Software Engineering Design - Summary

Gina Häussge

9th February 2006

Contents

1 Design Principles **2**

- 1.1 External and internal criteria . . . 2
- 1.2 Modularity . . . 2
  - 1.2.1 Five criteria . . . 2
  - 1.2.2 Five rules . . . 3
  - 1.2.3 Five principles . . . 4
- 1.3 Coupling . . . 4
- 1.4 Cohesion . . . 4

2 Design Patterns **5**

- 2.1 What is a pattern? . . . 5
- 2.2 Pattern categories . . . 5
- 2.3 Abstract Factory . . . 6
- 2.4 Prototype . . . 6
- 2.5 Singleton . . . 7
- 2.6 Adapter . . . 7
- 2.7 Composite . . . 7
- 2.8 Decorator . . . 8
- 2.9 Flyweight . . . 8
- 2.10 Observer . . . 9
- 2.11 State . . . 9
- 2.12 Strategy . . . 10
- 2.13 Template Method . . . 10
- 2.14 Visitor . . . 11
- 2.15 Layers . . . 11
- 2.16 Microkernel . . . 11
- 2.17 Model-view-controller . . . 12
- 2.18 Pipes and Filters . . . 12
- 2.19 Blob/God Class . . . 12
- 2.20 Lava Flow/Dead Code . . . 13
- 2.21 Poltergeist/Gypsy . . . 13

3 Refactorings **14**

4 Frameworks **14**
4.1 Patterns versus Frameworks 14
4.2 Types of Frameworks 15
4.3 Framework integration 15

5 Aspect-Oriented Programming **15**
5.1 AspectJ 16
5.2 CaesarJ 17

6 Heuristics **17**
6.1 The God Class problem 17
6.2 The Proliferation of Classes Problem 17
6.3 Heuristics for the Uses Relationship 17

1 Design Principles

1.1 External and internal criteria

External criteria are

- **Correctness:** The ability to perform the task as specified by the specification
- **Robustness:** The ability to react appropriately to abnormal conditions / conditions outside the specification
- **Efficiency:** The ability to demand as few as possible from hardware resources
- **Extensibility:** The ease of adapting to changes of the specification
- **Reusability:** The ability of software elements to serve for the construction of other systems

Internal criteria (modularity, readability, ...) are used to build up a good base for external criteria.

1.2 Modularity

Bertrand Meyer defines five criteria, five rules and five principles to ensure modularity:

1.2.1 Five criteria

- **Decomposability:** The software structure helps to decompose the problem into smaller problems, whose solving implementations are then connected among each other but also independent enough to be solved separatly (*Divide et impera*, top-down design, ...). Counter example: Global initialisation modules.

- **Composability:** A design method that favours the production of software elements that may freely be recombined to form new (and possibly totally different) systems. Directly connected to the goal of *reusability*. Example: Unix shell commands piped together to form new commands. Counter example: Preprocessors to extend code, usually not combinable.
- **Understandability:** A human reader can understand each module without the need to know the others or at most a few of the others. Counter example: Sequential dependencies (Piping, ...).
- **Continuity:** A small change in the problem specification triggers a change of only one or at most a few modules. Example: Usage of symbolic constants instead of hardcoded values, Uniform Access Principle. Counter example: Static arrays, design that depends on the physical representation of data.
- **Protection:** An abnoral condition occuring at run time in one module will remain confined to that module or only propagate to a small number of adjacent modules in the worst case. Example: Validating input at source. Counter example: Undisciplined exceptions (use with extreme caution, don't let them ripple through the whole system).

1.2.2 Five rules

- **Direct Mapping:** The modular structure of the implementation follows the moular structure of the initial design. Follows from: Continuity (Direct Mapping makes it easier to limit the impact of changes), Decomposability.
- **Few Interfaces:** Every module should communicate with as few others as possible. Follows from: Continuity & Protection (too many relations between modules hinder the limitation of the impact of a change or an exception). Composability (a module may not depend on too many other modules to be usable in another environment).
- **Small Interfaces:** Two communicating modules should exchange as little information as possible. Counter example: Common block in Fortran, every accessing module could damage the data or misuse it. Problems ahead: Nested block structures.
- **Explicit Interfaces:** Whenever two modules A and B communicate, this must be obvious from the text of A or B or both. Follows from: Decomposability & Composability (when combining modules with each other, or decomposing a module into smaller submodules, any outside connection should be clearly visible), Continuity (elements that are effected by a change should be spotted easily), Understandability (A cannot be understandable by itself if B can influence it). Counter example: Data sharing.
- **Information Hiding:** Only the interface of the module is public. The goal is the separation of function from implementation, encapsulation.

