A Catalogue of General-Purpose Software Design Patterns

Walter F. Tichy
University of Karlsruhe
Karlsruhe, Germany

Abstract

Software design patterns describe proven solutions to recurring software design problems. Knowledge of these patterns increases designers' abilities, leads to cleaner and more easily maintained software, speeds up implementation and test, and helps programmers document and communicate their designs.

This paper catalogues over 100 general-purpose design patterns. The organizing principle of the catalogue is the use of patterns, i.e., the problems they solve. Other considerations, such as whether a pattern is behavioral or structural, how it is implemented, or whether it is high or low level, are secondary, because these aspects are less important for a designer looking for a solution to a design problem.

The catalogue collects general-purpose patterns from a variety of sources. It includes older patterns such as Module and Layers as well as modern, object-oriented patterns such as Observer and Visitor.

Introduction

A software design pattern describes a family of solutions to a software design problem. It consists of one or several software design elements (such as interfaces, classes, objects, methods, functions, processes, threads, etc.), relationships among the elements (for example association, inheritance, delegation, invocation, and creation), and a behavioral description. Example design patterns are Layered System and Model-View-Controller,

The purpose of design patterns is to capture design know-how and make it reusable. Design patterns can improve the structure of software, speed up implementation, simplify maintenance, and help avoid architectural drift. Design patterns also improve communication among software developers and can empower less experienced developers to produce high-quality designs.

Over the past few years, the number of documented software design patterns has increased greatly. Multitude brings with it a need to organize and classify. This document catalogues a large number of software design patterns from a variety of sources. In particular, it includes all patterns from [Gamma 95] and [Shaw 95], a selection of patterns from [Buschmann 96], [PLOP 95], and [PLOP 96], and some other sources. The "classics" such as Layered System, Pipes and Filters, Module, Event Channel, and Repository are also included. We selected only general-purpose patterns, i.e., patterns that can be used in many systems regardless of application domain. The catalogue is organized according to the purpose of these patterns.

Related Work

A number of classifications of software design patterns exist. Gamma et al. [Gamma 95] use two orthogonal dimensions. The first dimension, purpose, differentiates between creational, structural, and behavioral patterns. The second dimension, scope, distinguishes whether a pattern applies primarily to classes or to objects. The distinction between behavioral and structural patterns is problematic when searching for patterns, because whether a pattern has one or the other property is difficult to determine when the pattern itself is not known. Scope exhibits the same problem and is not suitable for non-object-oriented patterns.

Buschmann et al. [Buschmann 96] distinguish architectural patterns, design patterns, and idioms. Within each of those categories, patterns are loosely organized according to purpose. Architectural patterns provide a top-level structural division of software, while design patterns refine components at a medium level, and idioms are low-level, language-specific code sequences. The problem with these categories is that it is difficult to place patterns consistently. The authors themselves seem to have trouble with this. For example, Model-View-Controller (MVC) appears as an architectural pattern. However, MVC builds on the Observer pattern, which is categorized under design. Noexplanation for this discrepancy is given. In fact, there are several patterns which could be placed into at least two categories. For example, Pipes and Filters is listed as an architectural pattern. However, one can easily imagine a situation in which this pattern appears somewhere deep inside the overall system, perhaps even at the detailed code level. For instance, buffered I/O uses Pipes and Filters and can appear at any level. Conversely, most of the patterns in the design category could also structure the top level of an architecture, in particular Whole-Part, Master-Slave, and Command Processor. Finally, the subordinate categorization according to purpose seems arbitrary and shows inconsistencies of its own. For example, Pipes and Filters appears under distributed systems, even though it works in centralized systems as well. Furthermore, the distinction of the distributed systems and communication categories is unclear. The interactive systems category seems to indicate incorrectly that patterns in other categories are not suitable for interactive systems. While we have found (and our own categorization shows this) that it is not always possible to place a pattern into exactly one category, it appears that the classification by Buschmann et al. suffers from an excessive number of these problems.

Zimmer [Zimmer 95] analyzes the internal structure of the patterns in [Gamma 95]. While such an analysis is extremely valuable for a deeper understanding of pattern relationships, the resulting classification tends to obscure the purpose of patterns.

