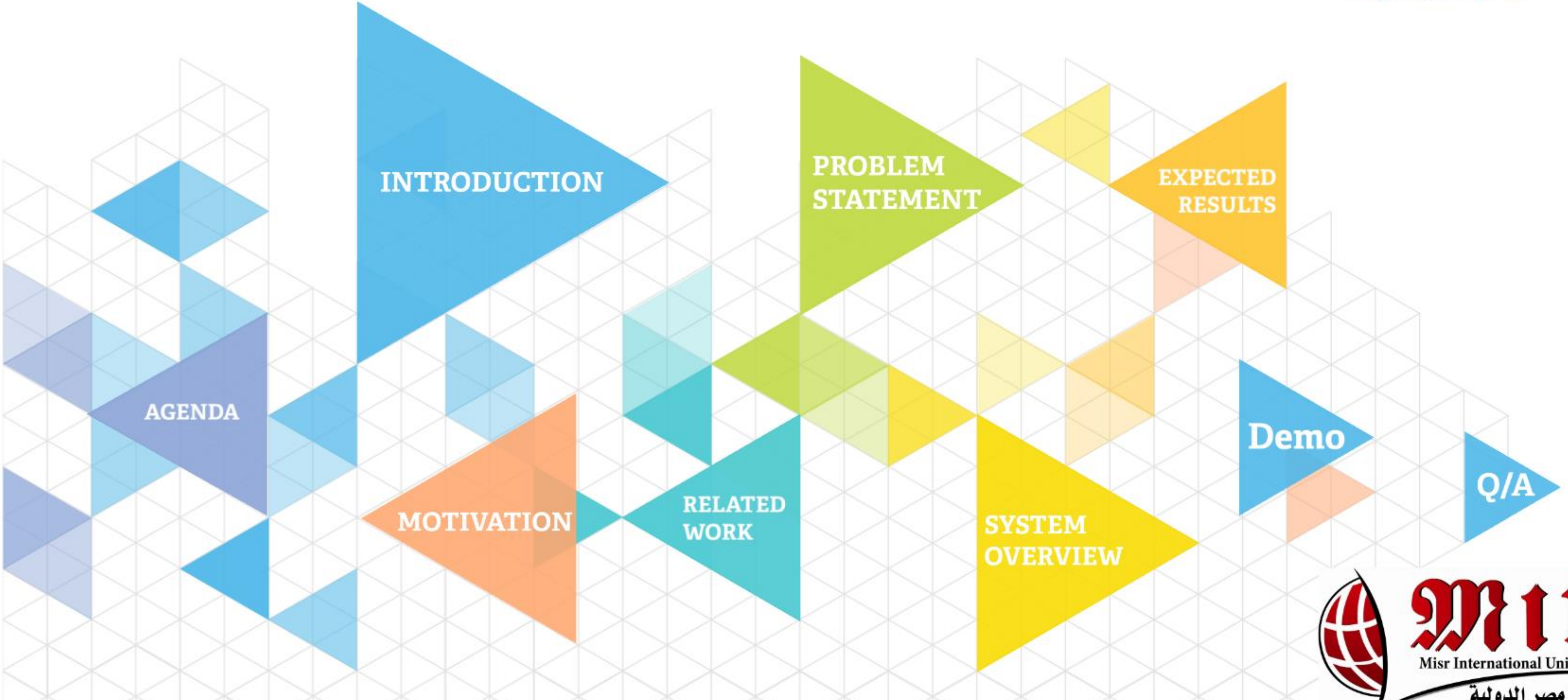
Automatic recognition of the appropriate Software Design Patterns

(Q/A tree like model approach)

By: Clara Kamal Youssef
Farida Mohamed Ahm
Hashem Mohamed Ha
Veronia Emad Talaat

Supervised By: Dr. Taraggy Ghanim
Eng. Nada Ayman



INTRODUCTION

PROBLEM STATEMENT

EXPECTED RESULTS

AGENDA

Demo

Q/A

MOTIVATION

RELATED WORK

SYSTEM OVERVIEW

Agenda

- Introduction
- Motivation
- Related Work
- Problem Statement
- System Overview
- Expected Results
- Demo

Misr International University

جامعة مصر الدولية

INTRODUCTION

Definition

Statistics 1

Statistics 2

Definition 1/2

Design Patterns [1] are reusable solutions to commonly occurring problems to help eliminate redundant coding.

They are not used directly in a machine code but as ready-made templates.

[1] “Design Patterns and Refactoring.”, SourceMaking, 2019, https://sourcemaking.com/design_patterns.

Definition 2/2

The concept of design pattern was initiated in 1994 when four software engineers published their book titled “Design patterns: Elements of reusable object oriented software” [2].

They proposed a two dimensional matrix categorization to the patterns based on two criterion which are purpose and scope.

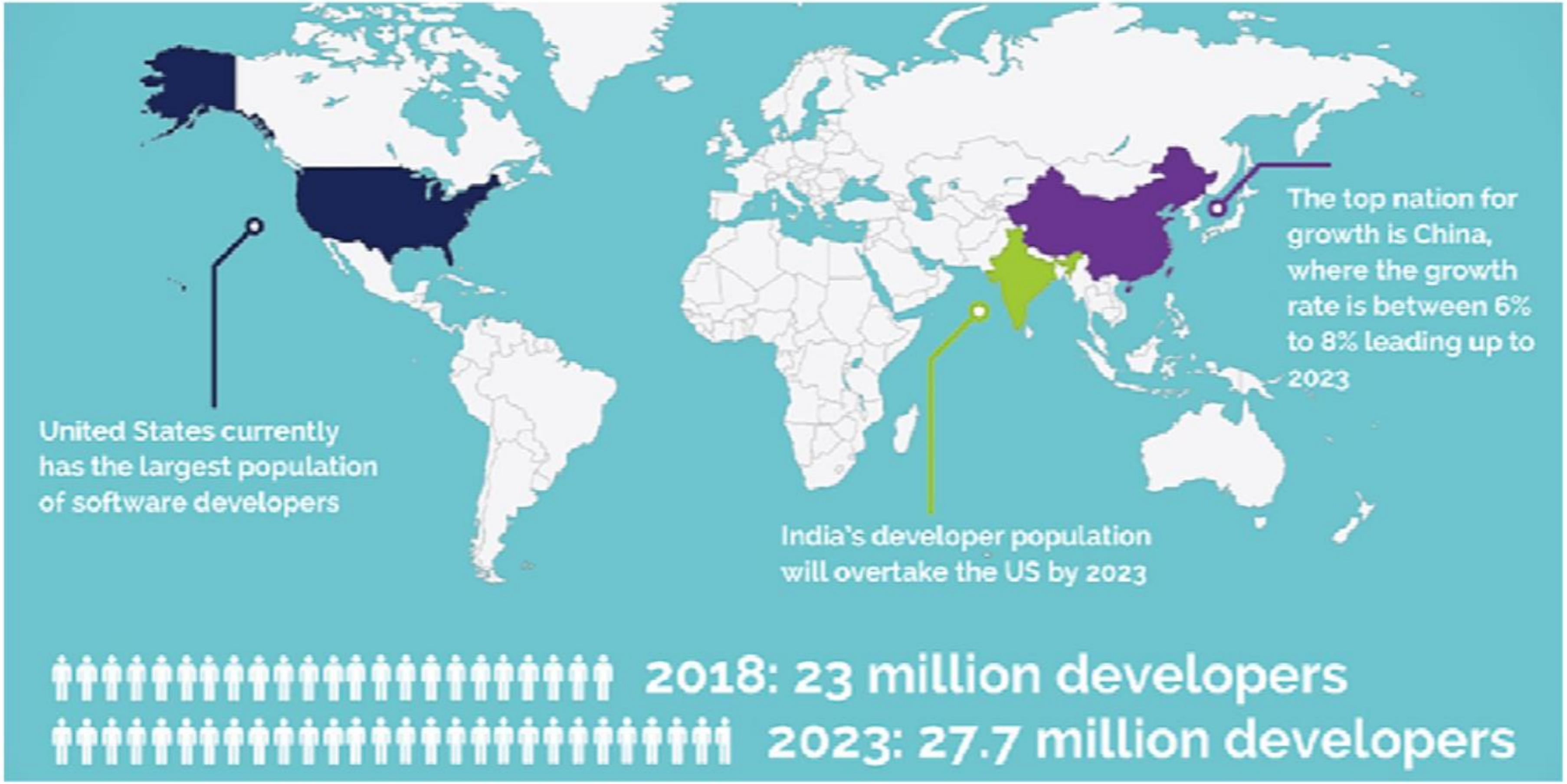
- **Purpose**: Creational, Behavioral and Structural.
- **Scope**: Class inheritance and Object composition patterns.

[2] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Pattern: Elements of Reusable Object-Oriented Software." Addison-Wesley, 1995.

Statistics

1

The number of software engineers in the world is growing every year:



The top nation for growth is China, where the growth rate is between 6% to 8% leading up to 2023

United States currently has the largest population of software developers

India's developer population will overtake the US by 2023

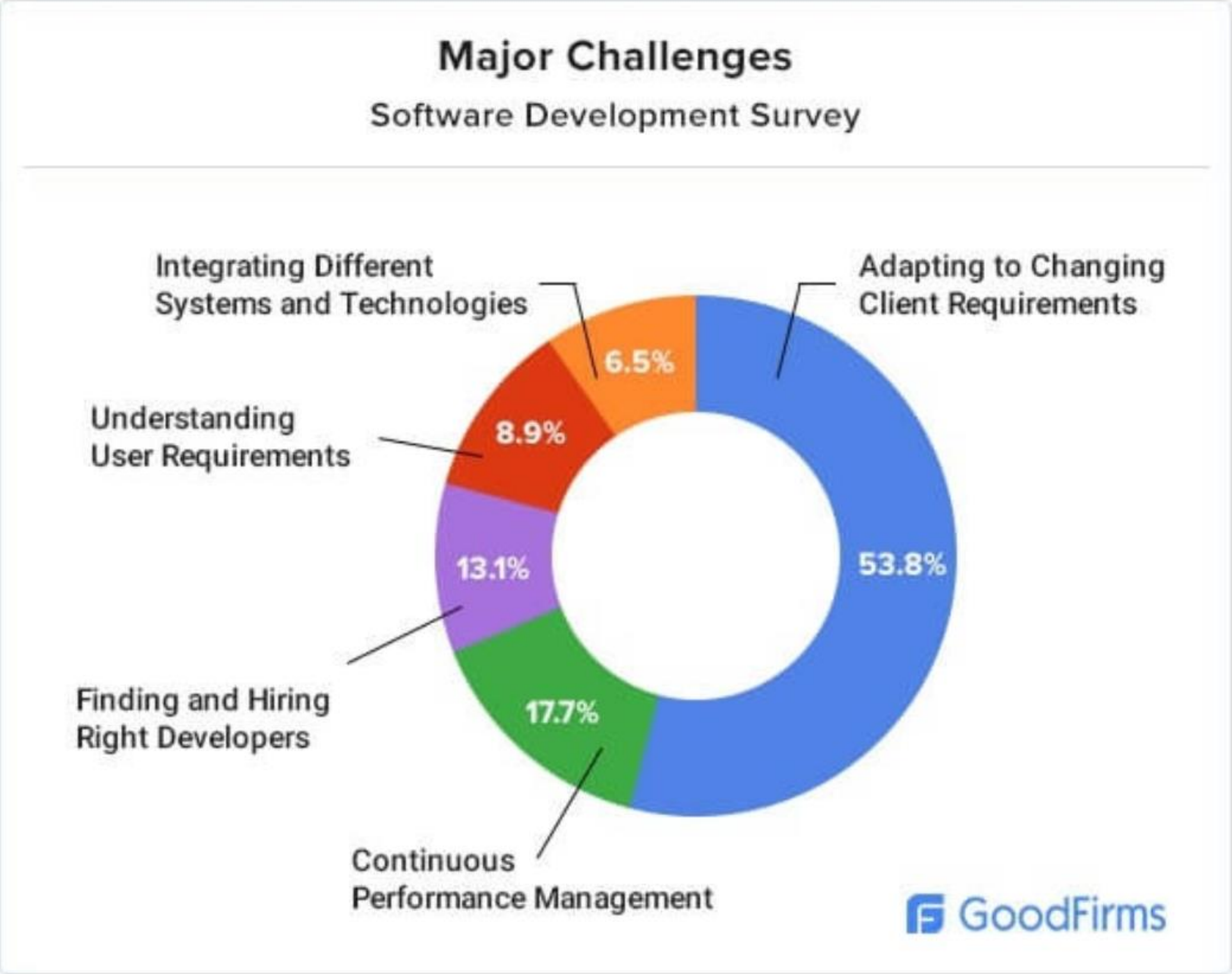
2018: 23 million developers

2023: 27.7 million developers

https://hackernoon.com/how-many-software-engineers-are-there-in-the-world-in-2019-us-europe-india-russia-and-china-c016d38oc

Statistics 2

Adapting and Changing user Requirements: **the #1 Challenge** that faces the software engineers



https://www.whatech.com/software/blog/archive/514579-around-54-software-developers-adapting-changing-client-requirements-challenge

MOTIVATION

Academic

Market

Academic Motivation

- Previous researches haven't achieved the most efficient accuracy.

