A new Pattern Systematics

Christian Heller <christian.heller@tu-ilmenau.de>
Detlef Streitferdt <detlef.streitferdt@tu-ilmenau.de>
Ilka Philippow <ilka.philippow@tu-ilmenau.de>

Technical University of Ilmenau
Faculty for Computer Science and Automation
Institute for Theoretical and Technical Informatics
PF 100565, Max-Planck-Ring 14, 98693 Ilmenau, Germany
http://www.tu-ilmenau.de, fon: +49-3677-69-1230, fax: +49-3677-69-1220

Abstract

Software patterns are a great aid in designing the architecture of application systems. They provide standard solutions for frequently occuring problems.
This paper introduces a new schema for systematizing patterns. It arose from firstly, the investigation of a whole spectrum of different patterns and secondly, the application of the principles of human thinking to the classification of these patterns.
This new way of sorting patterns uncovers their common advantages but also disadvantages and may have the potential to better support developers in choosing the right pattern solutions for their problems.
Keywords. Software Design, Pattern Systematics, Cybernetics Oriented Programming, CYBOP

1 Introduction

Patterns are a popular architecture instrument of current systems and languages – in the first line, however, of *Object Oriented Programming* (OOP). They describe design solutions that belong to a higher conceptual level, as opposed to the programming paradigms which are inherent to languages.

A common critics on the existence of patterns is put into words by the free *Wikipedia* encyclopedia [5] that writes:

> Some feel that the need for patterns results from using computer languages or techniques with insufficient abstraction ability. Under ideal factoring, a concept should not be copied, but merely referenced. But if something is referenced instead of copied, then there is no pattern to label and catalog.

In other words, patterns would become superfluous, if they could be applied just *once* to a system, in a manner that allowed any other parts of that system to reference and reuse-, instead of copy them.

The investigation of possibilities to better abstract knowledge in software belongs to the aims of the *Cybernetics Oriented Programming* (CYBOP) project [24]. It wants to eliminate the need for repeated pattern usage, and such enable application programmers, and possibly even domain experts, to faster create better application systems.

On the way to reaching such sublime aims, a first step is to look at current pattern solutions and try to identify what their common characteristics are. This is what the next sections will do. Those experts who sufficiently know the patterns explained following, may skip over section 2 and continue reading at section 3.

2 Pattern

2.1 Definition

Patterns, in a more correct form called *Software Patterns*, represent solutions for recurring software design problems and can be understood as recommendations for how to build software in an elegant way. In the past, more detailed definitions have been given by meanwhile well-known authors.

Christopher Alexander, an architect and urban planner, writes [1]: *Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.* He gave this definition primarily for problems occuring in architecture, construction, and urban/regional planning, but it can be applied in the same manner to software design, as done first by Ward Cunningham and others [16].

The systems designer Swift [7] sees a pattern as: *essentially a morphological law, a relationship among parts (pattern components) within a particular context. Specifically, a pattern expresses a relationship among parts that resolves problems that would exist if the relationship were missing. As patterns express these relationships, they are not formulae or algorithms, but rather loose rules of thumb or heuristics.*

The *Gang of Four* (Erich Gamma et al.) applied Alexander's definition to object oriented software and created a whole catalogue of design patterns [10]. After them, patterns are: *Structured models of thinking that represent reusable solutions for one-and-the-same design problem. They shall support the development, maintenance and extension of large software systems, while being independent from concrete implementation languages.* The experts identified four basic elements of each pattern: *Name*, *Problem*, *Solution* and *Consequences* (advantages and disadvantages).

For Frank Buschmann et al., software patterns contain the knowledge of experienced software engineers and help to improve the quality of decision making [3]. In his opinion, they are *Problem Solution Pairs*, that is basic solutions for problems that already occured in a similar way before.

Martin Fowler means that: *A pattern is some idea that already was helpful in a practical context and will probably be useful in other contexts, too.* [8]. After him, patterns, however they are written, have four essential parts: *Context*, *Problem*, *Forces* and *Solution*.

Depending on their experience, software developers produce good or bad solutions. One possibility to improve less well-done designs or to extend legacy systems are *Anti-Patterns* (telling how to go from a problem to a bad design), or the contrasting *Amelioration Patterns* (telling how to go from a bad- to a good solution) [16]. Both help finding patterns in wrong-designed systems, to improve these.

There are efforts to combine patterns to form a *Pattern Language*, also called *Pattern System* [3]. Such systems describe dependencies between patterns, specify rules for pattern combination and show how patterns can be implemented and used in software development practice.

2.2 Classification

Several schemes of *Pattern Classification* exist. One possible is shown in figure 1. Considering the level of abstraction (granularity), it distinguishes between *Architectural-*, *Design-* and *Idiomatic* patterns [3]. Design patterns, in turn, are divided after their functionality (problem category) into *Creational-*, *Structural-* and *Behavioural* patterns [10]. The Wikipedia Encyclopedia [5] mentions three further problem categories: *Fundamental-*, *Concurrency-* and *Real-time* patterns. Other criteria (dimensions) of classification exist. Fowler introduces a completely different category which he calls *Analysis Patterns* [8]. These are applicable early in the *Software Engineering Process* (SEP). And he defines patterns that are more often used for describing the modelling *Language* than the actual *Models* as *Meta Model Patterns*.

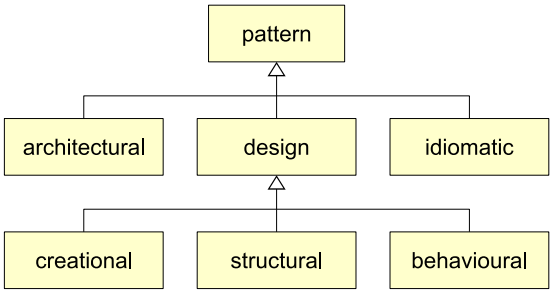

Fig. 1. Software Pattern Classification

2.3 Examples

This section briefly describes a greater number of known patterns. They are basic examples referenced by the pattern systematics introduced in section 4.2, later-on. However, since this section does not want to copy the work accomplished by the aforementioned authors, it refers to the corresponding literaric source for more detailed explanation.

Architectural *Architectural Patterns* are templates for the gross design of software systems. They describe concrete software architectures and provide basic structuring (modularization) principles.