In the following, we propose a classification scheme in which the top-level categories are the problem classes solved by patterns. A word of caution to the reader is in order: The descriptions of the patterns are too brief to be clear to the uninformed. So the catalogue is only useful to those who are already familiar with most patterns or willing to follow up the references.

The Catalogue

Concentrating on the problems solved by patterns leads to the following top-level categories:

1. **Decoupling:** dividing a software system into independent parts in such a way that the parts can be built, changed, replaced, and reused independently.
2. **Variant Management:** treating different objects uniformly by factoring out their commonality.
3. **State Handling:** generic manipulation of object state.
4. **Control:** control of execution and method selection.
5. **Virtual Machines:** simulated processors.
6. **Convenience Patterns:** simplified coding.
7. **Compound Patterns:** patterns composed from others, with the original patterns visible.
8. **Concurrency:** controlling parallel and concurrent execution.
9. **Distribution:** problems germane to distributed systems.

Categories should be mutually exclusive, with few exceptions. Subcategories are strict subsets of the parent category, also mutually exclusive as far as possible. It is not necessary for the categories to be "balanced," i.e., of approximately equal size. This can be seen in the decoupling category: It is the largest single category, reflecting the importance of dividing a system into independent units.

Similarities in implementation of patterns are ignored for categorization, although certain techniques such as inheritance and virtual functions tend to appear in clusters.

For each pattern, we provide its name and a reference, followed by three short paragraphs outlining the purpose of the pattern, what flexibility it provides, and how it is implemented.

1 Decoupling

Decoupling patterns divide a software system into several independent parts in such a way that they can be built, changed, and replaced independently as well as reused and recombined in unforeseen combinations. An important advantage of decoupling is local change, i.e., a system consisting of decoupled parts can be adapted and extended by modifying or adding a single or only a few parts, instead of modifying everything. The idea of decoupling is quite old. It goes back at least to the late 1950s, when programmers began to use each other's programs and write software libraries. A substantial number of decoupling patterns has evolved. Many of these patterns actually include an identifiable coupling component, so it would be more appropriate, though more cumbersome, to call this category "coupling/decoupling".

Decoupling patterns differ greatly in their range of applicability. The patterns Module, Abstract Data Type, and Hierarchical Layers, for example, have extremely broad applicability, whereas Iterator, Proxy, and Facet apply in much narrower design situations.

1.1 Module (Encapsulation, Information Hiding) [Parnas 72]

Purpose: cluster data structures and operations that change together and hide them behind a change-insensitive interface, so that changes to these components affect nothing outside the module.

Flexibility: change or replace implementation, hardware, I/O, OS, memory resources, network, etc., without affecting clients.

Implementation: make interface independent of likely changes.

Note: Examples of support for Modules are Modula's modules, C's header files, and Ada's packages. Normally, it is not possible to instantiate a module more than once in a single program. Instead, compiling and linking a module statically allocates the data for it.

1.2 Abstract Data Type (ADT, Class) [Dahl 68]

Purpose: hide data structure and access algorithms behind a change-insensitive interface.

Flexibility: change or replace implementation, hardware, I/O, OS, memory resources, network, etc. without affecting clients.

Implementation: make interface independent of likely changes.

Note: The purpose of ADT is similar to that of Module, but an ADT is typically smaller, containing a single data type. It is also possible to create multiple instances of an ADT. For a Module, there is normally only one instance. A Module, however, may combine several related ADTs.

There are numerous examples of ADTs. Some follow below.

1.2.1 Repository (Database) [Shaw 95]

Purpose: provide a central data structure with an access interface for multiple clients.

Flexibility: clients are independent of each other; add/remove clients; change implementation of data structure.

Implementation: get/set or query/update methods plus synchronization for parallel access (locks or transaction mechanism).

