type (Known to the application) and environment (preconfigured). *(E.g. creating GUI controls, given control type and knowing the GUI system being emulated).*
2. An array of factory methods with the same implementation criterion. *(Same example as above, where the need for each control type has aroused on a separate occasion.)*

Challenge: To encapsulate the decisions in a single object.

Skill:
View Wirechart

The system shall have the capability
to construct objects by criterion and preconfigured type,
which further requires...

1: Client**: "to obtain *Concrete Product*"**
2: Abstract Factory Class**: "to obtain the *Abstract Factory*"**
3: *OPTION first time*: Abstract Factory Class**: "to configure the *Abstract Factory*"**
4: Abstract Factory Class**: "to find the pending Product family"**
5: Abstract Factory**: "to initialize the *Abstract Factory*"**
6: *ABSTRACT*: Abstract Factory**: "to create product"**
7: *EXAMPLE* by Concrete Factory
8: Concrete Product**: "to initialize *Concrete Product*"**

Participants:

Abstract Factory:	A Singleton. The capability to create discrete products for a pre-configured environment.
Concrete Factory:	A set of methods for creating discrete products for a concrete environment.
(Some) Abstract Product:	(A passive entity.) One of the Abstract Factory's declared Product types.
(Some) Concrete Product:	(A passive entity.) The implementation of one of the Abstract Factory's declared Product types for a particular environment.

Signature: Abstract factory responds to creation messages explicitly named after each conceptual product type. Each concrete factory implements this interface for a concrete environment. The products are arranged in respective discrete type families where environment implementations derive from product.

Used patterns and idioms: singleton.

Source: GOF.

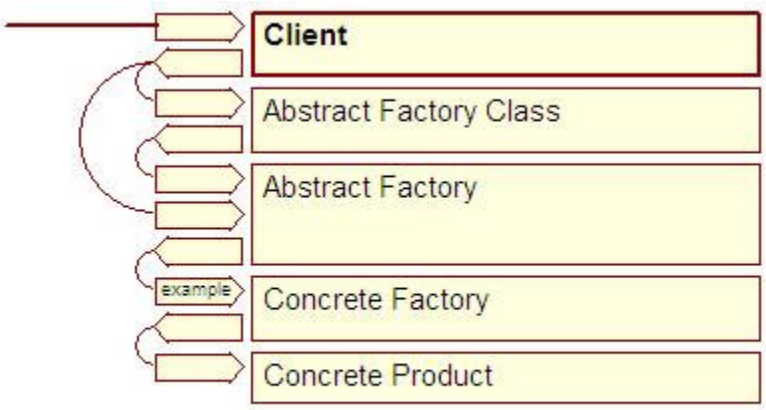
Scope: General.

Foil 1A - Abstract Factory

Foil 1.2.1A: To construct objects by criterion and preconfigured type

The system shall have the capability
to construct objects by criterion and preconfigured type,
which involves...

Send feedback



Foil 1.2.1A - Description:

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 2 - Adapter

www.skilldesign.com

Context: The requirement for concrete type implementation is already met, but by a non-native type.

Send feedback

Challenge: To make the different type substitutable within the native interface.

Skill: View Wirechart

The system shall have the capability
to access a foreign implementation of native functionality,
which further requires...

1: Adapter**: "to initialize *Adapter*"**
2: Adaptee**: "to initialize *Adaptee*"**
3: *MANY*: Client**: "to use the native interface"**
4: Adapter**: "to implement the native interface"**
5: Adaptee**: "to do the job"**
6: *OPTION*: Adapter**: "to adapt the result"**

Participants:

Native Interface:	The set of permissible message selectors for a common functionality.
Adaptee:	A set of methods for the native functionality, provided with the wrong message selectors. *(In a strongly-typed language:)* not derived from Native Interface.
Adapter:	Represents the Adaptee in the system, interpreting to Adaptee language. Usually, does little more than forwarding, but may also aggregate functionally, separate functionality or emulate missing functionality.

Signature: Adapter references (or contains) Adaptee, representing it. Adapter implements Interface.

Used by: clients of legacy code.

Scope: General. (Preferably, languages where late binding is obligatory.)

www .skilldesign .com

A Design Pattern Dictionary

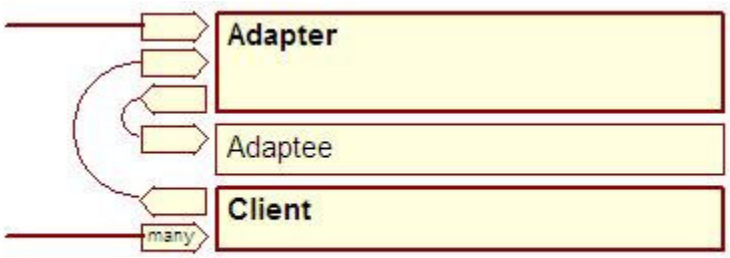
Section 2 - The Dictionary

Foil 2A - Adapter

Foil 1.2.2A: To access a foreign implementation of native functionality

The system shall have the capability
to access a foreign implementation of native functionality,
which involves...

Send feedback



Foil 1.2.2A - Description:

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 3 - Algebraic Hierarchy

www .skilldesign .com

Context: multi-method in a type family that features type promotion (with strictly linear precision order. *E.g. a sequence of Numbers, by precision.)*

Send feedback

Challenge: Given two objects - to convert the lesser to the type of the higher.

Skill: View Wirechart

The system shall have the capability
to promote object type,
which further requires...

1: Context**: "to perform operation with two *Quantities*"**
2: left hand Quantity (a Quantity)**: "to compare *Quantity* precision"**
3: *OPTION*: Context**: "to promote *Quantity*"**
4: *MANY*: Quantity In Conversion (a Quantity)**: "to convert *Quantity* one stage up"**
5: *OPTION suggested*: Context**: "to replace Left hand *Quantity*"**

Participants:

Quantity:	The capability to be sequenced by precision. *Specifically:* The capabilities for comparison with another quantity and for upward conversion.

Used by: Metamorphic Bridge (Generic Number).

Signature: Quantity responds to messages to compare its precision with another Quantity, implemented by Concrete Quantity (e.g. forwarded from template method in quantity).

Comments:

- The alternative Solution is double dispatch. Algebraic hierarchy is less efficient, but easier to write and maintain.
- While conversion is possible but up and down the precision ladder, both up conversion is more natural, as long as it does not create a cyclic namespace dependency.

Source: Coplien.

Scope: General.

Foil 3A - Algebraic Hierarchy

Foil 1.2.3A: To promote object type

The system shall have the capability
to promote object type,
which involves...

Send feedback

Foil 1.2.3A - Description:

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 4 - Bridge

www.skilldesign.com

Context: Send feedback

Two alternative classification schemes seem equally valid (e.g. stream opened for either input or output and also encapsulates a specific device). A replacement for multiple inheritance, possibly with dynamic classification.

Challenge:

To base the design on one classification while retaining the latter. Possibly, to allow the latter to vary during object lifetime.

Skill: View Wirechart

The system shall have the capability
to separate choice of Implementation from interface,
which further requires...

1: Client**: "to create configured *Behavior*"**
2: Concrete Implementation**: "to initialize *Concrete Implementation*"**
3: Behavior**: "to initialize *Behavior*"**
4: Behavior**: "to configure *Implementation*"**
5: Concrete Behavior**: "to perform concrete action"**
6: *ABSTRACT*: Implementation**: "to perform Low-level concrete action"**
7: *EXAMPLE* by Concrete Implementation

Participants:

Behavior:	Basic services used by client (e.g. Stream).
Particular Behavior:	The complete set of services to be used by client, typically non-polymorphically (e.g. Input Stream).
Implementation:	A set of primitives to be used internally by Behavior (e.g. to fill buffer).
Concrete Implementation:	Methods for the set of primitives to be used internally by Behavior.

Signature:

Behavior contains Implementation. Client uses Concrete Behavior and typically provides the Concrete Implementation.

Known issues:

Since Implementation typically features the entire spectrum of primitives needed by particular Behaviors, the signing its provision to the client and possibly allowing its replacement during Behavior's lifetime may invalidate the Concrete Behavior (e.g. resulting in an Input-Stream with Buffer open for output). widely using multiple inheritance instead would not allow that to happen, it would not allow dynamic classification (where desired) in the first place.

Used by:

I/O libraries.

Source:

GOF.

Scope:

General.

Foil 4A - Bridge

Foil 1.2.4A: To separate choice of implementation from interface

The system shall have the capability
to separate choice of Implementation from interface,
which involves...

Send feedback

Foil 1.2.4A - Description:

www .skilldesign .com

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 5 - Caching Pool (with reference counting)

Context: *See Flyweight.*

Send feedback

Challenge: To prevent redundant deletion and creation of often-used Resources (regardless of whether used at the moment).

Skill: View Wirechart

The system shall have the capability
to allow efficient creation of global Resources,
which further requires...

1: Smart Resource Pointer**: "to initialize *Smart Resource Pointer*"**
2: *OPTION not in pool*: Flyweight Pool**: "to create *Resource*"**
3: Smart Pointer Counter**: "to increase ownership"**
4: Smart Resource Pointer**: "to finalize *Smart Resource Pointer*"**
5: Smart Pointer Counter**: "to decrease ownership"**
6: Flyweight Pool**: "to handle unused *Resource*"**
7: Caching Strategy**: "to decide the fate of unused *Resource*"**
8: *OPTION*: Flyweight Pool**: "to cease managing *Resource*"**
9: Resource**: "to finalize *Resource*"**

Participants:

See Flyweight.

Caching Strategy: A function object used to decide the fate of an unused Resource. Configurable during runtime.

Signature: Flyweight contains Caching Strategy. *(See Also Flyweight.)* The flyweight is unlikely to reference the strategy, since it may contain data.

Used patterns and idioms: Strategy, (*See also Flyweight*.).

