Uppsala Universitet
Computing Science Department, Institute of Technology

Course:	Patterns and Frameworks (Spring 1999)
Instructor:	Amnon H. Eden
Authors:	Andreas Gustavsson
	Mattias Ersson

2nd submission July 2nd, 1999

Formalizing the Intent of Design Patterns

An Approach Towards a Solution to The Indexing Problem

Abstract

The *intent* section of a pattern description is written in easily understood, natural language, which unfortunately has the drawback of being too imprecise and unstructured for certain applications of the *intent* section.

We will in this essay try to formalize the intent section of a pattern description. Our aim will be to find a structure within the *intent* description that will reduce ambiguities and at the same time make the classification of patterns easier. The classification of patterns addresses the problem of “labeling” patterns into one of the following categories: Creational, Structural or Behavioral. Succeeding in classifying patterns by the *intent* does require that enough information for doing so is contained in the one to two sentences that make up the *intent*. Whether this is the case or not will be discussed in the essay.

A formalized *intent* section of a pattern description can not replace the understandability of the natural language description but can be thought of as a complement to the standard structure to patterns today.

Background

Our main goal with the formalization of the intent section in the pattern description is to make the task of finding an appropriate pattern, when looking to solve a problem, easier. This is often referred to as the indexing problem. If you are familiar with patterns and work with them a lot, you will probably know most of, for example the [GoF] patterns by heart and will have no trouble picking the correct one when you are searching for a design solution. If you on the other hand are not familiar with the patterns, you will have to read through all of the patterns in the book before you will be able to make your pick. The best index in the [GoF] book is the listing of the patterns with the intent descriptions.

Since a pattern catalogue is basically a catalogue of reusable design ideas it will become really usable when it contains a large number of patterns that are indexed in a way the provides for the catalogue to be used as almost any other reference. The cost of finding a reusable component or design solution has to be relatively low. Otherwise the developer will not use it, but come up with a solution of her own, that has not been tested and proved effective in the same way as an existing design pattern. [ISSE]

A problem with indexing patterns is however that the name of the pattern, which otherwise would be an intuitive, primary candidate to index on, does not convey enough information about the pattern. The name is perfectly fine to use in an index or table if the reader is already well acquainted with the patterns. If not, an ever so well-chosen name for a pattern is (in most cases) not sufficient to help the reader to determine if the pattern provides a solution to a particular problem. This leads us too look for another way of indexing patterns so that a developer that is searching a solution to a problem in a patterns catalogue can find a such in reasonable time, without having to read the whole catalogue.

Our appreciation is that the *intent* section provided in the way that the Gang of Four presents patterns drastically narrows down the search space of patterns that are applicable to the solution for a particular problem. As of today the intent part is not appropriate for indexing though. It consists of a couple of sentences in natural language, describing the essence of a pattern. How does one index on sentences though? Alphabetically on the first word of the sentence (several words if necessary of course), is probably the only feasible answer. If there was a way to index on the meaning, or semantics of the sentences this would be a much better way, since the alphabetical order has very little to do with finding a good solution to a problem. What different patterns really achieve does.

The index of intent that is presented in the GoF book, listed in three different groups, ordered alphabetically on the name of the pattern, is perhaps good enough for finding a pattern solving a problem, since there are relatively few patterns presented in the book. If a larger number of patterns were listed, the effort of finding a pattern would be too high.

Proposals

In this paper we present two heuristics that are possible steps on the way of finding a solution to the indexing problem. Even if the pattern community today is somewhat reluctant to formalize pattern material, since this would decrease creativity, this is necessary to a certain extent in order to be able to compose an index without ambiguities. When you are looking to solve a problem, you'll probably want to find a solution that **does** help you with the problem rather than a solution that could be applicable.

Therefore, we will first present a technique for rewriting the intent descriptions of the [GoF] book, in order to further increase the structure in the intent-index compared to what the case is in the book today. This will be achieved by formalizing the structure of the sentences that makes up the intent description.

Secondly we will present a graphical representation of the intent description. This is made of a modular system, presenting different entities, common to object oriented design, in combinations that will make it possible to group and list these combinations in a way that provides for indexing on them.

