Design Patterns And Refactoring in Functional Languages

by Roman Boiko

A thesis submitted to the
School of Graduate Studies
(Software for Automated Systems)
in partial fulfilment of the
requirements for the degree of
Master of Science
Faculty of Computer Sciences
National University of "Kyiv-Mohyla Academy"

Approved:

_____

Candidate of Physics and Mathematics,
Associate Professor
Volodymyr Boublik, Major Thesis Adviser

Kyiv 2012

Abstract

The purpose of this study was to collect in one research a set of methods of functional code refactoring using design patterns. Another aim was to find out whether mathematical theory can be applied to the most "mathematical" programming paradigm with aim to create common design solutions/optimise functional code. Finally, comparison of well-known design patterns from object-oriented paradigm to functional approach was done in the thesis.

The results of the study were organised as a set of patterns and refactoring methods and results of their application in project - XSLT transformer.

It was discovered that functional paradigm does not have most of problems that OOP patterns try to solve. As a result - applications written in functional languages could be much less complex and verbose than same ones in OOP.

The principal conclusion was that it is much easier to write/maintain software written in pure functional languages. It is the result of the fact that we don't need massive amount of code to realise pattern-like behaviour.

Contents

Abstract **ii**

List of Tables **vi**

1 Introduction **1**

2 Review of the Literature and Research **8**

2.1 The most fundamental works 8

2.2 Works supporting idea that we don't need patterns in FP 9

2.3 Existing descriptions of FP patterns 11

2.4 OOP And FP Cooperation 12

2.5 Refactoring and code structure in FP 13

3 OO Patterns To Functional **15**

3.1 Purpose of Design Patterns - Intro 15

3.2 Creational design patterns 18

3.2.1 Abstract Factory 18

3.2.2 Builder 19

3.2.3 Factory Method 20

3.2.4 Prototype 21

3.2.5 Singleton 21

3.3 Structural design patterns 22

3.3.1 Adapter . . . 23
3.3.2 Bridge . . . 24
3.3.3 Composite . . . 25
3.3.4 Decorator . . . 26
3.3.5 Facade . . . 26
3.3.6 Flyweight . . . 27
3.3.7 Proxy . . . 28
3.4 Behavioral design patterns . . . 29
3.4.1 Chain of responsibility . . . 30
3.4.2 Command . . . 32
3.4.3 Interpreter . . . 33
3.4.4 Iterator . . . 34
3.4.5 Mediator . . . 35
3.4.6 Memento . . . 36
3.4.7 Null Object . . . 37
3.4.8 Observer . . . 38
3.4.9 State . . . 38
3.4.10 Strategy . . . 39
3.4.11 Template method . . . 40
3.4.12 Visitor . . . 43
3.5 Criticism . . . 46

4 Pure Functional Patterns **48**
4.1 Intro . . . 48
4.2 Transactional Computation . . . 50
4.3 Catamorphism and Anamorphism . . . 50
4.4 Hylomorphism . . . 51
4.5 Other FP patterns . . . 53

5 Patterns And Refactoring In Project **54**

6 Conclusion **56**

6.1 Patterns usage feedback 56

6.2 Refactoring to Patterns feedback 60

Bibliography **62**

A List of Acronyms and Definitions **64**

List of Tables

3.1 Summary: functional implementation of OO Creational patterns . . . 16
3.2 Summary: functional implementation of OO Structural patterns . . . 16
3.3 Summary: functional implementation of OO Behavioral patterns . . . 17

Chapter 1

Introduction

> *"For refactoring to be valuable it must be going somewhere, not just an abstract intellectual exercise. Patterns document program structures with known good properties. Put the two together and you have Refactoring to Patterns."*
>
> —Kent Beck, Director, Three Rivers Institute

Today it is realy hard to find software developer who hadn't heard about design patterns and refactoring techniques. This terms became very popular few years ago and are very widely used in modern software development. The formal dfinition of this tems is the next:

- A design pattern is a general solution to a common problem in software design. It should systematically name, motivate and explain the problem, the solution, when to apply the solution, and its consequences. This solution might be customized and implemented to solve the problem in a particular context. [10]

- Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. Its heart is a series of small behavior preserving transformations. Each transformation (called a 'refactoring') does little, but a sequence of transformations can produce a significant restructuring. Since each refactoring is small, it's less likely

to go wrong. The system is also kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring. [9]

When I tried to sound my subject for thesis the first time a lot of my friends and advisors were worried that I am trying to mix in one study two quite different topics. but this is a half-true that this topics differ so much. There is a natural relation between patterns and refactorings. "Patterns are where you want to be; refactorings are ways to get there from somewhere else". [16]

Refactoring to Patterns is the marriage of refactoring - the process of improving the design of existing code – with patterns, the classic solutions to recurring design problems. Refactoring to Patterns suggests that using patterns to improve an existing design is better than using patterns early in a new design. This is true whether code is years old or minutes old. We improve designs with patterns by applying sequences of low-level design transformations, known as refactorings.

The main ideas in gathering both Refactoring and Patterns in same study were the next:

- refactor to Patterns when appropriate and away from Patterns when something simpler is discovered
- use Patterns to communicate intention
- know and continue to learn a large body of Patterns
- understand how to implement Patterns in simple and sophisticated ways
- use Patterns to clean, condense, clarify and simplify code
- evolve designs

The idea of Refactoring To Patterns of course is not new itsef - it had a long history with it's own classical authors and materials, the first in this queue is fundamental

book of Joshua Kerievski [16]. As the essential Gang of Four book says, "Design Patterns... provide targets for your refactorings." [9]

The answer for question "so what is than new in your work?" would be the second part of subject - we are trying to bring/apply our knowledge and experience to Functional world.

I remember when I heard the first time at second year of study about patterns and read few articles about it - I was very delighted with this new world and interesting methods of code organization and become very enthusiastic about patterns. The problem which really caused me to choose the subject for thesis is that when "Refactoring" term is quite general and can be easily applied to any language/paradigm, "Design Patterns" in our mind is very closely associated with Object-Oriented paradigm, and here is why.

Design patterns gained popularity in computer science after the book "Design Patterns: Elements of Reusable Object-Oriented Software" was published in 1994 by the so-called "Gang of Four" (Gamma et al.). That same year, the first Pattern Languages of Programming Conference was held and the following year, the Portland Pattern Repository was set up for documentation of design patterns. From another side we had perfect work of XP-creator Kent Back about patterns in Smalltalk. Since that time design patterns where very popular subject for different researches, discussions and pulications, they had a lot of heads involved and thats why the topic was developed very quickly and have now a huge base.

But the problem was that most of this researches and publications were about implementation and application of patterns/refactoring to Object-Oriented paradigm. It was most popular and authors used it as primary in their work. Other paradigms had no such careful attention, and it is black side of OOP monopolism in enterprise application development.

I always have been a loyal supporter of functional paradigm, so I aways wanted to have this gap filled at least in this paradigm. Historically, functional programming

languages have not been very popular for various reasons. Recently, however, a few of these languages are entering the computer industry.

As software becomes more and more complex, it is more and more important to structure it well. Well-structured software is easy to write, easy to debug, and provides a collection of modules that can be re-used to reduce future programming costs. Conventional languages place conceptual limits on the way problems can be modularised. Functional languages push those limits back. In order to do classic patterns in functional languages we should escape from OO paradigm limitations. As well we should have new patterns suggested by functional languages.

After developing for over four years in Java and Perl, I was pretty comfortable with best practices for creating applications in OO languages. When I tried to create my first application in Erlang(am not taking into account Lisp samples written during very short academic course), I ran into all sorts of problems. I couldn't figure out how to create table rows in a loop, or append values to a string, or to do two statements in one call. You can feel it even trying to write any complex XSLT with parsing and recursion.

But I have realized that what I was trying to do was use an OO approach in a functional programming language. There were fundamental differences between the two approaches, and not until I made the paradigm shift was I able to become proficient. The phrase "paradigm shift" is used a lot in programming, and I suppose sometimes it is justified. It actually helped me to understand OO better, since I was able to contrast the two approaches.

The history of Object-Oriented and Functional programming shows that they came from two different branches of the programming language tree. Functional is its own main branch with a venerable history going back to Lisp and the Lambda calculus. On the other hand OO is apparently and offshoot of procedural programming. I believe the second is a coincidence of history and the development of ideas. Today we see the beginnings of a growing together of Functional and OO programming.

Another idea is that to make functional paradigm more popular and widely used we need to involve to it a part of experienced OOP "Coryphaeus". Because functional programming encompasses a very different way of composing programs, programmers who are used to the imperative paradigm can find it difficult to learn. So I think that it would be very useful to show comparison/make mapping between well-known OOP design patterns and functional analogues, and to create last ones if we require that.

In process of making such comparison we would find out problems and gaps of each paradigm in regard to patterns and refactoring techniques. Hope it would be helpful for students as well when they try to learn course in functional languages. As at such courses we usually do not have time to look at FP in depth, and are onlu learning basic syntax. This fact causes us not to think about FP as something which can be used in real life, in production. But as far as most of students are aware of OOP design principles and patterns, it would be interesting for them to find out that the same problems/tasks can be solved in functional languages in much easier and straight-forward way.

To support this idea I have included to my thesis two parts:

- comparison of well-known design patterns and refactoring methods from object-oriented paradigm to functional approach
- description of my open-source project(XSLT transformer in Erlang) as an example of production-value library in functional language which was written with help of patterns and refactoring.

So the objectives of the study were:

- to collect in one research a set of methods of functional code refactoring using design patterns
- to find out whether mathematical theory can be applied to the most "mathematical" programming paradigm with aim to create common design solutions/optimise functional code

- to make a comparison of well-known design patterns from object-oriented paradigm to functional approach
- to create utility for making XSLT transformations which is missed in current Erlang/OTP distribution

The "XSLT Transformer in Erlang" project was chosen as far as we don't have tool for XSLT transformations in native Erlang and it is one of the most requested features in Erlang community.

Erlang is a general-purpose pure functional, concurrent, garbage-collected programming language with dynamic typing. It is used to build massively scalable soft real-time systems with requirements on high availability. Some of its uses are in telecoms, banking, e-commerce, computer telephony and instant messaging. Erlang's runtime system has built-in support for concurrency, distribution and fault tolerance.

There are a lot of libraries and addons written in and for Erlang, but we still do not have native XSLT transformer written in erlang to apply XSL stylesheets to XML documents. Current solutions are based on adapters to C++ - transformers which brings a set of problems and restrictions(we need to install native system libraries, specific to platform so our erlang app is not platform-independant any more, we depend on version of external library which could be not supported any more, it is slow as we need inter-languages communication throughout erlang virtual machine and we loose our benefits of fault talerancy and internal concurrency model support).

In native Erlang we have set of functions for working with XML/XPATH evaluation(xmerl package), so our XSLT transformer is using this library for underlying transformation. XSLT (Extensible Stylesheet Language Transformations) is a declarative, XML-based language used for the transformation of XML documents. So to write XSLT transformer in functional language means "to write interpreter for functional/declarative language in functional language", which brings additional challange to work.

In process of my work on thesis I found that "Patterns" that are widely used in one

paradigm may be invisible, trivial or built in as native in another paradigm, so I agree with statement made in [7], where author claims that "Patterns are created often as attempt to bring another-paradigm feature to language" Yes patterns might be a sign of weakness, but they might be a sign of simplicity. There is a trade-off between putting something in your programming language and making it be a convention, or perhaps putting it in the library.

When we identify and document one, that should not be the end of the story. Rather, we should have the long-term goal of trying to understand how to improve the language so that the pattern becomes invisible or unnecessary. But on other side, no matter how complicated your language will be, there will always be things that are not in the language. These things will have to be patterns. So, we can eliminate one set of patterns by moving them into the language, but then we'll just have to focus on other patterns. We don't know what patterns will be important 50 years from now, but for sure programmers will still be using patterns of some sort.