- Selection of the suitable Design Pattern is considered as one of the most critical and confusing phases.

- Very limited researches about this approach as most of the researches concerns Reverse Engineering.

- According to our research, 2 researches only tested their approach on the most common 23 design patterns.

Reverse Engineering

- Known as Backward Engineering
- Deconstruction of an existing product to reveal its design & architecture
- Have some risk factors:
    - Loss of embedded business knowledge
    - Difficulty to retrieve an efficient design and requirements
    - Most commonly associated with the theft of intellectual property

Market Motivation

Silicon
Unions

Valeo

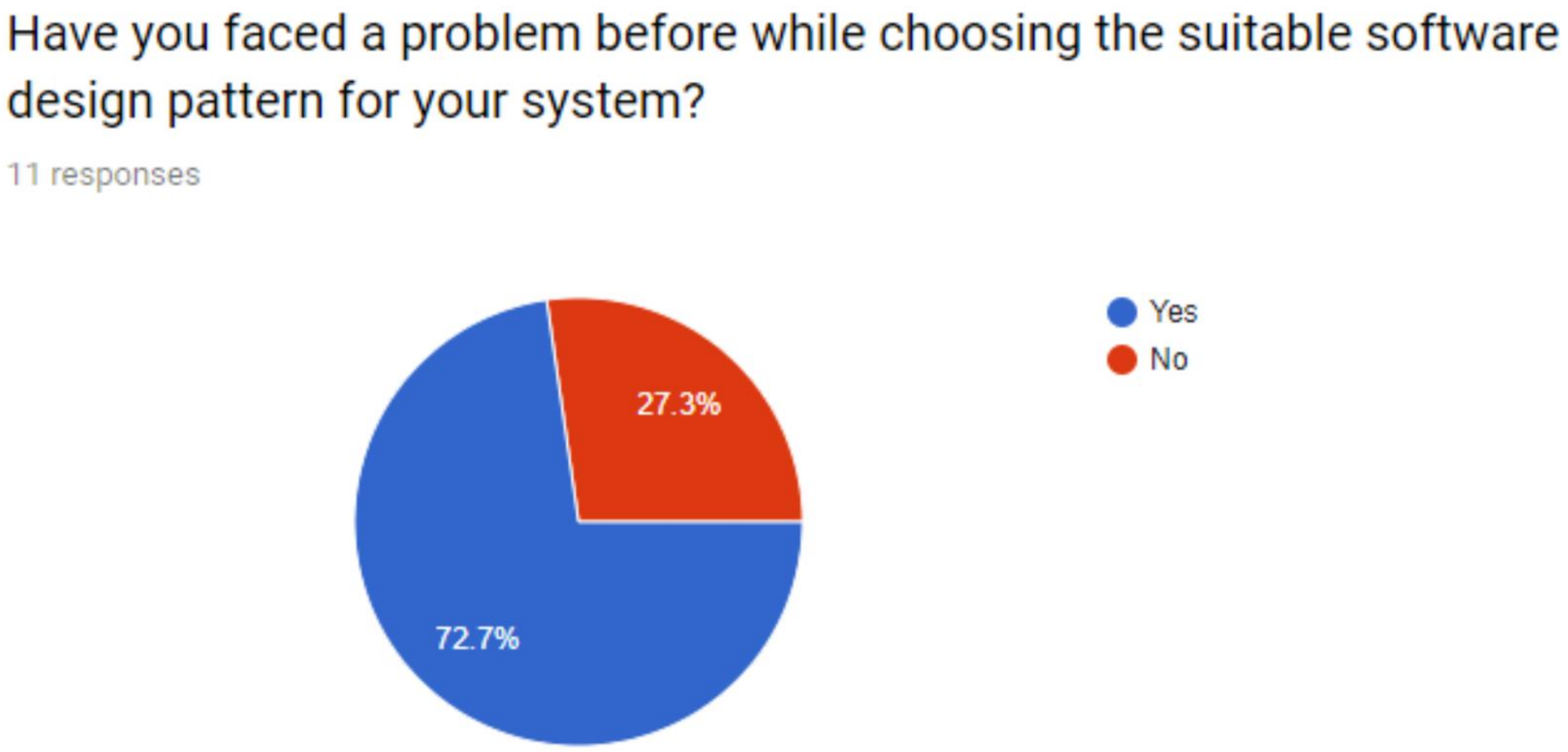
eBenefits
NETWORK

ASSET®
Technology Group
an EOH company

vodafone

iSky

CENTRO
GLOBAL SOLUTIONS

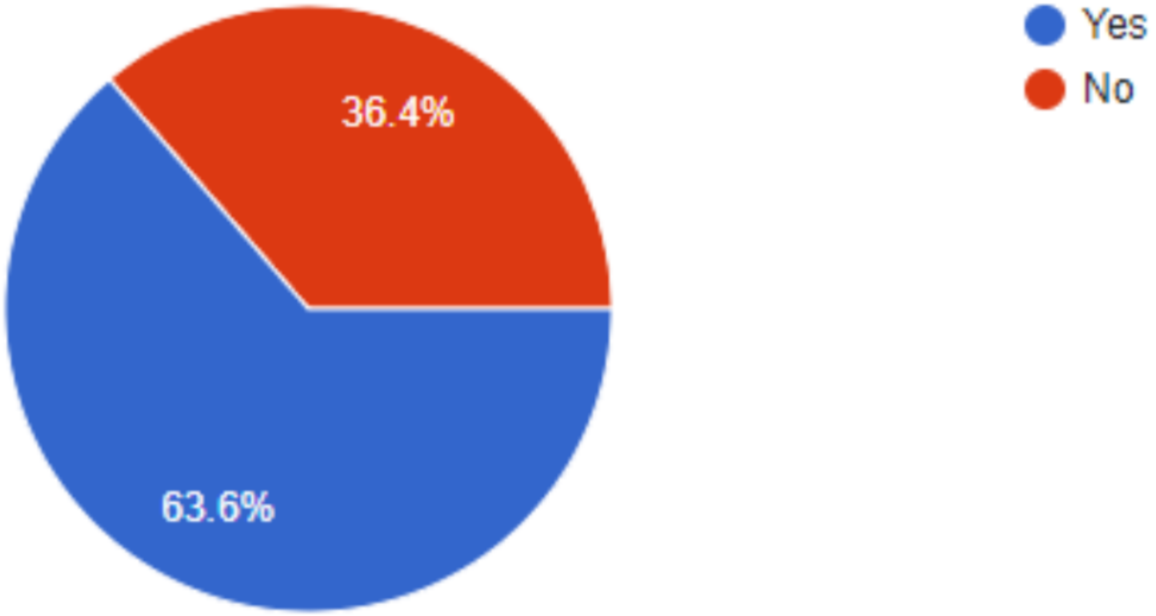


Have you faced a problem before while choosing the suitable software design pattern for your system?

11 responses

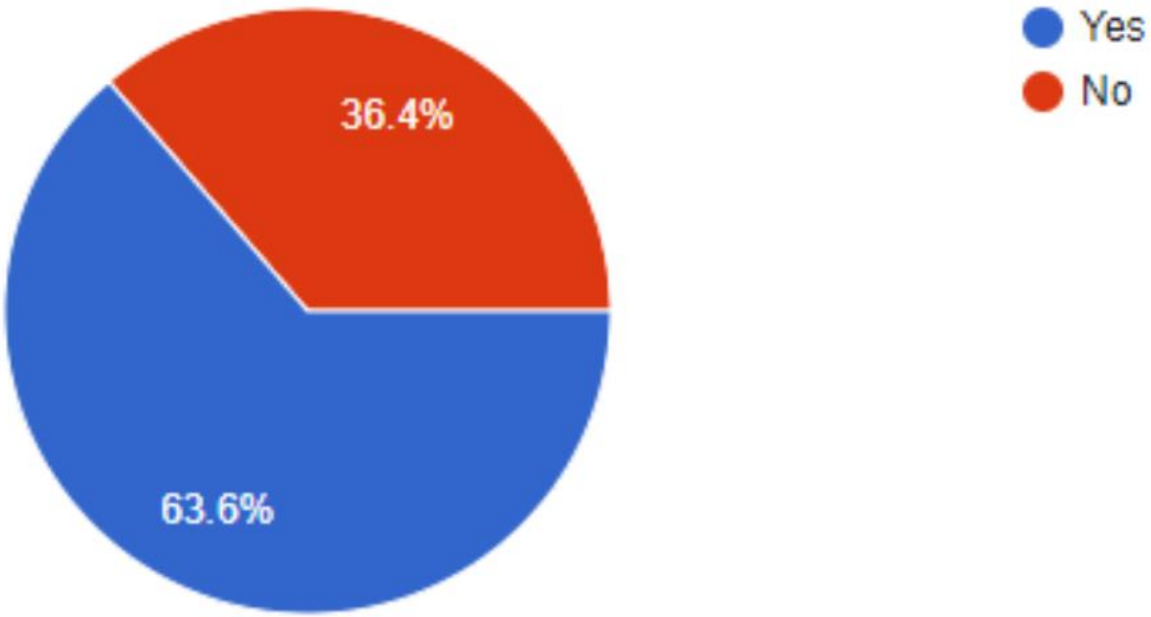
Have you faced a problem before while creating the correct class diagram of your system?