1.2.3 Five principles

- **Linguistic Modular Units principle:** Modules must correspond to syntactic units in the language used. In the case of programming languages, modules should be separately compilable. Follows from: Continuity, Direct Mapping (maintain a clear correspondence between the structure of the model and the structure of the solution), Decomposability (every decomposed unit needs to be a well-delimited, well-defined syntactic unit), Composability, Protection (modules need to be syntactically limited in order to be controllable in the scope of errors).

- **Self-Documentation principle:** Make all information about the module part of the modules itself. Make programs look like design! Follows from: Understandability (obvious), Continuity (treating software and documentation as different entities would make it difficult to keep both in sync on changes).

- **Uniform Access principle:** All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation. Uniform access hides complexity and avoids reimplementation. Example: object.getPrice() instead of direct access via object.price makes it possible to calculate the price on the fly if necessary without making a change to the clients necessary.

- **Open-Closed principle:** Modules should be both open and closed. Open: The module is still available for extension. Closed: The modules is available for use by other modules, it may be compiled, stored away into a library, ... and has a stable interface. Inheritance is the key to accomplish this: Allows to extend a closed module by simply inheriting from it, adding new functionality on base of the old one.

- **Single Choice principle:** Whenever a software system must support a set of alternatives, one and only one moduler in the system should know their exhaustive list. Follows from: Information Hiding, Open-Closed principle (include more alternatives in a subclass).

1.3 Coupling

Coupling (or dependency) measures the degree of interconnectness of a system. The more modules a single modules depends on, the higher its coupling. Low coupling is desirable as it makes the recombination and maintenance of modules easier (Continuity, Protection, Understandability, Reusability). Too little coupling though is not desirable: A system is made of communicating modules, too little coupling leads to a few highly active modules doing all the work instead of a fair distribution among the whole system.

1.4 Cohesion

Cohesion measures how related the responsibilities of a single module are. The higher the cohesion, the better. Modules with low cohersion are hard to comprehend, hard to reuse and difficult to maintain as they are easily affected by changes.

There are different types of cohesion:

- Coincedental cohesion: The different parts of the module have no significant relationship. Example: A module which groups a set of frequently used utility functions.
- Logical cohesion: Elements of a class perform one kind of logical function. Example: A module providing methods to communicate with hardware or the user.
- Temporal cohesion: Parts of a module are grouped by when they are processed. Example: A method providing some standard way of handling an exception, e.g. logging to a file, displaying an error message, ...

A module with very low cohesion would be one responsible for many things in different functional cases. Low cohesion means a module is responsible only for one task, but this task is very complicated, making a split of the module into several submodules each handling a part of the task recommendable. A module with moderate cohesion has moderate responsibilities in a few different areas logically related to the class concept but not to each other. High moderation finally indicates a module with lightweight responsibilities in one area and appropriate communcation with other modules in order to fulfill a more complicated task.

2 Design Patterns

2.1 What is a pattern?

> A 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.
>
> – *Christopher Alexander*

2.2 Pattern categories

- **Creational Patterns:** Deal with object creation mechanisms.
- **Structural Patterns:** Define relationship between entities.
- **Behavioural Patterns:** Identify common communication patterns between objects and realise them.
- **Architectural Patterns:** Describe typical software architectures.
- **Anti Patterns:** Counter patterns, describing what you should *not* do.
- **Idioms:** Common structures in programming below the design level (e.g. while loops).

2.3 Abstract Factory

Type: Creational

An abstract factory defining methods for creating other objects, but not implementing them. Example: Look and feel factory for widgets, the abstract factory would specify methods for creating a window or a button, certain subclasses then would return objects according to the defined style.