We have this situation already. At the very beginning of "Design Patterns Boom" we had only quite low-level structural, creational and behavioral patterns. In contrast now we already have a number of groups of high-level design and architecture patterns(Integration patterns [13], Workflow Patterns, Patterns for Distributed Processing, Enterprise Patterns, so on).

In the same sense that comprehensive application frameworks support larger-scale reuse of design and code than do stand-alone functions and class libraries, pattern languages will support larger-scale reuse of software architecture and design than individual patterns. Developing comprehensive pattern languages is challenging and time consuming, but will provide the greatest payoff for pattern-based software development during the next few years. And it is extremely important to develop patterns and create standarts in Functional languages, as they are a future of large-scale, fault-tolerant applications, cloud computing and non-relational data storages.

Chapter 2

Review of the Literature and Research

2.1 The most fundamental works

For current moment we have no any book on patterns in Functional programming or group of authors who are the law-makers in this topic. The only author who is making a try on gathering functional techniques in single dictionary/book is professor Jeremy Gibbons, who is an academic in the Computing Laboratory at Oxford University, specializing in programming languages and editor of "Journal of Functional Programming" He is in progress of writing book "Patterns in Functional Programming", about patterns of computation in functional programming. This is his sabbatical project for the academic year 2010/2011. But it is not finished yet and not seams that the progress is good as last record on his blog about the book [12] dated by summer. But he has quite interesting articles about accumulations on data structures, which distribute information across the data structure. List instances are familiar from the Haskell standard libraries, and professor Gibbons presents instances for a variety of tree datatypes; and the later work was about making it datatype-generic.

The most exciting article to my mind is "Functional Programming For The Rest

of Us" [5] by Slava Akhmechet. In this article he explains the most widely used ideas from functional languages using examples written in OO language(Java) Author claims that functional languages are extremely expressive and in a functional language one does not need design patterns because the language is likely so high level, you end up programming in concepts that eliminate design patterns all together. Author also tries to apply mathematical theory to FP, as functional programming is a practical implementation of Alonzo Church's ideas. He says that not all lambda calculus ideas transform to practice because lambda calculus was not designed to work under physical limitations. Therefore, like object oriented programming, functional programming is a set of ideas, not a set of strict guidelines. Lambda calculus was designed to investigate problems related to calculation. Functional programming, therefore, primarily deals with calculation, and uses functions to do so. A function is a very basic unit in functional programming. But it is hard to apply most of mathematical theory to FP. The problems are in article and the main question is "Why is the universe described with mathematical laws? Can all of the phenomena of our universe be described by mathematics?"

2.2 Works supporting idea that we don't need patterns in FP

Further there is a number of different discussions which I want to mention and which support the idea that we don't need analogues of OOP patterns in FP as most of them are already build in languages.

For instance, Mark Jason Dominus(a leading Perl programmer) tries to prove that patterns are language-agnostic [7] and that "Patterns" that are used recurringly in one language may be invisible or trivial in a different language

Also there is a very interesting discussion on StackOverflow("Does Functional Programming Replace GoF Design Patterns?") [2] Different authors there agree that

if a problem exists in OOP languages, which does not exist in FP languages, then clearly that is a shortcoming of OOP languages. The problem can be solved, but the language does not do so, so we require a bunch of boilerplate code in form of pattern to work around it. But they are at the same time joking that they want in this case to have programming language which will magically make all problems go away. Any problem that is still there is in principle a shortcoming of the language. At the same time they mention a lot of useful ideas like that the main features of functional programming include functions as first-class values, currying, immutable values, etc. and it doesn't seem obvious that OO design patterns are approximating any of those features. Also it sounds that FP doesn't eliminate the need for design patterns. The term "design patterns" just isn't widely used to describe the same thing in FP languages. But they exist. Functional languages have plenty of best practice rules of the form "when you encounter problem X, use code that looks like Y", which is basically what a design pattern is. So FP has its design patterns too, people just don't usually think of them as such. And unfortunately the functional programming community do not come up with as communicative pattern names as the object-oriented programming community. However, they agree that a lot of OOP-specific design patterns are pretty much irrelevant in functional languages. And if another language can solve the same problem trivially, that other language won't have need of a design pattern for it. Users of that language may not even be aware that the problem exists, because it's not a problem in that language. When you work in a FP language, you no longer need the OOP-specific design patterns. But you still need some general design patterns, like MVC or other non-OOP specific stuff, and you need a couple of new FP-specific "design patterns" instead. All languages have their shortcomings, and design patterns are usually how we work around them. An article "Scala's Pattern Matching = Visitor Pattern on Steroids" [18] attempts to prove this as well by showing built-in ability of scala's features to create Visitor. It describes PatternMatching as a key-feature in functional programming languages because it

facilitates definition of recursive functions in a way that maps closely to functions in lambda calculus. Author of article says that another way of looking at the Visitor Pattern is that it's simply a special case of pattern matching, which matches only on the type of method parameters. And it doesn't require all the boiler-plate code for the Visitor Pattern (e.g. Visitor superclass and double-dispatch accept method on expressions)

2.3 Existing descriptions of FP patterns

About attempts to treat some FP techniques as patterns the most useful article for me was [19], where Peter Norvig describes his view of design patterns in dynamic languages, and shows examples how to write classical patterns in dynamic languages, apparently there are some examples in Lisp. Another interesting one is [3] - where author describes FP patterns he found as usable and which refactorings can be done to implement this patterns:

- Structural Recursion
- Interface Procedure
- Mutual Recursion
- Accumulator Variable
- Syntax Procedure
- Local Procedure
- Program Derivation

I will make an overview of them in next chapter.

2.4 OOP And FP Cooperation

Furthermore there are resources that propagate idea of close cooperation of OOP and FP. The on of them is site [8] about Workshop MPOOL - the first Multiparadigm Programming with Object-Oriented languages workshop. MPOOL was created out of the need to bring together people who try to use or extend object-oriented tools in ways inspired by different programming paradigms, functional at most. The organizators claim that different paradigms roughly correspond to different communities and it is often the case that programming tools, concepts, and methodologies remains isolated in a single community because of lack of communication. So communication across paradigm-based communities is certainly hard - often the common background and vocabulary is limited. Nevertheless, they have found that the interaction of different paradigms can be very fruitful. Also I really want to mention the article "Patterns vs. Higher Order Functions" [15]. Author suggests that objects can be used to simulate higher order functions[FP], and believes the connection is more fundamental. He considers the concept of class and function to be synonymous. OO programming has traditionally lacked the power and flexibility provided by functional languages in relation to functions. However, these powers are not in conflict with OO and so there is no reason they shouldn't be added to the OO perspective. In addition to considering class and functions synonymous, he defines object as the environment resulting from the invocation of a function. FP has traditionally handled immutable models while OO handles mutable ones. So author believes they are complementary, and OO can bring way to incorporate mutability into to functional programming. As well as functional programming is the correct way to integrate immutability into OO programming. What remains is to discover the interconnections and interplay between these two concepts.

2.5 Refactoring and code structure in FP

There is also a number of publications influence of FP on refactoring methods and code structure. "How does functional programming affect the structure of your code?" [17], for instance, describes the specific braught by F#(functional language for .NET platform) to project. Microsoft lead developer says that functional programming affects the structure on a number of levels. "FP's influence is weakest "in the large"; at this level, the structure of the problem itself dominates, and FP's influence comes mostly shaping one's thinking about architectures or high-level designs. FP's influence is more apparent "in the medium", where the design of some APIs can be simplified sometimes dramatically in a few specific domains that are especially well-suited to FP. FP differentiates itself the most "in the small" this is the level where computations happen, this is level where the majority of code (for programs of all sizes) is written, this is where most FP-specific constructs take hold."

The very detailed fundamental article "Why Functional Programming Matters" [14] by John Hughes - is indeed must-to-read for every FP developer, even though it was written in 80-th. Author mentions that software becomes more and more complex and so it is more and more important to structure it well. Well-structured software is easy to write, easy to debug, and provides a collection of modules that can be re-used to reduce future programming costs. Conventional languages place conceptual limits on the way problems can be modularised. Functional languages push those limits back. In this paper he shows that two features of functional languages in particular, higher-order functions and lazy evaluation, can contribute greatly to modularity. "Since modularity is the key to successful programming, functional languages are vitally important to the real world."

And more modern follow-up to that article "Why 'Functional Programming Matters' matters" [6] give us explanation of old one in modern view. The great advantage of FP is that programs in it are factored more purely, and the factors are naturally along the lines of responsibility. The author defines "Factoring of program" as the

act of dividing it into units that are composed to produce the working software. Factoring happens as part of the design. (Re-factoring by him is the act of rearranging an existing program to be factored in a different way). The most important precondition for refactoring is that programs can be factored in orthogonal ways. The article's value is that it expresses an opinion about what makes programs better. It backs this opinion up with reasons why modern functional programming languages are more powerful than imperative programming languages. And at the end we see the suggestion that even if we don't plan to try functional programming tomorrow, the lessons about better programs are valuable for your work in any language today. "That's why Why Functional Programming Matters matters."

Chapter 3

OO Patterns To Functional

3.1 Purpose of Design Patterns - Intro

Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns.

Often, people only understand how to apply certain software design techniques to certain problems. These techniques are difficult to apply to a broader range of problems. Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem.

In addition, patterns allow developers to communicate using well-known, well understood names for software interactions. Common design patterns can be improved over time, making them more robust than ad-hoc designs.

To make intro I want to notice as well the common thing in most implementations forced by FP nature: we have to forget about encapsulation as all actions should be performed by utility functions, we can not bind method to data structure, which should confuse us at the beginning.

OOP Pattern	Comment
Abstract Factory	Natively supported with currying - we can configure function to create another functions with some property
Builder	implemented with utility functions
Factory Method	Summurized to FactoryFunction and described in Abstract Factory section
Prototype	Language-agnostic as works on data-structure level
Singleton	very language agnostic and should not be supported by pure functional language. But in Erlang can be replaced with registered light-weight server

Table 3.1: Summary: functional implementation of OO Creational patterns

OOP Pattern	Comment
Adapter	Adopts interace of existing function according to client's expectations
Bridge	Same as Adapter in FP
Composite	Native language-specific tree-like data structures and utilities to work with them
Decorator	In FP - composition of functions. F1 wraps F2 and decorates/processes result of F2 invocation
Facade	Very similar in implementation to Adapter in FP but with different purpose
Flyweight	Couldn't be implemented efficiently in pure functional style as objects are immutable and are copied each time
Proxy	In FP usually can be used for lazy evaluation only

Table 3.2: Summary: functional implementation of OO Structural patterns

OOP Pattern	Comment
Chain of responsibility	List of first-class functions with same signatures
Command	Obvious - any function as it is first-class object
Interpreter	Simple in implementations as we have pattern matching and recursion built in language
Iterator	External utility function specific to data structure, usually is not implemented as outstanding one but as part of recursive procedure-visitor
Mediator	Simplest - same as Memento. Pure Functional - in next chapter
Memento	The same idea with context as in State but client is saving history in stack
Null Object	Specific to language
Observer	Very similar to OO one
State	State in FP is not encapsulated, but should be present in both IN and OUT of function
Strategy	Obvious - any function as it is first-class object
Template method	Implemented in static and dynamic variants
Visitor	Most widely used in FP and has few natural forms and implementations

Table 3.3: Summary: functional implementation of OO Behavioral patterns

3.2 Creational design patterns

This design patterns is all about class instantiation. This pattern can be further divided into class-creation patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.