11 responses

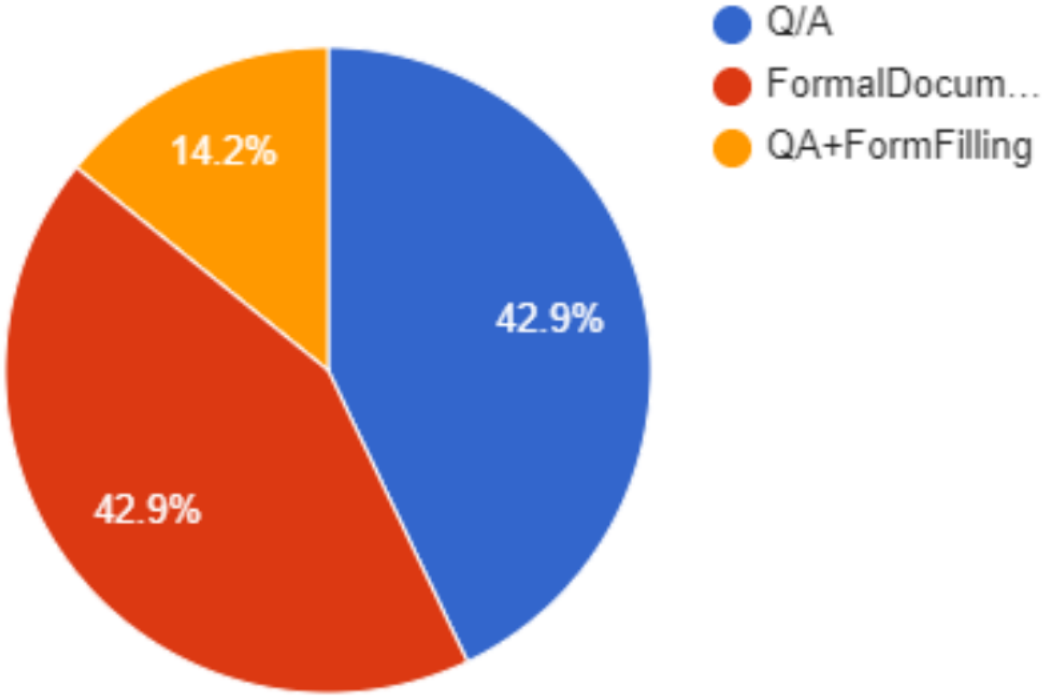


Have you ever discovered after writing your code that you used the wrong software design patterns?

11 responses

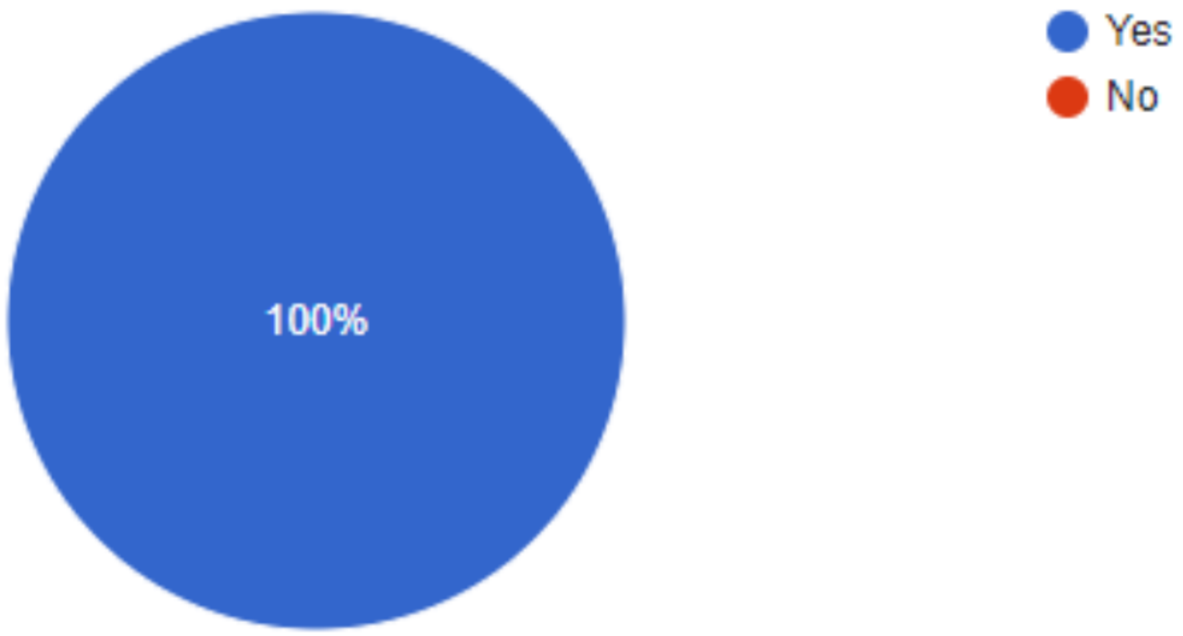


From your point of view, which input format would help in getting the most accurate results? (Ex: Q/A approach, Formal Documents…etc.)



If there's a system that helps in selecting the suitable software design patterns or creating the class diagram, Will you use this system ?

11 responses



Related Works

First approach

Second approach

Third approach

Fourth Approach

Comparison with proposed project

First Approach

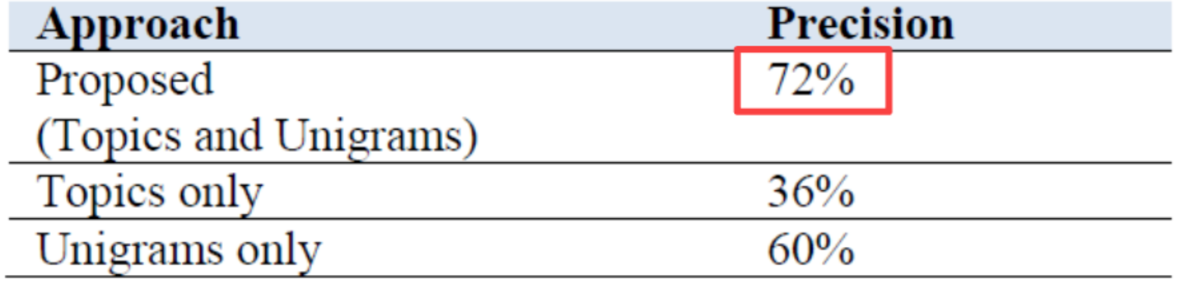
Towards more accurate automatic recommendation of software design patterns [3]

Dataset:

- 14 patterns from the catalog of GoF design patterns
- 32 real design problem scenarios

Approach Details:

- Tokenization and Normalization
- Stop Word Removal
- Porter stemming algorithm
- The natural language toolkit NLTK
- Gensim
- Vector Space Model (VSM)
- TF*IDF (Term Frequency Inverse Document Frequency)
- Improved Sqrt-cosine similarity



Approach	Precision
Proposed (Topics and Unigrams)	72%
Topics only	36%
Unigrams only	60%

[3] Hamdy, Abeer & Elsayed, M.. (2018). Towards more accurate automatic recommendation of software design patterns. Journal of Theoretical and Applied Information Technology. 96. 5069-5079.

Second Approach

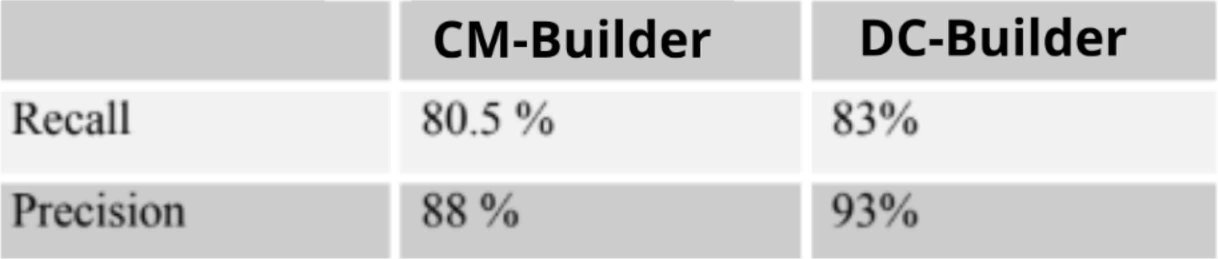
User Requirements To Class Diagram Analysis [4]

Dataset:

- Eleven rules proposed by Chen [4]

Approach Details:

- Gate Framework
- Information extraction system called ANNIE
- Sentence splitter
- Part of speech (POS tagger)
- Syntactic parser

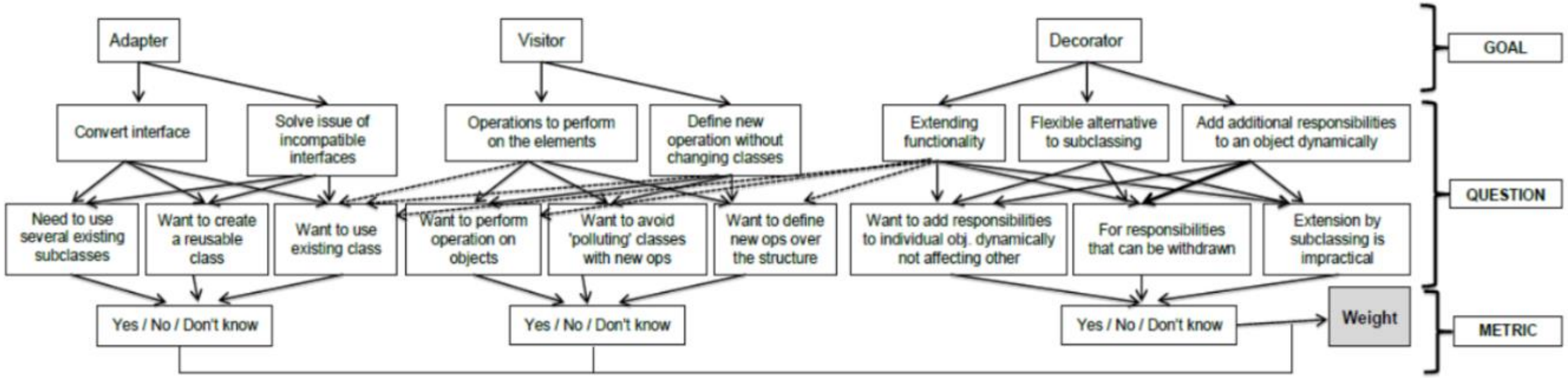


	CM-Builder	DC-Builder
Recall	80.5 %	83%
Precision	88 %	93%

[4] Herchi, Hatem & Ben Abdessalem, Wahiba. (2012). From user requirements to UML class diagram.

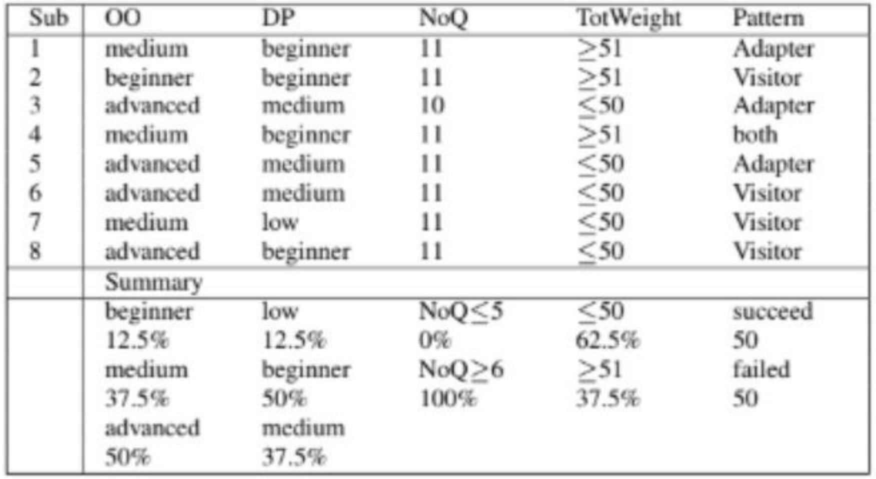
Third Approach

Recommendation System for Design Patterns in Software Development: An DPR [5]


Adapter
Visitor
Decorator
GOAL
Convert interface
Solve issue of incompatible interfaces
Operations to perform on the elements
Define new operation without changing classes
Extending functionality
Flexible alternative to subclassing
Add additional responsibilities to an object dynamically
QUESTION
Need to use several existing subclasses
Want to create a reusable class
Want to use existing class
Want to perform operation on objects
Want to avoid 'polluting' classes with new ops
Want to define new ops over the structure
Want to add responsibilities to individual obj. dynamically not affecting other
For responsibilities that can be withdrawn
Extension by subclassing is impractical
Yes / No / Don't know
Yes / No / Don't know
Yes / No / Don't know
Weight
METRIC

Dataset:

- 23 design patterns by GoF [2].