See also: Blackboard

Decoupling
- Module (Encapsulation)
- Abstract Data Type (Class)
  - Repository (Database)
    - Client/Server
  - Manager (Collection)
  - Iterator
  - Other Instances of ADT
- Layers
  - Sandwich
  - Façade
  - Mediator
  - Bridge
  - Adapter (Wrapper)
  - Proxy (Surrogate)
    - Decorator
    - Buffer Proxy
    - Logging/Counting Proxy
    - Firewall
    - Synchronization Proxy
    - Remote Access Proxy
  - Facet (Extension Object)
- Pipeline (Pipes and Filters)
  - Synchronous Pipe
    - Push Pipe
    - Pull Pipe
  - Asynchronous (Buffered) Pipe
- Event Notification
  - Catch and Throw
  - Callback
  - Event Loop
  - Event Channel
  - Propagator
    - Strict Propagator
    - Lazy Propagator
    - Adaptive Propagator
    - Observer
- Framework

Figure 1: Decoupling Patterns

1.2.1.1 Client-Server [Shaw 95]

Purpose: clients and repository (=server) run potentially on separate computers connected by a network.

1.2.2 Manager (Collection) [PLOP 96]

Purpose: place collection-related functions such as creation/deletion, registration, search, layout and display into a manager class, separate from the objects in the collection.

Flexibility: change, replace, or reuse manager.

Implementation: a large variety of data structures for implementing collections exist; they differ mainly in the speeds of update and lookup.

1.2.3 Iterator (Robust Iterator) [Gamma 95]

Purpose: provide an interface for sequentially accessing components in an aggregate/container.

Flexibility: vary internal structure of aggregate/container; multiple iterators may operate simultaneously on the same object.

Implementation: separate iterator state from aggregate/container; standardize interface for iteration.

1.2.4 Other Examples of ADT

Many more examples of abstract data types exist (container classes, graphs, matrices, etc.).

1.3 Layers (Hierarchical Layers) [Dijkstra 68, Shaw 95]

Purpose: a (software) layer provides an interface and an implementation of this interface. Layers are partially ordered with respect to the "Uses"-relation. The implementation of a layer may only use layers beneath itself in the partial order.

Flexibility: extension, contraction (partial reuse), replacement, combination, incremental build and test.

Implementation: the uses-relation among layers must be acyclic.

Examples:

- Operating system kernel
- Protocol stack
- Information system (layers are database, communication layer, core application, user interface).

1.3.1 Sandwich [Habermann76]

Purpose: break up cycle in "Uses"-relation between modules, classes, or objects by factoring out mutually used components and placing them into a separate module, class or object at a lower level. This process results in a three- or four-level "sandwich" structure.
Flexibility: Same as for layers.

1.3.2 Façade [Gamma 95]

Purpose: provide unified, convenient interface to a set of existing interfaces that are too "rich" or complicated to be exported in their entirety; also, hide some components.

Flexibility: change or replace "hidden" components including their interfaces.

Implementation: façade calls hidden component interfaces.

1.3.3 Mediator [Sullivan96, Gamma 95]

Purpose: encapsulate how sets of objects interact.

Flexibility: interacting objects do not know about each other (loose coupling); therefore it is easy to change objects and mediator.

Implementation: use separate classes for objects and mediator; objects inform mediator of significant events by direct call (upcall) or through event notification; mediator then invokes appropriate actions on other objects.

1.3.4 Bridge [Gamma 95]

Purpose: let abstraction and implementation layers evolve separately.

Flexibility: abstraction and implementation may evolve independently; it is possible to add a new abstraction or change the implementation without affecting the other.

Implementation: separate layers for abstraction and implementation, fixed interface of implementation.

1.3.5 Adapter (Wrapper) [Gamma 95]

Purpose: convert a given interface into another given interface.

Flexibility: no need to rewrite adaptee and its clients.

Implementation: adapter translates one interface into another.

1.3.6 Proxy (Surrogate) [Gamma 95]

Purpose: add or withdraw unplanned functionality transparently.

Flexibility: add or withdraw functionality without affecting original object or clients, without subclassing.

Implementation: proxy has the same interface as original; delegates request to original before/after adding functionality.

Variations:

- Decorator (Cascading Proxies)
- Buffer Proxy (Cache Proxy)
- Logging Proxy, Counting Proxy
- Firewall (Protection Proxy)
- Synchronization Proxy
- Remote Access Proxy

1.3.7 Facet (Extension Object) [PLOP 96]

Purpose: add new interfaces to existing classes without changing the classes; provide multiple views.

Flexibility: extend interface, extend functionality.

Implementation: An object registers its extensions and returns them if queried.