- Abstract Factory [3.2.1]
- Builder [3.2.2]
- Factory Method [3.2.3]
- Prototype [3.2.4]
- Singleton [3.2.5]

3.2.1 Abstract Factory

AbstractFactory - creates an instance of several families of classes

What is the abstract factory pattern, if not currying? We can pass parameter to a function once, to configure what kind of functions it would create in future.

Currying is the process of transforming a function that takes multiple arguments into a function that takes just a single argument and returns another function if any arguments are still needed.

So in example below we have "Abstract factory" which should convert two-arguments function to single-argument one if one argument should be constant.

Two concrete factories(multiplyByConstantFunctionFactory and addConstantFunctionFactory) return functions which will multiply by/add constant passed to factory.

An example is too verbose but just with aim to show similarity to OOP Abstract Factory.

```
1 -module(abstractfactory).
2 -export([start/0]).
```

```
3
4  start() ->
5      Doubler  = multiplyByConstantFunctionFactory(2),
6      Trippler = multiplyByConstantFunctionFactory(3),
7      io:format("==MultiplyByConstantFactory==~n",[]),
8      io:format("=>Calling doubler for argument 3: result=~p~n",[Doubler(3)]),
9      io:format("=>Calling trippler for argument 3: result=~p~n",[Trippler(3)]),
10
11     Add2 = addConstantFunctionFactory(2),
12     Add3 = addConstantFunctionFactory(3),
13     io:format("==AddConstantFactory==~n",[]),
14     io:format("=>Calling Add2 for argument 5: result=~p~n",[Add2(5)]),
15     io:format("=>Calling Add3 for argument 5: result=~p~n",[Add3(5)]).
16
17
18
19 abstractTwoArgumentsFunctionToOneArgumentFactory(ConstantArgument, Fun) ->
20     fun(VariableArgument) -> Fun(ConstantArgument, VariableArgument) end.
21
22 multiplyByConstantFunctionFactory(Multiplier) ->
23     abstractTwoArgumentsFunctionToOneArgumentFactory(Multiplier, fun(X, Y)->X*Y end).
24
25 addConstantFunctionFactory(Constant) ->
26     abstractTwoArgumentsFunctionToOneArgumentFactory(Constant, fun(X, Y)->X+Y end).
27
28 %~ ==========OUT==========
29 %~ ==MultiplyByConstantFactory==
30 %~ =>Calling doubler for argument 3: result=6
31 %~ =>Calling trippler for argument 3: result=9
32 %~ ==AddConstantFactory==
33 %~ =>Calling Add2 for argument 5: result=7
34 %~ =>Calling Add3 for argument 5: result=8
```

Listing 3.1: `code/01-AbstractFactory.erl`

3.2.2 Builder

Builder-Separates object construction from its representation

The only difference between functional builder from listing below and OOP ones is that we need to pass constructed object to build utility functions each time and we have overhead copying attributes each time adding new one - as it is a payment for immutability.

Builder methods ”newDir”, ”newFile”, ”withName” and ”withChildren” from listing are hiding structure of object behind them.

```
1 -module(builder).
2 -export([start/0]).
3 -record(dir, {name, type, children}).
```

```
4
5
6  start() ->
7      Dir1BuiltWithoutChildren = withName(newDir(), "parentDir"),
8      Dir1WithChildrenAdded = withChildren(Dir1BuiltWithoutChildren, [withName(newDir(), "subDir"),
           withName(newFile(), "subFile")]),
9      io:format("=>BuilderEx: built dirresult=>~n",[]),
10     printDir(Dir1WithChildrenAdded).
11
12 newDir() -> #dir{type=dir, children=[]}.
13 newFile() -> #dir{type=file, children=[]}.
14
15 withName(FSObject, Name) ->
16     #dir{name=Name, type=FSObject#dir.type, children=FSObject#dir.children}.
17
18 withChildren(FSObject, Children) when FSObject#dir.type==dir ->
19     #dir{name=FSObject#dir.name, type=FSObject#dir.type, children=Children}.
20
21 printDir(D) ->
22     io:format("Dir=~p, type=~p~n",[D#dir.name, D#dir.type]),
23     printChildren(D#dir.children).
24
25 printChildren([])              ->
26     ok;
27 printChildren([FirstDir|Tail]) ->
28     printDir(FirstDir),
29     printChildren(Tail).
30
31 %~ ==========OUT==========
32 %~ =>BuilderEx: built dirresult=>
33 %~ Dir="parentDir", type=dir
34 %~ Dir="subDir", type=dir
35 %~ Dir="subFile", type=file
```

Listing 3.2: `code/02-Builder.erl`

We are building exactly the same composite structure as in listing 3.6, so you can check differences between direct structure fileds popullation and the same action made with Builder methods.

3.2.3 Factory Method

FactoryMethod - creates an instance of several derived classes. As in FP we don't have object's methods, it would be the same as any other "Factory" - please see [3.2.1]

3.2.4 Prototype

Prototype - a fully initialized instance to be copied or cloned.

As functions always return new immutable values our prototype in Functional language could be just a function which creates and fills data each time.

As an alternative in Erlang we can simply create constant macros(-define(prototypename, value)) which will work as prototype. We can't modify the fields later anyway.

```
1  -module(prototype).
2  -export([start/0]).
3  -record(lecturer, {name, middlename}).
4
5  start() ->
6      Prototype=prototypeTeacherFromMultiMediaDepartment(),
7      io:format("==Teacher from MultiMedia Department -
           middlename=~p~n",[Prototype#lecturer.middlename]).
8  prototypeTeacherFromMultiMediaDepartment()-> #lecturer{middlename="Oleksandrovych"}.
9
10 %~ ==========OUT==========
11 %~ ==Teacher from MultiMedia Department - middlename="Oleksandrovych"
```

Listing 3.3: `code/05-Prototype.erl`

3.2.5 Singleton

Singleton - a class of which only a single instance can exist.

In functional paradigm there is no such thing as a singleton class as we have no state. The singleton is something of an anti-pattern here, so if we anyway need to do this, there is probably a better architecture. But we can emulate singleton in language-specific manner. For Haskell it would be state monad, for Erlang - lightweight process, which hangs in memory and continuously loops with one argument. Since this is done with message passing, it's safe for concurrent use.

In example you can see that I am trying to create singleton on module level, which represents single Connection object(for example for case when it is too expensive to create this connection or we are prohibited to create more than one at all):

```
1  -module(singleton_connection).
2  -export([start/0]).
3  -export([get/0, loop/1]).
4
```

```
5  start() ->
6      io:format("==GetSingletonFirstTime=>~n",[]),
7      singleton_connection:get(),
8      io:format("==GetSingletonSecondTime=>~n",[]),
9      singleton_connection:get().
10
11 get() ->
12     createSingletonIfNotCreatedYet(registered()),
13     ?MODULE ! {get, self()},
14     receive
15         Connection ->
16             io:format("~p~n",[Connection]),
17             Connection
18     end.
19
20 createSingletonIfNotCreatedYet([?MODULE|_REST_OF_REGISTERED_PROCESSES]) ->
21     io:format("==>>Cached Connection returned: ",[]),
22     ok;
23 createSingletonIfNotCreatedYet([_NOT_OUR_PROCESS|REST_OF_REGISTERED_PROCESSES]) ->
24     createSingletonIfNotCreatedYet(REST_OF_REGISTERED_PROCESSES);
25 createSingletonIfNotCreatedYet([]) ->
26     Connection = createExpensiveConnectionObject(),
27     io:format("==>>Expensive Connection Created: ",[]),
28     register(?MODULE, spawn(?MODULE, loop, [Connection])).
29
30 loop(CachedConnection) ->
31     receive
32         {get, From} ->
33             From ! CachedConnection,
34             loop(CachedConnection)
35         end.
36
37 createExpensiveConnectionObject() ->
38     connection_id1.
39
40 %~ ==========OUT==========
41 %~ ==GetSingletonFirstTime=>
42 %~ ==>>Expensive Connection Created: connection_id1
43 %~ ==GetSingletonSecondTime=>
44 %~ ==>>Cached Connection returned: connection_id1
```

Listing 3.4: `code/06-Singleton.erl`

3.3 Structural design patterns

This design patterns is all about Class and Object composition. Structural class-creation patterns use inheritance to compose interfaces. Structural object-patterns define ways to compose objects to obtain new functionality.

- Adapter [3.3.1]

- Bridge [3.3.2]
- Composite [3.3.3]
- Decorator [3.3.4]
- Facade [3.3.5]
- Flyweight [3.3.6]
- Proxy [3.3.7]

3.3.1 Adapter

Adapter - match interfaces of different classes

In functional variant we can redefine Adapter pattern as one that will adapt parameters to the structure which mathces our existing function. In result we should be able to call existing procedures which have different type of input with the same argument using our Adapter.

For instance, in next listing we are using adapter if we have list but existing function accepts only tuples.

```
1  -module(functionalAdapter).
2  -export([start/0]).
3
4  start() ->
5      List = [3,14,15,92],
6      io:format("=>We have List, so we call function which accepts lists without adapter:~n",[]),
7      printList(List),
8
9      io:format("=>We use Adapter if we have function which accepts tuples but we want to pass same
           List:~n",[]),
10     tuplesInputAdapter(fun printTuple/1, List).
11
12 tuplesInputAdapter(_Function, []) ->
13     ok;
14
15 tuplesInputAdapter(Function, [Key, Value|Tail]) ->
16     Function({Key, Value}),
17     tuplesInputAdapter(Function, Tail).
18
19 printList([])->
20     io:format("~n",[]);
21 printList([H|T])->
```

```
22      io:format("~p ",[H]),
23      printList(T).
24
25  printTuple({Key, Value}) ->
26      io:format("~p=>~p~n",[Key, Value]).
27
28  %~ ==========OUT==========
29  %~ =>We have List, so we call function which accepts lists without adapter:
30  %~ 3 14 15 92
31  %~ =>We use Adapter if we have function which accepts tuples but we want to pass same List:
32  %~ 3=>14
33  %~ 15=>92
```

Listing 3.5: `code/07-AdapterFunctional.erl`

This example is quite similar to Facade one but has different purpose. While in Facade we are hiding signatures and functions behind single interface, we use Adapter here when we have legacy functions with different signatures and we do want to pass same argument.

3.3.2 Bridge

Bridge - separates an object's interface from its implementation

- decouples an abstraction from its implementation so that the two can vary independently.
- "Beyond encapsulation, to insulation" [1]

"Hardening of the software arteries" has occurred by using subclassing of an abstract base class to provide alternative implementations. This locks in compile-time binding between interface and implementation. The abstraction and implementation cannot be independently extended or composed.

Adapter makes things work after they're designed; Bridge makes them work before they are.

But in FP we don't have hierarchy of functions, se the implementation of Bridge would be completely the same as in Adapter[3.3.1]

Exceptional case - monads, which can have hierarchy, are described in FP-specific patterns

3.3.3 Composite

Composite - a tree structure of simple and composite objects.

I don't think we need to find analogy/implementation for functional languages. As in them we would onlyy need separate tree-like structure for data and separate util functions to operate with. For instance, in Erlang we have a lot of choices to make "Composite" based on lists ([1, [2, 3, 4], [5,6,7]]), tuples (1, 2, 3, 4, 5,6,7) or records:

```
1  -module(composite).
2  -export([start/0]).
3  -record(dir, {name, type, children}).
4
5
6  start() ->
7      CompositeDir1 =
8          #dir{name="parentDir", type=d, children=[
9              #dir{name="subDir", type=d, children=[]},
10             #dir{name="subFile", type=f, children=[]}
11         ]},
12
13     io:format("=>CompositeEx: component - dir:~n",[]),
14     printDir(CompositeDir1).
15
16 printDir(D) ->
17     io:format("Dir=~p, type=~p~n",[D#dir.name, D#dir.type]),
18     printChildren(D#dir.children).
19
20 printChildren([])              ->
21     ok;
22 printChildren([FirstDir|Tail]) ->
23     printDir(FirstDir),
24     printChildren(Tail).
25
26 %~ ========OUT==========
27 %~ =>CompositeEx: component - dir:
28 %~ Dir="parentDir", type=d
29 %~ Dir="subDir", type=d
30 %~ Dir="subFile", type=f
```

Listing 3.6: `code/09-Composite.erl`

But this representation is too language-agnostic and doesn't have FP-specific theory underneath so I will skip wrapping record to module to have real composite with

methods.