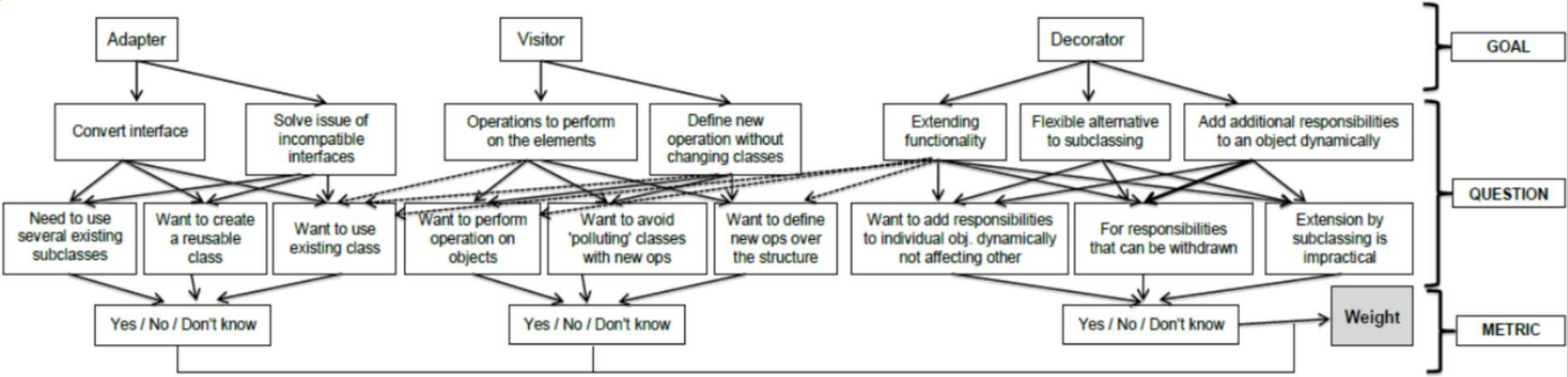
Sub	OO	DP	NoQ	TotWeight	Pattern
1	medium	beginner	11	≥51	Adapter
2	beginner	beginner	11	≥51	Visitor
3	advanced	medium	10	≤50	Adapter
4	medium	beginner	11	≥51	both
5	advanced	medium	11	≤50	Adapter
6	advanced	medium	11	≤50	Visitor
7	medium	low	11	≤50	Visitor
8	advanced	beginner	11	≤50	Visitor
	Summary				
	beginner	low	NoQ≤5	≤50	succeed
	12.5%	12.5%	0%	62.5%	50
	medium	beginner	NoQ≥6	≥51	failed
	37.5%	50%	100%	37.5%	50
	advanced	medium			
	50%	37.5%			

Techniques:

- Goal-Question-Metric (GQM) approach.

[5] Palma, Francis & Farzin, Hadi & Guéhéneuc, Yann-Gaël & Moha, Naouel. (2012). Recommendation System for Design Patterns in Software Development: An DPR Overview. 2012 3rd International Workshop on Recommendation Systems for Software Engineering, RSSE 2012 - Proceedings. 1-5. 10.1109/RSSE.2012.6233399.

Third Approach


Adapter
Visitor
Decorator
GOAL
Convert interface
Solve issue of incompatible interfaces
Operations to perform on the elements
Define new operation without changing classes
Extending functionality
Flexible alternative to subclassing
Add additional responsibilities to an object dynamically
QUESTION
Need to use several existing subclasses
Want to create a reusable class
Want to use existing class
Want to perform operation on objects
Want to avoid 'polluting' classes with new ops
Want to define new ops over the structure
Want to add responsibilities to individual obj. dynamically not affecting other
For responsibilities that can be withdrawn
Extension by subclassing is impractical
Yes / No / Don't know
Yes / No / Don't know
Yes / No / Don't know
Weight
METRIC

Fourth Approach

A GQM-based Approach for Software Process Patterns Recommendation [6]

Dataset:

- Process pattern library, each pattern is described in the form of Name, Intent, Domain, Solution, Initial Context.
- 57 questions for 89 patterns

Techniques:

- Latent Dirichlet Allocation (LDA).
- K-means.
- Euclidean Distance.
- TF-IDF.
- Goal-Question-Metric (GQM).

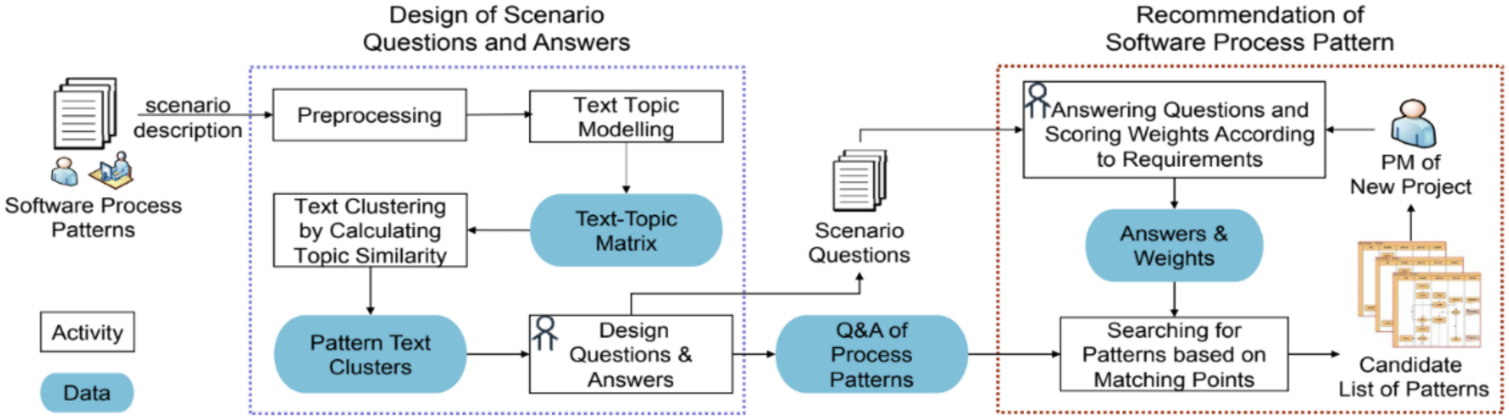

Design of Scenario Questions and Answers
Recommendation of Software Process Pattern
scenario description
Software Process Patterns
Preprocessing
Text Topic Modelling
Text Clustering by Calculating Topic Similarity
Text-Topic Matrix
Pattern Text Clusters
Design Questions & Answers
Scenario Questions
Q&A of Process Patterns
Answering Questions and Scoring Weights According to Requirements
PM of New Project
Answers & Weights
Searching for Patterns based on Matching Points
Candidate List of Patterns
Activity
Data

Fig. 1: Our Approach to Software Process Pattern Recommendation

[6] Meng, Zhangyuan et al. “A GQM-based Approach for Software Process Patterns Recommendation.” SEKE (2017).

GQM Tree

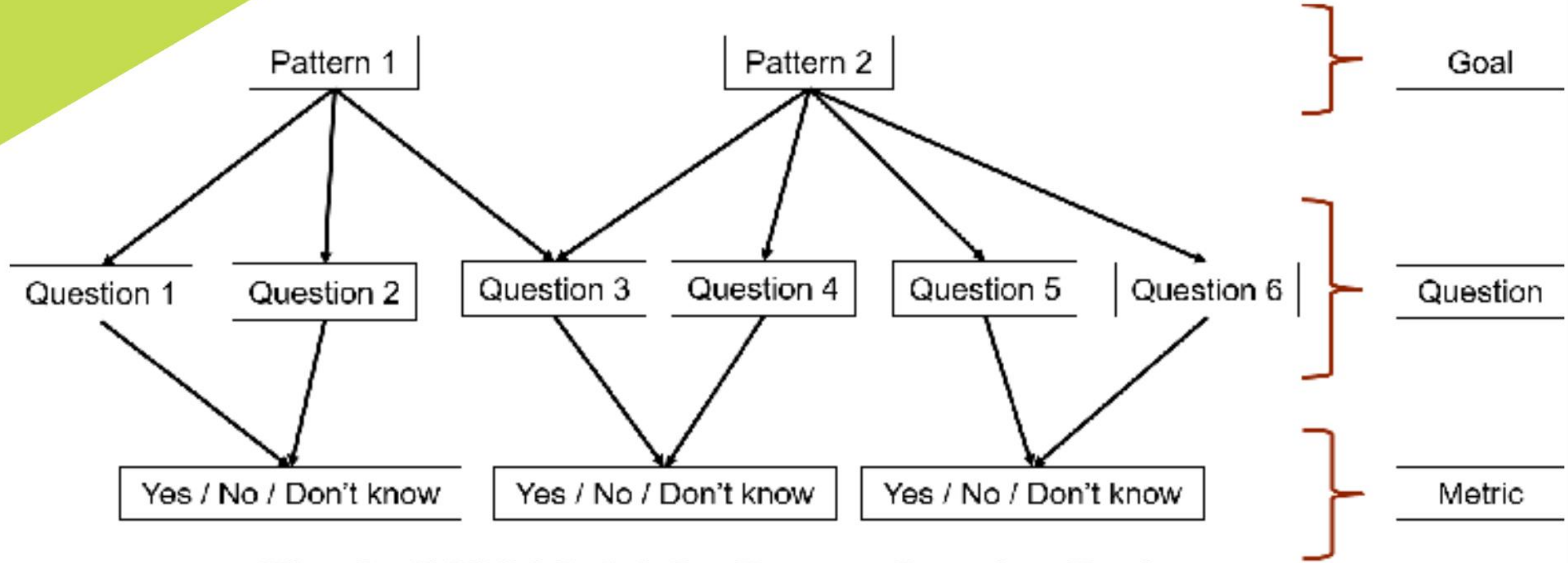
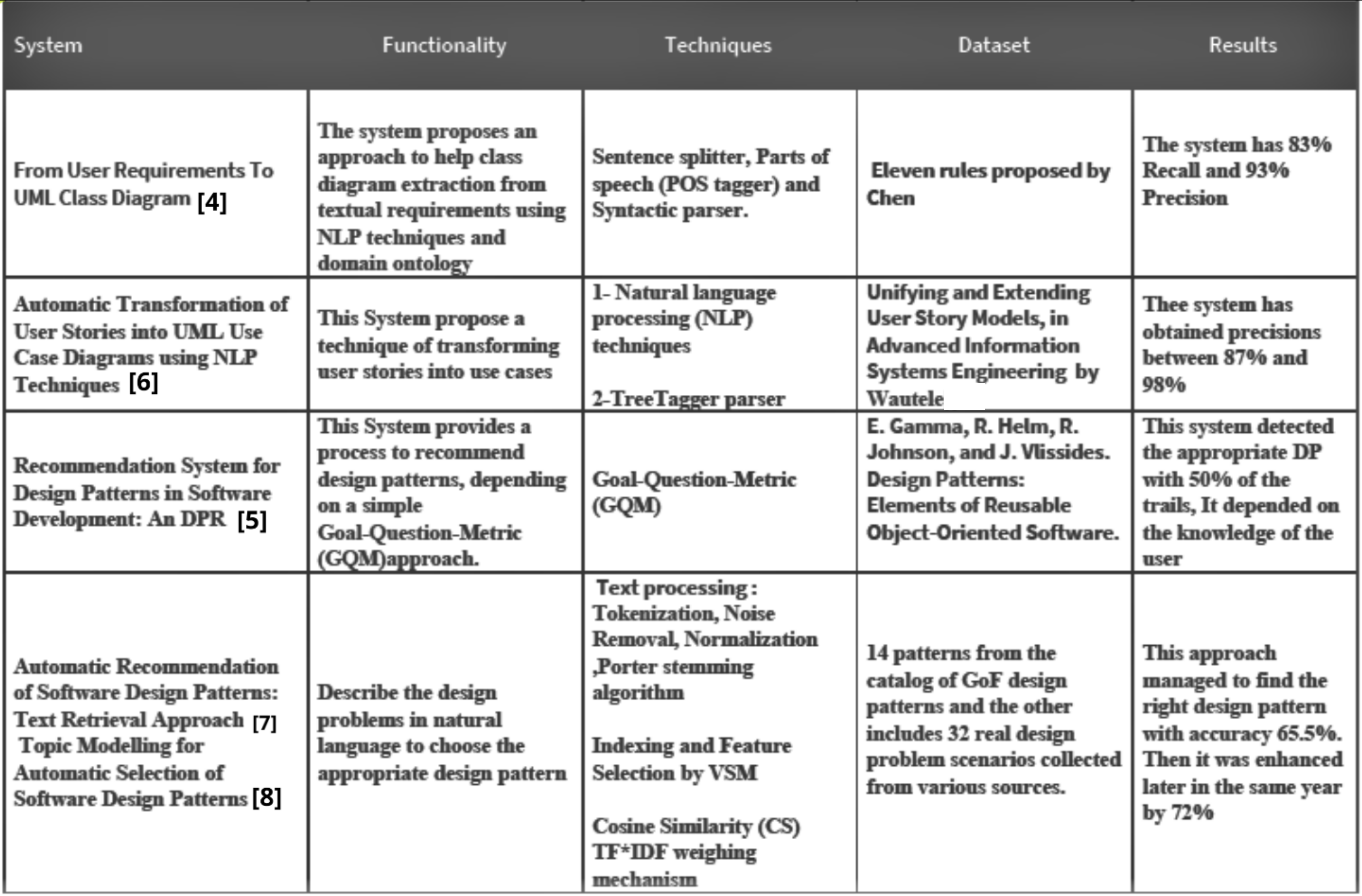