Sentence structure of the intent

The patterns in [GoF] have a fairly coarse classification into groups of creational, structural and behavioral patterns. Within these groups they are listed in alphabetical order, based on the name of the pattern. The intent itself does not give us any clue as to where it belongs, in any other way than being related to a pattern that conforms to this particular classification.

If the intent description itself could convey such information, it would be easier for a reader of the intent to immediately see what type of pattern it belongs to. And by that, also what kind of problem the author or authors of the pattern were addressing when they wrote down the intent.

The above is not entirely true though, it is possible to extract some semantic information from the description when taking a closer look at them. The trick is to extract this information and transfer it to the grammatical or syntactic structure, which is easier to *get a feel for* intuitively, without having to actually grasp the meaning of the sentence. This of course has to be done in a way that does not destroy or alter any of the semantic information presented in the intents, before rewriting them.

Creational patterns

For the creational patterns we can see that all but one (the Singleton pattern) directly talk about creating some entity with some behavior by performing some action or actions to provide for the result. In some cases supplementary information on the outcome or result of use of the pattern is also provided.

The Singleton pattern does this indirectly. In this pattern, instances *can not* be created, so it is a simple negation of the creation of some entity.

The structure of the intent descriptions for creational patterns could therefore be built in the following way:

Negation	**Create**	Entity	Action	Force	Supplementary information

Applying this to the [GoF] descriptions gives us the following rewritten intents:

Abstract factory

Provide an interface for creating families of related or dependent objects without specifying their concrete classes. [GoF p.87]

New structure:

Create	Entity	Action	Force
Create	Families of related or dependent objects	By providing an interface for doing so	Without specifying their concrete class

New description:
Create families of related or dependent objects by providing an interface for doing so, without specifying their concrete class.

Builder

Separate the construction of a complex object from its representation so that the same construction process can create different representations. [GoF p.97]

New structure:

Create	Entity	Action	Supplementary information
Create	Complex objects	By separating the construction from its representation	So that the same construction process can create different representations.

New description:
Create complex objects by separating the construction from its representation so that the same construction process can create different representations.

Factory method

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. [GoF p.107]

New structure:

Create	Entity	Action	Supplementary information
Create	Objects	By defining an interface that lets subclasses decide which objects to instantiate.	Factory Method lets a class defer instantiation to subclasses.

New description:
Create objects by defining an interface that lets subclasses decide which objects to instantiate. Factory Method lets a class defer instantiation to subclasses.

Prototype

> Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype. [GoF p.117]

New structure:

Create	Entity	Action	Force
Create	Objects	By copying a prototype.	Specify the kinds of objects to create using this prototypical instance.

New description:
Create objects by copying a prototype. Specify the kinds of objects to create using this prototypical instance.

Singleton

> Ensure a class only has one instance, and provide a global point of access to it. [GoF p.127]

New structure:

Negation	**Create**	Entity	Action
Do not	Create	More than one instance of a class.	Provide a global point of access to it.

New description:
Do not create more than one instance of a class. Provide a global point of access to it.

Structural patterns

The structural patterns, that deal with the composition of objects or classes all talk about some action that will be applied to an entity and what result or effects that will be achieved for doing so. In some of the patterns the outcome of the action is some form of transformation of the entity. In those cases the intent specifies this outcome. The common semantic structures of the intents are as follows:

Action to apply	Entity	Outcome	Result

Some of the intents described among the structural patterns fits perfectly fine into this model in their original form. However, if not all of them do so, we can not get any help with the indexing problem by looking at the structure of the sentences.

The intent description can, using this scheme be rewritten as follows:

Adapter

> Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. [GoF p.139]

New structure:

Action to apply	Entity	Outcome	Result
Convert	The interface of a class	Into another interface clients expect.	Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

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

Bridge

> Decouple an abstraction from its implementation so that the two can vary independently. [GoF p.151]

New structure:

Action to apply	Entity	Result
Decouple from each other	An abstraction and its implementation	So that the two can vary independently

New description:
Decouple from each other, an abstraction and its implementation, so that the two can vary independently.