Layers The *Layers* pattern [3] is one of the most often used principles to subdivide a system into logical levels. One famous variant contains the three layers *Presentation*, *Domain Logic* and *Data Source*. Another well-known example making use of this pattern is the *Open Systems Interconnection* (OSI) reference model, defined by the *International Organization for Standardization* (ISO). Numerous books [25, 23] describe this model and its protocols.

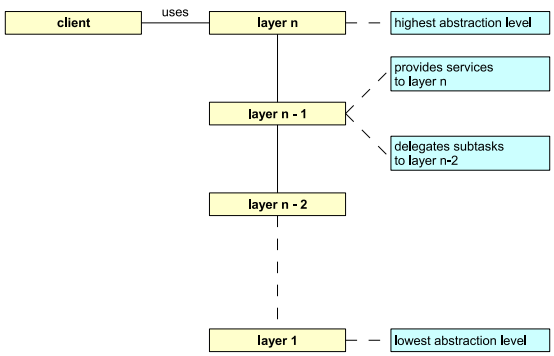

Fig. 2. Layers Pattern

A more general illustration can be seen in figure 2. It shows a client using the functionality encapsulated in a layer. That top-most layer delegates subtasks to lower-level layers which are specialized on solving them.

One variant of this pattern, mentioned by Buschmann [3], is the *Relaxed-Layered-System*. It permits a layer to not only use the services of its direct base layer, but also of yet lower-situated layers. The base layer in this case is called *transparent*.

Data Mapper Besides the *Domain Logic*, standard three-tier architectures contain a *Data Source* layer which may for example represent a database. Both layers need to exchange data. Modern systems use OOP methods to implement the domain model. Database models, on the other hand, are often implemented as *Entity Relationship Model* (ERM).

In order to avoid close coupling and a mix-up of both layers, the introduction of an additional *Data Mapper* layer [9] in between the two others may be justified (figure 3). The most important idea of this pattern is to abolish the interdependencies of domain- and persistence model (database).

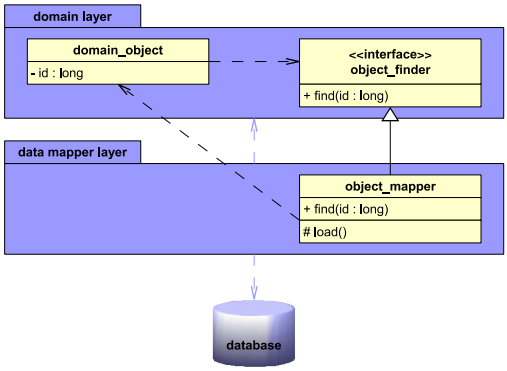

Fig. 3. Data Mapper Pattern

The dashed arrows in figure 3 indicate dependencies. The data mapper layer knows the domain model- as well as the data source layer, via *unidirectional* relations. Its task is to *translate* between the two, in both directions. Domain model and data source know nothing from each other.

Each domain model class knows its appropriate interface (*object_finder*) but does not know the implementation of the same. That is, persistence- and data retrieval mechanisms are hidden in front of the domain model. The implementation (*object_mapper*) is part of the mapping package and also implements all finder methods. It maps data of the received result sets to the special attributes of the domain model objects.

The *Mediator* pattern [10] is similar to the *Mapper*, in that it is used to decouple different parts of a system. Fowler [9] writes: *. . . the objects that use a mediator are aware of it, even if they aren't aware of each other; the objects that a mapper separates aren't even aware of the mapper.*

Although the *Data Mapper* pattern is very helpful at implementing OO systems, two things are to be criticised:

Firstly, since the *object_finder* relies on functionality specific to the retrieval of persistent data, it does actually belong into the data mapper layer what, if done, would create bidirectional dependencies between the domain model- and data mapper layer. But also with the *object_finder* remaining in the domain model layer, dependencies are not purely unidirectional. It is true that from an OO view, they are. Internally, however, a super class or interface relates to its inheriting classes, so that it can call their methods to satisfy the polymorphic behaviour.

Secondly, the layers do not truely build on each other. Taken a standard architecture consisting of the following five – instead of only three – layers:

1. Presentation
2. Application Process
3. Domain Model
4. Data Mapper
5. Data Source

. . . the application process does not only access the domain model layer, it also has to manage (create and destroy) the objects of the data mapper layer. In other words, it surpasses (disregards) the domain model layer when accessing the data mapper layer directly.

Data Transfer Object It is a well-known fact that many small requests between two processes, and even more between two hosts in a network need a lot of time. A local machine with two processes has to permanently change the *Program Context*; a network has a lot of *Transfers*. For each request, there is a necessity of at least *two* transfers – the *Question* of the client and the *Answer* of the server.

Transfer methods are often expected to deliver common data such as a Person's address, that is surname, first name, street, zip-code, town and so on. These information is best retrieved by only *one* transfer call. That way, the client has to wait only once for a server response and the server does not get too many single tasks. In this example, all address data would best be packaged together and sent back to the client.

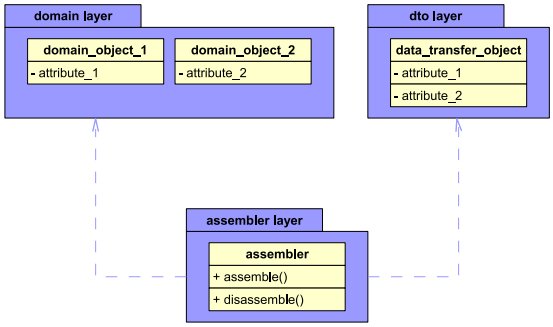

Fig. 4. Data Transfer Object Pattern

A scenario of that kind is exactly what the *Data Transfer Object* pattern [9] proposes a solution for: A central *Assembler* object takes all common data of the server's domain model objects and assembles them together into a special *Data Transfer Object* (DTO), which is a flat data structure (figure 4). The server will then send this DTO over network to the client. On the client's side, a similar assembler takes the DTO, finds out all received data and maps (disassembles) them to the client's domain model. In that manner, a DTO is able to drastically improve the communication performance.

Model View Controller After having had a closer look at design patterns for persistence (*Data Mapper*) and communication (*Data Transfer Object*), this section considers the presentation layer of an application, which is often realized in form of a *Graphical User Interface* (GUI).