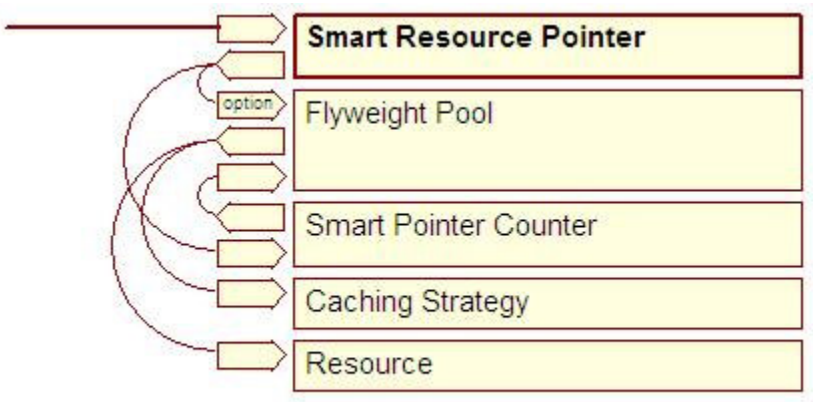
Scope: General (system programming).

Foil 5A - Caching Pool (with reference counting)

Foil 1.2.5A: To allow efficient creation of global resources

The system shall have the capability
to allow efficient creation of global Resources,
which involves...

Send feedback



Foil 1.2.5A - Description:

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 6 - Command

www.skilldesign.com

Context: Initiator of action cannot (or need not) be there in time to launch the action (e.g. scheduling, selection from menu). Possibly, action may be undone later.

Send feedback

Challenge: To encapsulate the execution request with its arguments (and possibly its undo arguments).

Skill: View Wirechart

The system shall have the capability
to separate request from execution,
which further requires...

1: Client**: "to create execution request"**
2: Command**: "to request execution with arguments"**
3: Invoker**: "to register execution request"**
4: *ASYNC,OPTION*: Invoker**: "to invoke execution request"**
5: Invoker**: "to Invoke *Receiver* into action"**
6: Receiver**: "to perform action"**

Participants:

Client:	Initiates the request. Creates a Concrete Command over the Receiver, which it registers with the Invoker.
Invoker:	A heterogeneous container of commands. Responsible for activating the commands at the proper event.
Command:	The Invoker's generic contents. The capability to delay execution and store its parameters. Possibly also to undo it.
Concrete Command:	the capability to activate a particular receiver.
Receiver:	The object of the request.

Signature: Invoker contains Commands. Concrete Command references Receiver. Client references (or contains) both Receiver and Invoker, creates the Command and registers it with Invoker.

Used by: Order of initialization pattern, Observer pattern, *menu systems, transaction-based systems.*

Scope: Languages that do not support bound methods / delegates. General (when involving more data or functionality than just the receiver, method and invoke-time arguments).

Foil 6A - Command

Foil 1.2.6A: To separate request from execution

The system shall have the capability
to separate request from execution,
which involves...

Send feedback

Foil 1.2.6A - Description:

www .skilldesign .com

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 7 - Composite

Context: Send feedback

A recursive object structure (e.g. file directory tree) has to be traversed from the outside (e.g. view traversing its document), applying routine functionality (e.g. formatting for display).

Challenges:

- To traverse the structure node by node, ignoring node type (possibly applying operations that affect the integrity of the structure).
- To copy or move nodes, ignoring Node type (counting on automatic validation) .

Skill: View Wirechart

The system shall have the capability
to traverse a recursive structure uniformly,
which further requires...

1: Context**: "to process heterogeneous subtree"**
2: *MANY in Component*: Context**: "to process *Component*"**
3: *ABSTRACT*: Component**: "to get *Component*"**
4: *EXAMPLE* by Composite
5: *ABSTRACT*: Component**: "to help processing *Component*"**
6: *EXAMPLE* by Particular Composite
7: *EXAMPLE* by Particular Leaf

Participants:

Component:	The capability to traverse a subtree. Optionally, the ability to match node types.
Particular Leaf:	Silently ignores the traversal protocol. (But does other useful jobs).
Composite:	Manages a heterogeneous collection of Components. Implements the default traversal protocol.
Particular Composite:	Implements particular collection management protocol. Optionally, re-implements the traversal protocol.

Signature: Both particular Leaves and default Composite are Concrete Components. Composite contains (or references) Components. The Particular Composite may choose to limit the scope of the inherited association (see rectangular inheritance of association).

Used patterns and idioms: Recursive Object-Structure idiom, Rectangular Inheritance of Association idiom, unified interface.

Used by: Document/View architectures (document side), generic presentation and report systems, arithmetic expressions, parse trees, rule bases.

Known issues: See Rectangular Inheritance of Association idiom.

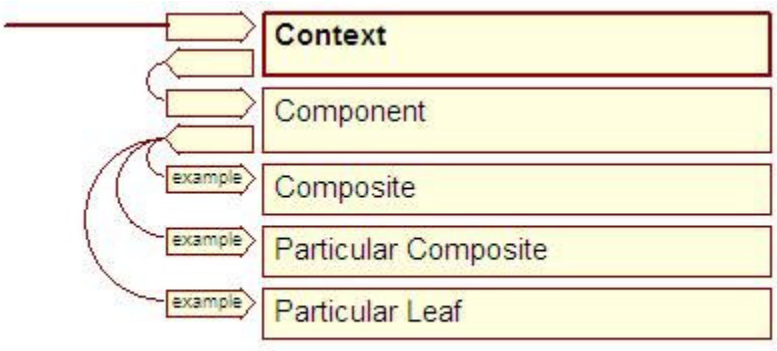
Scope: General.

Foil 7A - Composite

Foil 1.2.7A: To traverse a recursive structure uniformly

The system shall have the capability
to traverse a recursive structure uniformly,
which involves...

Send feedback



Foil 1.2.7A - Description:

www
.skilldesign
.com

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 8 - Decorator

Context: "Dynamic and multiple classification": The implementation of some facet of behavior may be extended during its object's lifetime.

Send feedback

Challenges:

- To encapsulate the difference in behavior without modifying the subject. To make the extended functionality over the subject substitutable with the subject.
- To practically replace an object base (setting it to an existing object).
- To replace supercall by delegation.

Skill: *The system shall have the capability*
to enhance discrete functionality dynamically,
which further requires...

View Wirechart

1: Client**: "to decorate concrete *Subject*"**
2: Concrete Decorator**: "to enhance *Subject* functionality"**
3: Client**: "to update *Subject* reference"**
4: Client**: "to use *Subject*"**
5: *ABSTRACT*: Subject**: "to do the *Subject* job"**
6: *EXAMPLE* by Concrete Decorator
7: Concrete Decorator**: "to decorate the *Subject* job"**
8: Concrete Decorator**: "to pre-process *Subject* work"**
9: *ABSTRACT*: **"to do the *Subject* job"**
10: Concrete Decorator**: "to post-process *Subject* work"**

Participants:

Subject:	The Subject interface. Allows the Decorator to be substitutable for Concrete Subjects.
Concrete Subject:	The object to be decorated. (A passive participant.)
Decorator:	The set of default methods for interfacing for Subject.
Concrete Decorator:	The set of methods that enhance Subject behavior.

Signature: Decorator is a concrete Subject and references (another) Subject, interfacing for it. Concrete Decorator derives from Decorator.

Notes:

- A degenerate form without an abstract Decorator is possible but undesirable *(see Recursive Object-Structure idiom).*
- In some cases, the decorator replaces (rather than enhances) subject functionality (e.g. Typedef replaces the "name" property of Type).

Used patterns and idioms: Recursive Object-Structure idiom.

Used by: (Typically:) parsing, coding and decoding algorithms.

Source: GOF.

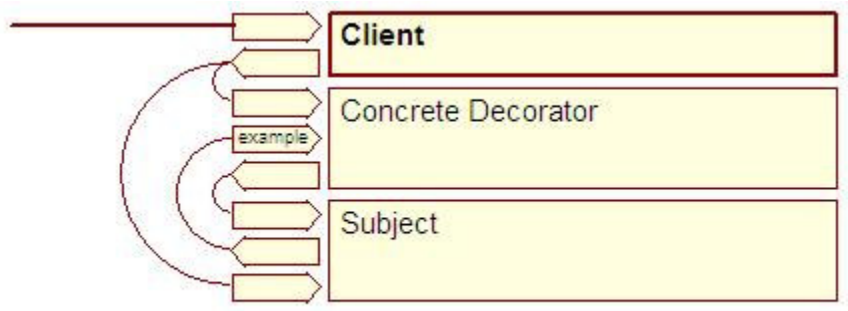
Scope: General. (Preferably, languages where late binding is obligatory.)

Foil 8A - Decorator

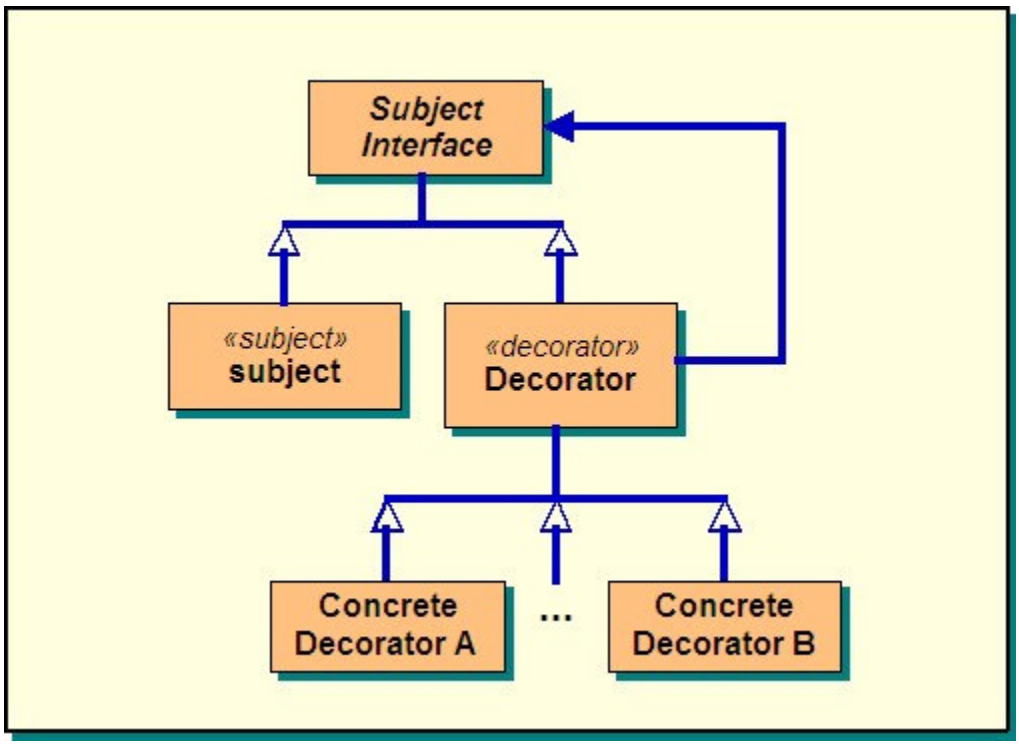
Foil 1.2.8A: To enhance discrete functionality dynamically

The system shall have the capability
to enhance discrete functionality dynamically,
which involves...