3.3.4 Decorator

Decorator - add responsibilities to objects dynamically.

We can use simple mathematical superset of functions, in which decorator function accepts decorated as param and additionally processes it's output:

```
1  -module(decorator).
2  -export([start/0]).
3
4  start() ->
5      List = [3,14,15,92,6],
6      io:format("==>INPUT LIST:~n",[]),
7      printList(List),
8
9      io:format("==Simple Decorated Invocation - adds 3 to each element in list==~n",[]),
10     printList(decorated(List)),
11
12     io:format("==Simple Decorator Invocation - adds 1 to each element of Decorated
           invocation==~n",[]),
13     printList(decorator(List, fun decorated/1)).
14
15 decorated(List) ->
16     lists:map(fun(X) -> X+3 end, List).
17
18 decorator(List, Decorated) ->
19     ResultOfDecoratedInvocation = Decorated(List),
20     lists:map(fun(X) -> X+1 end, ResultOfDecoratedInvocation).
21
22 printList([])->
23     io:format("~n",[]);
24 printList([H|T])->
25     io:format("~p ",[H]),
26     printList(T).
27
28 %~ ==========OUT==========
29 %~ ==>INPUT LIST:
30 %~ 3 14 15 92 6
31 %~ ==Simple Decorated Invocation - adds 3 to each element in list==
32 %~ 6 17 18 95 9
33 %~ ==Simple Decorator Invocation - adds 1 to each element of Decorated invocation==
34 %~ 7 18 19 96 10
```

Listing 3.7: `code/10-Decorator.erl`

3.3.5 Facade

Facade - a single class that represents an entire subsystem.

In example below we are hiding underlying procedures printList and printTuples behind Facade function with constant interface.

```
-module(facade).
-export([start/0]).

start() ->
    List = [3,14,15,92],
    io:format("=>Call Facade to print List as list:~n",[]),
    facadeWhichHidesUnderlyingSubProcedures(as_list, List),

    io:format("=>Call Facade to print List as set of tuples:~n",[]),
    facadeWhichHidesUnderlyingSubProcedures(as_tuples, List).

facadeWhichHidesUnderlyingSubProcedures(as_list, List) ->
    printList(List);

facadeWhichHidesUnderlyingSubProcedures(as_tuples, []) ->
    ok;

facadeWhichHidesUnderlyingSubProcedures(as_tuples, [Key, Value|Tail]) ->
    printTuple({Key, Value}),
    facadeWhichHidesUnderlyingSubProcedures(as_tuples, Tail).

printList([])->
    io:format("~n",[]);
printList([H|T])->
    io:format("~p ",[H]),
    printList(T).

printTuple({Key, Value}) ->
    io:format("~p=>~p~n",[Key, Value]).

%~ ==========OUT==========
%~ =>Call Facade to print List as list:
%~ 3 14 15 92
%~ =>Call Facade to print List as set of tuples:
%~ 3=>14
%~ 15=>92
```

Listing 3.8: `code/11-Facade.erl`

3.3.6 Flyweight

Flyweight - a fine-grained instance used for efficient sharing.

Unfortunately, we can't implement same in FP because objects in FP are immutable and we will always copy them.

The only possibility to make memory usage more efficient is to have associative array in which keys would be small id's and values would be expensive objects we

want to represent. In this case we will win in space but loose in speed - we will need to lookup real objects from map in real time.

3.3.7 Proxy

Proxy - an object representing another object.

The big deal here is that you could potentially craft a function that's computationally expensive but the user of this function might not need all the results all at once. Example would be using these sorts of functions to obtain database records and to display them on a webpage

So, naturally we can craft functions such that we can have lazy evaluation.

Lazy evaluation is usually implemented by encapsuling the expression in a parameterless anonymous function, called a thunk. For example, let's take a function when_null(A, B, C) that returns B if A equals the null atom, otherwise C.

Like in an if statement we want B only be evaluated when A equals null otherwise we want to get C.

```
1  -module(proxy).
2  -export([start/0]).
3
4  start() ->
5          io:format("====>expensiveMethod WhenNull~n"),
6
7          io:format("====>MethodResFor1=~p~n",[
8                  expensive_when_null(someFunction(1), expensiveMethod(1), expensiveMethod(2))
9          ]),
10
11         io:format("====>MethodResFor2=~p~n",[
12                 expensive_when_null(someFunction(2), expensiveMethod(1), expensiveMethod(2))
13         ]),
14
15         io:format("~n====>proxyMethod WhenNull~n"),
16
17         io:format("====>MethodResFor1=~p~n",[
18                 proxy_when_null(someFunction(1), fun()->expensiveMethod(1) end,
                        fun()->expensiveMethod(2) end)
19         ]),
20         io:format("====>MethodResFor2=~p~n",[
21                 proxy_when_null(someFunction(2), fun()->expensiveMethod(1) end,
                        fun()->expensiveMethod(2) end)
22         ]).
23
24
25
```

```
26 someFunction(N) when N==1 -> null;
27 someFunction(_N)           -> 1.
28
29
30 expensiveMethod(N) ->
31         io:format("==>expensiveMethod with input ~B~n", [N]), N.
32
33 expensive_when_null(null, B, _) -> B;
34 expensive_when_null(_, _, C) -> C.
35
36
37 proxy_when_null(null, B, _) -> B();
38 proxy_when_null(_, _, C) -> C().
39
40
41 %~ ==========OUT==========
42 %~ ##########STARTING###########
43 %~ ====>expensiveMethod WhenNull
44 %~ ==>expensiveMethod with input 1
45 %~ ==>expensiveMethod with input 2
46 %~ ====>MethodResFor1=1
47 %~ ==>expensiveMethod with input 1
48 %~ ==>expensiveMethod with input 2
49 %~ ====>MethodResFor2=2
50
51 %~ ====>proxyMethod WhenNull
52 %~ ==>expensiveMethod with input 1
53 %~ ====>MethodResFor1=1
54 %~ ==>expensiveMethod with input 2
55 %~ ====>MethodResFor2=2
```

Listing 3.9: `code/14-Proxy.erl`

3.4 Behavioral design patterns

This design patterns is all about Class's objects communication. Behavioral patterns are those patterns that are most specifically concerned with communication between objects.

- Chain of responsibility [3.4.1]
- Command [3.4.2]
- Interpreter [3.4.3]
- Iterator [3.4.4]

- Mediator [3.4.5]
- Memento [3.4.6]
- Null Object [3.4.7]
- Observer [3.4.8]
- State [3.4.9]
- Strategy [3.4.10]
- Template method [3.4.11]
- Visitor [3.4.12]

3.4.1 Chain of responsibility

ChainOfResponsibility - a way of passing a request between a chain of objects

```
1  -module(chain).
2  -export([start/0]).
3  -define(CAN_PROCESS, 0).
4  -define(CANNOT_PROCESS, 1).
5
6  start() ->
7          ChainWithConditions = [fun accept1/1, fun accept2/1, fun accept3/1],
8          ChainWithGuards = [fun acceptGuarded1/1, fun acceptGuarded2/1, fun acceptGuarded3/1],
9
10         lists:foreach(
11                         fun(N)->io:format("=CHAIN with Conditions -> ARG=~B=~n",
                                [N]),processArg(N, ChainWithConditions) end,
12                         [1,2,3,4]
13                 ),
14
15         lists:foreach(
16                         fun(N)->io:format("=CHAIN with Guards -> ARG=~B=~n", [N]),processArg(N,
                                ChainWithGuards) end,
17                         [1,2,3,4]
18                 ).
19
20 accept1(N) ->
21         if
22                 N==1 -> io:format("acc1: I can accept~n"     ), ?CAN_PROCESS;
23                 true -> io:format("acc1: I can not accept~n"), ?CANNOT_PROCESS
24         end.
25
26 accept2(N) ->
```

```
27          if
28                  N==2 -> io:format("acc2: I can accept~n"     ), ?CAN_PROCESS;
29                  true -> io:format("acc2: I can not accept~n"), ?CANNOT_PROCESS
30          end.
31
32  accept3(N) ->
33          if
34                  N==3 -> io:format("acc3: I can accept~n"     ), ?CAN_PROCESS;
35                  true -> io:format("acc3: I can not accept~n"), ?CANNOT_PROCESS
36          end.
37
38
39  acceptGuarded1(N) when N==1 -> io:format("acc1: I can accept~n"     ), ?CAN_PROCESS;
40  acceptGuarded1(_)           -> io:format("acc1: I can not accept~n"), ?CANNOT_PROCESS.
41
42  acceptGuarded2(N) when N==2 -> io:format("acc2: I can accept~n"     ), ?CAN_PROCESS;
43  acceptGuarded2(_)           -> io:format("acc2: I can not accept~n"), ?CANNOT_PROCESS.
44
45  acceptGuarded3(N) when N==3 -> io:format("acc3: I can accept~n"     ), ?CAN_PROCESS;
46  acceptGuarded3(_)           -> io:format("acc3: I can not accept~n"), ?CANNOT_PROCESS.
47
48
49  processArg(N, [H|T]) ->
50          HeadRes = H(N),
51          if
52                  HeadRes==?CANNOT_PROCESS -> processArg(N, T);
53                  true                     -> ok
54          end;
55  processArg(N, []) -> io:format("==>No handler to process arg ~B found in chain~n", [N]).
56
57  %~ ==========OUT==========
58  %~ #########STARTING############
59  %~ =CHAIN with Conditions -> ARG=1=
60  %~ acc1: I can accept
61  %~ =CHAIN with Conditions -> ARG=2=
62  %~ acc1: I can not accept
63  %~ acc2: I can accept
64  %~ =CHAIN with Conditions -> ARG=3=
65  %~ acc1: I can not accept
66  %~ acc2: I can not accept
67  %~ acc3: I can accept
68  %~ =CHAIN with Conditions -> ARG=4=
69  %~ acc1: I can not accept
70  %~ acc2: I can not accept
71  %~ acc3: I can not accept
72  %~ ==>No handler to process arg 4 found in chain
73  %~ =CHAIN with Guards -> ARG=1=
74  %~ acc1: I can accept
75  %~ =CHAIN with Guards -> ARG=2=
76  %~ acc1: I can not accept
77  %~ acc2: I can accept
78  %~ =CHAIN with Guards -> ARG=3=
79  %~ acc1: I can not accept
80  %~ acc2: I can not accept
81  %~ acc3: I can accept
82  %~ =CHAIN with Guards -> ARG=4=
83  %~ acc1: I can not accept
84  %~ acc2: I can not accept
```

```
85 %~ acc3: I can not accept
86 %~ ==>No handler to process arg 4 found in chain
```

Listing 3.10: `code/15-ChainOfResponsibility.erl`

3.4.2 Command

Command - encapsulate a command request as an object

What is the command pattern, if not an approximation of first-class functions?