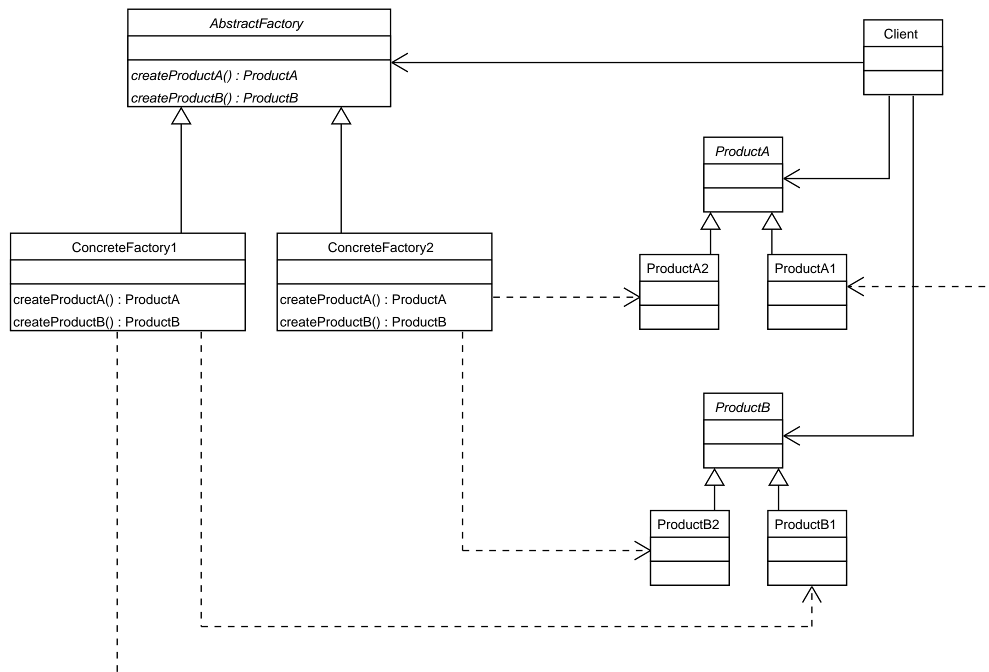

Figure 1: Structure of the Abstract Factory pattern

2.4 Prototype

Type: Creational

The class implements a method that can be used for cloning the object, including its internal state. Suitable for example to enable a framework method to create a certain type of object without it having the knowledge about the object to create - the target object is simply given as a prototype and the new object is created by using its clone method.

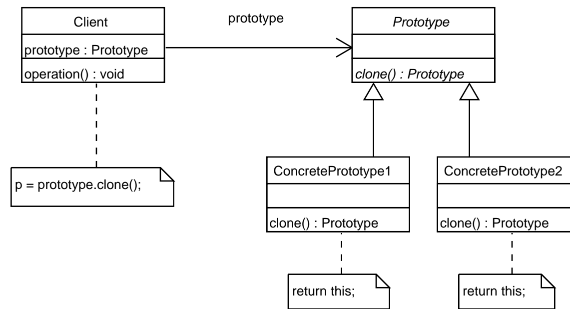

Figure 2: Structure of the Prototype pattern

2.5 Singleton

Type: Creational

Ensures that at most one instance of a given class is existing at any given time. The class manages a static field of its own type holding the only instance of it. The constructor is declared protected, to get the instance of the class a static method instead is used which either constructs a new instance if none already exists and saves it to the instance field, or returns the already existing instance.

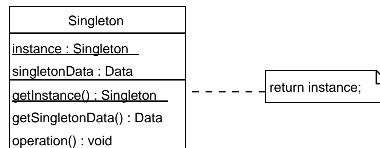

Figure 3: Structure of the Singleton pattern

2.6 Adapter

Type: Structural

Used for connecting the interface of one module with the interface of another module. Makes it possible to let module operate that would not be able to do so otherwise due to incompatible interfaces.