Send feedback



Object Model



A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 9 - Double Dispatch Idiom

www.skilldesign.com

Context: "Double Polymorphism", "Multi-method" requirement: Functionality distributed between two objects of unknown type (e.g. finding the intersection area between two shapes).
Send feedback

Challenge: To resolve the multi-method efficiently, (e.g. without runtime searching).

Skill: View Wirechart

The system shall have the capability
to resolve multiple-object functionality among objects,
which further requires...

1: Left Side Subject (a Concrete Subject)**: "to begin performing action from left side"**
2: Right Side Subject (a Concrete Subject)**: "to end performing action from right side on left side"**

Participants:

Subject:	The skill to dispatch the operation to the right hand object plus the set of all skills to resolve the operation by the right hand object. *(In a dynamically scoped language: May also implement the dispatch.)*
Concrete Left-Side Subject:	*(In a statically scoped language: Dispatches the operation to the right hand object.)* The collection of all methods for a particular operation involving concrete subject a on the right hand and all other known subject types.
Concrete Right-Side Subject:	*(The same, respectively.)*

Signature: Left-side Subject responds to messages involving the entire known range of right-side Subject types. Concrete subjects implement this interface.

Used by: Visitor, efficient multi-method implementations.

Known issues: Limited to a "stable hierarchy". *(In a statically scoped language:)* creates cyclic namespace dependency.

Scope: Languages that do not support multi-methods.

www
.skilldesign
.com

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 9A - Double Dispatch Idiom

Foil 1.2.9A: To resolve multiple-object functionality among the objects

The system shall have the capability
to resolve multiple-object functionality among objects,
which involves...

Send feedback

Foil 1.2.9A - Description:

www
.skilldesign
.com

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 10 - Dynamic Pluggable Factory

Context: Creating objects by keyword, where types of objects to be created are not known in advance or are extensible during runtime.

Send feedback

Challenges:

- To separate the capability to actually create specific objects from the factory.
- To add the capability to create objects to the factory during runtime.
- To fill factory with default population as part of product type definition.

Skill: *The system shall have the capability* **to create objects by unique key**, *which further requires...*

View Wirechart

1: *ASYNC,MANY*: Client**: "to register *Product* creator"**
2: Concrete Creator Plug**: "to initialize *Concrete Creator Plug*"**
3: Factory Class**: "to obtain the *Factory*"**
4: *OPTION*: Factory**: "to initialize the *Factory*"**
5: Factory**: "to register prototype"**
6: Client**: "to obtain *Product* by unique key"**
7: Factory Class**: "to obtain the *Factory*"**
8: Factory**: "to create *Product*"**
9: *ABSTRACT*: Creator Plug**: "to create default *Product*"**
10: *EXAMPLE* by Concrete Creator Plug
11: Concrete Product**: "to initialize *Concrete Product*"**
12: Product**: "to use *Product*"**

Participants:

Factory:	A Singleton. Registers Creator Plugs. Responsible for creating products by unique key.
Creator Plug:	Plugs into factory to create products of a type by unique key. (Typically - Factory Method. Also Class Object, Prototype.)
Concrete Creator Plug:	Registers itself (or is registered) in the factory. Responsible for creating a product of a fixed type and/or content known only to itself. Typically istantiates from a template.
Product:	What the factory is declared to create.
Concrete Product:	What the factory will create for a certain key.
Client:	Obtains products from the factory, supplying key.

Signature: Factory references Creator Plugs by key and creates products for Client.

Used patterns and idioms: Factory Method. *Optionally:* Static Registration idiom.

Used by: Serialization mechanisms, visual selection mechanisms and generally: entry-points for

objects from outside program memory.

Variations:

- **Virtual constructor**. Creation method is a static method of the abstract base-class. *(Note: the virtual constructor can be implemented by means of any factory, e.g. Dynamic Pluggable Factory.) Source:* Stroustrup.
- **Product Trader** . Factory loads Creator Plug if not found. *Source:* Lauder.

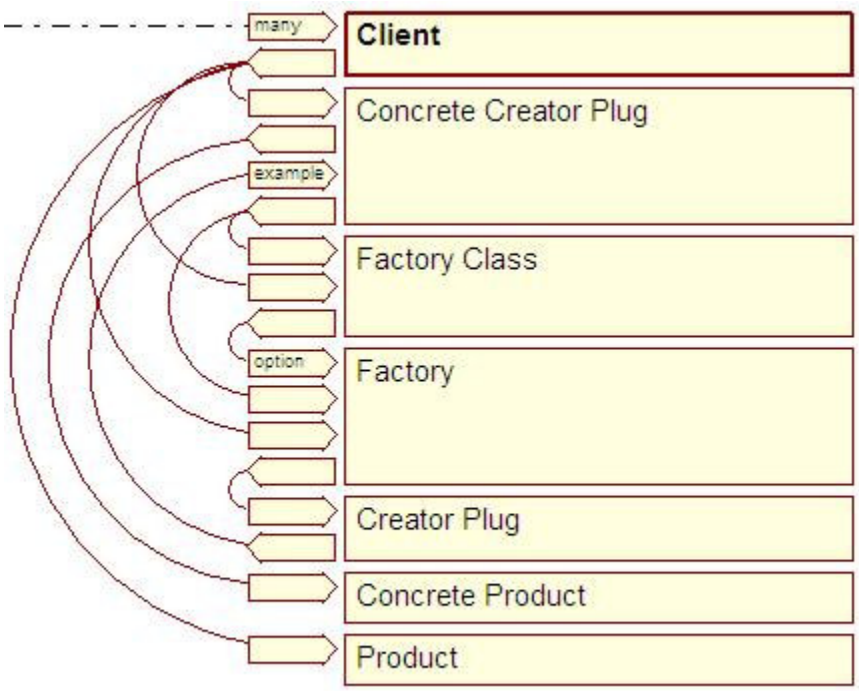
Scope: General.

Foil 10A - Dynamic Pluggable Factory

Foil 1.2.10A: To create objects by unique key

The system shall have the capability
to create objects by unique key,
which involves...

Send feedback



Foil 1.2.10A - Description:

www .skilldesign .com

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 11 - Dynamic Pluggable Factory (prototypes)

Context: Creating objects by pointing at an example in a series of objects, which may be of diverse types, or differ in content.

Send feedback

Challenges:

- To separate the capability to actually create specific objects from the factory.
- To add the capability to create objects to the factory during runtime.
- To create a true replica of the selection.

Skill: *The system shall have the capability* **to create objects by example**, *which further requires...*

View Wirechart

1: *ASYNC,MANY*: Client**: "to assign concrete prototype as creator for its type"**
2: Factory Class**: "to obtain the *Factory*"**
3: *OPTION*: Factory**: "to initialize the *Factory*"**
4: Concrete Product**: "to initialize *Product*"**
5: Factory**: "to register prototype"**
6: Client**: "to obtain *Concrete Product* by unique key"**
7: Factory Class**: "to obtain the *Factory*"**
8: Factory**: "to create *Product*"**
9: *ABSTRACT*: Product**: "to copy *Product*"**
10: *EXAMPLE* by Concrete Product

Participants:

Factory:	A Singleton. Registers Product protoypes. Responsible for creating products by position.
Product:	What the factory is declared to create. One or more prototypes for each concrete Product type are registered in the factory as creators.
Concrete Product:	What the factory will create for a certain position. Responsible for creating replicas of itself.
Client:	Obtains products from the factory, supplying key.

Signature: Factory references Product protoypes and creates products for Client.

Used patterns and idioms: Either Prototype. *Optionally:* Static Registration idiom.

Used by: Serialization mechanisms, visual selection mechanisms and generally: entry-points for objects from outside program memory.

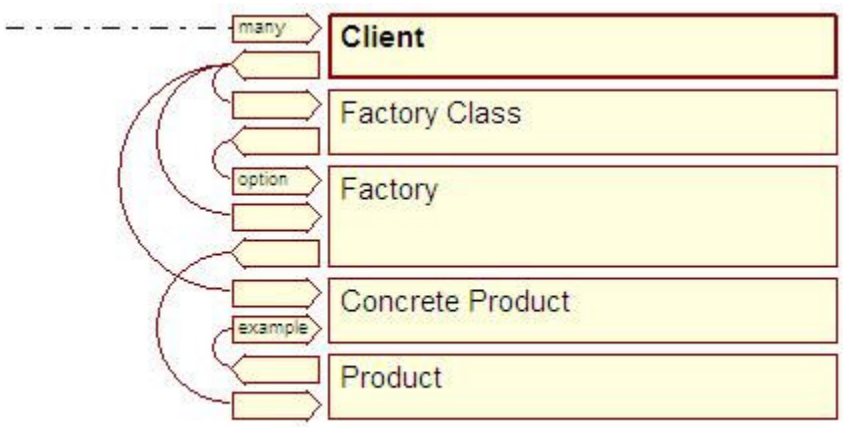
Scope: General.