In FP language, you'd simply pass a function as the argument to another function.

```
1  -module(command).
2  -export([start/0]).
3
4  start() ->
5          lists:foreach(fun(F)->F() end, [fun command1/0, fun command2/0, fun command3/0]).
6
7  command1() ->
8          io:format("executing command1~n").
9
10 command2() ->
11         io:format("executing command2~n").
12
13 command3() ->
14         io:format("executing command3~n").
15
16 %~ ==========OUT==========
17 %~ executing command1
18 %~ executing command2
19 %~ executing command3
```

Listing 3.11: `code/16-Command.erl`

In an OOP language, you have to wrap up the function in a class, which you can instantiate and then pass that object to the other function.

```
1  import java.util.*;
2
3  public class CommandQueue {
4
5          interface Command {
6                  void execute();
7          }
8
9          static class command1 implements Command {
10                 public void execute() {
11                         System.out.println("command1");
12                 }
13         }
14         static class command2 implements Command {
```

```
15                  public void execute() {
16                          System.out.println("command2");
17                  }
18          }
19          static class command3 implements Command {
20                  public void execute() {
21                          System.out.println("command3");
22                  }
23          }
24
25          public static List produceRequests() {
26                  List queue = new ArrayList();
27                  queue.add(new command1());
28                  queue.add(new command2());
29                  queue.add(new command3());
30                  return queue;
31          }
32
33          public static void workOffRequests(List queue) {
34                  for (Iterator it = queue.iterator(); it.hasNext();)
35                          ((Command) it.next()).execute();
36          }
37
38          public static void main(String[] args) {
39                  List queue = produceRequests();
40                  workOffRequests(queue);
41          }
42  }
```

Listing 3.12: `code/16-Command.java`

The effect is the same, but in OOP it's called a design pattern, and it takes a whole lot more code. [2]

3.4.3 Interpreter

Interpreter - a way to include language elements in a program.

See also example of Visitor[3.4.12] - interpret() function taken from there

```
1   -module(interpreter).
2   -export([start/0]).
3
4   start() ->
5       calculate("(1+(2*3))"),
6       calculate("((1+2)*3)").
7
8   calculate(Expr)->
9       AST = createAbstractSyntaxTree(Expr),
10      CalculationResult = interpret(AST),
11
12      io:format("Input: ~p~n", [Expr]),
13      io:format("AST: ~p~n", [AST]),
```

```
14      io:format("InterpetedExpr2: ~p ~n~n", [CalculationResult]).
15
16  createAbstractSyntaxTree(Expr) -> ast(Expr).
17  ast([BS, OL, A, OR, BE | _T]) when ([BS]=="(") and ([BE]==")") ->
18      {LeftOperand, _}=string:to_integer([OL]),
19      {RightOperand, _}=string:to_integer([OR]),
20      {LeftOperand, [A], RightOperand};
21  ast([_BS, OL, A, OR, BE | T]) when [OR]=="(" ->
22      {LeftOperand, _}=string:to_integer([OL]),
23      {LeftOperand, [A], ast(lists:concat([[OR], [BE], T]))};
24  ast([_BS1, BS2, OL, A, OR, BE2, A2, OR2 | _T]) when [BS2]=="(" ->
25      {RightOperand, _}=string:to_integer([OR2]),
26      {ast(lists:concat([[BS2], [OL], [A], [OR], [BE2]])), [A2], RightOperand}.
27
28  interpret(Expr)        -> evalExpr(Expr).
29  evalExpr({L, "+", R}) -> evalExpr(L)+evalExpr(R);
30  evalExpr({L, "*", R}) -> evalExpr(L)*evalExpr(R);
31  evalExpr(Number)       -> Number.
32
33  %~ ==========OUT==========
34  %~ Input: "(1+(2*3))"
35  %~ AST: {1,"+",{2,"*",3}}
36  %~ InterpetedExpr2: 7
37
38  %~ Input: "((1+2)*3)"
39  %~ AST: {{1,"+",2},"*",3}
40  %~ InterpetedExpr2: 9
```

Listing 3.13: `code/17-Interpreter.erl`

3.4.4 Iterator

Iterator - sequentially access the elements of a collection

```
1   -module(iterator).
2   -export([start/0]).
3
4   start() ->
5       List = [3,14,15,92,6],
6
7       io:format("==Simple Iterator==~n",[]),
8       simpleIterator(List),
9
10      io:format("~n==Iterator Which Stops When Number Of Left Elements Is N==~n",[]),
11      iteratorWhichStopsWhenNLeft(List, 2),
12      io:format("~n==Same over empty list==~n",[]),
13      iteratorWhichStopsWhenNLeft([], 2),
14
15      Tree = {1, {3, 4, {5, 8, 9}}, 3},
16      io:format("~n==Tree Iterator==~n",[]),
17      treeIterator(Tree).
18
19  simpleIterator([]) ->
20      io:format("==>Iteration finished~n",[]);
```

```
21
22 simpleIterator([Head|Tail]) ->
23     io:format("currentElem: ~p~n", [Head]),
24     simpleIterator(Tail).
25
26
27 iteratorWhichStopsWhenNLeft(List, N) when length(List)=<N ->
28     io:format("==>Iteration finished~n",[]);
29 iteratorWhichStopsWhenNLeft([Head|Tail], N)                ->
30     io:format("currentElem: ~p~n", [Head]),
31     iteratorWhichStopsWhenNLeft(Tail, N).
32
33 treeIterator({P, L, R}) ->
34     io:format("ParentElem: ~p~n", [P]),
35     treeIterator(L),
36     treeIterator(R);
37
38 treeIterator(Leaf) ->
39     io:format("Leaf: ~p~n", [Leaf]).
40
41 %~ ==========OUT==========
42 %~ ==Simple Iterator==
43 %~ currentElem: 3
44 %~ currentElem: 14
45 %~ currentElem: 15
46 %~ currentElem: 92
47 %~ currentElem: 6
48 %~ ==>Iteration finished
49
50 %~ ==Iterator Which Stops When Number Of Left Elements Is N==
51 %~ currentElem: 3
52 %~ currentElem: 14
53 %~ currentElem: 15
54 %~ ==>Iteration finished
55
56 %~ ==Same over empty list==
57 %~ ==>Iteration finished
58
59 %~ ==Tree Iterator==
60 %~ ParentElem: 1
61 %~ ParentElem: 3
62 %~ Leaf: 4
63 %~ ParentElem: 5
64 %~ Leaf: 8
65 %~ Leaf: 9
66 %~ Leaf: 3
```

Listing 3.14: `code/18-Iterator.erl`

3.4.5 Mediator

Mediator - defines simplified communication between classes.

In OO Mediator is statefull and can process calls from clients using it's current

state. In FP we should use "Inversion of Control" - we are never saving context(state) inside function but always injecting - retrieving it from Fun. In this case the simplest Mediator in FP would look like Memento[3.4.6]. But I will show another example of FP-specific "Mediator" in pure functional patterns, which will use Monad.

3.4.6 Memento

Memento - captures and restores an object's internal state.

In example I am using the same approach and functions as in State example([3.4.9]), but a bit adopted. To be able to recover state/redo actions I am saving contexts(=function results, =states) in history stack, so using Memento description terminology from GoF, I am "Caretaker". Like in State example, it is possible only if function I am using is accepting and returns similar Context objects, which I am using as Memento states.

```
1  -module(memento).
2  -export([start/0]).
3
4  start() ->
5      io:format("==>I am client and I am starting to operate with Snack machine~n",[]),
6      doWhatMachineSaysToDo(start,[]).
7
8  doWhatMachineSaysToDo(ok, History) ->
9      io:format("Machine stopping, full history stack: ",[]), printList(History);
10
11 doWhatMachineSaysToDo(Action, History) when length(History)==3 ->
12     [LastAction|_Tail]= History,
13     io:format("==>I am client and I want to redo previous action: ~p, when machine asks me to do:
           ~p~n",[LastAction,Action]),
14     NextAction = operateWithSnackMachine(LastAction),
15     doWhatMachineSaysToDo(NextAction, lists:concat([[LastAction], History]));
16
17 doWhatMachineSaysToDo(Action, History) ->
18     io:format("==>I am client and I doing action machine asks me: ~p~n",[Action]),
19     NextAction = operateWithSnackMachine(Action),
20     doWhatMachineSaysToDo(NextAction, lists:concat([[Action], History])).
21
22 operateWithSnackMachine(start)           -> put_coins;
23 operateWithSnackMachine(put_coins)       -> select_snack;
24 operateWithSnackMachine(select_snack)    -> take_your_snack;
25 operateWithSnackMachine(take_your_snack) -> get_out;
26 operateWithSnackMachine(get_out)         -> ok.
27
28 printList([])   -> io:format("~n",[]);
29 printList([H|T])-> io:format("~p ",[H]), printList(T).
30
```

```
%~ ==========OUT==========
%~ ==>I am client and I am starting to operate with Snack machine
%~ ==>I am client and I doing action machine asks me: start
%~ ==>I am client and I doing action machine asks me: put_coins
%~ ==>I am client and I doing action machine asks me: select_snack
%~ ==>I am client and I want to redo previous action: select_snack, when machine asks me to do:
    take_your_snack
%~ ==>I am client and I doing action machine asks me: take_your_snack
%~ ==>I am client and I doing action machine asks me: get_out
%~ Machine stopping, full history stack: get_out take_your_snack select_snack select_snack
    put_coins start
```

Listing 3.15: `code/20-Memento.erl`

3.4.7 Null Object

NullObject - designed to act as a default value of an object.

In FP would be very language-agnostic as depends on data representation.

Example shown for Erlang - constant NullObject to substitute Record value. Will help if function have to return null, but we don't wont client code to check on null result each time or crash trying to process null result

```
-module(null_object).
-export([start/0]).
-record(nameDataObject, {name}).
-define(NULL_NAME_DATA, #nameDataObject{name=null}).

start() ->
    io:format("==Real NameDataObject==~n",[]),
    printNameDataObject(createRealObject()),
    io:format("==Predefined Null NameDataObject==~n",[]),
    printNameDataObject(createNullObject()),
    io:format("==Null returned - should throw exception==~n",[]),
    printNameDataObject(createNull()).

printNameDataObject(NameData) ->
    io:format("==>NameData.name=~p~n",[NameData#nameDataObject.name]).

createRealObject()-> #nameDataObject{name="someName"}.
createNullObject()-> ?NULL_NAME_DATA.
createNull()      -> null.


%~ ==========OUT==========
%~ ==Real NameDataObject==
%~ ==>NameData.name="someName"
%~ ==Predefined Null NameDataObject==
%~ ==>NameData.name=null
%~ ==Null returned - should throw exception==
```

```
28 %~ {"init terminating in
       do_boot",{{badrecord,nameDataObject},[{proxy,printNameDataObject,1,[{file,"c:/UBS/Dev/ws/mt/code/proxy.erl"},{
```

Listing 3.16: `code/21-NullObject.erl`

3.4.8 Observer

Observer - a way of notifying change to a number of classes. In FP - our observed function should be responsible for notification of number of registered observers.

```
1  -module(observer).
2  -export([start/0]).
3
4  start() ->
5      Observers = [
6          fun observer1/1,fun observer2/1
7      ],
8      io:format("==Executing Observed==~n", []),
9      subjectFunction1(Observers).
10
11
12 subjectFunction1(Observers) ->
13     io:format("==OBSERVED CALLED~n", []),
14     notify(Observers, process_info(self(), current_function)).
15
16 notify([], _Observed) -> ok;
17 notify([Observer|RestOfObservers], Observed)->
18     Observer(Observed),
19     notify(RestOfObservers, Observed).
20
21 observer1(Observed)->
22     {current_function, {M, F, _Arity}} = Observed,
23     io:format("==>OBSERVER1: I was notified by ~p:~p~n",[M, F]).
24 observer2(Observed)->
25     {current_function, {M, F, _Arity}} = Observed,
26     io:format("==>OBSERVER2: I was notified by ~p:~p~n",[M, F]).
27
28 %~ ==========OUT==========
29 %~ ==Executing Observed==
30 %~ ==OBSERVED CALLED
31 %~ ==>OBSERVER1: I was notified by observer:subjectFunction1
32 %~ ==>OBSERVER2: I was notified by observer:subjectFunction1
```

Listing 3.17: `code/22-Observer.erl`

3.4.9 State

State - alter an object's behavior when its state changes.