2.7 Composite

Type: Structural

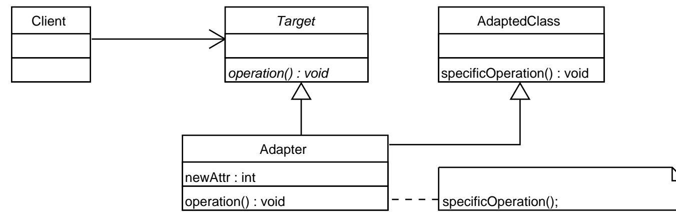

Figure 4: Structure of the Adapter pattern - Class Adapter

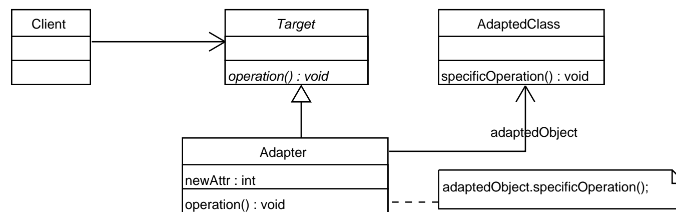

Figure 5: Structure of the Adapter pattern - Object Adapter

Composes objects into tree structures to represent part-whole hierarchies. Allows clients to treat individual objects and objects composed of others identically. Inheritance is organised hierarchically as well. Example: Expression tree.

2.8 Decorator

Type: Structural

Dynamically extends an object with abilities. Example: Border or scrollbars around a widget.

2.9 Flyweight

Type: Structural

Uses object of a small granularity to efficiently manage a large number of them. Differentiates between extrinsic and intrinsic attributes, the extrensic ones being context sensitive and therefore managed by the client using the object, which manages its instrinsic attributes. Example: A document consists of many characters. A single character would have its value (e.g. 'c') as an intrinsic attribute, but its position in the text and formatting attributes like bold or italic as an extrensic attribute.

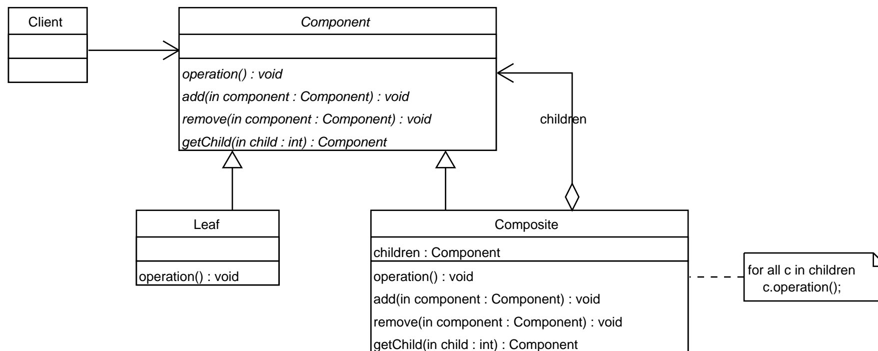

Figure 6: Structure of the Composite pattern

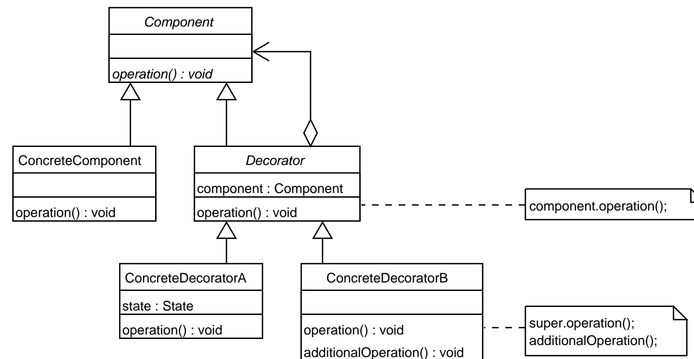

Figure 7: Structure of the Decorator pattern

2.10 Observer

Type: Behavioural

One or more objects (called observers or listeners) either are registered or register themselves at another object in order to be notified when a certain event is raised. Example: In order to repaint a gui component upon changes on its data, it might implement an observer pattern to notify the drawing routines of the gui to initiate the repaint.

2.11 State

Type: Behavioural

Makes it possible for an object to change its behaviour during runtime upon the change of its internal change. It looks like the object changes its class during runtime.