Foil 11A - Dynamic Pluggable Factory (prototypes)

Foil 1.2.11A: To create objects by example

The system shall have the capability
to create objects by example,
which involves...

Send feedback



Foil 1.2.11A - Description:

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 12 - Factory Method (singleton)

www.skilldesign.com

Context: Creating an object by fixed criterion. Possibly reconfiguring it later.

Send feedback

Challenge: to create the object without specifying its type.

Skill: View Wirechart

The system shall have the capability
to construct objects by fixed type criterion,
which further requires...

1: Client**: "to obtain concrete product"**
2: Factory Method Class**: "to obtain the *Factory Method*"**
3: *OPTION*: Factory Method Class**: "to configure the *Factory Method*"**
4: Factory Method Class**: "to find the pending *Factory Method* type"**
5: Factory Method**: "to initialize the *Factory Method*"**
6: Client**: "to create product"**

Participants:

Client:	Uses Factory Method to obtain Products.
Factory Method:	The capability to create Products of pre-configured type. An efficient data-driven replacement for a switch/case statement.
Concrete Factory Method:	The capability to create a Concrete Product. There is usually one instance of the Concrete Factory Method, accessed through global Factory Method reference (typically singleton). Where the Concrete Factory Method serves no other function, (and where genericity is supported) it can be parameterized over the type of Concrete Product that it creates.
Product:	The type of Product *declared* to be created.
Concrete Product:	The Product actually created, as configured via the current Concrete Factory Method.

Signature: Factory Method (typically Singleton) creates Products for Client. Each Concrete Factory Method is built to produce the respective Concrete Product. In the extreme case, Concrete Factory Method is parameterized over Concrete Product.

Used patterns and idioms: Singleton.

Used by: Dynamic Pluggable Factory.

Scope: Languages that that lack the class object. *(Elsewhere, where objects are created by default, a global reference to the respective class object will suffice.)*

www .skilldesign .com

A Design Pattern Dictionary

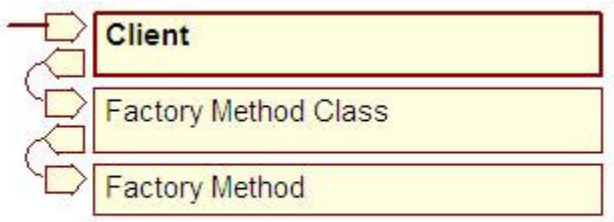
Section 2 - The Dictionary

Foil 12A - Factory Method (singleton)

Foil 1.2.12A: To construct objects by fixed type criterion

The system shall have the capability
to construct objects by fixed type criterion,
which involves...

Send feedback



Foil 1.2.12A - Description:

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 13 - Flyweight Pool (reference count)

www.skilldesign.com

Context: Objects are heavyweight or encapsulate system-critical resources and are read only.

Send feedback

Challenge: To prevent resource duplication, system wide.

Skill:

View Wirechart

1: **"to prevent redundant creation of global *Resources*"**
2: Smart Resource Pointer**: "to ihitialize *Smart Resource Pointer*"**
3: *OPTION*: The Flyweight Pool**: "to create *Resource*"**
4: Smart Pointer Counter**: "to increase ownership"**
5: Smart Resource Pointer**: "to finalize *Smart Resource Pointer*"**
6: Smart Pointer Counter**: "to decrease ownership"**
7: The Flyweight Pool**: "to cease managing *Resource*"**
8: Resource**: "to finalize *Resource*"**

Participants:

Flyweight:	The sole supplier of Resources. A singleton. Responsible for creating Resources, managing their lifespan and guaranteeing their uniqueness.
Client:	Uses Resources obtained from the Flyweight.
Smart Resource Pointer:	A go-between, obtained from the flyweight to control Resource lifespan. A Reference-Counting Smart Pointer.
Smart Pointer Counter:	Stored in the Flyweight. Accessed through a Smart Resource Pointer. Responsible for alerting the Flyweight to initiate Resource (i.e. its own) disposal.
Resource:	Stored inside the Flyweight. Obtained exclusively through the Flyweight. Required to allow creation using unique key.

Signature: Flyweight contains Smart Pointer Counters by key, creates Resources (which it keeps via the Smart Pointer Counters) as well as Smart Pointers (which it does not keep) that share a Smart Pointer Counter. Smart Pointer Counter contains a Resource and notifies the Flyweight. Client uses Smart Resource Pointers, obtained from the Flyweight.

Notes: The "RC Smart pointer" may look at first like a viable alternative. However, it prevents duplication inside copy-conceived groups, rather than globally as required here.

Used patterns and idioms: Reference-Counting Smart Pointer, Singleton.

Variations: Invisible Flyweight Pool, Caching Flyweight Pool.

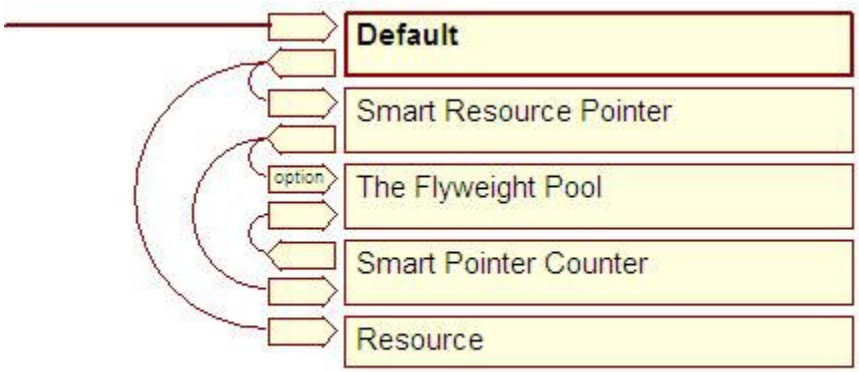
Used by: Database replicators and similar infrastructure, typical of client/server architectures .

Source: GOF (without the interface of Smart-Pointer and without reference counting).

Scope: General.

Foil 13A - Flyweight Pool (reference count)

Foil 1.2.13A: To prevent redundant creation of global resources



Foil 1.2.13A - Description:

Send feedback

www
.skilldesign
.com

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 14 - Invisible Flyweight Pool (reference count)

Context: Send feedback

Objects of a type may be heavyweight but tend to have equal values and are read-only all or most of the time *(e.g. character string)*.

Challenge:

To prevent resource duplication system wide, hiding the existence of the Flyweight Pool.

Skill: View Wirechart

The system shall have the capability
to globalize the implementation of local Resources with shareable values, *which further requires...*

1: Resource Proxy**: "to initialize *Resource Proxy*"**
2: *OPTION*: Flyweight Pool**: "to create *Resource*"**
3: Resource Reference Counter**: "to increase ownership"**
4: Smart Resource Proxy**: "to detach from *Resource*"**
5: Smart Pointer Counter**: "to decrease ownership"**
6: Flyweight Pool**: "to cease managing *Resource*"**
7: Resource**: "to finalize *Resource*"**

Participants:

Flyweight:	The (hidden) sole source for Resources. A singleton. Responsible for creating resources, managing their lifespan and guaranteeing their uniqueness. Not exposed to the Client.
Resource Proxy:	A transparent resource wrapper used directly by the client (thus, effectively hiding the existence of the flyweight). A reference-counting proxy.
Resource Reference Counter:	A resource wrapper obtained by the Resource Proxy from the flyweight to control Resource lifespan. Responsible for alerting the flyweight to initiate its own disposal.
Resource:	stored inside the flyweight. Accessed through a Resource Proxy.

Signature:

Flyweight contains Smart Pointer Counters by key and creates Resources. Resource Proxies share Smart Pointer Counter, obtained from the Flyweight. Smart Pointer Counter contains Resource and notifies the flyweight.

Used patterns and idioms:

Singleton, Reference-counting proxy.

Used by:

Character string and similar widely used low-level types with significant duplication overhead.

Scope:

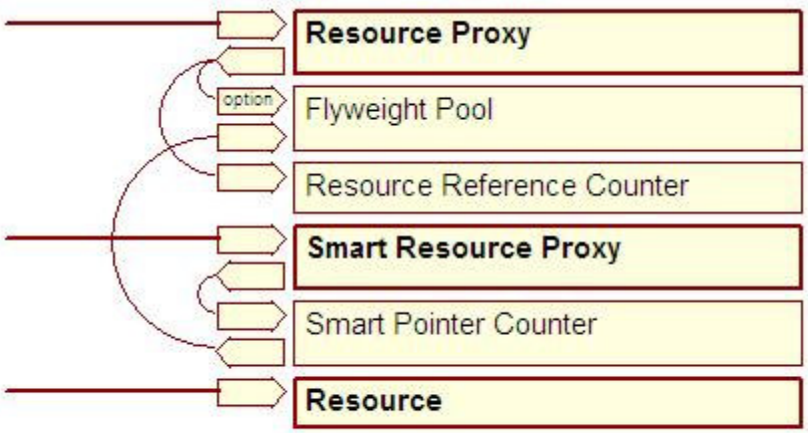
General (system programming).

Foil 14A - Invisible Flyweight Pool (reference count)

Foil 1.2.14A: To globalize the implementation of shareable local Resources

The system shall have the capability
to globalize the implementation of local Resources with shareable values,
which involves...

Send feedback



Foil 1.2.14A - Description:

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 15 - Lazy Copy idiom (copy on write)

www.skilldesign.com

Context: Shared aggregation of read/write resource.

Send feedback

Challenge: To allow changing the resource without affecting its other users.

Skill: *(Does not involve significant functionality.)*

View Wirechart

Participants:

Resource Context:	References Resource, possibly interfacing for it. Responsible, in addition to initiating reference counting, for lazy-copying of the resource.
Resource:	A passive participant. It's only responsibility, in addition to initialization, is to allow (deep) copying.

Signature: Resource Contexts reference Resource.

Used by: Reference Counting Proxy, Invisible Flyweight Pool.