Fig. 2: GQM Model for Pattern Question Design.

Comparison with proposed project 1



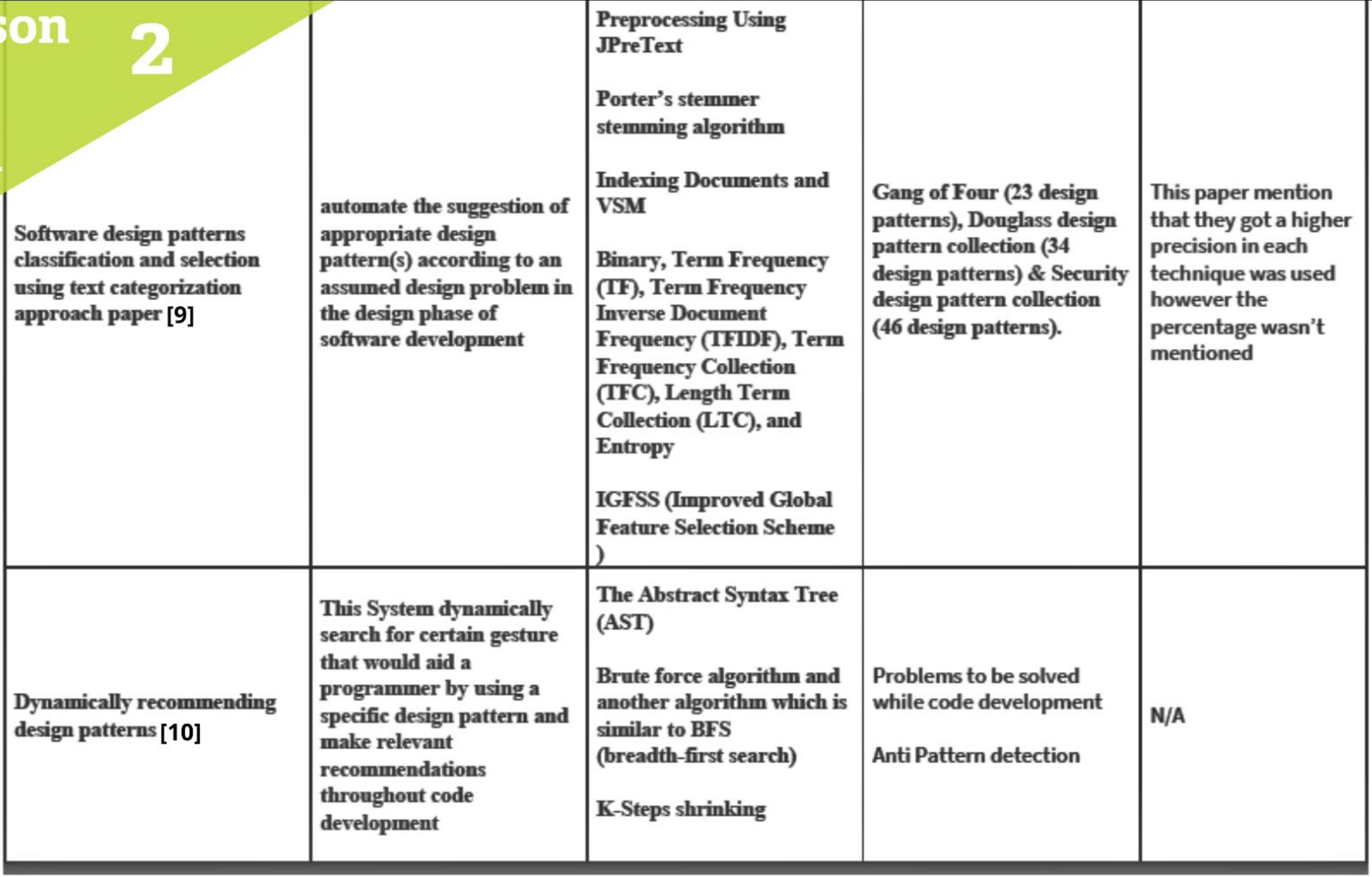
System	Functionality	Techniques	Dataset	Results
From User Requirements To UML Class Diagram [4]	The system proposes an approach to help class diagram extraction from textual requirements using NLP techniques and domain ontology	Sentence splitter, Parts of speech (POS tagger) and Syntactic parser.	Eleven rules proposed by Chen	The system has 83% Recall and 93% Precision
Automatic Transformation of User Stories into UML Use Case Diagrams using NLP Techniques [6]	This System propose a technique of transforming user stories into use cases	1- Natural language processing (NLP) techniques 2-TreeTagger parser	Unifying and Extending User Story Models, in Advanced Information Systems Engineering by Wautele	Thee system has obtained precisions between 87% and 98%
Recommendation System for Design Patterns in Software Development: An DPR [5]	This System provides a process to recommend design patterns, depending on a simple Goal-Question-Metric (GQM)approach.	Goal-Question-Metric (GQM)	E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software.	This system detected the appropriate DP with 50% of the trails, It depended on the knowledge of the user
Automatic Recommendation of Software Design Patterns: Text Retrieval Approach [7] Topic Modelling for Automatic Selection of Software Design Patterns [8]	Describe the design problems in natural language to choose the appropriate design pattern	Text processing : Tokenization, Noise Removal, Normalization ,Porter stemming algorithm Indexing and Feature Selection by VSM Cosine Similarity (CS) TF*IDF weighing mechanism	14 patterns from the catalog of GoF design patterns and the other includes 32 real design problem scenarios collected from various sources.	This approach managed to find the right design pattern with accuracy 65.5%. Then it was enhanced later in the same year by 72%

[6] Elallaoui, Meryem et al. "Automatic Transformation of User Stories into UML Use Case Diagrams using NLP Techniques." ANT/SEIT (2018).

[7] Hamdy, Abeer and Mohamed Elsayed. "Automatic Recommendation of Software Design Patterns: Text Retrieval Approach." JSW 13 (2018): 260-268.

[8] Hamdy, Abeer and Mohamed Elsayed. "Topic modelling for automatic selection of software design patterns." ICGDA (2018).

Comparison with proposed project 2



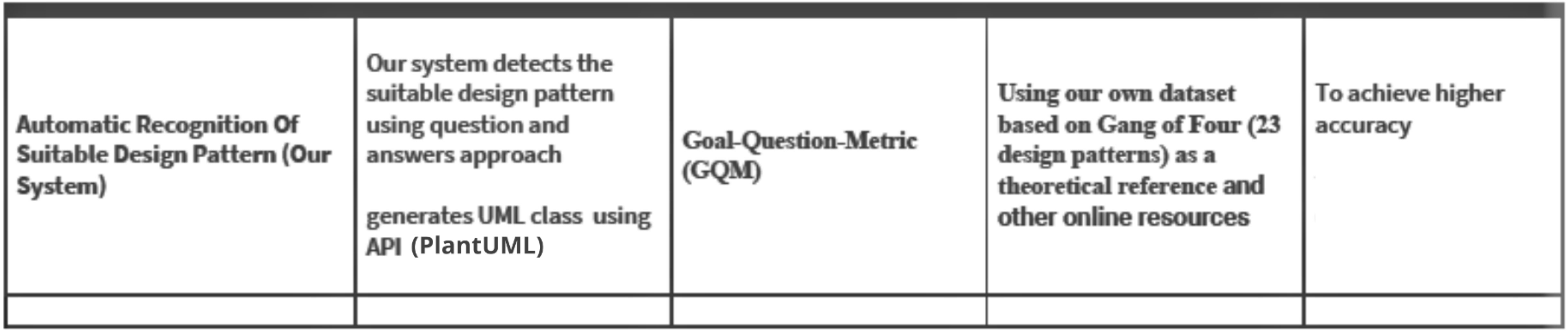
Software design patterns classification and selection using text categorization approach paper [9]	automate the suggestion of appropriate design pattern(s) according to an assumed design problem in the design phase of software development	Preprocessing Using JPreText Porter's stemmer stemming algorithm Indexing Documents and VSM Binary, Term Frequency (TF), Term Frequency Inverse Document Frequency (TFIDF), Term Frequency Collection (TFC), Length Term Collection (LTC), and Entropy IGFSS (Improved Global Feature Selection Scheme)	Gang of Four (23 design patterns), Douglass design pattern collection (34 design patterns) & Security design pattern collection (46 design patterns).	This paper mention that they got a higher precision in each technique was used however the percentage wasn't mentioned
Dynamically recommending design patterns [10]	This System dynamically search for certain gesture that would aid a programmer by using a specific design pattern and make relevant recommendations throughout code development	The Abstract Syntax Tree (AST) Brute force algorithm and another algorithm which is similar to BFS (breadth-first search) K-Steps shrinking	Problems to be solved while code development Anti Pattern detection	N/A

[9] Hussain, Shahid et al. "Software design patterns classification and selection using text categorization approach." Appl. Soft Comput. 58 (2017): 225-244.

[10] Smith, Steven O and D. R. Plante. "Dynamically recommending design patterns." SEKE (2012).