Composite

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. [GoF p.163]

New structure:

Action to apply	Entity	Outcome	Result
Compose	Objects	Into tree structures to represent part-whole hierarchies	Composite lets clients treat individual objects and compositions of objects uniformly

New structure:
Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Decorator

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. [GoF p.175]

New structure:

Action to apply	Entity	Result
Dynamically attach additional responsibilities to	An object	Decorators deliver a flexible extending mechanism.

New description:
Dynamically attach additional responsibilities to an object. Decorators deliver a flexible extending mechanism.

Façade

Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use. [GoF p.185]

New structure:

Action to apply	Entity	Result
Provide a unified interface to	A set of interfaces in a subsystem	Façade defines a higher-level interface that makes the subsystem easier to use.

New description:
Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

Flyweight

Use sharing to support large numbers of fine-grained objects efficiently. [GoF p.195]

New structure:

Action to apply	Entity	Result
Use sharing on	Objects	Flyweight allows for large numbers of fine-grained objects to be handled efficiently

New description:
Use sharing on objects. Flyweight allows for large numbers of fine-grained objects to be handled efficiently.

Proxy

Provide a surrogate or placeholder for another object to control access to it. [GoF p.207]

New structure:

Action to apply	Entity	Result
Provide a surrogate or placeholder for another	Object	To control access to it

New description:
Provide a surrogate or placeholder for another object to control access to it.

Behavioral patterns

For the behavioral patterns the important issue is the (run time) behavior of a certain entity or construct in the program. This is achieved by performing some action in the program, and the intents descriptions sometimes provides information on what the results of using the patterns are. This leads to the following structure of the behavioral descriptions:

Objective	Course of action / proceedings	Result

The central part of this structure is the behavior, and using this heuristic with the model as above does give us a conflict with the classification as they are presented in [GoF]. The Visitor pattern only handles behavior in an indirect way and does not easily resolve to just a rephrasing of the sentences. The action performed is the central part in this pattern. It is still left in this section here though, in order to conform with the classification in [GoF].
The intents are rewritten as follows:

Chain of responsibility

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

New structure:

Objective	Course of action / proceedings
Avoid coupling the sender of a request to its receiver	By giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

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

Command

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations. [GoF p.233]

New structure:

Objective	Course of action / proceedings	Result
Parameterize clients with different requests	By encapsulating a request as an object.	Command lets you queue or log requests and support undoable operations.

New description:
Parameterize clients with different requests, by encapsulating requests as an object. Command lets you queue or log requests and support undoable operations.

Interpreter

> Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language. [GoF p.243]

New structure:

Objective	Course of action / proceedings
Interpret sentences in a given language	By using a representation that is defined along with the grammar for the language.

New description:
Interpret sentences in a given language, by using a representation that is defined along with the grammar for the language.

Iterator

> Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation. [GoF p.257]

New structure:

Objective	Result
Access elements of an aggregate object sequentially	Iterator provides a way to access the elements so that underlying representation is not exposed.

New description:
Access elements of an aggregate object sequentially. Iterator provides a way to access the elements so that underlying representation is not exposed.

Mediator

> Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring from each other explicitly, and lets you vary their interaction independently. [GoF p.273]

New structure:

Objective	Course of action / proceedings	Result
Encapsulate the interaction of a set of objects	By defining an object for this	Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and lets you vary their interaction independently

New description:
Encapsulate the interaction of a set of objects, by defining an object for this. Mediator promotes loose coupling by keeping objects from referring from each other explicitly, and lets you vary their interaction independently.

Memento

> Without violating encapsulation, capture and externalize an object’s internal state so that the object can be restored to this state later. [GoF p.283]

New structure:

Objective	Result
Capture and externalize an object’s internal state so that this state can be restored later	Memento does so without violating encapsulation.

New description:
Capture and externalize an object’s internal state so that this state can be restored later. Memento does so without violating encapsulation.

Observer

> Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. [GoF p.293]

New structure:

Objective	Course of action / proceedings
All dependencies are notified and updated automatically when one object changes state	By defining a one-to-many relationship between objects.

New description:
All dependencies are notified and updated automatically when one object changes state, by defining a one-to-many relationship between objects.