Nowadays, the well-known *Model View Controller* (MVC) pattern [3, 9] is used by a majority of standard applications. Its principle is to have the *Model* holding domain data, the *View* accessing and displaying these data and the *Controller* providing the workflow of the application by handling any action events happening on the view (figure 5). This separation eases the creation of applications with many synchronous views on the same data. Internally, the MVC may consist of design patterns like:

- *Observer* notifying the views about data model changes
- *Strategy* [10] encapsulating functionality of the controller, to make that functionality easily exchangeable
- *Wrapper* delegating the controller functionality to the strategy mentioned before
- *Composite* equipping graphical views with a hierarchical structure

Some MVC implementations like parts of the *Java Foundation Classes* (JFC) use a simplified version not separating controllers from their views. The *Microsoft Foundation Classes* (MFC) C++ library calls its implementation *Document-View*.

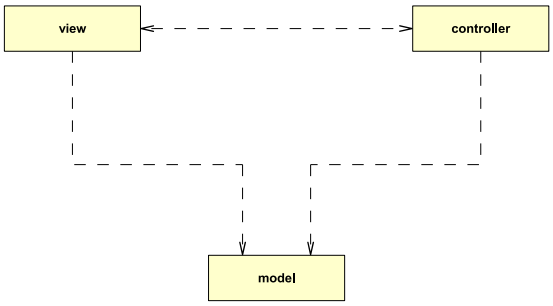

Fig. 5. Model View Controller Pattern

Hierarchical Model View Controller There exist several extensions of the MVC pattern, one of them being the *Hierarchical Model View Controller* (HMVC) [4]. It combines the patterns *Composite*, *Layers* and *Chain of Responsibility* into one conceptual architecture (figure 6).

This architecture divides the presentation layer into hierarchical sections containing so-called *MVC Triads*. The triads conventionally consist of *Model*, *View* and *Controller*, each. They communicate with each other by relating over their controller object. Following the *Layers* pattern, only neighbouring layers know from each other.

As a practical example, the upper-most triad could represent a graphical *Dialog* and the next lower one a *Panel*. Being a container, too, the panel could hold a third triad like for example a *Button*. Events occuring at the button are then normally processed by the corresponding controller belonging to the button's triad. If, however, the button controller cannot handle the event, that is forwarded along the chain of responsibility to the controller of the higher-next layer. If also the panel controller does not know how to handle the event, the final responsibility falls to the controller of the dialog's triad.

The HMVC is similar to the *Presentation Abstraction Control* (PAC) pattern [3]. A *PAC Agent* is comparable to an *HMVC Triad*.

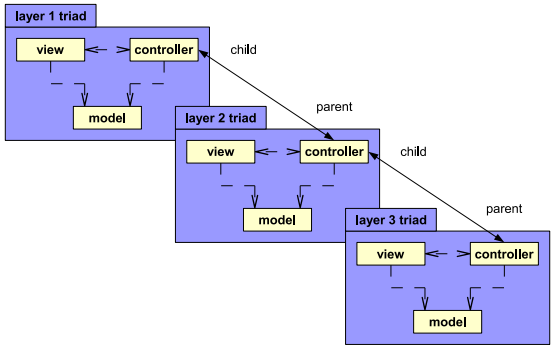

Fig. 6. Hierarchical Model View Controller Pattern

Microkernel The *Microkernel* pattern [3] allows to keep a system flexible and adaptable to changing requirements or new technologies. A minimal functional *Kernel* gets separated from extended functionality. The kernel may call internal- or external servers (figure 7) to let them solve special tasks which do not belong to its own core responsibility. Internal servers are often called *Daemons*.

This pattern provides a *Plug & Play* environment and serves as base architecture for many modern *Operating Systems* (OS). Andrew S. Tanenbaum recommends its use as well [26].

Broker The *Broker* pattern [3] may support the creation of an IT infrastructure for distributed applications. It connects decoupled components which interact through remote service invocations (figure 8).

The broker is responsible for coordinating all communication, for forwarding requests as well as for transmitting results and exceptions.

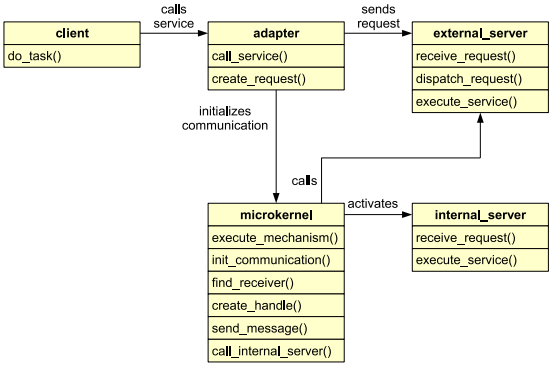

Fig. 7. Microkernel Pattern

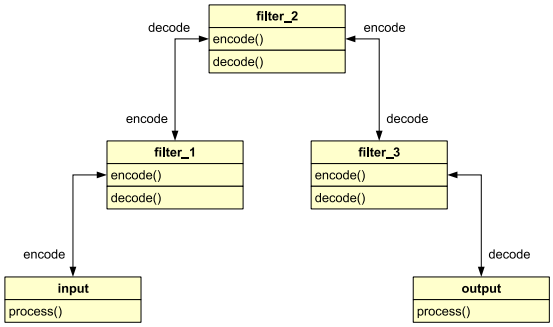

Fig. 9. Pipes and Filters Pattern

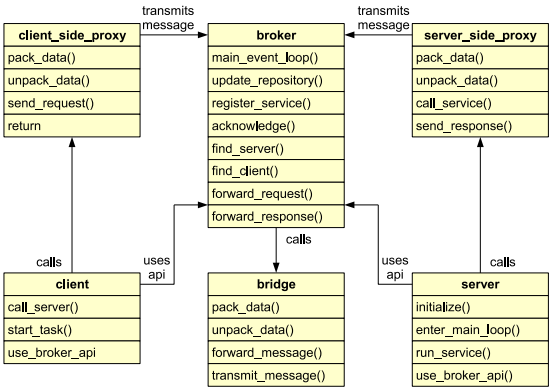

Fig. 8. Broker Pattern

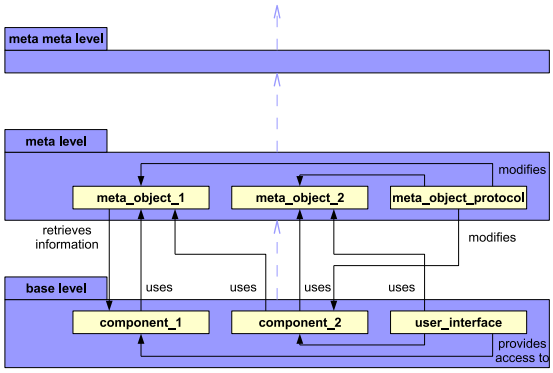

Fig. 10. Reflection Pattern

Pipes and Filters Systems that process streams of data may use the *Pipes and Filters* pattern [3]. It encapsulates every processing step in an own *Filter* component and forwards the data through channels which are called *Pipeline* (figure 9). Families of related systems can be formed by changing the single filter positions. The data forwarding can follow various scenarios:

- *Push:* active filter pushes data to passive filter
- *Pull:* active filter pulls data from passive filter
- *Mixed Push-Pull-Pipeline:* all filters push or pull data
- *Independent Loops:* all filters actively access pipeline

Reflection The *Reflection* pattern [3] (also known under the synonyms *Open Implementation* or *Meta-Level Architecture*) provides a mechanism to change the structure and behaviour of a software system *dynamically*, that is at runtime, which is why that mechanism is sometimes called *Run Time Type Identification* (RTTI). A reflective system owns information about itself and uses these to remain changeable and extensible.

Reflective information *about* something is called *Meta Information*. Therefore, the level above the *Base Level* in figure 10 is labelled *Meta Level*. The base level depends on the meta level, so that changes in the meta level will also affect the base level. All manipulation of meta objects happens through an interface called *Meta Object Protocol* (MOP), which is responsible for checking the correctness of- and for performing a change. If a further level holds information about the meta level, then that additional level is called *Meta Meta Level*, and so forth.

Many examples of meta level architectures exist. In his book *Analysis Patterns* [8], Fowler uses them extensively. He talks of *Knowledge Level* (instead of meta level) and *Operational Level* (instead of base level). Elements of the *Unified Modeling Language* (UML) are defined in an own meta model [22]. And the principles of reflection are also supported by several programming languages, such as *Smalltalk* [20] and *Java* [17].

Classes (types) in a system have a static structure, as defined by the developer at design time. Normally, most classes belong to the base level containing the application

logic. As written before, one way to change the structure and behaviour of classes at runtime is to introduce a meta level providing type information, in other words functionality that *all* application classes need. This helps avoid redundant implementations of the same functionality.

Looking closer at functionality, it turns out that some basic features like persistence and communication occur repeatedly in almost all systems, while other parts are specific to one concrete application. Traditionally, the application classes in the base level have to cope with general system functionality although that is not in their original interest. It therefore seems logical to try to divide application- and system functionality, and to put the latter into a meta level.

Design Gamma et al. [10] define a design pattern as: *description of collaborating objects and classes which are taylored to solve a general design problem in a special context.* Mostly, patterns are in relation to each other. They can be combined to master more complex tasks.

Command The *Command* pattern [10], also known as *Action* or *Transaction*, sometimes also *Signal*, encapsulates a command in form of an object. That way, operations can get parameterised; they can be put in a queue, be made undone or traced in a log book. Figure 11 shows the structure of the pattern.

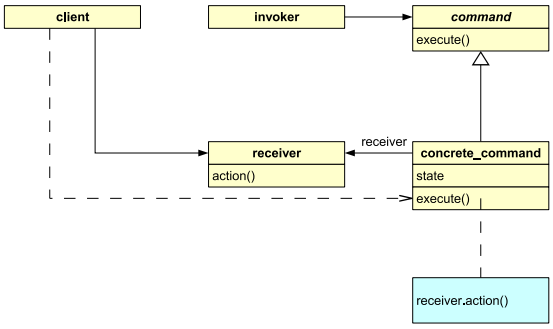

Fig. 11. Command Pattern

Wrapper The *Wrapper* pattern [10] allows otherwise incompatible classes to work together. It can be seen as skin object enclosing (wrapping) an inner core object, to which it provides access. In other words: It adapts the interface of a class which is why Gamma et al. call the pattern *Adapter*.

As can be seen in figure 12, this pattern makes heavy use of *Delegation*, where the *Delegator* is the adapter (or wrapper) and the *Delegate* is the class being adapted [16].

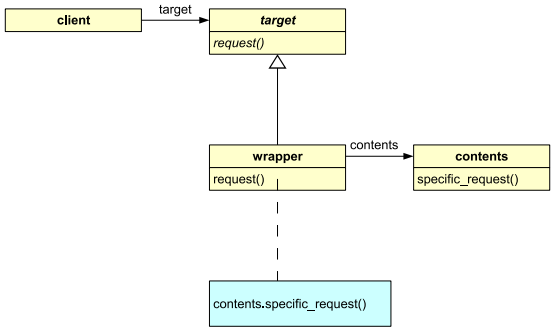

Fig. 12. Wrapper Pattern

Whole Part Whenever many components form a semantic unit, they can be subsumed by the *Whole-Part* pattern [3]. It encapsulates single part objects (figure 13) and controls their cooperation. Part objects are not addressable directly.

Almost all software systems contain components or sub systems which could be organized by help of this pattern. In some way, it is quite similar to the previously introduced *Wrapper*, only that not just one- but many objects are wrapped.

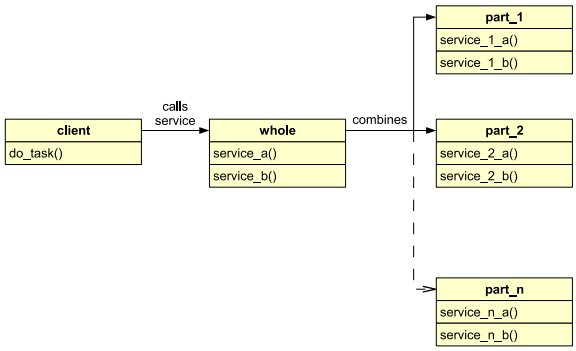

Fig. 13. Whole-Part Pattern

Composite A hierarchical object structure, also called *Directed Acyclical Graph* (DAG) or *Tree*, can be represented by a combination of classes called *Composite* pattern [10]. It describes a *Component* that may consist of *Children* (figure 14), which makes it comparable to the *Whole-Part* pattern. The difference is that the *Composite* is a more generalized version, with a dynamically extensible number of child (part) objects.

The *Composite* is a pattern based on *Recursion*, which is one of the most commonly used programming techniques at all. The pattern's split into *Leaf*- and *Composite* sub classes helps distinguish primitive- from container objects. A composite tree node holds objects of type *Component*.