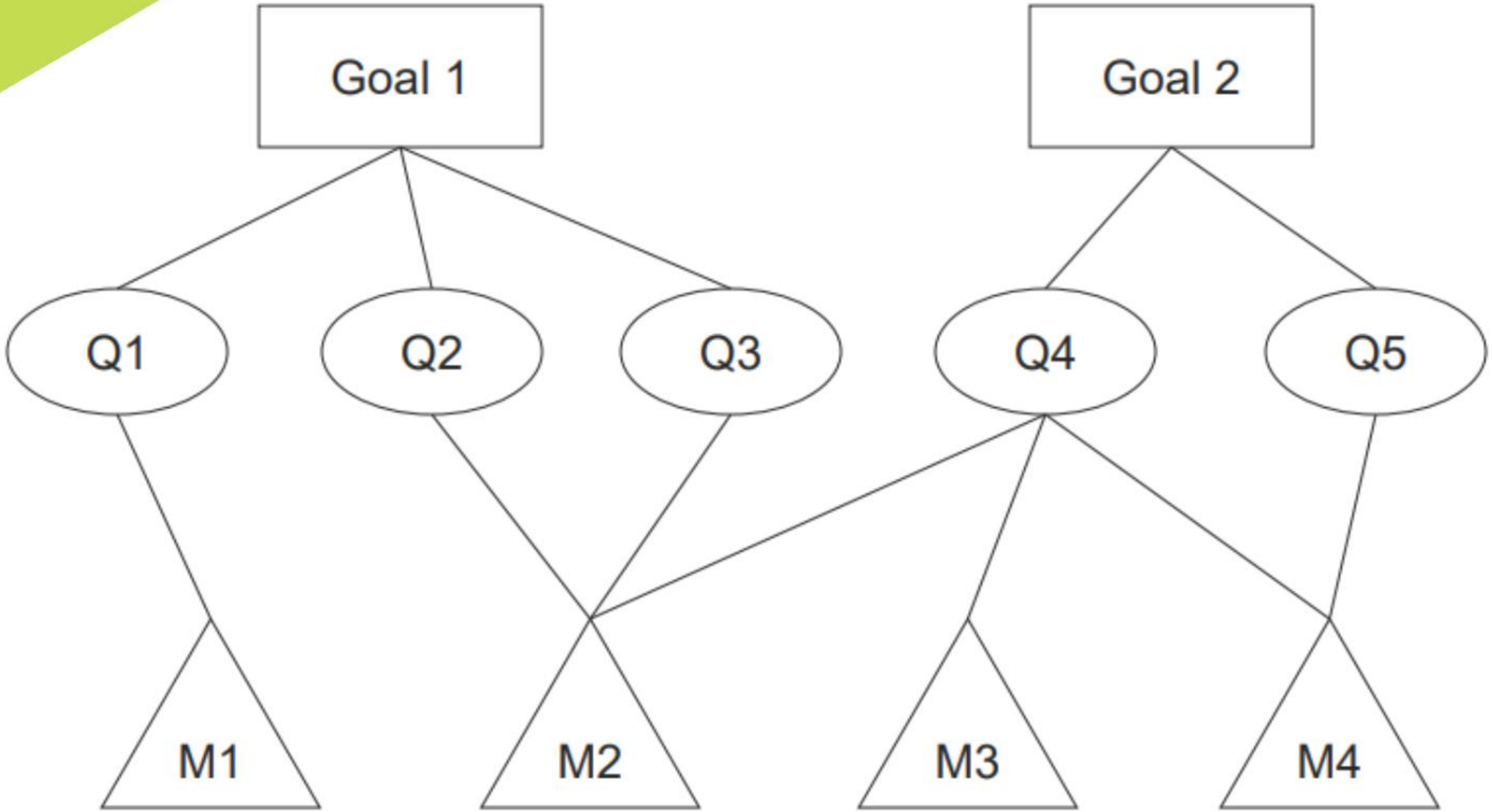
Comparison with proposed project

3

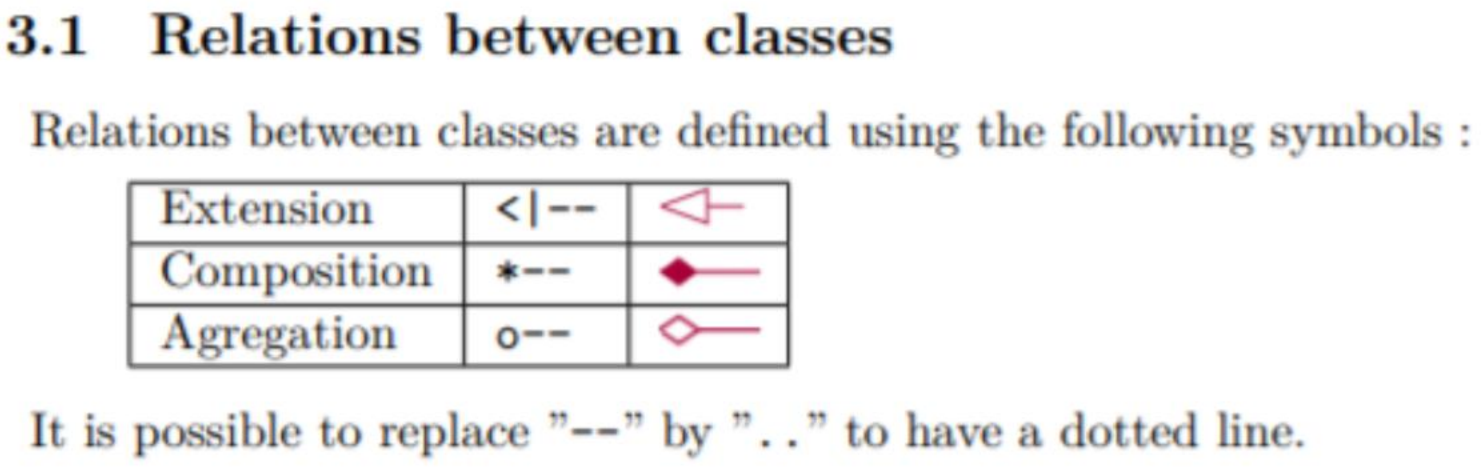


Automatic Recognition Of Suitable Design Pattern (Our System)	Our system detects the suitable design pattern using question and answers approach generates UML class using API (PlantUML)	Goal-Question-Metric (GQM)	Using our own dataset based on Gang of Four (23 design patterns) as a theoretical reference and other online resources	To achieve higher accuracy

GQM Tree



PlantUML

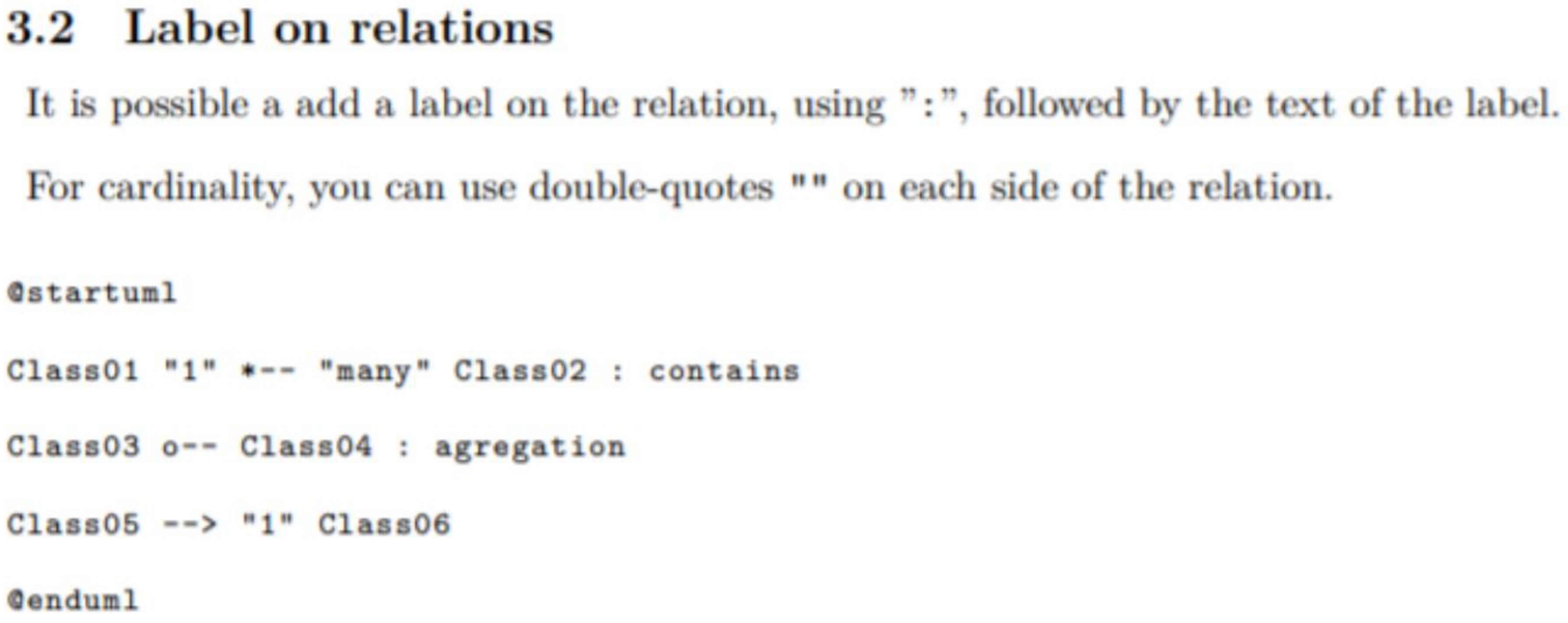


3.1 Relations between classes

Relations between classes are defined using the following symbols :

Extension	`<|--`	◁—
Composition	`*--`	◆—
Agregation	`o--`	◇—

It is possible to replace "--" by ".." to have a dotted line.



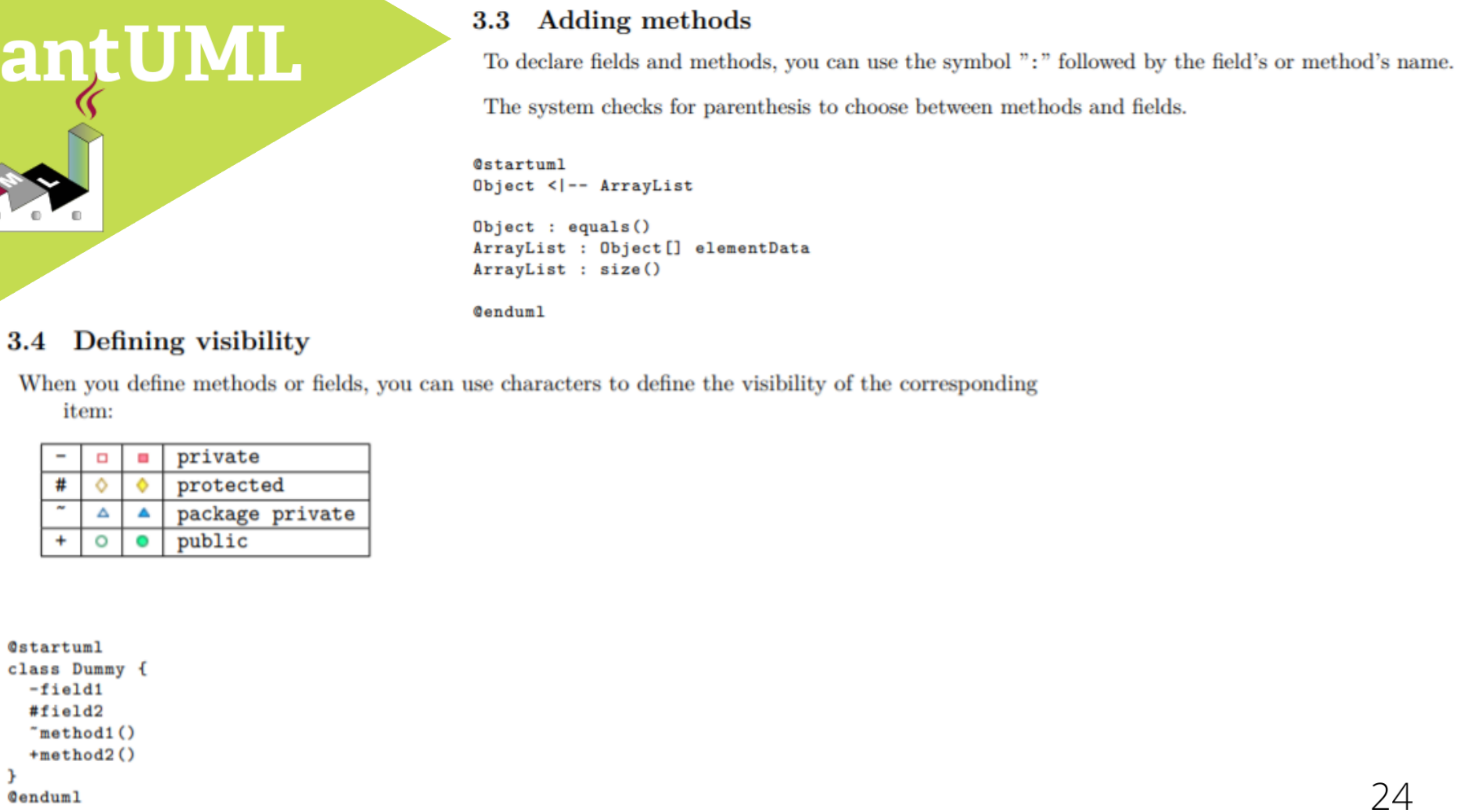
3.2 Label on relations

It is possible a add a label on the relation, using ":", followed by the text of the label.