Scope: General (system programming).

www.skilldesign.com

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 15A - Lazy Copy idiom (copy on write)

Foil 1.2.15A: To defer the duplication of temporarily shared resource

The system shall have the capability
to defer the duplication of temporarily shared resource,
which involves...

Send feedback

Foil 1.2.15A - Description:

www .skilldesign .com

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 16 - Metamorphic Bridge

•

Context: "Dynamic classification": The implementation of object functionality, possibly related to internal representation (e.g. number internal representation) is determined during object creation time and may change later due to assignment.

Send feedback

Challenge: To separate the interface from implementation, possibly allowing the replacement of the latter.

Skill: *The system shall have the capability* **to configure internal representation automatically**, *which further requires...*

View Wirechart

1: Context**: "to initialize *Context*"**
2: Context**: "to configure *Implementation*"**
3: Context**: "to perform action in *Context*"**
4: Implementation**: "to implement action"**
5: *OPTION*: Context**: "to Replace *Implementation*"**

Participants:

Context:	Used by Client, typically non-polymorphically. Responsible for constructing the appropriate Implementation (normally, understood from argument) and replacing it (typically, due to assignment). *(E.g. Number, hiding internal number representation, constructed according to numeric-constant argument.)*
Implementation:	Internal representation, hidden inside Context. The capability to do the actual work in Context. *(E.g. to execute arithmetic operations in Number.)*
Concrete Implementation:	Constructed by Context. *(E.g. Integer or Real in Number.)* Responsible for doing the actual work in Context.

Signature: Context contains Implementation and represents it.

Used patterns and idioms: *(Optionally)* Double Dispatch idiom, *(optionally)* Algebraic Hierarchy.

Comment: A simplified form of the GOF pattern by the same name (lacking the Concrete Behavior).

Source: Coplien.

Scope: General.

www .skilldesign .com

A Design Pattern Dictionary

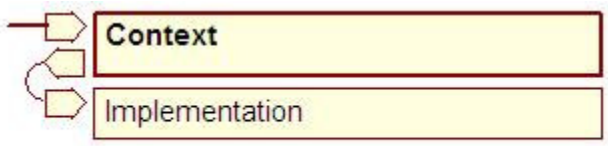
Section 2 - The Dictionary

Foil 16A - Metamorphic Bridge

Foil 1.2.16A: To configure internal representation automatically

The system shall have the capability
to configure internal representation automatically,
which involves...

Send feedback



Foil 1.2.16A - Description:

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 17 - Observer

www.skilldesign.com

Context: "Controlled data redundancy": The state of one object must reflect the current state of another object (e.g. document and its views, server and its clients, the result of a formula and its values in a spreadsheet).

Send feedback

Challenge: To keep the dependent object up-to-date at minimum cost. (The alternative of polling the data source by its observers is both expensive and intrusive).

Skill: *The system shall have the capability* **to synchronize state change**, *which further requires...*

View Wirechart

1: Observer**: "to prepare for *Subject* change"**
2: Observer Delegate**: "to initialize *Observer Delegate*"**
3: Subject**: "to register *Observer Delegate*"**
4: *ASYNC,MANY*: Subject**: "to change the *Subject* state"**
5: Observer Delegate**: "to notify *Observer*"**
6: *OPTION*: Observer**: "to update *Observer* state"**

Participants:

Observer:	An object that reflects the state of some resource, e.g. a view. Responsible for registering a notifier over itself for modification by that resource. Knows how to respond to change notification.
Subject:	A resource whose state is reflected by other objects. Responsible for notifying its observers when its state changes.
Observer Delegate:	A command registered by an observer at its subject. Buffers subject from observer. Allows the observer to be of any type. also: **Notifier.**

Signature: Subject references notifiers, each referencing an observer.

Used patterns and idioms: Command.

Used by: Document/view architectures.

Source: Java.

Scope: General.

www .skilldesign .com

A Design Pattern Dictionary

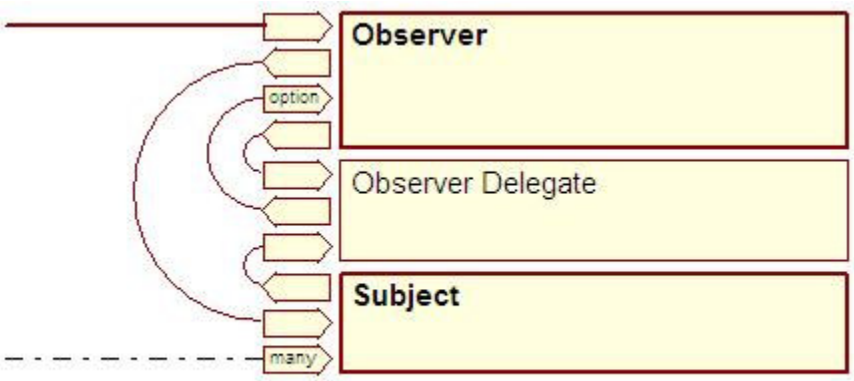
Section 2 - The Dictionary

Foil 17A - Observer

Foil 1.2.17A: To synchronize state change

The system shall have the capability
to synchronize state change,
which involves...

Send feedback



Foil 1.2.17A - Description:

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 18 - Observer (inheritance)

www.skilldesign.com

•

Context: *See Observer.*
Send feedback

Challenge: *See Observer.*

Skill: *The system shall have the capability*
View Wirechart **to synchronize state change**, *which further requires...*

1: Observer**: "to prepare for *Subject* change"**
2: Subject**: "to register *Observer*"**
3: *ASYNC*: Subject**: "to change the *Subject* state"**
4: Subject**: "to notifier *Observer*"**
5: *OPTION*: Observer**: "to update *Observer* state"**

Participants:

Observer:	An object that reflects the state of some Resource, *(e.g. view over documents)*. Responsible for registering itself for modification by that Resource.
Subject:	A resource whose state is reflected by other objects. Responsible for notifying its observers when it has suffered change.
Concrete Observer:	An object that implements Observer skills. Knows how to respond to change notification.
Concrete Subject:	An object that implements Subject skills. Knows when to notifie Observers of its own change.

Signature: Subject references Observers. Often, Concrete Observer contains or references Concrete Subjects (e.g. view and document).

Used patterns and idioms: (Variation on) Rectangular Inheritance of Association.

Source: GOF.

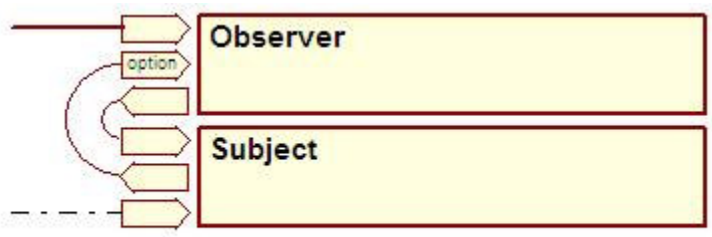
Scope: General.

Foil 18A - Observer (inheritance)

Foil 1.2.18A: To synchronize state change

The system shall have the capability
to synchronize state change,
which involves...

Send feedback



Foil 1.2.18A - Description:

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 19 - Order of Initialization

www.skilldesign.com

Context: Application framework is responsible for initializing some resources and make them globally available (e.g. to each other), but does not - itself - use them.

Send feedback

Challenge: To make each resource ready on time (without applying the singleton pattern).

Skill: *The system shall have the capability* **to prioritize the initialization of global Resources**, *which further requires...*

View Wirechart

1: *ASYNC,MANY*: Resource**: "to initialize global object"**
2: Resource Initializer**: "to initialize global *Resource*"**
3: Initialization Manager**: "to register global *Resource* initialization request"**
4: Application Framework**: "to initialize the system"**
5: Initialization Manager**: "to conclude initialization of global *Resources*"**
6: *MANY*: Initialization Manager**: "to make global *Resource* conclude initialization"**
7: Resource**: "to conclude global *Resource* initialization"**

Participants:

Application Framework:	Responsible for driving the Initialization Manager.
Initialization Manager:	A singleton. Registers prioritized initialization commands. Responsible for activating the initialization commands in the correct order.
Resource:	The target of the initialization. Its constructor leaves it in undefined state. Optionally, responsible for launching the initialization command during its own (empty) initialization.
Resource initializer:	A command, waiting at the initialization manager, in order of priority. Responsible for waking the resource to finish it initialization process.

Signature: *Initialization Manager* references Resource Initializers, each referencing a Resource. Resource typically creates a Resource Initializer which typically registers itself at the Initialization Manager. The application-Framework uses the (singleton) Initialization Manager.

Used patterns and idioms: Command, Singleton.

Used by: generalized application frameworks.

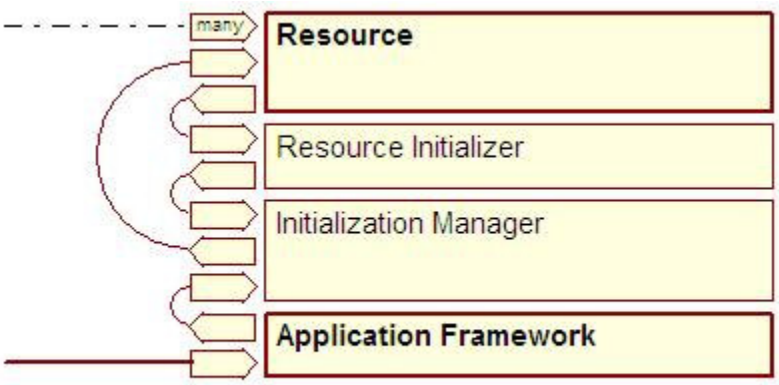
Scope: General (infrastructure).

Foil 19A - Order of Initialization

Foil 1.2.19A: To prioritize the initialization of global resources

The system shall have the capability
to prioritize the initialization of global Resources,
which involves...

Send feedback



Foil 1.2.19A - Description:

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 20 - Policy (Traits)

www.skilldesign.com

Context: The implementation of part of object behavior varies from object to object, but is known before each object's creation and is never re-configured for it. A fixed strategy.