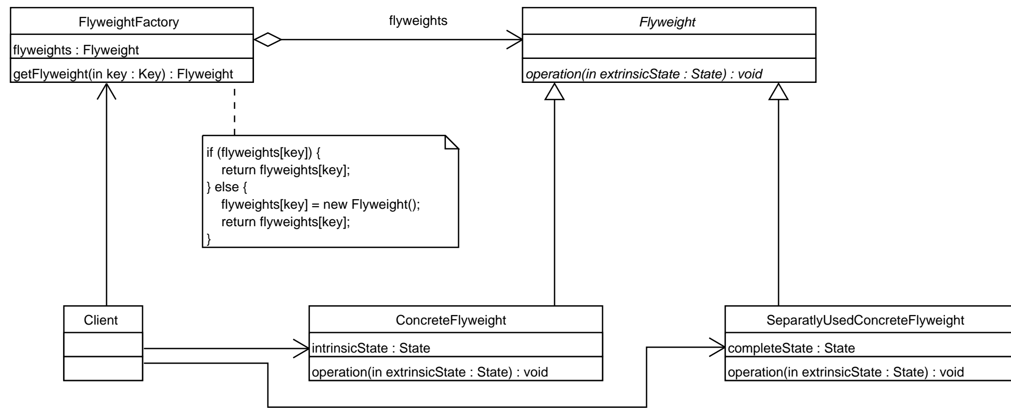

Figure 8: Structure of the Flyweight pattern

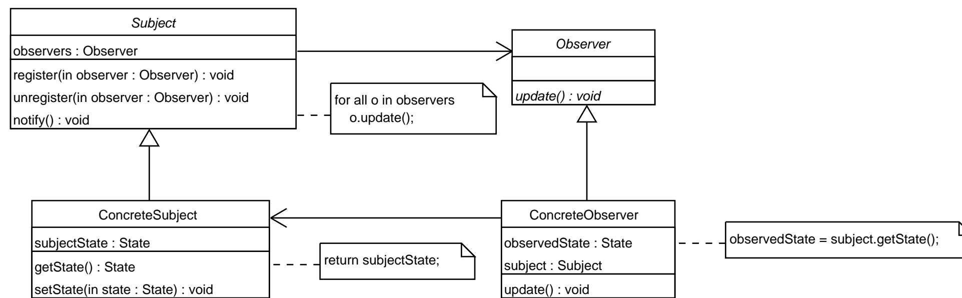

Figure 9: Structure of the Observer pattern

2.12 Strategy

Type: Behavioural

Define a family of algorithms by encapsulating each one and making it interchangable, makes it possible to vary the algorithm independently from the client using it.

2.13 Template Method

Type: Behavioural

Defines a skeletal structure of operations and delegates the implementation of the single operations to subclasses. This allows to redefine certain parts of the functionality of a module without changing its structure, therefore making it possible to use two modules with differently implemented operations from the same client.

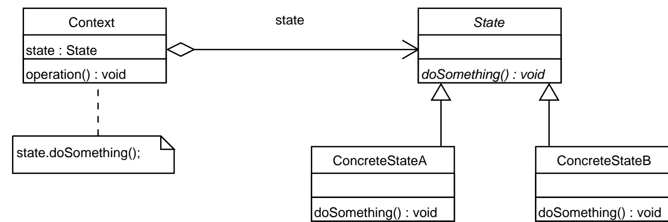

Figure 10: Structure of the State pattern

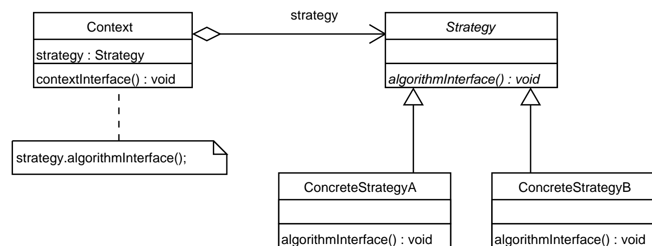

Figure 11: Structure of the Strategy pattern

2.14 Visitor

Type: Behavioural

Encapsulates an operation that needs to be run on the elements of a structure in an object. The visitor pattern allows to define new types of operations on a set of elements without the need to redefine the set of elements.