For cardinality, you can use double-quotes "" on each side of the relation.

```
@startuml

Class01 "1" *-- "many" Class02 : contains

Class03 o-- Class04 : agregation

Class05 --> "1" Class06

@enduml
```

PlantUML Language Reference Guide (1.2019.9)



3.3 Adding methods

To declare fields and methods, you can use the symbol ":" followed by the field's or method's name.

The system checks for parenthesis to choose between methods and fields.

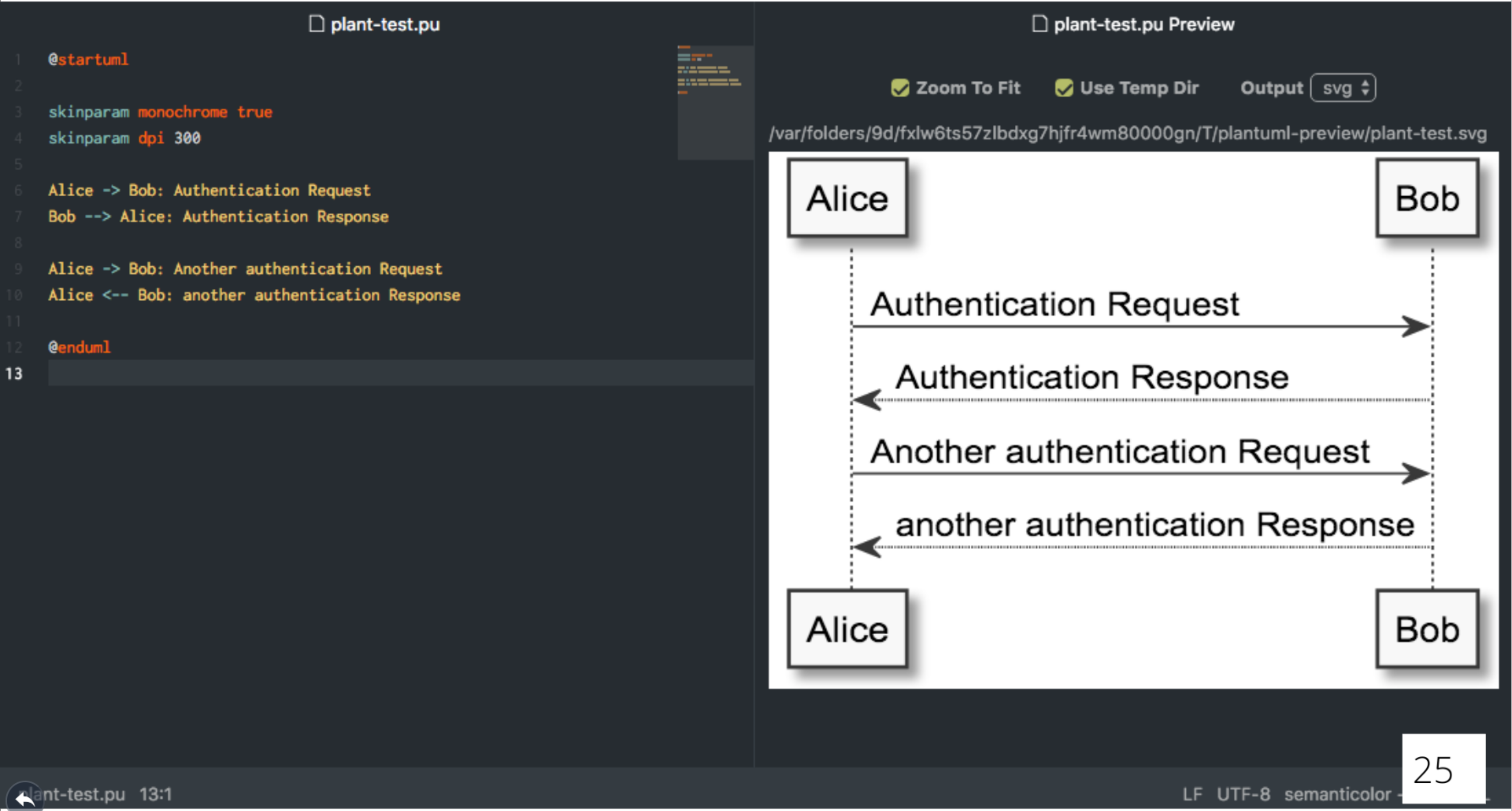
```
@startuml
Object <|-- ArrayList

Object : equals()
ArrayList : Object[] elementData
ArrayList : size()

@enduml
```

3.4 Defining visibility

When you define methods or fields, you can use characters to define the visibility of the corresponding item:

-	□	■	private
#	◇	◆	protected
~	△	▲	package private
+	○	●	public

```
@startuml
class Dummy {
  -field1
  #field2
  ~method1()
  +method2()
}
@enduml
```



plant-test.pu

```
@startuml

skinparam monochrome true
skinparam dpi 300

Alice -> Bob: Authentication Request
Bob --> Alice: Authentication Response

Alice -> Bob: Another authentication Request
Alice <-- Bob: another authentication Response

@enduml

```

plant-test.pu Preview

☑ Zoom To Fit ☑ Use Temp Dir Output svg

/var/folders/9d/fxlw6ts57zlbdxg7hjfr4wm80000gn/T/plantuml-preview/plant-test.svg

Alice
Bob
Authentication Request
Authentication Response
Another authentication Request
another authentication Response
Alice
Bob

lant-test.pu 13:1 LF UTF-8 semanticolor

Problem Statement

Our proposed approach will help the software engineers to find the suitable design pattern for a specific problem scenario and generate the class diagram easily to avoid following problems:

- Solve the problem of Anti-Pattern:
    - Big ball of mud
    - God object
    - Cargo cult programming
- Complicated code.
- The selection is confusing for novice engineers.

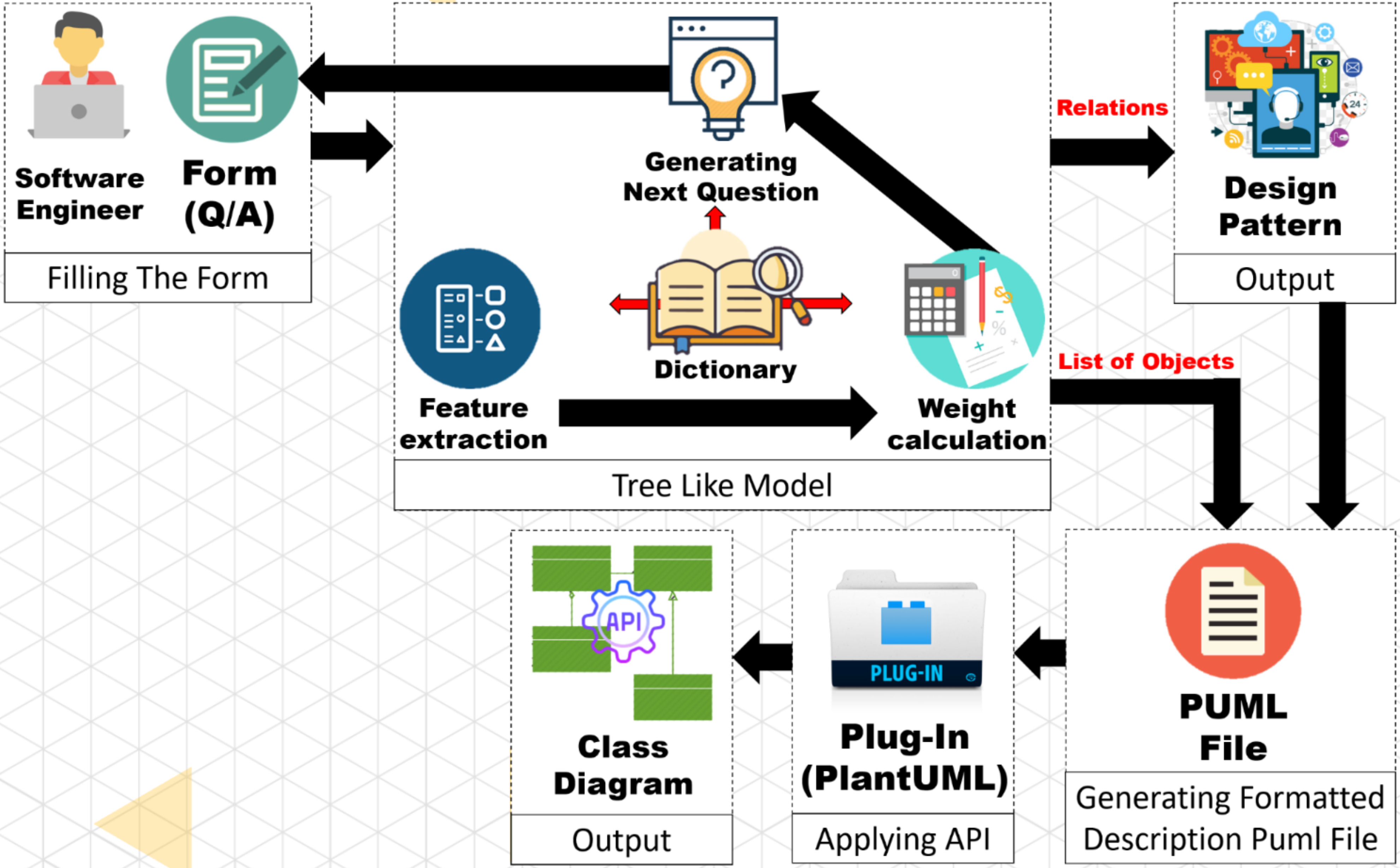
Why ?

WHY?

As there are similar design patterns with the same objective but with different functionality

Factory Pattern	Abstract Factory Pattern	Builder Pattern	Composite Pattern
• Create object through inheritance • Produce only one product	• Create object through composition • Produce families of products	It is used to create group of objects of predefined types.	It creates Parent - Child relations between our objects.

System Overview



EXPECTED RESULTS

Contribution

- Provide an automatic selection of the suitable design pattern according to the user's specific design problem.

- Create a class diagram depending on the design pattern chosen.

- More efficient Design Pattern selection than previous researches as it won't depend on the engineer's knowledge level.