We are changing faunction behaviour in same way - depending on context(State pattern's terminology), using matching ability. In example below - user interacts with Snack machine. Machine is reacting on current state and instructs user what to do next.

```
1  -module(state).
2  -export([start/0]).
3
4  start() ->
5      io:format("==>I am client and I am starting to operate with Snack machine~n",[]),
6      doWhatMachineSaysToDo(start).
7
8  doWhatMachineSaysToDo(ok) ->
9      io:format("Machine can be used by someone else now",[]);
10
11 doWhatMachineSaysToDo(Action) ->
12     io:format("==>I am client and I doing action machine asks me: ~p~n",[Action]),
13     NextAction = operateWithSnackMachine(Action),
14     doWhatMachineSaysToDo(NextAction).
15
16 operateWithSnackMachine(start)           -> put_coins;
17 operateWithSnackMachine(put_coins)       -> select_snack;
18 operateWithSnackMachine(select_snack)    -> take_your_snack;
19 operateWithSnackMachine(take_your_snack) -> get_out;
20 operateWithSnackMachine(get_out)         -> ok.
21
22 %~ ==========OUT==========
23 %~ ==>I am client and I am starting to operate with Snack machine
24 %~ ==>I am client and I doing action machine asks me: start
25 %~ ==>I am client and I doing action machine asks me: put_coins
26 %~ ==>I am client and I doing action machine asks me: select_snack
27 %~ ==>I am client and I doing action machine asks me: take_your_snack
28 %~ ==>I am client and I doing action machine asks me: get_out
29 %~ Machine can be used by someone else now
```

Listing 3.18: `code/23-State.erl`

3.4.10 Strategy

Strategy - encapsulates an algorithm inside a class. In case of Functional Paradigm we don't need to create separate wrapper like class if we want to write algorithm - it would be just new function, which we can pass as argument(as it is first-class object). See Command[3.4.2] as well

3.4.11 Template method

TemplateMethod-Defer the exact steps of an algorithm to a subclass

We usually use the TemplateMethod pattern too much. When we see a duplicated algorithm, it seems that the natural tendency is to push up the skeleton into a superclass. This creates an inheritance relationship within the algorithm, which in turn makes it harder to change. Later, when we do need to change the algorithm, we have to change the superclass and all of the subclasses at the same time. For example, one particular superclass contained three or four template methods, which made the subclasses look quite odd; and each little complex of template-plus-overrides significantly hampered design change in each of the others. Why? Is it the cost of extra classes, or my mathematical background, or coding habits ingrained before the rise of object-oriented languages? So, note to self: If we need to make more than one template method in class - we should rather break out State/Strategy objects instead of relying on TemplateMethod. [20]

Template method pattern mandates that the *template method* must model the invariant part of the algorithm, while the concrete classes will implement the variabilities that will be called back into the template method. While the pattern encourages implementation inheritance, which may lead to unnecessary coupling and brittle hierarchies, the pattern has been used quite effectively in various frameworks developed over times.

Outside the OO world, template method pattern has been put to great use by many programming languages and frameworks. In many of those applications, like many of the other design patterns, it has been subsumed within the language itself. Hence its usage has transcended into the idioms and best practices of the language itself. [11]

!!!In languages that support higher order functions, template methods are ubiquitous, and used as a common idiom to promote framework style reusability. Languages like Haskell allow you to do more advanced stuff in the base abstraction through func-

tion composition and monadic sequencing, thereby making the invariant part more robust in the face of implementation variabilities. Even with IoC frameworks like Spring, template methods promote implementing monadic combinators (to the extent you can have in OO without HOF) that result in autowiring and sequencing operations.

```
-module(templateMethod).
-export([start/0]).

start() ->
    %Instantiate 'TEMPLATE CLASS1'
    TemplateClass1 = fun()-> templateInvariant(fun templateVariabilityPart1/0) end,
    io:format("===call 'TEMPLATE METHOD' from Object(Class1)==~n",[]),
    TemplateClass1(),

    %Instantiate 'TEMPLATE CLASS2'
    TemplateClass2 = fun()-> templateInvariant(fun templateVariabilityPart2/0) end,
    io:format("===call 'TEMPLATE METHOD' from Object(Class2)==~n",[]),
    TemplateClass2().

templateInvariant(F) ->
    io:format("invariant Do before Variability part~n",[]),
    F(),
    io:format("invariant Do after Variability part~n",[]).

templateVariabilityPart1() ->
    io:format("ping1 executed~n",[]).

templateVariabilityPart2() ->
    io:format("ping2 executed~n",[]).

%~ ==========OUT==========
%~ ===call 'TEMPLATE METHOD' from Object(Class1)==
%~ invariant Do before Variability part
%~ ping1 executed
%~ invariant Do after Variability part
%~ ===call 'TEMPLATE METHOD' from Object(Class2)==
%~ invariant Do before Variability part
%~ ping2 executed
%~ invariant Do after Variability part
```

Listing 3.19: `code/25-TemplateMethodStatic.erl`

Higher Order Template Methods through Message Passing

Template method pattern has two distinct aspects that combine to form the wholeness of the pattern - the commonality of the algorithm described by the base class and the concrete variabilities being injected by the derived implementations. Replace class by module and we have some great examples in Erlang / OTP Framework using

asynchronous message passing

```
-module(templateMethod).
-export([start/0, callTemplateMethod/2, templateVariabilityPart1/0, templateVariabilityPart2/0]).

start() ->
    spawn(fun() -> templateInvariant() end).

templateInvariant() ->
    receive
        {Pid, F} ->
            io:format("invariant Do before Variability part~n",[]),
            Result=F(),
            io:format("invariant Do after Variability part~n",[]),
            Pid ! {self(), Result}
    end.

callTemplateMethod(Pid, Q) ->
    Pid ! {self(), Q},
    receive
        {Pid, Reply} -> Reply
    end.

templateVariabilityPart1() ->
    io:format("ping1 executed~n",[]).

templateVariabilityPart2() ->
    io:format("ping2 executed~n",[]).

%~ ==========OUT==========
%~ 1> Pid1=templateMethod:start().
%~ <0.33.0>
%~ 2> templateMethod:callTemplateMethod(Pid1, fun templateMethod:templateVariabilityPart1/0).
%~ invariant Do before Variability part
%~ ping1 executed
%~ invariant Do after Variability part
%~ ok
%~ 3> Pid2=templateMethod:start().
%~ <0.37.0>
%~ 4> templateMethod:callTemplateMethod(Pid2, fun templateMethod:templateVariabilityPart2/0).
%~ invariant Do before Variability part
%~ ping2 executed
%~ invariant Do after Variability part
%~ ok
```

Listing 3.20: `code/25-TemplateMethodDynamic.erl`

Erlang implements variability through message passing, dynamic typing and dynamic hot swapping of code. Isn't this some template method on steroids ? But teh Erlang people never call it by this name. This is because, it is one of the most common idioms of Erlang programming. Only difference with OO is that in some other languages, the pattern gets melded within the language syntax and idioms.

But- HOFs are more like the strategy pattern than the template method pattern. Template methods are a form of static composition while HOFs are based on dynamic composition.

Answer- In a language that supports HOFs, the difference between Strategy and Template Method goes away. In fact in Java also, we can design Strategy as a Template Method.

3.4.12 Visitor

Visitor-Defines a new operation to a class without change

One of the most powerful things in functional languages is Pattern Matching. Apparently, it's a key-feature in functional paradigm because it facilitates definition of recursive functions in a way that maps closely to functions in lambda calculus. [18]

Let's define an abstract class to represent math expressions, and three implementations to represent numbers, sum operation, and product operation.

For example, the math expression "1 + (2 x 3)" can be composed using the expression record as follows: 1, '+', 2, '*', 3 Here are examples of using them on Expression: evaluate(Expr) outputs 7 print(Expr) outputs (1+(2*3))

Let's try to apply PatternMatching to solve the problem in Erlang

```
1  -module(visitorMatching).
2  -export([start/0]).
3
4  start() ->
5      Expr = {1, '+', {2, '*', 3}},
6      io:format("EvalExpr res: ~p ~n", [evaluateVisitor(Expr)]),
7      io:format("PrintExpr res: ~s ~n", [printVisitor(Expr)]).
8
9  evaluateVisitor(Expr) -> evalExpr(Expr).
10
11 evalExpr({L, '+', R}) ->
12     evalExpr(L)+evalExpr(R);
13 evalExpr({L, '*', R}) ->
14     evalExpr(L)*evalExpr(R);
15 evalExpr(Number) ->
16     Number.
17
18 printVisitor(Expr) -> printExpr(Expr).
19
20 printExpr({L, '+', R}) ->
```

```
21      lists:concat(["(", printExpr(L),"+",printExpr(R), ")"]);
22  printExpr({L, '*', R}) ->
23      lists:concat(["(", printExpr(L),"*",printExpr(R), ")"]);
24  printExpr(Number) ->
25      lists:concat(["",Number]).
26
27  %~ ==========OUT==========
28  %~ EvalExpr res: 7
29  %~ PrintExpr res: (1+(2*3))
```

Listing 3.21: `code/26-VisitorMatching.erl`

Now in Java, how would this have been implemented with the Visitor Pattern?

```
1   abstract class Expr {
2           public <T> T accept(ExprVisitor<T> visitor) {
3                   visitor.visit(this);
4           }
5   }
6
7   class Num extends Expr {
8           private int value;
9
10          public Num(int value) {
11                  this.value = value;
12          }
13
14          public int getValue() {
15                  return value;
16          }
17  }
18
19  abstract class BiCompositeExpr extends Expr {
20          private Expr left;
21          private Expr right;
22
23          protected BiCompositeExpr(Expr left, Expr right) {
24                  this.left = left;
25                  this.right = right;
26          }
27
28          public Expr getLeft() {
29                  return left;
30          }
31
32          public Expr getRight() {
33                  return right;
34          }
35  }
36
37  class Sum extends BiCompositeExpr {
38          protected Sum(Expr left, Expr right) {
39                  super(left, right);
40          }
41  }
42
43  class Prod extends BiCompositeExpr {
```

```
44          protected Prod(Expr left, Expr right) {
45                  super(left, right);
46          }
47  }
48
49  interface ExprVisitor<T> {
50          T visit(Num num);
51
52          T visit(Sum sum);
53
54          T visit(Prod prod);
55  }
56
57  class EvalExpr implements ExprVisitor<Integer> {
58          public Integer visit(Num num) {
59                  return num.getValue();
60          }
61
62          public Integer visit(Sum sum) {
63                  return sum.getLeft().accept(this) + sum.getRight().accept(this);
64          }
65
66          public Integer visit(Prod prod) {
67                  return prod.getLeft().accept(this) * prod.getRight().accept(this);
68          }
69  }
70
71  class PrintExpr implements ExprVisitor<Void> {
72          public Void visit(Num num) {
73                  print(" " + num.getValue() + " ");
74                  return null;
75          }
76
77          public Void visit(Sum sum) {
78                  sum.getLeft().accept(this);
79                  print("+");
80                  sum.getRight().accept(this);
81                  return null;
82          }
83
84          public Void visit(Prod prod) {
85                  prod.getLeft().accept(this);
86                  print("*");
87                  prod.getRight().accept(this);
88                  return null;
89          }
90  }
```

Listing 3.22: `code/26-Visitor.java`

So we can look at pattern matching as for thing, which provides another way of accomplishing the goals of the Visitor Pattern.