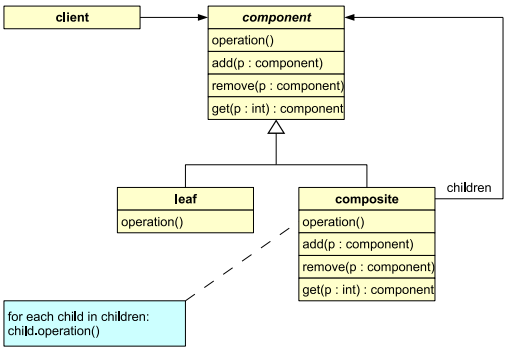

Fig. 14. Composite Pattern

Chain of Responsibility The *Chain of Responsibility* pattern [10] is similar to the *Composite*, in that it represents a recursive structure as well. Objects destined to solve a task are linked with a corresponding *Successor* (figure 15), such forming a chain. If an object is not able to solve a task, that task is forwarded to the object's successor, along the chain.

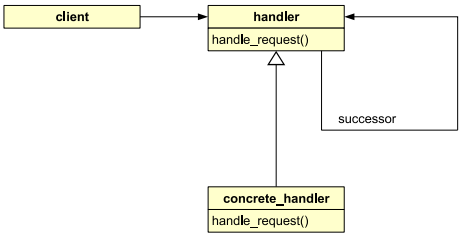

Fig. 15. Chain of Responsibility Pattern

The pattern found wide application, for example in help systems, in event handling frameworks or for exception handling. Its *Handler* class is known under synonyms like *Event Handler*, *Bureaucrat* or *Responder*.

Frequently, the pattern gets misused by delegating messages not only to children but also to the parent of objects. The *Hierarhical Model View Controller* (HMVC) pattern is one example for this. It causes unfavourable bidirectional dependencies (section 3.2) and leads to stronger coupling between the layers of a framework, because parent- and child objects then reference each other.

Observer Another pattern that found wide application is the *Observer* [10], an often-used synonym for which is *Publisher-Subscriber*. It provides a notification mechanism for all objects that registered as *Observer* at a *Subject* in whose state changes they are interested, leading to an automatic update of all dependent objects (figure 16).

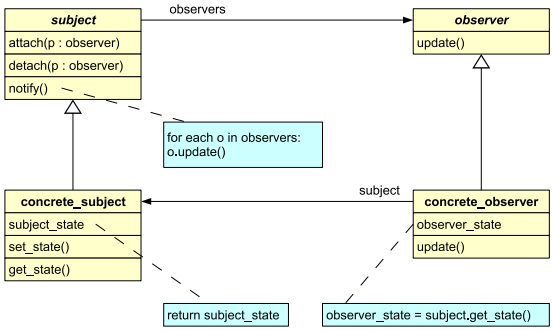

Fig. 16. Observer Pattern

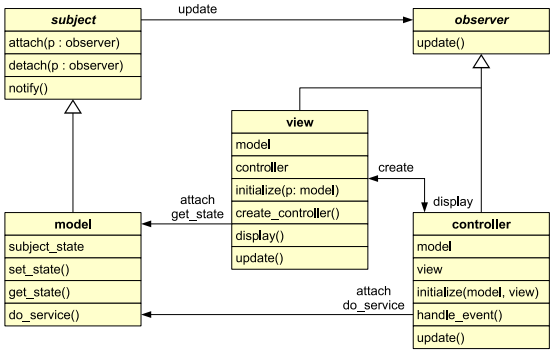

Fig. 17. MVC- using Observer Pattern

Similar notification mechanisms are used for *Callback* event handling in frameworks, where the framework core

calls functionality of its extensions. The *Model View Controller*-(MVC) uses the *Observer* pattern to let the model notify its observing views about necessary updates (figure 17).

A disadvantage of the *Observer* pattern is that it relies on bidirectional dependencies (section 3.2), so that circular references can occur, when a system is not programmed very carefully.

Idiomatic An *Idiom* is a pattern on a low abstraction level. It describes how certain aspects of components or the relations between them can be implemented using the means of a specific programming language. Idioms can such be used to describe the actual realization of design patterns. Besides the *Counted-Pointer* pattern, Buschmann [3, p. 377] also categorizes *Singleton*, *Template Method*, *Factory Method* and *Envelope-Letter* [6] as *Idiom*.

Template Method The *Template Method* pattern [10], also called *Hook Method*, is an abstract definition of the *Skeleton* of an algorithm. The implementation of one or more steps of that algorithm is delegated to a sub class (figure 18).

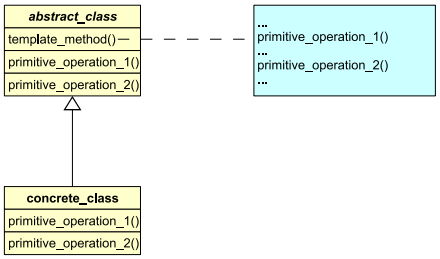

Fig. 18. Template Method Pattern

Counted Pointer The *Counted Pointer* pattern [3] supports memory management in the *C++* programming language, by counting references to dynamically created objects (figure 19). That way, it can avoid the destruction of an object through one client, while still being referenced by other clients. Also, it helps avoiding memory leaks by cleaning up forgotten objects.

Singleton Whenever an object-oriented system wants to ensure that only one instance of a certain class exists, the *Singleton* pattern [10] can be used. It essentially is a class which encapsulates its instance's data and provides global access to them, via *static*, sometimes called *class* methods (figure 20).

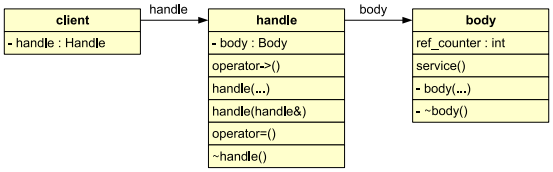

Fig. 19. Counted Pointer Pattern

A *Registry* object as described by Fowler [9] often uses the *Singleton* pattern, to be unique and to become globally accessible. Similarly do many so-called *Manager* objects, for example change managers which are also responsible for the caching of objects.