1.4 Pipeline (Pipes and Filters) [Shaw 95, Buschmann 96]

Purpose: pass data through a sequence of transformations (filters) connected by channels (pipes).

Flexibility: transformations do not depend on each other; stages may be replaced, added, or deleted; topology may be changed.

Implementation: defined input/output format and data passing protocol: add adapters to match filters.

1.5 Event Notification

Purpose: let independent parts interact by announcing and responding to events (loose coupling).

Flexibility: announcers and respondents are independent; the choice and number of respondents to an event is dynamic.

Implementation: respondents register their interest in an event.

1.5.1 Catch and Throw

Purpose: register error or event handlers dynamically; raising an event invokes the most recently registered handler for the given event type.

Flexibility: dynamically select handlers; an event may pass through any number of procedure or method invocations until an appropriate handler is found.

Implementation: supported by some programming languages directly, such as Common Lisp and Java.

1.5.2 Callback

Purpose: register an event handler (the callback) on an object. When the object receives an event, it calls the registered handler.

Flexibility: Associate different handlers with different objects for the same event; avoid cascaded if-statements to determine how to react to an event.

Implementation: The object receiving the event delegates to the callback. The callback can be a single function or an object containing several handlers.

1.5.3 Event Loop

Purpose: process events that are being posted on an event queue by other system components.

Flexibility: add/remove event posting components.

1.5.4 Event Channel [Barret96]

Purpose: let independent parts communicate indirectly, without knowing of each other.

Flexibility: integrate parts that know nothing of each other.

Implementation: participants register at common channel for events or data; notification sends event and data to channel; channel multicasts event and data to participants registered for the event.

1.5.5 Propagator [Feiler 98]

Purpose: propagate changes through a network of dependent objects.

Flexibility: extend/shrink network, add new classes of objects to network.

Implementation: registration and notification interface; direct notification through call or indirect notification through event channel.

1.5.5.1 Strict Propagator with/without failure

Purpose: propagate changes forward to the network all the way to the leaves.

1.5.5.2 Lazy Propagator

Purpose: Mark changed objects, but don't propagate until an object is accessed and then bring it up to date; or perform periodic updates.

1.5.5.3 Adaptive Propagator

Combination of Strict and Lazy Propagator: out-of-date markers are propagated strictly, but the updates themselves lazily.

1.5.5.4 Observer [Gamma 95]

Purpose: propagator limited to one level of dependents.

1.6 Framework [Lewis 95]

Purpose: provide a complete or nearly complete application layer that can be extended by subclassing.

Flexibility: add new subclasses where permitted, without changing application logic in framework.

Implementation: use subclassing, template methods, factory methods, builders, and abstract factories.

2 Variant Management

Variant management patterns treat different but related objects uniformly by factoring out their commonality. The variant patterns rely strongly on features found in object-oriented programming languages.

2.1 Superclass (Family)

Purpose: provide uniform treatment of variant classes by placing common interface into a superclass; variants are subclasses.

Flexibility: add new subclasses as variants without changing client programs.

Implementation: subclassing, inheritance, overriding.

Note: the difference between an ADT and subclassing is as follows. With an ADT it is possible to change its implementation without affecting the clients of the ADT. However, it is not possible to have several variants of an ADT in a single program at the same time; the decision of which one to use is made at compile or link time. With superclasses, however, a client program, can treat instances of different subclasses uniformly and in the same program.

- Variant Management
  - Superclass(Family)
    - Strategy
    - Composite
  - Visitor
    - Default Visitor
    - Extrinsic Visitor
    - Acyclic Visitor
  - Template Method
    - Factory Method
    - Builder
  - Abstract Factory

Figure 2: Variant Management Patterns

2.1.1 Strategy [Gamma 95]

Purpose: dynamically exchange algorithm variants.

Flexibility: replace or add algorithms used.

Implementation: common interface or superclass for algorithm variants, multiple implementations of algorithms, delegation of implementation.

Note: could also be listed under Control.

2.1.2 Composite [Gamma 95]

Purpose: handle parts-whole hierarchies; clients treat atomic objects and composites uniformly.

Flexibility: multiple atomic objects and composites.

Implementation: common interface for composites and atoms defined in superclass.

2.2 Visitor [Gamma 95]

Purpose: add new variations of operations to stable class hierarchy (e.g. a parts hierarchy, see Composite) without changing the class hierarchy.