Send feedback

Challenge:

- To integrate the strategy type with the client type.
- To eliminate the cost of run-time polymorphism in the communication between them. (E.g. formatting, localization, comparison).

Skill:

The system shall have the capability
to parameterize object behavior on partial functionality,
which further requires...

View Wirechart

1: Context**: "to initialize *Context*"**
2: Policy**: "to initialize *Policy*"**
3: Context**: "to perform partially-configurable action"**
4: Context**: "to perform configurable part of action"**

Participants:

Context:	Does part of its job by consulting an internal Policy.
Policy:	A collection of methods on one facet of Context implementation, used in Context. Policy methods are typically short and inline.

Signature: Context is parameterized over Policy type. Context contains Policy.

Used by: Localization, configurable low-level types.

Source: C++, Alexandrescu.

Scope: Languages that support genericity.

www .skilldesign .com

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 20A - Policy (Traits)

Foil 1.2.20A: To parameterize object behavior on partial functionality

The system shall have the capability
to parameterize object behavior on partial functionality,
which involves...

Send feedback

Foil 1.2.20A - Description:

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 21 - Prototype

www.skilldesign.com

Context:
Send feedback

- Selecting objects by visual example.
- Creating objects by content (rather than type).
- Copying heterogeneous collections.

Challenge: To copy each object without having to tell its type.

Skill:
View Wirechart

The system shall have the capability
to create objects by example,
which further requires...

1: Client**: "to prepare example set"**
2: *MANY*: Concrete Prototype**: "to initialize *Concrete Prototype*"**
3: *MANY*: Client**: "to create object by example"**
4: Prototype**: "to copy example"**

Participants:

Prototype:	represents the capability to copy any object of a type family.
Concrete Prototype:	knows how to copy itself.

Signature: prototype responds to *clone* message, returning base object.

Used by: Dynamic Pluggable Factory, selection bars, deep copy of heterogeneous collections.

Scope: General. *(Preferably, languages that support deep-copying by default.)*

www .skilldesign .com

A Design Pattern Dictionary

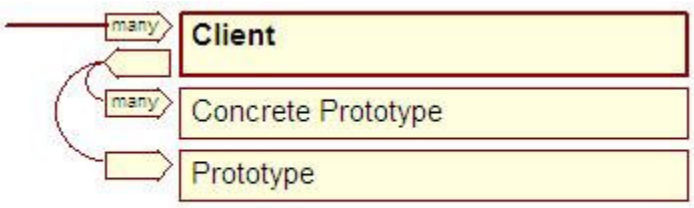
Section 2 - The Dictionary

Foil 21A - Prototype

Foil 1.2.21A: To create objects by example

The system shall have the capability
to create objects by example,
which involves...

Send feedback



Foil 1.2.21A - Description:

www .skilldesign .com

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 22 - Rectangular Inheritance of Association Idiom

Context: Send feedback

Objects of a type family are covariantlly associated one-to-many with objects of a parallel type family (e.g. editors and edited documents).

Challenge:

to encapsulate the fact of association and its management in the abstract base class, still, allowing a derived-class to limit the contained population safely.

Skill: View Wirechart

The system shall have the capability
to encapsulate heterogeneous collection management,
which further requires...

1: Concrete Container**: "to use *Resource* in *Concrete Container*"**
2: Container**: "to get *Resources* in *Container*"**
3: Resource**: "to use *Resource*"**
4: Concrete Container**: "to use *Concrete Resource* in *Container*"**
5: Concrete Container**: "to downcast *Resource*"**
6: Concrete Resource**: "to use *Concrete Resource*"**

Participants:

Container:	Contains the heterogeneous collection and features the generic access sub-interface.
Concrete Container:	Uses the heterogeneous collection in part of its capacity. Responsible for performing significant operations on the objects inside, depending on their true type.
Resource:	The declared content of the collection. A passive participant. Optionally, features a "fat" interface.
Concrete Resource:	The actual content of the collection, in a concrete container. The Concrete Container optionally downcasts each Resource into a Concrete Resource before applying particular operations.

Signature:

Container contains (or references) Resources. Concrete Container specializes in Concrete Resources (but does not contain them directly).

Used patterns and idioms:

Optionally - Template Method.

Used by:

Composite pattern, observer (GOF), general-purpose editors.

Scope:

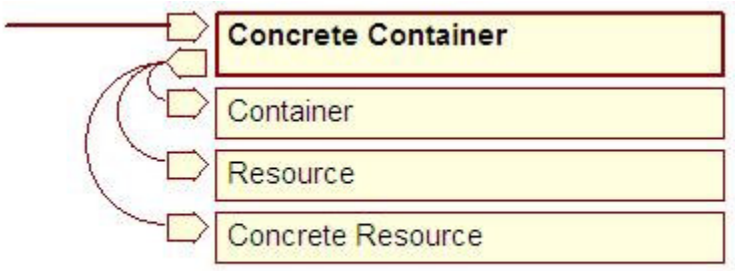
General.

Foil 22A - Rectangular Inheritance of Association Idiom

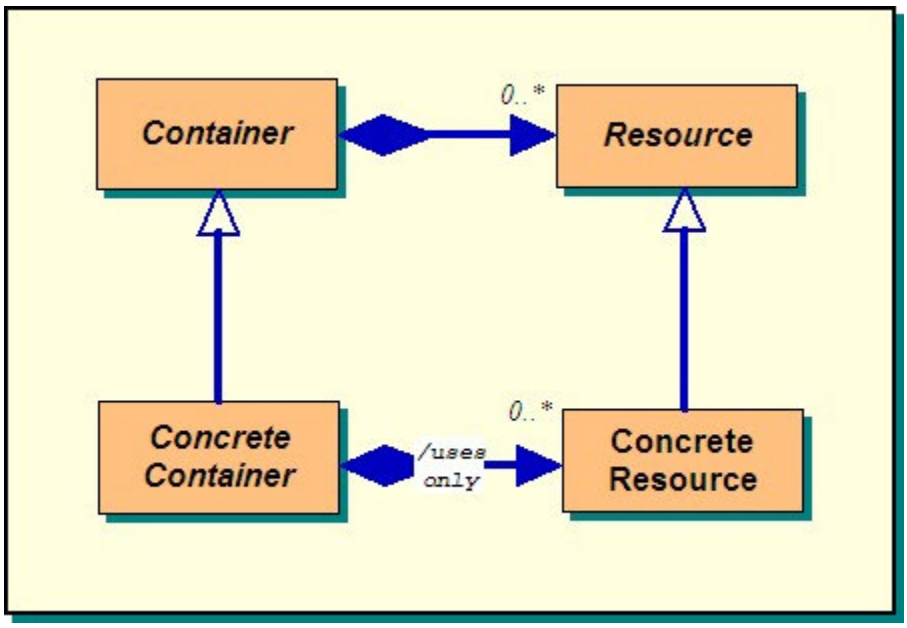
Foil 1.2.22A: To encapsulate heterogeneous collection management

Send feedback

The system shall have the capability
to encapsulate heterogeneous collection management,
which involves...



Object model



Foil 23 - Recursive Object-Structure Idiom

www.skilldesign.com

Context: A hierarchical object structure of variable depth.

Send feedback

Challenge: To express the hierarchy using object-structure.

Skill: *(Does not involve significant functionality.)*

View Wirechart

Participants:

Subtree:	A node that contains other nodes.
Leaf:	A node that does not contain other nodes.
Node:	A node in the tree.

Signature: Subtrees contain (or reference) Nodes, where each may be either a Leaf or itself a Subtree (thus, giving a tree structure).

Comments: The Node is a flat interface. In case of an abstract base class it should be devoid of implementation. A binary version (where subtree is a concrete class) is possible but error-prone. A unary version (i.e. simple reflexive association) is possible where all Nodes feature the same behavior, with their tree position reflecting a changeable status, rather than permanent role).

Used by: Decorator (1:1), Composite (1:n).

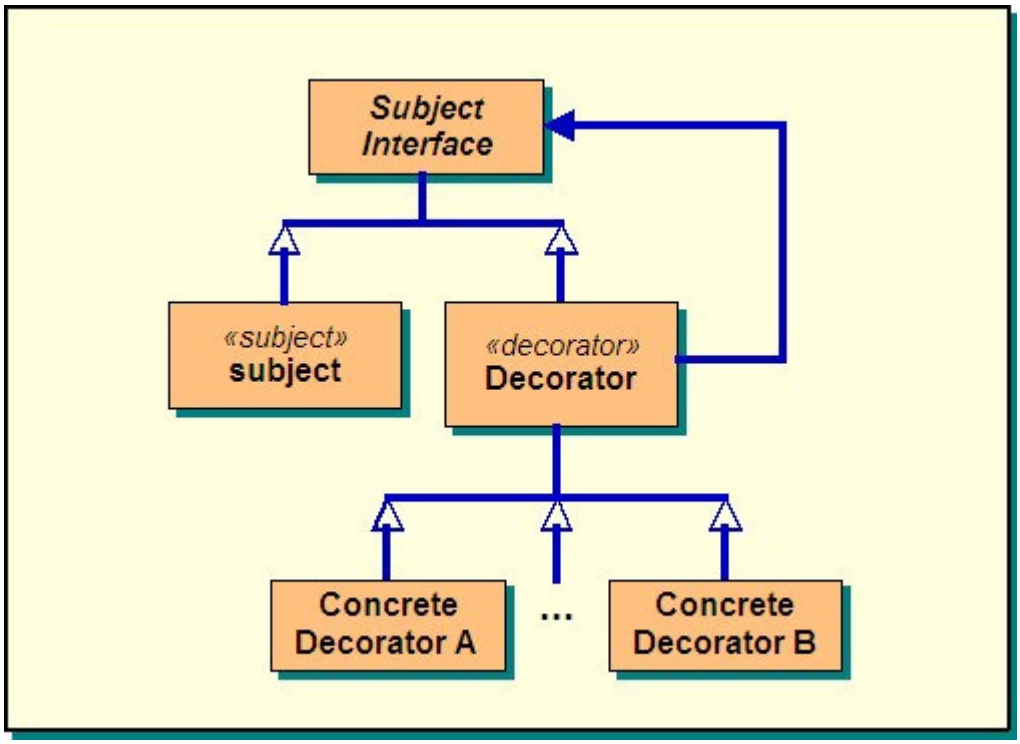
Scope: General.