2.15 Layers

Type: Architectural

Solves the problem of a system consisting of both low- and high-level issues by splitting them into separate layers, which communicate to each other via well-defined interfaces. Makes it possible to easily exchange parts of a system (e.g. data sources). A Layer only communicates with its next neighbour.

2.16 Microkernel

Type: Architectural

Separates minimal functionality from extended functionality, encapsulates the minimal functionality of the system and serves as a socket for extension. There are four

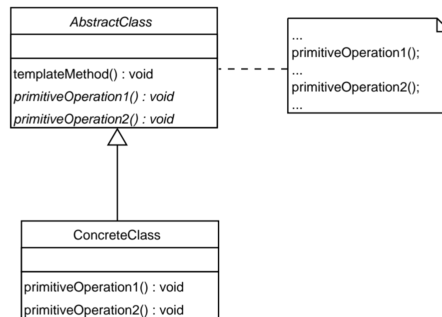

Figure 12: Structure of the Template Method pattern

major components: Internal Servers (which extend the microkernel functionality), External Servers (which use the microkernel), Adapters (which hide system dependencies from the clients and invoke methods of the External Servers on behalf of the clients) and finally the Client itself.

2.17 Model-view-controller

Type: Architectural

Separates an application's data model, user interface and control logic into three distinctive components. This make it possible to modify one of those components with a very low impact on the others.

2.18 Pipes and Filters

Type: Architectural

Divides a systems task into a sequence of processing stages. At the beginning there is a data source (which delivers the input data into the pipeline), which is connected to a data sink (which consumes the output) via a series of pipes (which transfers, buffers and synchronises data between its in- and output) and filters (which executes a certain function on the incoming data and outputs the result of this applied function). Example: InputStream and OutputStream in Java, concatenation of unix shell commands.

2.19 Blob/God Class

Type: Anti

God Classes actually have a procedural design and consist of masses of methods with a low cohesion. Consequences are bad extendability, bad reusability and a complication

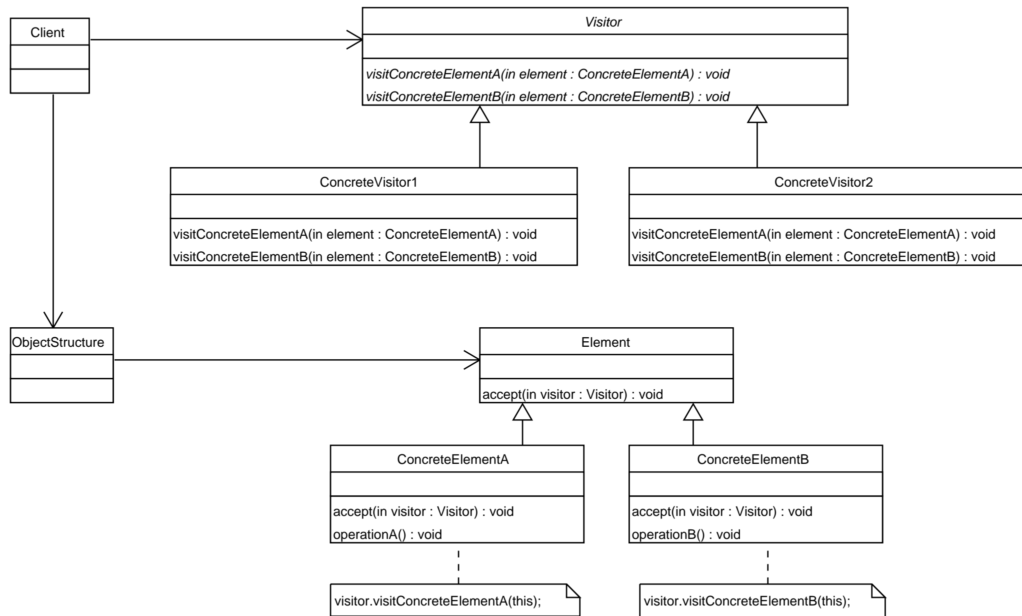

Figure 13: Structure of the Visitor pattern

of debugging and testing the code. Reasons are "organic growth" of classes, adding functionality to a single class instead of extracting it to a new one.