Flexibility: adding or removing operation variants by touching only one class.

Implementation: use a common accept interface in classes; one visitor class per variant collects the methods for all classes in the hierarchy.

Note: Instead of extending all classes with a new operation, these operations are collected in one class. So this pattern has a dual role as a decoupling pattern: it decouples operations that exist in variants for many classes from those classes.

Variations:

- Default Visitor
- Extrinsic Visitor
- Acyclic Visitor

2.3 Template Method [Gamma 95]

Purpose: specify algorithm skeleton using primitives; vary primitives in subclasses or by delegation.

Flexibility: multiple families of primitives.

Implementation: define common interface for primitives across families as abstract functions, implement them in subclasses.

2.3.1 Factory Method [Gamma 95]

Purpose: specify creation skeleton using constructor primitives; vary primitives in subclasses.

Flexibility: multiple families of constructor primitives.

Implementation: common constructor interface across families as abstract functions, implementation in subclasses.

2.3.2 Builder [Gamma 95]

Purpose: Same as Factory Method, but uses delegation instead of subclassing to choose construction primitives.

Flexibility: multiple families of constructor primitives; add variants of skeleton without subclassing.

2.4 Abstract Factory [Gamma 95]

Purpose: bundle the constructors of a family of related objects into one object (the factory).

Flexibility: supply multiple families of constructors.

Implementation: common constructor interface across families; multiple implementations. Factory invokes creation operations directly from chosen family (delegation).

3 State Handling

State Handling patterns manipulate the state of objects generically. This means that these patterns work on the state of any object or class, independent

3.1 Singleton [Gamma 95]

Purpose: guarantee single instance of a class.

Implementation: record existence of instance in static member. of their actual purpose.

- State Handling
  - Singleton
  - Prototype (Cloneable)
  - Flyweight
  - Memento
    - Persistent Object

Figure 3: State Handling Patterns

3.2 Prototype (Cloneable) [Gamma 95]

Purpose: create new objects by cloning an existing one (the prototype).

Flexibility: prototype's content can be determined dynamically.

Implementation: interface for cloning.

3.3 Flyweight [Gamma 95]

Purpose: save space by sharing common state among objects.

Implementation: differentiate between extrinsic state and intrinsic state; share intrinsic (common) state.

3.4 Memento [Gamma 95]

Purpose: save and restore an object's internal state.

Flexibility: vary pack and unpack routines.

Implementation: write pack and unpack routines for externalizing and restoring state.

Variation: save object state in persistent store (deep or shallow copy).

4 Control

Control patterns deal with control of execution and selecting the right methods at the right time.

4.1 Blackboard [Shaw 95]

Purpose: dynamically decide which transformers ("knowledge sources") to apply to a shared data structure.

Flexibility: add and replace transformers; replace controller.

Implementation: use the following components: shared data structure, set of transformers, controller that selects transformers.

See also: Repository

4.2 Command (Command Processor) [Gamma 95]

Purpose: separate composition of a request from the time it is executed.

Flexibility: multiple commands; add functionality such as undo or redo, scheduling.

Implementation: encapsulate command with additional state (the objects on which to operate); add command processor.

- Control
  - Blackboard
  - Command
  - Chain of Responsibility
  - Strategy
  - Control State
  - Master/Slave
  - Process Control
    - Open-loop system
    - Closed-loop system
      - Feedback control
      - Feedforward control

Figure 4: Control Patterns

4.3 Chain of Responsibility [Gamma 95]

Purpose: pass a request down a chain of objects until an object handles it.

Flexibility: a) decoupling handler from request (handler is determined dynamically); b) add new handlers.

Implementation: common interface for handlers, delegation along chain.

Note: could also be listed under decoupling.

4.4 Strategy [Gamma 95]

See Strategy under Variant Management. Though strategy has to do with control (choosing the right algorithm), it fits more naturally under variants.

4.5 Control State (State) [Gamma 95]

Purpose: choose behavior according to state of an object.

Flexibility: add/remove states.

Implementation: distributed form of a finite state machine: uniform interface for actions; each state object implements behavior appropriate for given state.

4.6 Master-Slave

Purpose: dynamically distribute work over several subordinate processes (slaves).