Section 2 - The Dictionary

Foil 23A - Recursive Object-Structure Idiom

Foil 1.2.23A: To compose a tree structure

Send feedback



Foil 1.2.23A - Description:

www
.skilldesign
.com

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 24 - Reference-counting idiom

Context: Some cases of shared aggregation - n:1 association with shared ownership. An object is considered as member (with the obvious restrictions) by one or more other objects at once.

Send feedback

Challenge: To eliminate the shared resource exactly when no longer in use.

Skill: *The system shall have the capability*
to time the disposal of a shared Resource,
which further requires...

View Wirechart

1: Client**: "to attach to *Resource*"**
2: Resource**: "to increase ownership"**
3: Client**: "to detach from *Resource*"**
4: Client**: "to decrease ownership"**
5: *OPTION*: Resource**: "to dispose of the *Resource*"**

Participants:

Client:	Shares ownership over the resource. Responsible for notifying the Resource about the start and end of the association.
Resource:	Manages the reference count. Responsible for actually deleting the Resource (i.e. itself).

Signature: *Inside an existing n:1 hierarchy:* Reference Count datum and interface added to Resource. Registration code added to Client.

Used by: Reference-Counting Smart Pointer, Reference-Counting Proxy.

Scope: Languages that do not support automatic garbage collection. (General:) control over a partial population.

www .skilldesign .com

A Design Pattern Dictionary

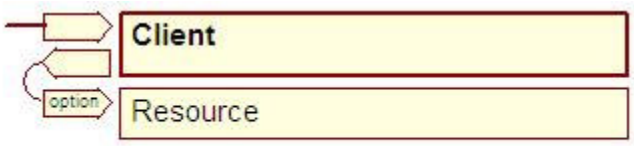
Section 2 - The Dictionary

Foil 24A - Reference-counting idiom

Foil 1.2.24A: To time the disposal of a shared resource

The system shall have the capability
to time the disposal of a shared Resource,
which involves...

Send feedback



Foil 1.2.24A - Description:

www
.skilldesign
.com

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 25 - Reference-Counting Proxy ("Counted Body")

Context:
Send feedback

1. Heavyweight resources that are referenced by different clients, typically read-only.
2. Heavyweight objects that are passed and returned by value (in languages that support value addressing).

Challenge: To avoid redundant storage, as well as its copying and disposal overhead.

Skill:
View Wirechart

The system shall have the capability
to share hidden Resource implementation,
which further requires...

1: Reference Counting Proxy**: "to obtain *Resource*"**
2: *ALTERNATIVES*:
3: *OPTION*: Reference Counting Proxy**: "to initialize *Reference Counting Proxy*"**
4: Resource**: "to increase ownership"**
5: *OPTION*: Reference Counting Proxy**: "to initialize copy of *Reference Counting Proxy*"**
6: Resource**: "to increase ownership"**
7: *OPTION suggested*: Reference Counting Proxy**: "to change *Resource*"**
8: *OPTION shared*: Reference Counting Proxy**: "to lazy-copy *Resource*"**
9: Reference Counting Proxy**: "to finalize *Reference Counting Proxy*"**
10: Resource**: "to decrease ownership"**
11: *OPTION*: Resource**: "to finalize *Resource*"**
12: Reference Counting Proxy**: "to copy *Resource*"**
13: Reference Counting Proxy**: "to finalize *Reference Counting Proxy*"**

Participants:

Resource Proxy :	Represents the resource implementation (for its Clients). Responsible for creating the Resource Implementation and notifying it of the duration of the link.
Resource Implementation :	Contains the data and knows the real methods over it. Responsible for actually deleting the resource (i.e. itself).

Signature: Resource Proxies share a Resource Implementation and represent it.

Used patterns and idioms: Reference-counting idiom, *optionally:* Lazy Copy.

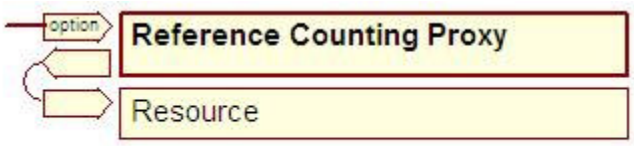
Used by: Flyweight-aided character string.

Scope: General (system programming).

Foil 25A - Reference-Counting Proxy ("Counted Body")

Foil 1.2.25A: To share hidden resource implementation

The system shall have the capability
to share hidden Resource implementation,
which involves...



Foil 1.2.25A - Description:

www
.skilldesign
.com

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 26 - Reference-Counting Smart Pointer

Context: *See Reference Counting idiom.*

Send feedback

Challenges:

- To *encapsulate* the deletion timing mechanism.
- To separate the association and its management code from both Client and Resource.
- *(In strong-typed languages that support genericity:)* To separate association management code from Resource type.
- *(See also Reference Counting Idiom.)*

Skill:

View Wirechart

The system shall have the capability
to control shared Resource lifespan automatically,
which further requires...

1: Smart Resource Pointer**: "to initialize *Smart Resource Pointer*"**
2: Smart Pointer Counter**: "to increase ownership"**
3: Smart Resource Pointer**: "to finalize *Smart Resource Pointer*"**
4: Smart Pointer Counter**: "to decrease ownership"**
5: *OPTION*: Resource**: "to finalize *Resource*"**

Participants:

Smart Pointer:	Contained by the Client. Represents pointer to Resource (by overloading the dereferencing operator).
Resource:	The object referenced (type insignificant). Its sole responsibility here is to allow deletion.
Smart-Pointer Counter:	Referenced by Smart Pointer. Manages the reference count. Responsible for actually deleting the Resource.

Signature: Smart Pointers share Smart Pointer Counter, which contains the Resource. Smart Pointer and Smart Pointer Counter are parameterized on Resource. (Smart Pointer Counter's declaration may nest in Smart Pointer). Nothing is required of the Resource (besides enabling deletion). Client contains Smart Pointer (rather than Resource).

Used patterns and idioms: Reference Counting idiom.

Scope: Languages that do not support automatic garbage collection and support value addressing.

www .skilldesign .com

A Design Pattern Dictionary

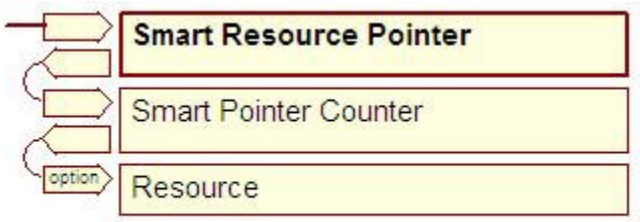
Section 2 - The Dictionary

Foil 26A - Reference-Counting Smart Pointer

Foil 1.2.26A: To control shared Resource lifespan automatically

The system shall have the capability
to control shared Resource lifespan automatically,
which involves...

Send feedback



Foil 1.2.26A - Description:

www
.skilldesign
.com

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 27 - Singleton

Context:
Send feedback

- A software module (in languages that do not support modularity).
- A generally available part of the programming infrastructure whose existence is taken for granted by the programmers who use it.
- A non-procedural flow of control implicit in the construction of infrastructure components prior to the main program.

Challenge:

- To guarantee the existence of only one singleton object.
- To complete the construction of the singleton instance prior to its first use.

Skill:
View Wirechart

The system shall have the capability
to encapsulate the existence of a globally-available resource,
which further requires...

1: Singleton Class**: "to access *The Singleton* instance"**
2: *OPTION*: The Singleton**: "to initialize *The Singleton* instance"**

Participants:

Client:	Uses the singleton object.
Singleton class:	Allows access to the singleton instance. Responsible for creating it by default.

Signature: Added static instance method, static singleton instance, private constructor and possibly destructor.

Known issues:

1. If never used, the singleton constructor will never be invoked (which is not normally a reason for concern).
2. *Automatic disposal of the Singleton object.* Where automatic garbage collection is not supported, the Singleton destructor will never be invoked. This is remedied (e.g. Meyers) by declaring the singleton object as a static variable in the instance method. The singleton destructor is invoked in the opposite order to that of the singleton constructor. It is the recommended to check that the singleton is not invoked after already being destroyed.
3. *Synchronizing Singleton access*. In a multi-threaded application, two simultaneous accesses to the singleton may result in two instances of the singleton, one being useless. To prevent this, either the entire instance method or only in the first time allocation block are to be synchronized. The latter is double-checked as precaution.

Used by: Factories, flyweights, dispatchers and similar globally available resource-management and synchronization mechanisms.

Sources: GOF, Meyers.

Scope: Languages that do not support modularity.

www .skilldesign .com

A Design Pattern Dictionary

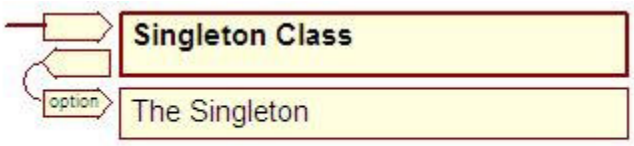
Section 2 - The Dictionary

Foil 27A - Singleton

Foil 1.2.27A: To encapsulate a globally-available resource

The system shall have the capability
to encapsulate the existence of a globally-available resource,
which involves...

Send feedback



Foil 1.2.27A - Description:

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 28 - State

www.skilldesign.com

Context: "Dynamic classification": Object that receives a small number of messages, but the entire set of methods it uses to respond to them depends upon its current state.

Send feedback

Challenge:

- To encapsulate state management.
- To virtually replace an object's effective virtual table.

Skill: *The system shall have the capability*
to switch among alternative implementations of entire functionality,
which further requires...