Global, that is static access – the main purpose of the *Singleton* pattern, is its main weakness, at the same time (section 3.3). One obvious solution to avoid singleton objects could be to forward global information from component to component, possibly using an own *Lifecycle Method*, as described in Apache Jakarta's *Avalon Framework* [2]. This approach, however, might bring with a rather large number of parameters to be handed over. The search for further alternatives therefore remains a topic of interest.

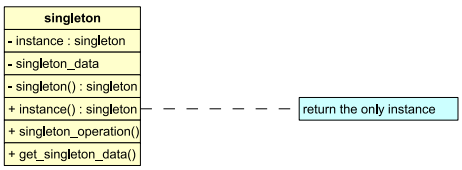

Fig. 20. Singleton Pattern

3 Problems

This section does not describe further patterns. Instead, it wants to come back to reflective- and other mechanisms as described in section 2 before, and elaborate their negative effects a bit more. Although the first of the following three reviews concentrates on the example of *Java*, many points surely count for other *Object Oriented Programming* (OOP) languages as well.

3.1 Broken Type System

Languages like *Smalltalk* or the *Common Lisp Object System* (CLOS) offer reflective mechanisms [3]. The *C++ Standard Library*, also known as *libstdc++* [18], has a *type_info* class providing meta information that *C++* innately does not have.

In the *Java* framework [17], finally, the basic *java.lang.** package contains the top-most super class *java.lang.Object*. All other classes in the framework inherit from it. Additionally, the package contains a class *java.lang.Class* which, among others, keeps reflective (meta) type information about a *Java* class':

- Package
- Name
- Superior Class
- Interfaces
- Fields
- Methods
- Constructors
- Modifiers
- Member Classes

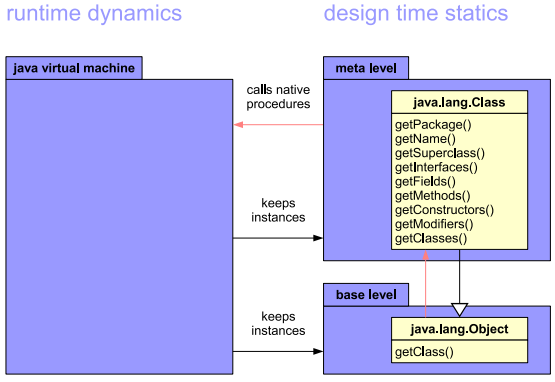

Fig. 21. Java Type System

Via the *getClass()* method which they inherit from *java.lang.Object* (figure 21), all Java classes have access to that reflective information in their meta class. The meta class *java.lang.Class* itself uses so-called *native* methods to access the information in the *Java Virtual Machine* (JVM).

The JVM operates on a level underneath the actual application, close to the *Operating System* (OS). It interprets the Java application source code and resolves all object-oriented- into procedural structures, and finally low-level system instructions. All runtime objects, that is class instances, are hold in dynamic structures internal to the JVM. That is why *native* methods need to be used to access and change the runtime structure or behaviour of objects.

One problem that becomes obvious when inspecting figure 21 is the existence of a *Bidirectional Dependency*, also called *Circular Reference*. The two sub dependencies causing it are:

1. *Inheritance* of *java.lang.Class* from *java.lang.Object* which is due to the rule that all Java classes need to inherit from the top-most framework class
2. *Association* from *java.lang.Object* to *java.lang.Class* which enables every object to access its meta class using the *getClass()* method

The avoidance of circular references is one of the most basic principles of computer programming (section 3.2). The disadvantage of bidirectional dependencies between meta- and basic level is also mentioned by Buschmann [3]. If meta classes in the kind of *java.lang.Class* define the structure and behaviour of all basic classes inheriting from *java.lang.Object*, then those meta classes in turn should *not* themselves inherit from *java.lang.Object*.

Another problem is the mixed and redundant storage of meta information which Jonathon Tidswell [14] even calls a *Broken Type System*. He writes: *A careful examination of the classes in the standard runtime will show that they are not strictly instances of java.lang.Class (hint: statics)*. Gilbert Carl Herschberger II [14] calls the separation of reflection and wrappers an *Inconsistent Design*. Java classes are based on many different kinds of type information:

- Structure applied by the JVM through the usage of the *class* keyword
- Meta information supplied by the *java.lang.Class* class
- Reflective information provided by *java.lang.reflect.**
- Wrapper classes for primitive types in *java.lang.**
- Dynamically created array classes, without having an array class file

The fact that the *java.lang.Class* class which is to provide meta information *about* classes is a *Class* itself is an antagonism. It is true that that meta class is made *final* to avoid its extension by inheriting subclasses. But correctly, it should not be a class at all.

Yet how can this paradoxon be resolved? Obviously, one of the two dependencies between *java.lang.Object* and *java.lang.Class* needs to be cut. But then either the *java.lang.Object* class would not be able to access its meta information anymore or the *java.lang.Class* class would not be available as runtime object to other polymorphic data structures. One solution could be to merge both classes, so that each object, by default, has the necessary methods to access its meta information. But as it turns out, this would not be a real solution, just a *Shift* of the problem to another level.

As mentioned above, the JVM keeps all instances (objects) in internal, dynamic structures. If objects were allowed to access these internal structures via native methods (procedures), a similar kind of bidirectional dependency, between the JVM and its stored objects, would occur.

One finally has to ask whether the usage and manipulation of meta information is really necessary at all! If objects did not have a *static* structure consisting of certain attributes and methods, as defined by the software developer at design time, but instead based on a uniform, *dynamically* changeable structure – the need to use reflective mechanisms might disappear. More research has to be done on this topic.

There are other Java-related points to be criticised. Although it is worth noting they exist, these are *not* explained in detail here, since this document wants to focus on general concepts. Gilbert Carl Herschberger II [14] mentions the problematic issue of *Pre-Conditions*, leading to corresponding *Assumptions*. After him, such work-arounds were necessary to break circular references in Java:

- Each JVM must pre-define an *Internal Meta Class*, implemented in machine code and *not* available as Java bytecode in a class file. The *java.lang.Class* as base meta class for all Java classes depends on that internal meta class and assumes its existance.
- A JVM pre-defines one *Primordial Class Loader*, implemented in machine code and resolved at compile-time. Since additional class loaders need to know their meta class when being created, they have to assume the primordial class loader exists so that, using it, their meta class can be created first.