Flexibility: add and remove slaves, e.g. to speed up completion, scale parallelism, balance load.

Implementation: master process creates slave processes, supplies tasks, waits for completion, then supplies additional tasks or may destroy slaves.

4.7 Process Control [Shaw 95]

Purpose: regulate a physical (continuous) process.

Flexibility: adjust or replace controller.

Implementation: contains controller, process variables, input variables, controlled variables, manipulated variables, sensors, set point.

Variations:

4.7.1 Open-loop system (process variables not used to adjust system)

4.7.2 Closed-loop system

- **Feedback control** (controlled variables are used to adjust system)
- **Feedforward control** (input or intermediate variables are used to adjust system)

5 Virtual Machines

A virtual machine simulates a processor. It is software that interprets a program written in a specific language.

```
Virtual Machines
├──Interpreter
│   └──Emulator
└──Rule-based Interpreter
```

Figure 5: Virtual Machine Patterns

5.1 Interpreter [Shaw 95]

Purpose: execute a program written in the interpreter's language.

Flexibility: replace interpreter (for portability, efficiency), extend language.

Implementation: contains program store, working memory, program counter, and methods to execute the instructions in the language.

Note: [Gamma 95] describes an object-oriented implementation of an interpreter without an explicit program counter.

5.1.1 Emulator

Purpose: simulate the instruction set of a hardware unit in software.

Flexibility: modifying emulator is easier than changing hardware.

5.2 Rule-based Interpreter [Shaw 95]

Purpose: execute a rule set using a fact base.

Flexibility: extend rules, replace interpreter.

Implementation: use components rule store, fact base, working memory, rule matcher (instead of program counter), and a rule interpreter.

Note: layer and facade could also be viewed as virtual machines, but lack a program counter.

6 Convenience Patterns

Convenience patterns simplify code by concentrating often-repeated code sequences in one place.

```
Convenience Patterns
├──Convenience Method
├──Convenience Class
├──Default Class
└──Null Object
```

Figure 6: Convenience Patterns

6.1 Convenience Method [PLOP 95]

Purpose: simplify method invocations by suppressing parameters whose values are the same for many calls.

Implementation: define specialized methods that call more general methods and supply frequently used parameter combinations.

6.2 Convenience Class

Purpose: simplify method invocations by storing parameter values in the class.

Implementation: define a class containing convenience methods plus variables to store the values of the suppressed parameters. Special operations change the parameter values.

6.3 Default Class

Purpose: provide a default implementation of a class that normally must be reimplemented by the client. Subclasses override only the methods for which the defaults are inappropriate.

6.4 Null Object (Stub) [PLOP 96]

Purpose: eliminate frequent tests for null references by replacing null references with a reference to the null object.

Implementation: a null object is an instantiation of a class with pseudo implementations of its methods. These methods do nothing or return default values appropriate for null references.

7 Compound Patterns

Compound patterns are composed from others, with the original patterns visible to the client programs. Most patterns are actually composed of others, but if a particular combination takes on a different purpose, then it is categorized according to this purpose.

Because compound patterns consist of several other patterns, they can usually be classified into several categories at once. To avoid the duplication, they are placed in this category.

Compound Patterns
- Model/View/Controller
- Bureaucracy
- Active Bridge

Figure 7: Compound Patterns

7.1 Model-View-Controller (MVC)

Purpose: provide multiple, dynamic views on shared data and dynamically change responses to user input.

Flexibility: add and remove views, change response, add and remove components.

Implementation: combination of observer, strategy, and composite.

7.2 Bureaucracy [PLOP 96]

Purpose: organize objects in a hierarchical structure such that it maintains its inner consistency by itself.

Flexibility: extend and shrink hierarchy, extend and shift responsibilities.

Implementation: combination of composite, observer, and chain of responsibility.

7.3 Active Bridge [Riehle 97]

Purpose: Connect an application to event-driven resouces in a portable way.

Flexibility: replace platform, change event handling, let platform and application evolve independently.

Implementation: combination of Bridge, Proxy, Observer, Abstract Factory, Factory Method.

8 Concurrency

Concurrency patterns control parallel and concurrent execution. The following patterns are in use (without further explanation).