View Wirechart

1: State Machine**: "to initialize *State Machine*"**
2: *MANY*: Concrete State**: "to initialize *Concrete State*"**
3: State Machine**: "to set startup *State*"**
4: *MANY*: State Machine**: "to respond to event"**
5: State**: "to perform action for event in *State*"**
6: *OPTION*: State Machine**: "to change *State*"**

Participants:

Finite State Machine (FSM):	Container of states, representing the current one. Responsible for state switching. May contain common data.
State:	The current FSM contents. The capability to actually respond to messages. Possibly, the default for some of them (e.g. to shift to next state).
Concrete State:	Encapsulates the methods for the FSM interface and possibly specific data

Signature: FSM contains States, references the current State and interfaces for it.

Scope: General.

Foil 28A - State

Foil 1.2.28A: To switch among alternative implementations of an entire functionality

The system shall have the capability
to switch among alternative implementations of entire functionality,
which involves...

Foil 1.2.28A - Description:

Send feedback

www .skilldesign .com

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 29 - Startup Registration idiom

Context: Some population in a Singleton registry (e.g. Dynamic Pluggable Factory) represents the application default .

Send feedback

Challenge: To make the default population register automatically.

Skill: *The system shall have the capability*
to let the default object population be registered spontaneously,
which further requires...

View Wirechart

1: *ASYNC,MANY*: Concrete Citizen**: "to initialize *Concrete Citizen*"**
2: *OPTION*: Registry**: "to register *Citizen*"**
3: *ASYNC,MANY*: Client**: "to request citizen-related action"**
4: *ABSTRACT*: Citizen**: "to perform citizen-related action"**
5: *EXAMPLE* by Concrete Citizen

Participants:

registry:	A Singleton. Registers Citizens, possibly by event type.
Citizen:	The declared contents of the register.
Concrete Citizen:	Registered at the registry as part of the definition of its own or a parallel type.

Signature: *(In modular languages:)* Registration code follows the Concrete Citizen type definition. *(In conventionally-linked languages:)* Global Concrete Citizen objects register themselves (in their own constructor).

Used by: Global registration mechanisms, e.g. Dynamic Pluggable Factory.

Scope: Languages that support either modularity or global objects.

Foil 29A - Startup Registration idiom

Foil 1.2.29A: To let the default object population be registered *spontaneously*

Send feedback

The system shall have the capability
to let the default object population be registered spontaneously, *which involves...*

Foil 1.2.29A - Description:

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 30 - Strategy

www.skilldesign.com

Context: "Dynamic classification": The implementation of a discrete object behavior (typically, less than a method) is determined during its creation and may be reconfigured later *(e.g. formatting, in a configurable environment).*

Send feedback

Challenge: To encapsulate the specific behavior. To make it easily replaceable.

Skill: *The system shall have the capability* **to reconfigure discrete functionality**, *which further requires...*

View Wirechart

1: Context**: "to initialize *Context*"**
2: Context**: "to configure *Strategy*"**
3: Concrete Strategy**: "to initialize *Concrete Strategy*"**
4: Context**: "to perform partially-configurable action"**
5: *ABSTRACT*: Strategy**: "to perform configurable part of action"**
6: *EXAMPLE* by Concrete Strategy

Participants:

Context:	Does part of its job by consulting an internal Strategy.
Strategy:	A procedure object, used in Context. Typically, does not contain data and receives all the arguments it needs when invoked (a fact that accounts for its expandability.) Strategies without data are often implemented as global variables, to be referenced by name.

Signature: Context references global Strategy. Alternatively, Context contains Strategy (which contains data).

Source: GOF.

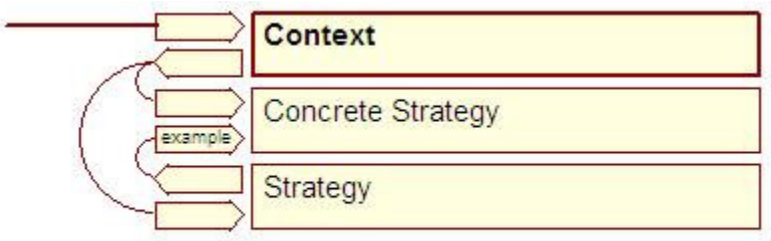
Scope: General. (In dynamically-typed languages, instance method override may do.)

Foil 30A - Strategy

Foil 1.2.30A: To reconfigure discrete functionality

The system shall have the capability
to reconfigure discrete functionality,
which involves...

Send feedback



Foil 1.2.30A - Description:

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 31 - Template Method

www .skilldesign .com

Context: "inheritance of process": A type family shares a sequential process with one or more stages being type-specific.
Send feedback

Challenge: To avoid repetition of the entire process in all implementations.

Skill: View Wirechart

The system shall have the capability
to outline and guarantee the execution of a generic process,
which further requires...

1: Processor**: "to execute a generic process"**
2: *ABSTRACT*: Processor**: "to perform a particular stage"**
3: *EXAMPLE* by Concrete Processor

Participants:

Processor:	Responsible for outlining a complete generic process, executing part of it, and coordinating the rest.
Concrete Processor:	*(for the current purpose:)* Implements the missing stages.

Signature: Processor features the template method (which should not be virtual) as well as one or more "primitives" - virtual (possibly abstract) functions, typically private. Concrete Processor implements the primitives.

Used by: *(Optionally:)* Rectangular Inheritance of Association idiom.

Known issues:

1. The pattern does not apply to constructor and destructor (due to inverted dependency - the state of the Concrete Implementation is undefined at these stages).
2. The pattern does not apply to recursive processes *(e.g. serialization).* These cases rather invite explicit supercall (i.e. normal - rather than inverted - control-flow).

Scope: General.

www .skilldesign .com

A Design Pattern Dictionary

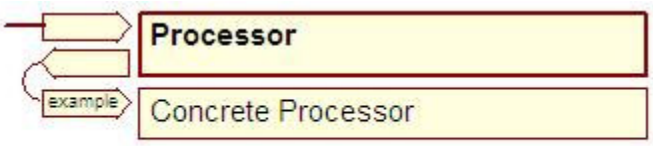
Section 2 - The Dictionary

Foil 31A - Template Method

Foil 1.2.31A: To outline and guarantee the execution of a generic process

Send feedback

The system shall have the capability
to outline and guarantee the execution of a generic process,
which involves...



Foil 1.2.31A - Description:

A Design Pattern Dictionary

Section 2 - The Dictionary

Foil 32 - Visitor

www.skilldesign.com

Context: Some polymorphic processing is configurable and, therefore, may not be part of the processed type family (e.g. formatting, user interface). Processing depends as much on the processor type as on the processed type and must be done by the processor.

Send feedback

Challenge: To add the virtual function hierarchy to a class hierarchy without opening it.

Skill: View Wirechart

The system shall have the capability
to process type family from the outside,
which further requires...

1: Context**: "to perform configurable operation over heterogeneous collection"**
2: Concrete Visitor**: "to initialize *Concrete Visitor*"**
3: *MANY*: Concrete Visitor**: "to process *Subject*"**
4: *ABSTRACT*: Subject**: "to forward to processing *Subject*"**
5: *EXAMPLE* by Concrete Subject
6: *ABSTRACT*: Visitor**: "to process *Concrete Subject*"**
7: *EXAMPLE* by Concrete Visitor

Participants:

Visitor:	The capability to process the types in a Subject family.
Concrete Visitor:	One implementation of the capability to process a type family, adding a discrete functionality (e.g. formatting, user interface).
Subject:	The capability to forward the visitor to process subject by type.
Concrete Subject:	Implementation of the capability to forward the visitor to process a concrete subject by type.

Signature: Visitor processes Subject by requesting Subject to "accept" it (the Visitor). Concrete Subject, requests Visitor to "visit" it (the Concrete Subject).

Used patterns and idioms: Double-Dispatch idiom.

Used by: Document/View architectures, dynamically configurable reporting/rendering systems.

Variations:

- **Acyclic visitor** . Prevent cyclic namespace dependency by making Visitor a "straw man", separating each Concrete visit-method to pseudo-concrete straw man, assembling the Visitor via multiple inheritance. Accept downcasts. This allows convenient implementation of default behavior. *Source:* Martin.
- *The above* , with straw man parameterized over Concrete Subject type, eliminating cyclic namespace dependency in C++. *Source:* Alexandrescu.

Known issues:

1. This pattern is strictly limited to stable hierarchies.
2. Default visit methods are error-prone unless the funtionality is nice to have.

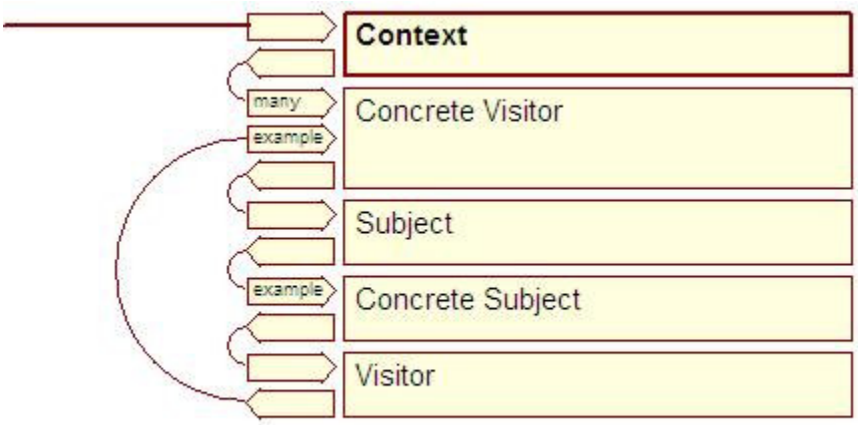
Scope: Languages that do not support multi-methods *(most commercial OO languages). The Visitor is really a multi-method over Concrete Visitor and Concrete Subject types).* Aspect oriented programming attacks the problem by from another direction.

Foil 32A - Visitor

Foil 1.2.32A: To assign discrete functionality by object type

The system shall have the capability
to process type family from the outside,
which involves...

Send feedback



Object model