State

> Allow an object to alter its behavior when its internal state changes. The object will appear to change its class. [GoF p.305]

New structure:

Objective	Result
Allow an object to alter its behavior when its internal state changes	The object will appear to change its class

New description:
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Strategy

> Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. [GoF p.315]

New structure:

Objective	Course of action / proceedings	Result
Make algorithms in a family of algorithms interchangeable	By encapsulating each one	Strategy lets the algorithm vary independently from clients that use it

New description:
Make algorithms in a family of algorithms interchangeable by encapsulating each one. Strategy lets the algorithm vary independently from clients that use it.

Template method

> Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. [GoF p.325]

New structure:

Objective	Course of action / proceedings	Result
Defer steps of algorithms to subclasses	By defining the skeleton of an algorithm in one operation	Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure

New description:
Defer steps of algorithms to subclasses, by defining the skeleton of an algorithm in an operation. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Visitor

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

New Structure

Objective	Course of action / proceedings	Result
_	Represent an operation to be performed on the elements of an object structure	Visitor lets you define a new operation without changing the classes of the elements on which it operates.

New description:
No rewriting has been performed, since the description is difficult to fit into the model used. This will be disputed further in the discussion.

Discussion

Sentence structure of the intent

What is important when looking at the proposed restructuring of the intent descriptions, is not primarily the results achieved here, but the technique used for achieving them. A language researcher would certainly do a lot better work at deciding in what parts to divide the sentence structure. What is interesting though, is that it is possible to use such a heuristic in order to satisfy the same structure requirements on the sentences of the intent descriptions within the same group. In only one case was this not fully applicable, in the case of the Visitor pattern. It can actually be argued that the Visitor pattern is not a behavioral pattern. It does talk about how to represent something in order to perform operations on elements, rather than a specific behavior.

With this in mind, it is possible to see that not only is a structuring like this a help for the reader of a intent, but also for the author of a pattern, or the person that is classifying the pattern. If it is hard to write an intent according to the structure, or rewrite it to fit the model, it is maybe the case that the pattern should belong to a class whose structure fits the intent better.

The heuristic does however not bring in any new or better information in the intent, but simply reorganizes the words in order to simplify classification and thereby indexing on the intent section of a pattern description.

Conclusions

With simple means it is possible to reorganize the natural language used in the intent descriptions, without changing the semantics of the sentences, so that classification and thereby indexing on the intent will be easier. It is also a helpful heuristic, when writing or rewriting the description, to see that the pattern really belongs to the class that it was originally intended for.

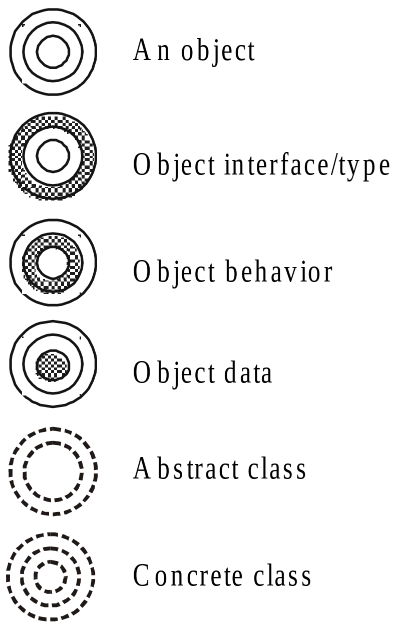
Graphical representation of the intent

In this section of this paper we try to interpret words and sentences of the intent part of all patterns in the [GoF] as graphical symbols. To compile and systematize a natural language into symbols is in fact hard. When every word which we use has a meaning to us, how is it possible to give definitions of common words as ”object” and ”class”. In this paper the scope is the ”natural language” of OOP which gives us a smaller domain of words and definitions then ordinary languages as English and Swedish. With this domain in mind we will try to illustrate the intent part of the patterns in simple symbols and entities so when looking at them understand the ”intent” of the pattern easily.

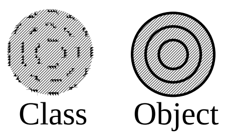
Because of the relation between natural language and graphical symbols it is inevitable that the making of these symbols and graphs are based on a more thorough knowledge of the pattern than just the intent section. The symbols sometimes give away more information than the intent itself. Especially when the intent does not speak about any object oriented terms. The purpose however is not to be a exact representation of the intent sections as they are written in [GoF] today, but rather extend them in a formalized way in order to make indexing easier.