As you can see, each matching statement in the Erlang example maps to a visit

method in the Java example. So, another way of looking at the Visitor Pattern is that it's simply a special case of pattern matching, which matches only on the type of method parameters.

Of course, if we add a new expression type in Java, such as Div, we have to add a new visit method for that type in every visitor. Likewise, to handle Div in Erlang, we would also have to add a new matching statement in every Erlang visitor that works with expressions. This is a famous trade-off for using the Visitor Pattern as it enables you to define new operations easily, but makes it difficult to add new types to visit.

3.5 Criticism

The concept of design patterns has been criticized by some in the field of computer science. [1]

- **Targets the wrong problem.** 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. Paul Graham writes in the essay Revenge of the Nerds. Peter Norvig provides a similar argument. He demonstrates that 16 out of the 23 patterns in the Design Patterns book (which is primarily focused on C++) are simplified or eliminated (via direct language support) in Lisp or Dylan.

- **Lacks formal foundations.** The study of design patterns has been excessively ad hoc, and some have argued that the concept sorely needs to be put on a more formal footing. At OOPSLA 1999, the Gang of Four were (with their full cooperation) subjected to a show trial, in which they were "charged" with numerous crimes against computer science. They were "convicted" by 2/3 of the "jurors" who attended the trial.

- **Leads to inefficient solutions.** The idea of a design pattern is an attempt to standardize what are already accepted best practices. In principle this might appear to be beneficial, but in practice it often results in the unnecessary duplication of code. It is almost always a more efficient solution to use a well-factored implementation rather than a "just barely good enough" design pattern.

- **Does not differ significantly from other abstractions.** Some authors allege that design patterns don't differ significantly from other forms of abstraction, and that the use of new terminology (borrowed from the architecture community) to describe existing phenomena in the field of programming is unnecessary. The Model-View-Controller paradigm is touted as an example of a "pattern" which predates the concept of "design patterns" by several years. It is further argued by some that the primary contribution of the Design Patterns community (and the Gang of Four book) was the use of Alexander's pattern language as a form of documentation; a practice which is often ignored in the literature.

Chapter 4

Pure Functional Patterns

4.1 Intro

In order to find out patterns and general rules for development using functional approach I tried to use mathematical theory and fundamental principles to be able to prove the concept.

So the main thesis statements for this chapter where recognized as following:

- Plato: "The universe is described with mathematical laws. Can all of the phenomena of our universe be described by mathematics?". Functional paradigm is the closest to mathematics. So it can be generalised and defined with patterns(mathematical laws) in the most natural and straightforward way;
- So Algorithms can be optimized due to mathematics theoretical rules
- We consider, that every variable in FP is final and constant. Since every symbol is non-mutable we cannot change the state of anything. Function can never cause side effects(purely functional function (or expression) have no memory or I/O side effects). But in real life we should have them anyway(I/O, persistance to DB, HTTP connections, ...)
- Higher Order Functions as superset of functions

- Basic wonderful abilities of FP, which could be used in patterns description:
  - Recursion (tail recursion particularly)
  - Currying: the process of transforming a function that takes multiple arguments into a function that takes just a single argument and returns another function if any arguments are still needed.
  - First-class functions (language supports constructing new functions during the execution of a program, storing them in data structures, passing them as arguments to other functions, and returning them as the values of other functions)
  - Lazy Evaluation of functions (technique of delaying a computation until the result is required)- i.e for concurrent calls and optimization in conditional clauses
  - Continuation (holds an instance of a computational process due to a given point in the process's execution)
  - Pattern Matching (the act of checking some sequence of tokens for the presence of the constituents of some pattern)
  - Closures (a first-class function with free variables that are bound in the lexical environment. Such a function is said to be "closed over" its free variables)
  - Monads(vice versa - pattern in FP) - a design pattern for "dealing with global state" (simple in OOP languages, so no equivalent design pattern exists there). In (pure) functional languages, side effects and mutable state are impossible, unless you work around it with the monad "design pattern", or any of the other methods for allowing the same thing.

4.2 Transactional Computation

Each function is allowed to abort the computation:

```
1  -module(monadic_transactions).
2  -export([start/0]).
3
4  start() ->
5      io:format("==>TransactionSuccessful - res:~p~n",[transactionalInvocation(4)]),
6      io:format("==>TransactionFailed     - res:~p~n",[transactionalInvocation(-1)]).
7
8  transactionalInvocation(In) ->
9      with(In, [
10         fun(X)->{ok, X-1} end,
11         fun(X)->{ok, X+2} end,
12         fun(X)-> if X==0->{error, '/0'}; true->{ok, 15/X} end end,
13         fun(X)->{ok, X*3} end
14     ]).
15
16 with(In, [])                -> In;
17 with(In, [Fun|RestofFuns]) ->
18     case Fun(In) of
19         {ok   , Res} -> with(Res, RestofFuns);
20         {error, Err} -> {error, Err}
21     end.
22
23 %~ ==========OUT==========
24 %~ ==>TransactionSuccessful - res:9.0
25 %~ ==>TransactionFailed     - res:{error,'/0'}
```

Listing 4.1: `code/F_MonadicTransactions.erl`

4.3 Catamorphism and Anamorphism

Catamorphism is a generalization of the folds on lists known from functional programming to arbitrary algebraic data types that can be described as initial algebras.

Anamorphism is a kind of generic function that can corecursively construct a result of a certain type and which is parameterized by functions that determine what the next single step of the construction is.

Examples would be shown as part of hylomorphic pattern in next section.

4.4 Hylomorphism

Hylomorphism is a recursive function, corresponding to the composition of an anamorphism (which first builds a set of results; also known as 'unfolding') and a catamorphism (which then folds these results into a final return value). Fusion of these two recursive computations into a single recursive pattern then avoids building the intermediate data structure. This is a particular form of the optimizing program transformation techniques. The categorical dual of a hylomorphism is called a metamorphism, and is a catamorphism followed by an anamorphism [4]

```
1  -module(hylomorphism).
2  -export([start/0]).
3  -record(hylomorphic_task, {
4      do,
5      till,
6      catamorfic_fun,
7      anamorfic_funs
8  }).
9
10 start() ->
11     io:format("==>evens(15)~n",[]),
12     printList(evens(15)),
13     io:format("~n==>factorial(4)~n~p~n",[factorial(4)]),
14     io:format("~n==>to_binary(11)~n",[]),
15     printList(to_binary(11)),
16     io:format("~n==>expand([{a,2},{b,3},{c,4}])~n",[]),
17     printList(expand([{'a',2},{'b',3},{'c',4}])).
18
19 printList([])    -> io:format("~n",[]);
20 printList([H|T])-> io:format("~p ",[H]), printList(T).
21
22 new(PL) when is_list(PL) ->
23     HMTask = #hylomorphic_task{
24         do              = proplists:get_value(do,              PL, fun(X) -> X + 1           end),
25         till            = proplists:get_value(till,            PL, fun(X) -> X =:= undefined end),
26         catamorfic_fun  = proplists:get_value(catamorfic_fun,  PL, fun(X) -> X               end),
27         anamorfic_funs  = proplists:get_value(anamorfic_funs,  PL, { [], fun(A,E) -> [E|A] end,
                fun lists:reverse/1 })
28     },
29     fun(InputData) -> eval(HMTask, InputData, element(1,HMTask#hylomorphic_task.anamorfic_funs))
           end.
30
31 eval(HMTask, Input, Result) ->
32     {_InjectingFlagAtom, FormatTempFoldingResultFunction, FormatResultFunction} =
           HMTask#hylomorphic_task.anamorfic_funs,
33     case (HMTask#hylomorphic_task.till)(Input) of
34         false ->
35             CatamorphicFoldResult  = (HMTask#hylomorphic_task.catamorfic_fun)(Input),
36             CurrentIterationResult = (HMTask#hylomorphic_task.do)(Input),
37             eval(HMTask, CurrentIterationResult, FormatTempFoldingResultFunction(Result,
                    CatamorphicFoldResult));
```

```
38          true -> FormatResultFunction(Result)
39      end.
40
41  %Example Hylomorphism calculations
42  evens(N) ->
43      HMTask = new([
44          {do,   fun(X)            -> X + 2  end},
45          {till, fun(X)            -> X >= N end}
46      ]),
47      HMTask(0).
48
49  factorial(N) when N > 0 ->
50      HMTask = new([
51          {do,   fun(X)            -> X - 1  end},
52          {till, fun(X)            -> X =< 1 end},
53          {anamorfic_funs, {1, fun(A,E) -> A * E end, fun(A) -> A end}}
54      ]),
55      HMTask(N).
56
57  to_binary(N) when N > 0 ->
58      HMTask = new([
59          {do,             fun(X) -> X div 2   end},
60          {till,           fun(X) -> X =< 0    end},
61          {catamorfic_fun, fun(X) -> (X rem 2) end},
62          {anamorfic_funs,  {[], fun(A,E) -> [E|A] end, fun(A) -> A end}}
63      ]),
64      HMTask(N).
65
66  expand(L) ->
67      HMTask = new([
68          {do, fun ([_|T]) -> T; ([])  -> []    end},
69          {till, fun ([]) -> true; (_) -> false end},
70          {catamorfic_fun, fun([{C,N}|_])  -> lists:duplicate(N, C) end}
71      ]),
72      HMTask(L).
73
74  %~ ==========OUT==========
75  %~ ==>evens(15)
76  %~ 0 2 4 6 8 10 12 14
77
78  %~ ==>factorial(4)
79  %~ 24
80
81  %~ ==>to_binary(11)
82  %~ 1 0 1 1
83
84  %~ ==>expand([{a,2},{b,3},{c,4}])
85  %~ [a,a] [b,b,b] [c,c,c,c]
```

Listing 4.2: `code/F_Hylomorphism.erl`

Fold is one of the most commonly used catamorphisms in functional languages. Erlang has foldl and foldr, which are used very frequently to encapsulate some of the very common patterns of computation as higher order operators instead of using

recursion directly. Fold's dual is unfold (anamorphism), which unfortunately does not enjoy an equal popularity amongst functional programmers. While fold is a recursion operator that consumes a collection, it's dual unfold encapsulates a common pattern that produces streams / collections from a single object.

4.5 Other FP patterns

Other FP patterns, which were identified but not covered in this thesis, are Convolution (function which maps a tuple of sequences into a sequence of tuples), Recursions(particularly Tail Recursion) and Monads(which I need to define anyway as it is basic pattern to deal with state in FP).

Monad is a programming structure that represents computations. Monads are a kind of abstract data type constructor that encapsulate program logic instead of data in the domain model. A defined monad allows the programmer to chain actions together and build different pipelines that process data in various steps, in which each action is decorated with additional processing rules provided by the monad; for example a sequence of arithmetic operations can be controlled to avoid division by zero in intermediate results. Programs written in functional style can make use of monads to structure procedures that include sequenced operations, or to define some arbitrary control flows (like handling concurrency, continuations, side effects such as input/output, or exceptions).

Chapter 5

Patterns And Refactoring In Project

The initial idea of project was to create utility to make XSLT transformations, which is missed in current Erlang/OTP distribution

The "XSLT Transformer in Erlang" project was chosen as far as we don't have tool for XSLT transformations in native Erlang and it is one of the most requested features in Erlang community.

Erlang is a general-purpose pure functional, concurrent, garbage-collected programming language with dynamic typing. It is used to build massively scalable soft real-time systems with requirements on high availability. Some of its uses are in telecoms, banking, e-commerce, computer telephony and instant messaging. Erlang's runtime system has built-in support for concurrency, distribution and fault tolerance.