2.20 Lava Flow/Dead Code

Type: Anti

Mostly in systems which started out as research projects, Dead Code consists of not fitting fields and code blocks, big blocks of code which was commented out without an explanation, and outdated interfaces. Code like this makes testing and verification hard to impossible, has a negative impact on performance and resources and tends to spread from project to project due to copy-and-paste. Causes can be the extension of an existing system without further planning or reconfiguration of an old architecture without removing the now unused parts.

2.21 Poltergeist/Gypsy

Type: Anti

Classes with very limited responsibilities, short lifed instances and a lack of states. Actually they are redundant and only a waste of resources and energy. Reasons: Not a

very profound understanding of OO-concepts, specification requirements or relicts of a solved God Class.

3 Refactorings

- **Extract Method:** Use on code fragments that can better be summarised into a method with a descriptive name.
- **Encapsulate Field:** Use on public fields, make them private and instead create access methods for getting and setting the value.
- **Rename:** Rename a field or a method if its not clear on first sight what it does.
- **Move:** Move functionality to the fitting classes where necessary.
- **Replace Type Code with State/Strategy:** Instead of a type code you can't make unnecessary by using subclasses, use a state object instead.
- **Replace Conditional with Polymorphism:** Use on conditionals which select different behaviour depending on the type of an object: Move the different possible behaviours as methods into subclasses of the original class and declare the original method abstract.
- **Form Template Method:** Two methods in subclasses share rather similar steps in the same order and only differ in a small number of steps. Extract the different steps into methods with the same signature and move the rest of the steps up with the new methods declared as abstract.

4 Frameworks

A framework is a collection of cooperating classes that together define a generic or template solution to a family of domain specific requirements. It should dictate the overall structure of a family of applications by decribing how the different responsibilities are shared among the modules and how the components interact with each other, while also not going into too much detail about the implementation of the hot spots of the framework. Usually, the Template Method pattern is used a lot in frameworks, making it possible to redefine the hot spots of the framework via inheritance.

In a framework, the control is inverted: The framework is the main program and uses the user defined functionality via its hot spots, it dictates the control flow.

4.1 Patterns versus Frameworks

- Patterns are smaller than frameworks: A framework generally include realisations of one or more design patterns.

- Patterns are more abstract than frameworks: Patterns define solutions for design issues and use code fragments only for illustration - frameworks on the other hand provide implementations that can be reused as is.
- Patterns are less specialised than frameworks: A framework is a solution that applies to a specific problem domain. Patterns are design techniques that apply to any OO software.

4.2 Types of Frameworks

- **White-Box Frameworks:** Customised by subclassing existing framework classes and providing concrete implementations. There are two types of clients: “customising clients”, which customise the framework for specific needs sing inheritance, and “use clients”, which use the abstract parts of the framework.
- **Black-Box Frameworks:** Customised by filling in parameters or plugging together single components (which must be fully implemented). Limitation of possible combinations though, as only such things are possible that were made possible by the framework developers. Again there are two types of clients, those that customise the framework via configuration/parameters, and those which use it.
- **Grey-Box Frameworks:** Frameworks that use both parameterisation and inheritance to customise the framework’s functionality. Usually a framework goes through a number of shades from white to black during its lifetime.

4.3 Framework integration

- When integrating more than one framework, there are problems to be expected due to the inversion of control, namely both frameworks claiming control over the application. A solution would be to use threading, giving each framework its own thread to control.
- Integrating with legacy code can prove difficult when the framework relies on inheritance for customisation as well. The use of the adapter pattern is recommended in such situations.
- Frameworks which share the same real world components though in different representations will cause problems.
- Different architectural models in frameworks make it difficult to integrate them.

5 Aspect-Oriented Programming

In every complex system you will sooner or later come across cross cutting concerns, concerns that exist in many different modules of the system. Using OOP such cross cuts are spread througout the system, usually implemented by using redundant code.

A necessary change then ripples through the whole system. It would be better if the implementation reflected the design here, if the cross-cuts where defined in one location and could easily changed at one single location as well. This is what aspect oriented programming introduces.