We begin with very simple concept, definitions, and so forth, and step by step build up a graphical representation of the intent part of all patterns in the [GoF].

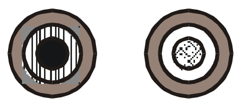
Entities of the graphical representation:



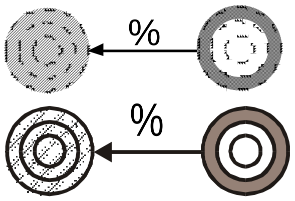
When representing an object of a certain class, a pattern fill will show the relation between the object and its class.



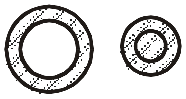
Two objects can have the same interface but different behaviors and states.



When objects or classes have an arrow line between them, they have some sort of interactions. The arrow line, have some sort of meaning depending of what symbol that is placed beside.
When the symbol ”%” appears it means change. The right entity describes what is changed in the left entity.

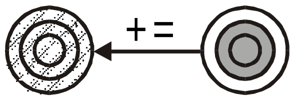


This picture represents the intent of the **Adapter** pattern.
To the left is a class or object that is changed and to the right is the symbol representing what is changed i.e. the interface. Note that the pattern does not imply that classes and objects have to be used in this way simultaneously, it is rather an *or* situation.

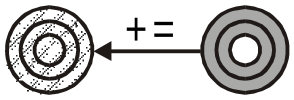


This picture represents the intent of the **Bridge** pattern.
The abstraction (the interface) is decoupled from the implementation, (behavior and data) so that the two can vary independently.

When the symbol ”+=” appears it means add, however not commutative. The right entity describes what is added to the left entity.

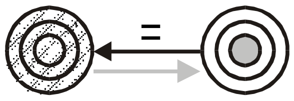


This picture represents the intent of the **Decorator** pattern.
To the left is an object that gets additional functionality.

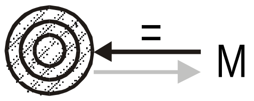


This picture represents the intent of the **Visitor** pattern.
To the left is an object that gets additional behaviors.

When the symbol ”=>” appears it means save and revert. The right entity describes what is saved and reverted to the left entity.

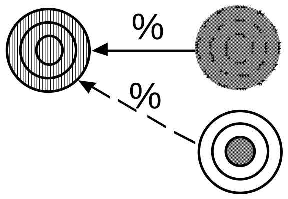


This picture represents the intent of the **Memento** pattern.
To the left is an object that saves(the ”=” symbol) the state of the right object. The light grey arrow means that the action can be reverted.



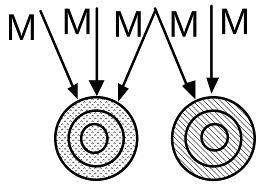
This picture represents the intent of the **Command** pattern.
To the left is an object that saves(the ”=” symbol) the message. The light gray arrow means that the action can be reverted.

When a dashed line appears it means that it is the real change but the appearing result is the change of the non dashed line.

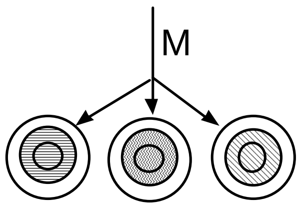


This picture represents the intent of the **State** pattern.
It is actually the internal state that changes (of the object to the left) but the object change its class.

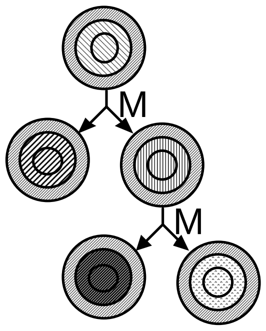
So far we have looked into changes of one object. Objects can also interact with each other using their interface. When using an object the client sends and the object reacts accordingly. So the symbol ”M” stands for a message that is send to the object



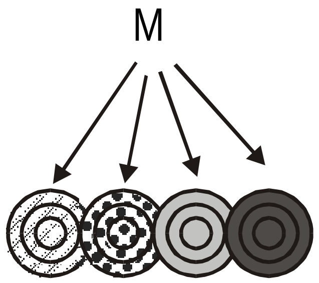
This picture represents the intent of the **Flyweight** pattern.
Several messages are send to the objects from different angles. This means that several clients share the same object since there is arrow lines from different clients.