There are a lot of libraries and addons written in and for Erlang, but we still do not have native XSLT transformer written in erlang to apply XSL stylesheets to XML documents. Current solutions are based on adapters to C++ - transformers which brings a set of problems and restrictions(we need to install native system libraries, specific to platform so our erlang app is not platform-independant any more, we depend on version of external library which could be not supported any more, it is

slow as we need inter-languages communication throughout erlang virtual machine and we loose our benefits of fault talerancy and internal concurrency model support).

This existing solutions are:

- **XMERL** - native erlang library with functions for exporting XML data to an external format.
- **ErlXSL** - the aim of this project is to provide a usable binding for Erlang to call native XSLT processors - alpha
- **Sablotron** - an adapter for a C++ XSL processor (sablotron) that allows Erlang programmers to perform transformations of XML data (binary or file) using an XSL stylesheet (binary or file).

In native Erlang we have set of functions for working with XML/XPATH evaluation(xmerl package), so our XSLT transformer is using this library for underlying transformation.

XSLT (Extensible Stylesheet Language Transformations) is a declarative, XML-based language used for the transformation of XML documents.

So to write XSLT transformer in functional language means "to write interpreter for functional/declarative language in functional language", which brought additional challange to work.

The project was developed using defined in previous sections patterns and their occurences were documented well, which you can found in source added to this thesis.

Chapter 6

Conclusion

6.1 Patterns usage feedback

The main part of my work was to define and describe patterns and common solutions in functional languages. That was driven by idea which was brought from Object-Oriented ones. During my attempts to port solutions from OOP I made a conclusion that fits idea claimed by Christopher Alexander - one of pioneers of common OOP design decisions:

PATTERNS ARE LANGUAGE AND PARADIGM AGNOSTIC(he claims to that as well).

In general, I would say that specific patterns are continuously being eliminated by new (or just rising-in-popularity) language features. This is the natural way of language design development; as languages become more high-level, abstractions that could previously only be called out in a book using examples now become particular language feature or library. As an example - Prototype; while it is a fundamental notion of JavaScript, it has to be implemented from scratch in other languages.

"Patterns" that are used recurringly in one language may be invisible or trivial in a different language. Therefore patterns are signs of weakness in programming languages.

When we identify and document one, that should not be the end of the story. Rather, we should have the long-term goal of trying to understand how to improve the language so that the pattern becomes invisible or unnecessary.

So nevertheless this work was not done without outcome, it gave us understanding of what could be added/fixed in existing pure languages, both OOP and FP ones.

Identification of patterns is an important driver of progress in programming languages. As in all programming, the idea is to notice when the same solution is appearing repeatedly in different contexts and to understand the commonalities. This is admirable and valuable. The problem with the "Design Patterns" movement is the use to which the patterns are put afterward: programmers are trained to identify and apply the patterns when possible. Instead, the patterns should be used as signposts to the failures of the programming language. As in all programming, the identification of commonalities should be followed by an abstraction step in which the common parts are merged into a single solution.

Multiple implementations of the same idea are almost always a mistake in programming and breaks the golden DRY rule ("Don't Repeat Yourself"). The correct place to implement a common solution to a recurring design problem is in the programming language, if that is possible.

The stance of the "Design Patterns" movement seems to be that it is somehow inevitable that programmers will need to implement Visitors, Abstract Factories, Decorators, and Facades. But these are no more inevitable than the need to implement Subroutine Calls or Object-Oriented Classes in the source language. These patterns should be seen as defects or missing features in Java and C++. The best response to identification of these patterns is to ask what defects in those languages cause the patterns to be necessary, and how the languages might provide better support for solving these kinds of problems.

People say that it's all right that Design Patterns teaches people to do this, because the world is full of programmers who are forced to use C++ and Java, and they need

all the help they can get to work around the defects of those languages. If those people need help, that's fine. The problem is with the philosophical stance of the movement. Instead of seeing the use of design patterns as valuable in itself, it should be widely recognized that each design pattern is an expression of the failure of the source language.

If the Design Patterns movement had been popular in the 1980's, we wouldn't even have C++ or Java; we would still be implementing Object-Oriented Classes in C with structs, and the argument would go that since programmers were forced to use C anyway, we should at least help them as much as possible. But the way to provide as much help as possible was not to train people to habitually implement Object-Oriented Classes when necessary; it was to develop languages like C++ and Java that had this pattern built in, so that programmers could concentrate on using OOP style instead of on implementing it.

As a result of ideas described above I want to recognize two rules of using patterns:

- Integration of design patterns to form pattern languages.
- Integration with current software development methods and software process models.

But here we have an interesting controversion: if we will try to add patterns implementation to language than we will definitely tend to create very heavy and messed multi-paradigm languages. Why?

As we said before, the patterns usually represent construction which belongs to different paradigm and we have to emulate it in current language. We have already examples of such languages: JavaScript, C#, Scala.

But at this stage we definitely should understand that we will pay for this multiparadigming by complexity, very different implentations of same thing by different developers and mess in maintaining. It is obvious that some developers would tend to realize some feature in more closer to their habits way. And if only language will

allow to do that not in one way - we will have mess. That's why it is horrible to support/read JavaScript code written by different developers. That's why Java is de-facto monopolistic languages in today's enterprize development. Because it is strict and allows us to do things only by objects and this code could be easily readable and checked by static analysis tools.

So what we would do? To include patterns to language and make it multi-paradigmic which means to have yet another dead unsupportable monster or to use pure languages like pure functional or OO and to proceed to emulate features we need with patterns? There is no definite unanimous answer.

I would rather suggest completely different alternative approach, which is the most valuable discover in my work as I think and here we go.

I think we have to avoid usage of monsterious and unpredictable multi-paradigmal languages but we should try to develop multi-language projects. We should not reinvent the wheel - we have already this approach working in other kinds of human activity. For example - army. We do not have universal soldiers which could drive an airplane and tank perfect simultaniously. Each kind of troops should do it's work and do it perfectly well. As well we do have different positions in companies and different hammers to break walls and to repair clocks. So why having this experience gathered during centures of human activity we are treying to use same language to implement all kinds of actions and abstractions?!

We MUST AVOID doing that. We should implement parts of application or modules which are used to transfer data and change it in one language(OO or Domain-Specific, DSL) and services and actions in another (Functional). Bad examples of mix - ORM(Object-Relational Mapping) and Monads in Functional languages which break rule to not have side effects. The best existing example of such separation - set of languages for JVM platform. We can use Java OO features to create objects and to use Jaskell or JErlang or Closure to use operate with this objects. Another example - SpringFramework support for dynamic languages. It does the same idea like one

implemented before. The only thing we should care about is that we need to make OO data structures immutable and cloneable(like Prototype) to be able to use all the power of pure functional scalability.

Functional programming describes only the operations to be performed on the inputs to the programs, without use of temporary variables to store intermediate results. The emphasis is to capture "what and why" rather than the "how." It emphasizes the definition of functions rather than the implementation of state machines, in contrast to procedural programming, which emphasizes execution of sequential commands.

That's why large-scale knowledge management applications benefit greatly from using a functional programming style as it simplifies development.

6.2 Refactoring to Patterns feedback

The second part of my work was about attempt to write required software in functional language and to refactor it introducing patterns discovered before.

Analyzing results I want to admit that refactoring using design patterns is a powerful approach to prepare application for new features development. It's benefit depends on at least two factors, mainly the effort required in the refactoring and how effective it is.

For instance, the benefit would be small if too much effort is required to translate a program correctly into a refactored form. A measure of effectiveness is the maintainers' performance, which can be affected by their work experience, in realizing the changes. So program refactoring for adding additional patterns is also beneficial regardless of the work experience of the maintainers.

Patterns help to alleviate software complexity at several phases in the software lifecycle. Although patterns are not a software development method or process, they complement existing methods and processes. For instance, patterns help to bridge the abstractions in the domain analysis and architectural design phases with the concrete

realizations of these abstractions in the implementation and maintenance phases. In the analysis and design phases, patterns help to guide developers in selecting from software architectures that have proven to be successful. In the implementation and maintenance phases, they help document the strategic properties of software systems at a level higher than source code and models of individual software modules.

At the end whant to make a guess that as now much of the existing literature on patterns is organized as design pattern catalogs and these catalogs represent a collection of relatively independent solutions to common design problems - more experience is gained using these patterns, developers and authors will increasingly integrate groups of related patterns to form pattern languages. These pattern languages will encompass a family of related patterns that cover particular domains and disciplines ranging from concurrency, distribution, organizational design, software reuse, real-time systems, business and electronic commerce, and human interface design.

Bibliography

[1] *Design patterns*, (2007), http://sourcemaking.com/design_patterns.

[2] *Does functional programming replace gof design patterns?*, (2008), http://stackoverflow.com/questions/327955/does-functional-programming-replace-gof-design-patterns.

[3] *Functional programming patterns*, (2009), http://www.fatvat.co.uk/2009/01/functional-programming-patterns.html.

[4] (2011), http://en.wikipedia.org/wiki/Hylomorphism_(computer_science).

[5] Slava Akhmechet, *Functional programming for the rest of us*, (2006), http://www.defmacro.org/ramblings/fp.html.

[6] Reginald Braithwaite, *Why why functional programming matters matters*, (2007), http://weblog.raganwald.com/2007/03/why-why-functional-programming-matters.html.

[7] Mark Dominus, *Design patterns of 1972*, (2006), http://blog.plover.com/prog/design-patterns.html.

[8] Jorg Striegnitz Editors Kei Davis, Yannis Smaragdakis, *Multiparadigm programming with object-oriented languages*, (2001), http://www2.fz-juelich.de/nic-series/Volume7/Volume7.html.

[9] Martin Fowler, *Refactoring: Improving the design of existing code.*, 2002.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design patterns*, 1995.

[11] Debasish Ghosh, *Subsuming the template method pattern*, (2009), http://debasishg.blogspot.com/2009/01/subsuming-template-method-pattern.html.

[12] Jeremy Gibbons, *Patterns in functional programming*, (2011), http://patternsinfp.wordpress.com/.

[13] Gregor Hohpe and Bobby Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, Addison-Wesley, 2004.

[14] J. Hughes, *Why Functional Programming Matters*, Computer Journal **32** (1989), no. 2, 98–107, http://www.cse.chalmers.se/ rjmh/Papers/whyfp.html.

[15] JefferyWalker, *Object oriented vs. functional styles*, http://cs.oberlin.edu/ jwalker/langDesign/OO_vs_Fun/notes.html.

[16] Joshua Kerievsky, *Refactoring to patterns*, 2004.

[17] Brian Longon, *How does functional programming affect the structure of your code?*, (2008), http://lorgonblog.wordpress.com/2008/09/22/how-does-functional-programming-affect-the-structure-of-your-code/.

[18] Andy Maleh, *Scala's pattern matching = visitor pattern on steroids*, (2008), http://andymaleh.blogspot.com/2008/04/scalas-pattern-matching-visitor-pattern.html.

[19] Peter Norvig, *Design patterns in dynamic languages*, (1998), http://www.norvig.com/design-patterns/.

[20] Kevin Rutherford, *Too much templatemethod*, (2009), http://silkandspinach.net/2009/01/04/too-much-templatemethod/.

Appendix A

List of Acronyms and Definitions

- **GoF** - Gang of Four - authors of fundamental book [10]
- **OOP** - Object-Oriented Programming
- **FP** - Functional Programming
- **XP** - Extreme programming - one of Agile software development methodologies
- **XML** - Extensible Markup Language - a markup language
- **XSLT** - Extensible Stylesheet Language Transformations - a declarative, XML-based language used for the transformation of XML documents
- **OTP** - Open Telecom Platform - most common-used Erlang framework for scalable applications development
- **I/O** - Input/Output