Concurrency
- Semaphore
- Critical Region
- Conditional Critical Region
- Monitor
- Double-Checked Locking
- Thread Management
- Event Loop
- Send/Receive
- Reactor
- Rendezvous
- Listener
- Daemon
- Readers/Writers
- Bounded Buffer
- Active Object (Future)
- Asynchronous Completion Token
- Half-Sync/Half-Async
- Transaction and Rollback

Figure 8: Concurrency Patterns

9 Distribution

Distribution patterns solve problems that arise in distributed systems, e.g. problems with communication and cooperation, finding resources, dealing with network outages, or the low bandwidth of network links. Patterns that deal with parallelism, but not necessarily with distribution appear under Concurrency.

Distribution
- Protocol Stack
- Remote Procedure Call
- Router
- Acceptor and Connector
- Gateway
- Forwarder/Receiver
- Client/Server
- Client/Dispatcher/Server
- Broker
- Distributed State (Recoverable Distributor)

Figure 9: Distribution Patterns

Conclusions

Software design patterns collect the growing body of knowledge about software design. While experienced developers know many (though usually not all) of the patterns, organizing and systematizing this knowledge is important to make it accessible to future generations of software developers. This paper is a first attempt at such an organization. It needs input from many viewpoints on software design before it can evolve into a catalogue of patterns.

Acknowledgements: Many people from the pattern community have contributed comments to this work. The most detailed critique came from Ralph Johnson, who helped me understand a number of different viewpoints and thoroughly discussed the categorization of a number of patterns.

Bibliography

[Barrett 96] Daniel J. Barrett et al., A framework for event-based software integration, *ACM Trans. on Software Engineering and Methodology* 5(4), Oct. 1996, 378-421.

[Bloss 93] Adrienne G. Bloss and J.A.N. Lee, Language Processors, Encyclopedia of Computer Science, Van Nostrand Reinhold, New York, 1993, 727-733

[Buschmann 96] F. Buschmann et al., *A System of Patterns*, John Wiley & Sons, 1996.

[Dahl 68] O.-J. Dahl, B. Myhrhaug, and K. Nygaard, The Simula 67 Common Base Language, Norwegian Computing Center, Oslo, Norway, 1968.

[Dijkstra 68] Edsger W. Dijkstra, The structure of the THE multiprogramming system, *Communications of the ACM* 11(5), May 1968, 314-346.

[Feiler 98] Peter Feiler and Walter Tichy, Propagator -- A Family of Patterns, *Proc. TOOLS USA 97*, IEEE 1998.

[Gamma 95] E. Gamma et al., *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley Professional Computing Series. 1995.

[Habermann76] A. Nico Habermann, Lawrence flon, and Lee W. Cooprider, Modularization and hierarchy in a family of operating systems, *Communications of the ACM*, 19(5), ay 1976, 266-272.

[Islam 96] Nayem Islam and Murthy Devarakonda, An essential design pattern for fault-tolerant distributed state sharing, *Communications of the ACM*, 39(10), Oct. 1996, 65-74.

[Lewis 95] Ted Lewis, Object-Oriented Application Frameworks, Manning, 1995.

[Parnas 72] David L. Parnas, On the criteria do be used in decomposing systems into modules, *Communications of the ACM* 15(2), Dec. 1972,1053-1058.

[PLOP 95] James O. Coplien and Douglas C. Schmidt (eds.), *Pattern Languages of Program Design*, Addison-Wesley, 1995.

[PLOP 96] Douglas C. Schmidt (ed.), *Proceedings of the 3rd Annual Conference on the Pattern Languages of Programs*, University of Illinois at Urbana-Champaign, Sept. 1996.

[Riehle 97] Dirk Riehle, Composite Design Patterns, *Proc. OOPSLA'97*, ACM SIGPLAN Notices 32(10), Oct. 1997, 218-228.

[Shaw 95] Mary Shaw and David Garlan, *Software Architecture*, Prentice Hall 1995.

[Sullivan 96] K.J. Sullivan et al., Evaluating the Mediator Method: Prism as a case study, *IEEE Trans. on Software En-*

gineering 22(8), Aug. 1996, 563-579.

[Zimmer 95] Walter Zimmer, Relationships between design datterns, in [PLOP 95], 346-36S4.