This picture represents the intent of the **Strategy** pattern.
Several messages are send to the objects but through the same way. The line split into many arrow lines means that there is an option of which path to choose; one of the paths can be chosen by the message.

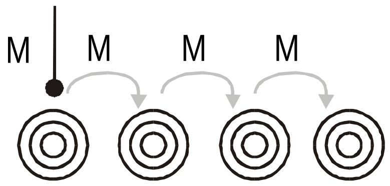


This picture represents the intent of the **Composite** pattern.
All nodes and leaves hava a uniform interface but different behaviors and states.



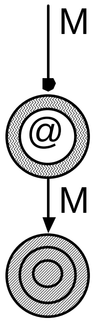
This picture represents the intent of the **Facade** pattern.
Messages are handled in the same way as the **Strategy** pattern. Intersecting objects means that they have combined their interfaces and the client sees them as one object.

When a client send a message indirectly to an object (i.e. the client send a message to representative that decides what to do with it) it is represented by a bullet line.

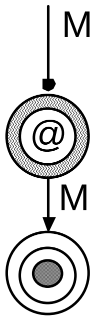


This picture represents the intent of the **Chain of Responsibility** pattern. The client (optionally) sends the message to a representative that send the message to the next object in the chain. The intent part does not mention anything about the type of the objects in the chain. Therefor the objects are represented without any pattern fill.

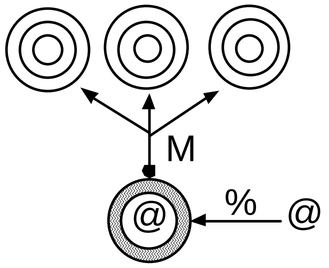
When an object has an access control to it, it's represented with an @-filled object.
This object handles the access to the object and works as an access point to it.



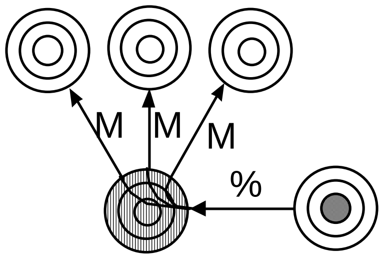
This picture represents the intent of the **Proxy** pattern.
The client sends the message to a representative handles the access to the object.



This picture represents the intent of the **Iterator** pattern.
The client sends the message to a representative that handles the access to the data of the object.

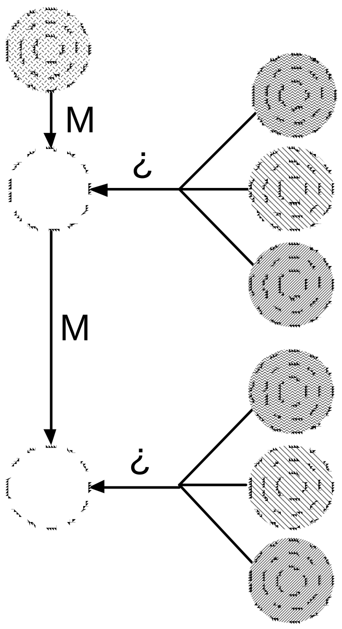


This picture represents the intent of the **Mediator** pattern.
The client sends the message to a representative (the bullet line to the access object) that handles the access for the interaction to the other objects(the arrow lines). The access can be changed.



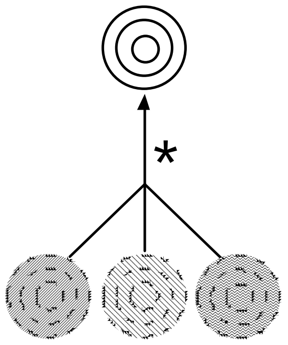
This picture represents the intent of the **Observer** pattern.
When the objects state change it send messages to all the dependent objects. Note that the message arrow line starts at the end of the change arrow line. This means that when the change occurs the message is sent.

When deferring instantiations to subclasses the ” ¿” symbol is used.

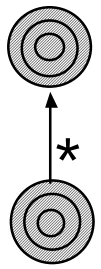


This picture represents the intent of the **Template Method** pattern. Subclasses are hooked (i.e. ”¿” symbol) to an abstract class (the subclass redefine the abstract class). The picture represents some related (steps) of an algorithm with conditional choices of subclasses i.e. let the subclass decide what to do in the specific part of the algorithm.