Jonathon Tidswell [14] is of the opinion that there are a number of security issues related to the design of Java, for example:

- Global names not local references are used for security
- Wrappers and names are used for reflection

Even though most of the issues raised in this section are rather Java-specific, many of them apply to other programming languages as well. *Smalltalk* [20] and *CLU* [19], for example, make primitive types look like classes and do not need special *Wrapper* classes like Java. But when digging deep enough, one will find that this is *Syntactic Sugar*, as Peter J. Landin used to call additions to the syntax of a computer language that do not affect its expressiveness but make it *sweeter* for humans to use [5].

3.2 Bidirectional Dependency

Bidirectional References are a nightmare for every software developer. They cause *Inter-Dependencies* so that changes in one part of a system can affect multiple other parts which in turn affect the originating part, which may finally lead to cycles or even endless loops. Also, the actual program flow and effects of changes to a system become very hard to trace. Therefore, the avoidance of such dependencies belongs to the core principles of good software design.

A *Tree*, in mathematics, is defined as *Directed Acyclic Graph* (DAG), also known as *Oriented Acyclic Graph* [21]. It has a *Root Node* and *Child Nodes* which can become *Parent Nodes* when having children themselves; otherwise they are called *Leaves*. Children of the same node are *Siblings*. *A common constraint is that no node can have more than one parent.*, as [15] writes and continues: *Moreover, for some applications, it is necessary to consider a node's children to be an ordered list, instead of merely a set.* A graph is *acyclic* if every node can be reached via exactly one path, which then also is the shortest possible.

In computing, trees are used in many forms, for example as *Process Tree* of an *Operating System* (OS) or as *Object Tree* of an object-oriented application. They represent *Data Structures* in databases or file systems and also the *Syntax Tree* of languages.

The violation of the principle of the *Acyclic Graph* can lead to the same loops, also called *Circular References*, as mentioned above, which can result in the crossing of memory limits and is a potential security risk.

3.3 Global Access

A pure tree of instances in a computer's *Random Access Memory* (RAM) represents an unidirectional structure that permits data access along *well-defined* paths. Global access via static types, on the other hand, allows *any* instance to address data in memory *directly*, which not only complicates software development and maintenance, but, due to the uncontrollable access, is a potential security risk.

The usage of static objects accessible by any other part in a system is an *Anti Pattern* to *Inversion of Control* (IoC) [2], highly insecure and hence undesirable.

4 New Systematics

Section 2 used traditional proposals [3, 10] to systematize patterns and divided them according to the first categorization level shown in figure 1. The following sections will work out a new systematics, to classify patterns.

4.1 Human Thinking

The new classification is based on the idea of categorizing software patterns after the principles of *Human Thinking*, that is concepts of the logical *Mind*, as opposed to *Artificial Neural Networks* (ANN) that want to imitate the functioning of the physical *Brain*.

The corresponding concepts were first introduced in [12]. After an investigation of the fundamentals of human thinking, that is how human beings understand their surrounding real world by abstracting it in *Models*, that paper concludes that there were three basic activities of abstraction:

1. Discrimination
2. Categorization
3. Composition

By discriminating their environment, humans are able to share it into discrete *Items*. Items with similar properties can be classified into a common super *Category*. Any abstract model of the universe is just an illusion, being made up of yet smaller models, and nobody knows where this hierarchy really stops, towards microcosm as well as towards macrocosm. Therefore, the third and last kind of abstraction, namely composition, lets humans perceive the items in their environment as *Compound* of smaller items.

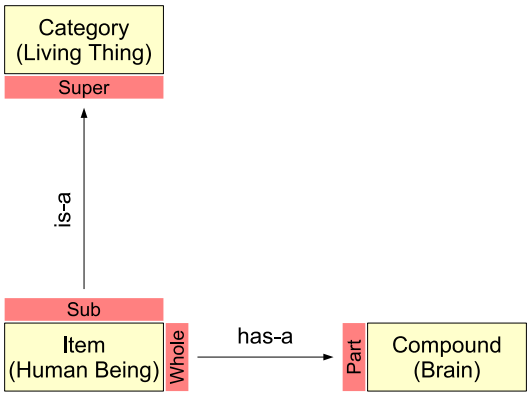

Fig. 22. Abstractions of Human Thinking [12]

The latter two activities of abstraction – categorization and composition – are based on special *Associations* (figure 22), between a *Super*- and a *Sub* model and between a *Whole*- and a *Part* model, respectively.

4.2 Categories

Most patterns heavily rely on associations, too. This paper therefore suggests to:

Take the Kind of Association as Criterion to sort patterns in a completely new way.

The opposite table shows a systematics of the new pattern categories with their equivalents in human thinking, some representative example patterns and a recommendation for their usage in software engineering. Patterns matching into more than one category are placed after the priority: *Recursion* over *Polymorphism*.

4.3 Recommendation

The first category *Itemization* (objectification) is the base of any modelling activity and clearly necessary.

The next three categories *1:1 Association*, *1:n Association* and *Recursion* are special kinds of associations that rely exclusively on *unidirectional* relations and result in a clean architecture which is why their usage is strongly recommended.

Category	Equivalent	Representative	Advice
Itemization	Discrimination	Command, Data Transfer Object, State, Memento, Envelope-Letter, Prototype	:-)
1:1 Association	Composition	Delegator, Object Adapter, Proxy (Surrogat, Client-/Server Stub), Wrapper, Handle-Body, Bridge	:-)
1:n Association	Composition	Whole-Part, View Handler, Broker (Mediator), Master-Slave, Command Processor, Counted Pointer, Chain of Responsibility	:-)
Recursion	Composition	Composite, Interpreter, Decorator, Linked Wrapper	:-)
Bidirectionalism	–	Observer (Callback, Publisher-Subscriber), Forwarder-Receiver, Chain of Responsibility, Visitor, Reflection	:-(
Polymorphism	Categorization	Template Method, Builder, Factory Method, Class Adapter, Abstract Factory (Kit), Strategy (Validator, Policy), Iterator (Cursor)	:-|
Grouping	Categorization	Layers, Domain Model, MVC	:-)
Global Access	–	Singleton, Flyweight, Registry, Manager	:-(

Bidirectionalism, on the other hand, is an *ill* variant of the three aforementioned categories and should be avoided wherever possible. Patterns in this category are one reason for endless loops and unpredictable behaviour since it becomes very difficult to trace the effects that changes in one place of a system have on others (section 3.2).