Some definitions:

- **Aspects:** Aspects are units of modularity for cross cutting concerns and consist of point cuts and advices applied upon them.
- **Join Points:** Join points are places in the system where cross cutting concerns might hook into, e.g. method calls.
- **Point Cuts:** Points cuts are subsets of all join points in a system that satisfy a special query, e.g. all method calls to methods of a special class.
- **Advices:** Advices are the implementation of the functionality of the cross cutting concerns and are applied at join points defined by a point cut.

5.1 AspectJ

AspectJ is a bytecode compatible extension of Java which introduces aspects. It defines the following syntax to define pointcuts:

- call, execution: Matches call/execution of methods and constructors via match of signature.
- get, set: Matches get/set of fields via match of signature.
- handler: Matches exception handler execution via match of handled exception.
- initialization, staticinitialization: Matches class (static) initialiser execution via match of class name.
- within, withincode: Matches any join point that is nested within class or method body of matching name.
- cflow, cflowbelow: Matches join points within the control flow of the join point matched by the contained point cut.
- this: Matches if current object is of given type.
- target: Matches if target object is of given type.
- args: Matches if arguments are of given type.

For a quick reference of how to create pointcuts and how to implement aspects, see the AspectJ quick reference at [1].

5.2 CaesarJ

- Extension of Java, CaesarJ code compiles to regular bytecode
- Introduces aspect oriented programming techniques (AspectJ style pointcuts, dynamic aspect deployment, bindings) and extended composition features (virtual classes, mix-in composition, dependent types)
- Advantages of virtual classes: No need to redefine inheritance relationships, virtual classes can use the fields and methods of their ancestors, functionality defined in the overriden virtual classes can be accessed without typecasts, overriden methods in newly defined classes can again be overriden in their subclasses.
- Declaration caesar class: `public cclass ClassName`
- Mix-in composition: `public cclass Class extends ClassA & ClassB { }`

For a quick reference of CaesarJ's syntax see [2].

6 Heuristics

6.1 The God Class problem

1. Distribute the system intelligence as uniformly as possible to that the top most classes share the work equally.
2. Many accessor methods in a class imply related data and behaviour not being kept in one place.

6.2 The Proliferation of Classes Problem

1. Only model classes, not roles.
2. Eliminate irrelevant classes from your system: Classes without a meaningful behaviour, classes outside of the system, only sends messages into the system and does not get any back.
3. Be sure not to turn operations into classes. Classes with verbs as names are highly suspicious. Instead maybe migrate the piece of behaviour into another class.

6.3 Heuristics for the Uses Relationship

1. Minimise the number of classes a class collaborates with. Maybe replace groups of classes by a class that contains the group where appropriate.
2. If a class A has a field of type B, it should send messages to this field. Prohibits get/set access to the field only, also disallows orphaned fields. Exception: Container classes.

3. Most of the methods defined in a class should use most of the fields most of the time.
4. Classes should not contain more fields than the short time memory of the developer can handle (usually 6, scientifically 7 ± 2).
5. Distribute system intelligence vertically.
6. Objects contained in the same class should not have a uses relationship between them.

List of Figures

1	Structure of the Abstract Factory pattern	6
2	Structure of the Prototype pattern	7
3	Structure of the Singleton pattern	7
4	Structure of the Adapter pattern - Class Adapter	8
5	Structure of the Adapter pattern - Object Adapter	8
6	Structure of the Composite pattern	9
7	Structure of the Decorator pattern	9
8	Structure of the Flyweight pattern	10
9	Structure of the Observer pattern	10
10	Structure of the State pattern	11
11	Structure of the Strategy pattern	11
12	Structure of the Template Method pattern	12
13	Structure of the Visitor pattern	13

References

[1] AspectJ Quick Reference
http://www.eclipse.org/aspectj/doc/released/progguide/quick.html
Accessed: 08. February 2006

[2] CaesarJ Basic Concepts
http://caesarj.org/index.php/ProgrammingGuide/BasicConcepts
Accessed: 09. February 2006

[3] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns.* Addison Wesley 1995

[4] M. Fowler. *Refactoring. Improving The Design Of Existing Code.* Addison Wesley 2000

[5] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall 1997