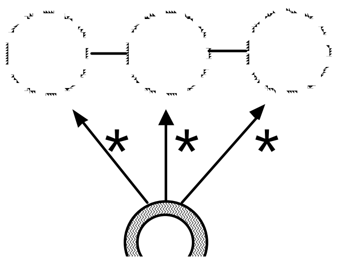
When creating an object the ”*” symbol is used.



This picture represents the intent of the **Builder** pattern.
The construction process (i.e. the ”*” symbol) can create objects with different representations. Therefor the conditional arrow line.

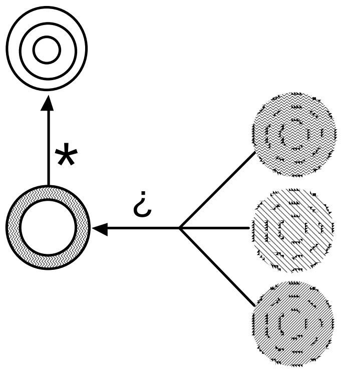


This picture represents the intent of the **Prototype** pattern.
One object(the prototype) is used to create another by copying it.

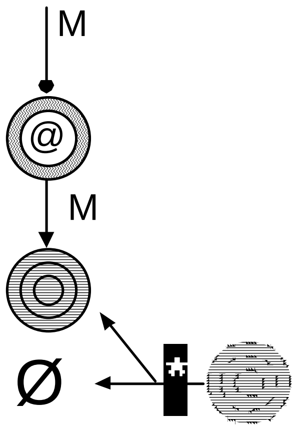


This picture represents the intent of the **Abstract Factory** pattern.

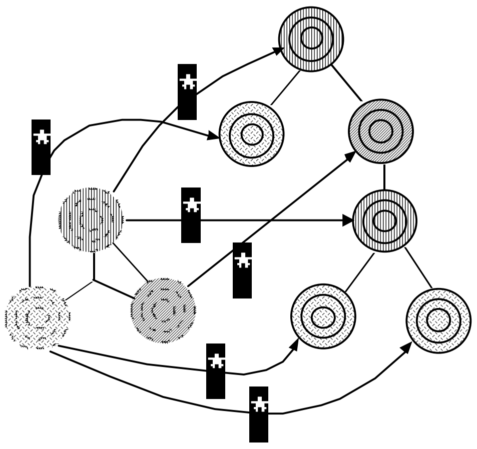
An interface is provided to create related objects without specifying their concrete classes.



This picture represents the intent of the **Factory Method** pattern.
An interface is provided to create an object.
Instantiation is deferred to subclasses.



This picture represents the intent of the **Singleton** pattern.
When creating an object of the class the conditional creation arrow line ensure that only one instance of the class exists. The ”empty set” symbol illustrated that no object is created.
Access control is provided for the object.



This picture represents the intent of the **Interpreter** pattern.
Related classes are used to create a structure of related objects.

Discussion

Graphical representation of the intent
When trying to illustrate the intent part we ran across many difficulties.
What is the key issue of the pattern according to the intent?
What is the best symbol for an object?
And so forth...

By using simple symbols as ”*” for creation to the more exotic ” ¿” to illustrate a hook we try to minimize the learning of the symbolic language we used. But this will also reduce the power to express the intent sentences.

In the quest for indexing the patterns we discovered that looking at similarities between the pictures one can see structures that repeatedly will be visible in several patterns. But this is both due to the small formalized symbol language used and hopefully similarities in the patterns.

Evidently the notation used resembles existing algebraic notations, and it most possibly would be a good idea to examine such for even better, and more widely used symbols. To do this in a good way would however require a substantial work in order to find the best existing symbols for this problem, and is out of scope for this paper.

Conclusions

With this simple graphical representation of the intent part of the [GoF] it is possible to fast and easy find a pattern that can help solving a problem. It will not replace the intent part but maybe be a good complement to the pattern catalog.

[GoF] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides “Design Patterns Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1998
[ISSE] Ian Sommerville “Software Engineering”, 5th ed., Addison-Wesley, 1997