Polymorphism is a good thing. It relies on categorization and due to inheritance can avoid a tremendous amount of otherwise redundant source code. However, it also makes understanding a system more difficult, since the whole architecture must be understood before being able to manipulate code correctly. Unwanted source code changes caused by inheritance dependencies are often described with the term *Fragile Base Class Problem* [3, section *Layers*]. They are just the opposite of what inheritance was actually intended to be for: *Reusability* [11, Vorwort].

Grouping models is essential to keep overview in a complex software system. A very promising technology to support this are *Ontologies* [13]. A lot of thought-work has to go into them but if they are well thought-out, they are clearly recommended.

The habit of globally accessing models is banned since OOP became popular. However, it is not banned completely. Patterns like *Singleton* encapsulate and bundle global access but they still permit it. They disregard any dependencies and relations in a system, such are a security risk and reason for untraceable data changes. This paper sees the whole category of *Global Access* as potentially dangerous and can *not* recommend its patterns.

5 Summary and Future

This paper investigated current software pattern solutions, to find their common characteristics. Furthermore, some of the good and rather bad sides of classical patterns were mentioned. The paper does not deliver solutions to these criticisms; it merely gives an overall view on patterns.

Using ideas of the so-called *Cybernetics Oriented Programming* (CYBOP), namely *Human Thinking* and its forms of abstraction, the paper categorized patterns in a new systematics, consisting of eight groups. It thereby hopes to provide a different view on software systems and to help identify patterns with similar concepts. By sorting patterns into these groups, developers might be able to faster recognize their advantages and disadvantages.

The search for solutions to the above-mentioned problems needs to continue. The CYBOP project [24] aims at finding a way for *pattern-less* application programming. The idea is to apply necessary patterns just once, in the *Cybernetics Oriented Interpreter* (CYBOI), to free application developers from the burden of repeatedly figuring out suitable patterns. Instead, they shall be enabled to concentrate on modelling pure application- and domain knowledge, by writing systems in the *Cybernetics Oriented Language* (CYBOL), which is based on the *Extensible Markup Language* (XML). Future papers will report about this progress.

References

1. Christopher Alexander, Sara Ishikawa, Muray Silverstein, and et al. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York, 1977. https://www.patternlanguage.com/cgi-bin/patternl/order.py.
2. Federico Barbieri, Stefano Mazzocchi, and Pierpaolo Fumagalli. *Apache Jakarta Avalon Framework*. Apache Project, 2002. http://avalon.apache.org/.
3. Frank Buschmann, Regine Meunier, Hans Rohnert, and et al. *Pattern-orientierte Softwarearchitektur. Ein Pattern-System.* Addison-Wesley, Bonn, Boston, Muenchen, 1. korr. nachdruck 2000 edition, 1998. http://www.aw.com/.
4. Jason Cai, Ranjit Kapila, and Gaurav Pal. Hmvc: The layered pattern for developing strong client tiers. *Java World*, July 2000. http://www.javaworld.com/javaworld/jw-07-2000/.
5. Collaborating contributors from around the world. Wikipedia – the free encyclopedia. Web Encyclopedia, October 2004. http://www.wikipedia.org.
6. J. O. Coplien. *Advanced C++ – Programming Styles and Idioms*. Addison-Wesley, Bonn, Boston, Muenchen, 1992.
7. Design Matrix – Systems and Product Design, http://www.designmatrix.com/bionics/. *Design Matrix*.
8. Martin Fowler. *Analysis Patterns. Reusable Object Models.* Addison-Wesley, Boston, Muenchen, 1997. http://www.aw.com.
9. Martin Fowler and et al. *Patterns of Enterprise Application Architecture (Information Systems Architecture)*. Addison-Wesley, Boston, Muenchen, 2001-2002. http://www.aw.com.
10. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Gang Of Four). *Design Patterns. Elements of reusable object oriented Software.* Addison-Wesley, Bonn, Boston, Muenchen, 1st edition, 1995. http://www.aw.com.
11. Volker Gruhn and Andreas Thiel. *Komponentenmodelle. DCOM, JavaBeans, Enterprise JavaBeans, CORBA.* Addison-Wesley, Boston, Muenchen, 2000. http://www.aw.com.
12. Christian Heller. Cybernetics oriented language (cybol). *IIIS Proceedings: 8th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2004)*, V:178–185, July 2004. http://www.iiisci.org/sci2004 or http://www.cybop.net.
13. Christian Heller, Torsten Kunze, Jens Bohl, and Ilka Philippow. A new concept for system communication. *Ontology Workshop at OOPSLA Conference*, October 2003. http://swt-www.informatik.uni-hamburg.de/conferences/oopsla2003-workshop-position-papers.html.
14. Gilbert Carl Herschberger II, Jonathon Tidswell, Stephen Crawley, and et al. The jos-general mailing list. jos-general@lists.sourceforge.net.
15. Denis Howe. Free on-line dictionary of computing (foldoc). Internet Database, September 2003. http://wombat.doc.ic.ac.uk/foldoc/Dictionary.gz, http://www.foldoc.org/.
16. Cunningham & Cunningham Inc. Portland pattern repository, 2004. http://c2.com/cgi/wiki?PortlandPatternRepository.
17. Sun Microsystems Inc. The java programming language. the java development kit (jdk). http://java.sun.com.
18. C++ standard library (libstdc++), 2004. http://gcc.gnu.org/libstdc++/.
19. Barbara Liskov and et al. The clu programming language, 2004. http://www.pmg.lcs.mit.edu/CLU.html.
20. Peter William Lount. Smalltalk.org, 2004. http://www.smalltalk.org.
21. National Institute of Standards and Technology (NIST). Dictionary of algorithms and data structures. Online Dictionary, July 2004. http://www.nist.gov/dads/.
22. Object Management Group (OMG). Unified modeling language (uml) specification, 2001. http://www.uml.org.
23. Margarete Payer and Alois Payer. Computervermittelte kommunikation / computer mediated communication (cmc). Lecture Notes on Website, November 2002. http://www.payer.de/cmclink.htm.
24. CYBOP Project. Cybernetics oriented programming (cybop), 2002-2004. http://www.cybop.net.
25. Andrew Stuart Tanenbaum. *Computernetzwerke*. Pearson Studium, Muenchen, 3rd edition, 2000. http://www.pearson-studium.com.
26. Andrew Stuart Tanenbaum. *Modern Operating Systems*. Prentice-Hall, New Jersey, London, Sydney, 2nd edition, 2001. http://www.prentice-hall.com.