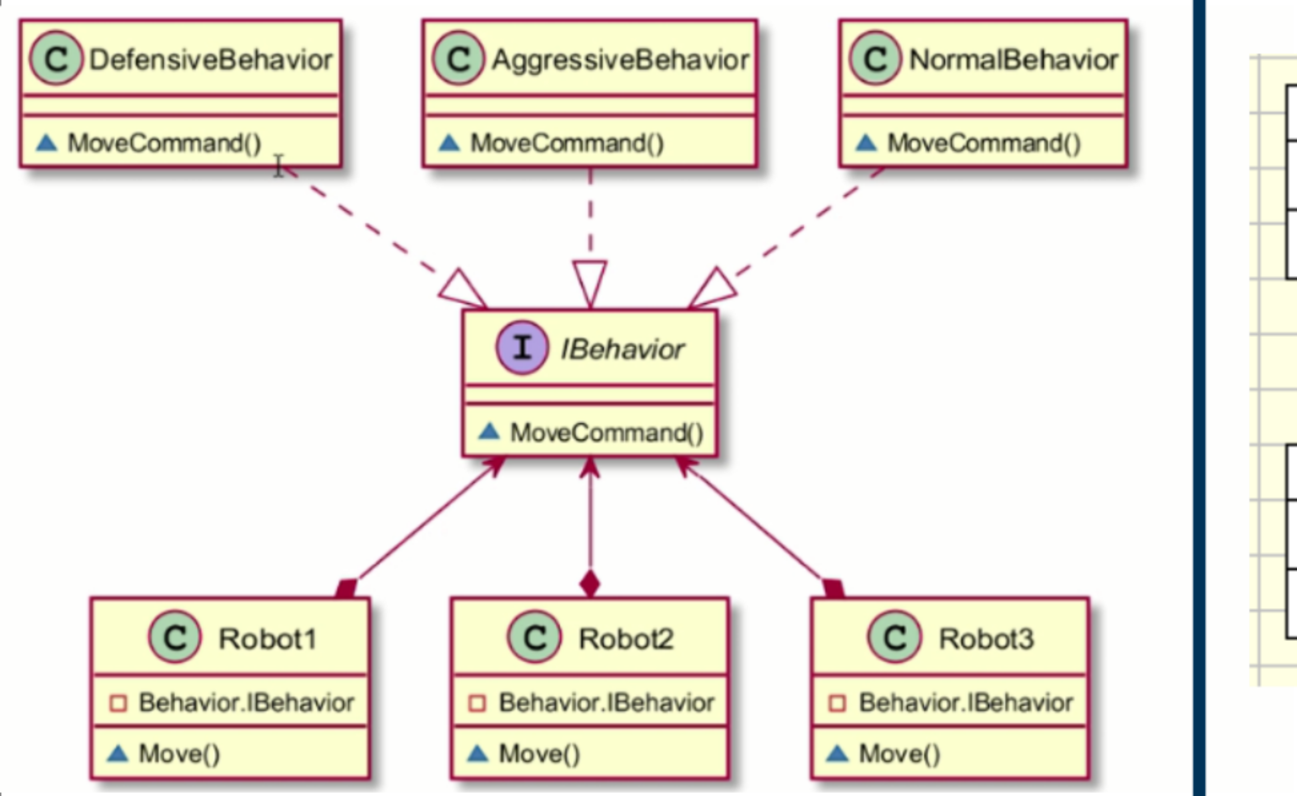
Contribution

- Achieving higher accuracy than previous researches which was 50%.

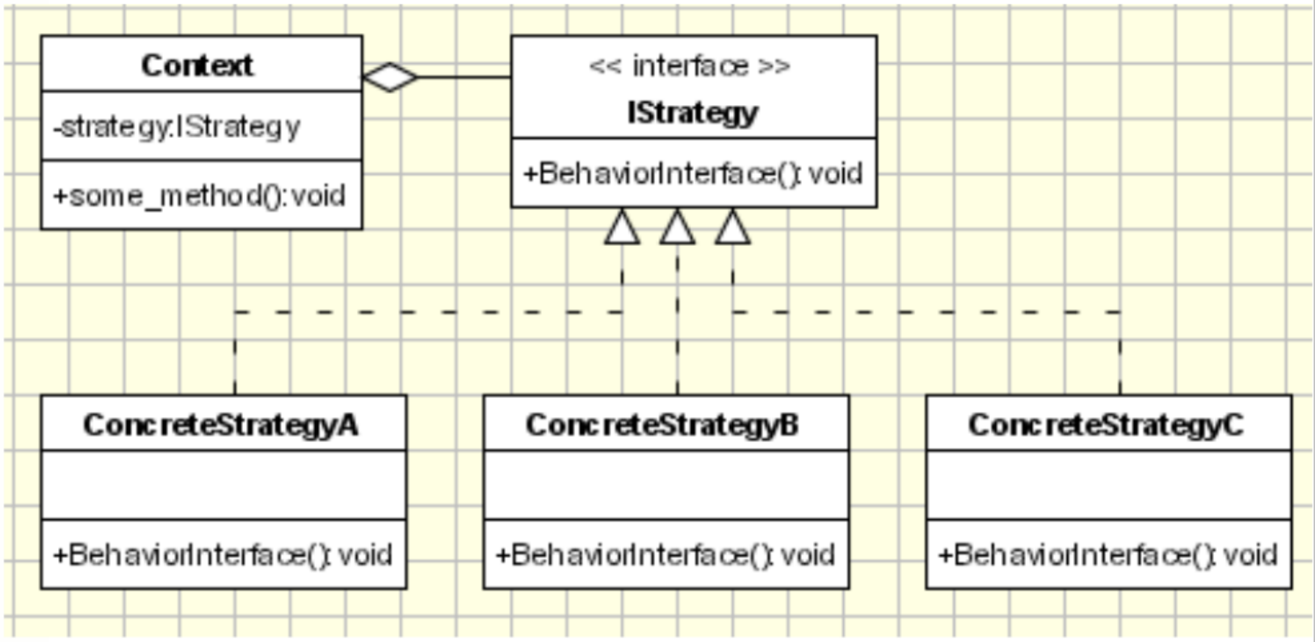
- Provide a system that can do both processes, selecting the suitable DP and creating the class diagram suitable for it.

- For the 1st time building a comprehensive dictionary that differentiate between the different design patterns, using a variety of books and researches.

Demo 1/2

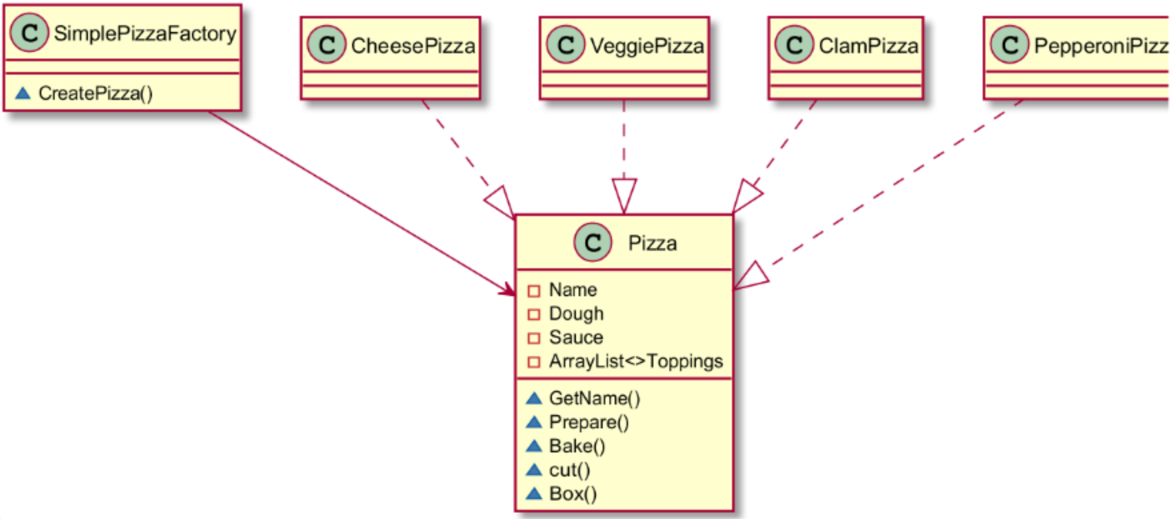


Strategy Demo Result

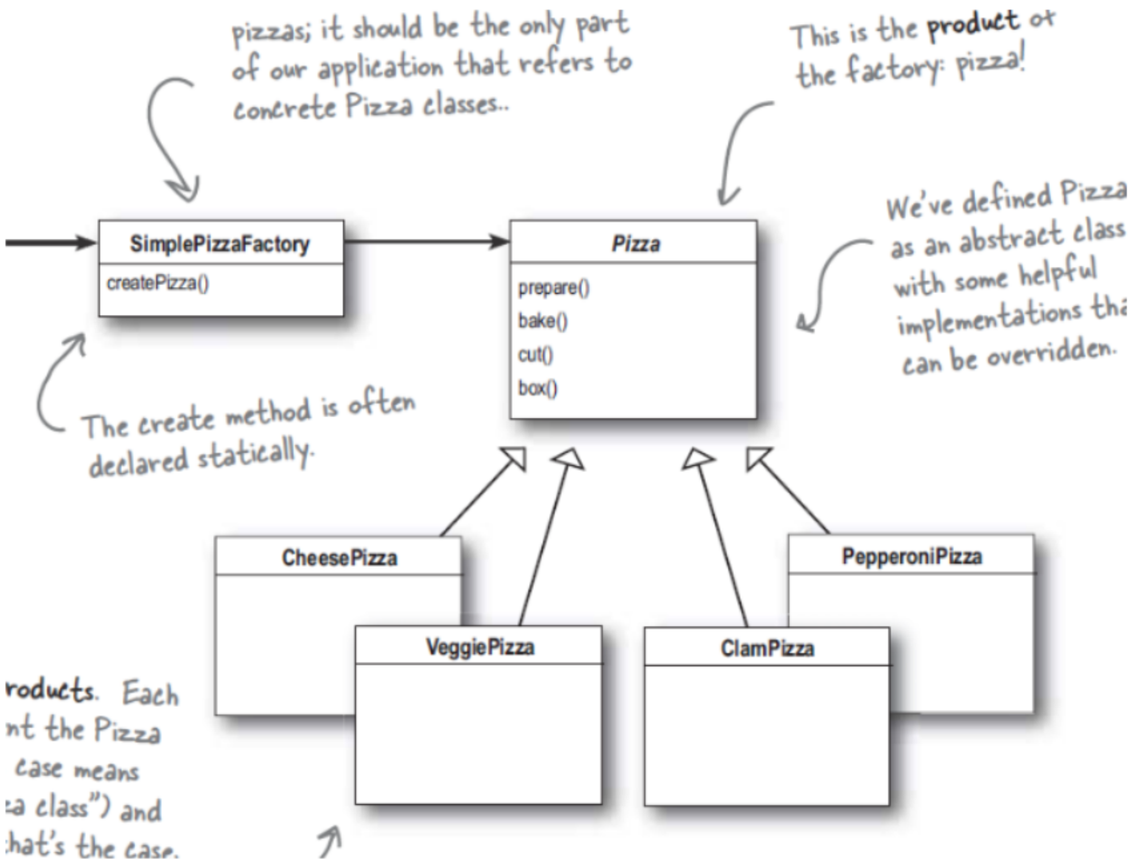


Strategy in book

Demo 2/2



Factory Demo Result



pizzas; it should be the only part of our application that refers to concrete Pizza classes..

This is the **product** of the factory: pizza!

We've defined Pizza as an abstract class with some helpful implementations tha can be overridden.

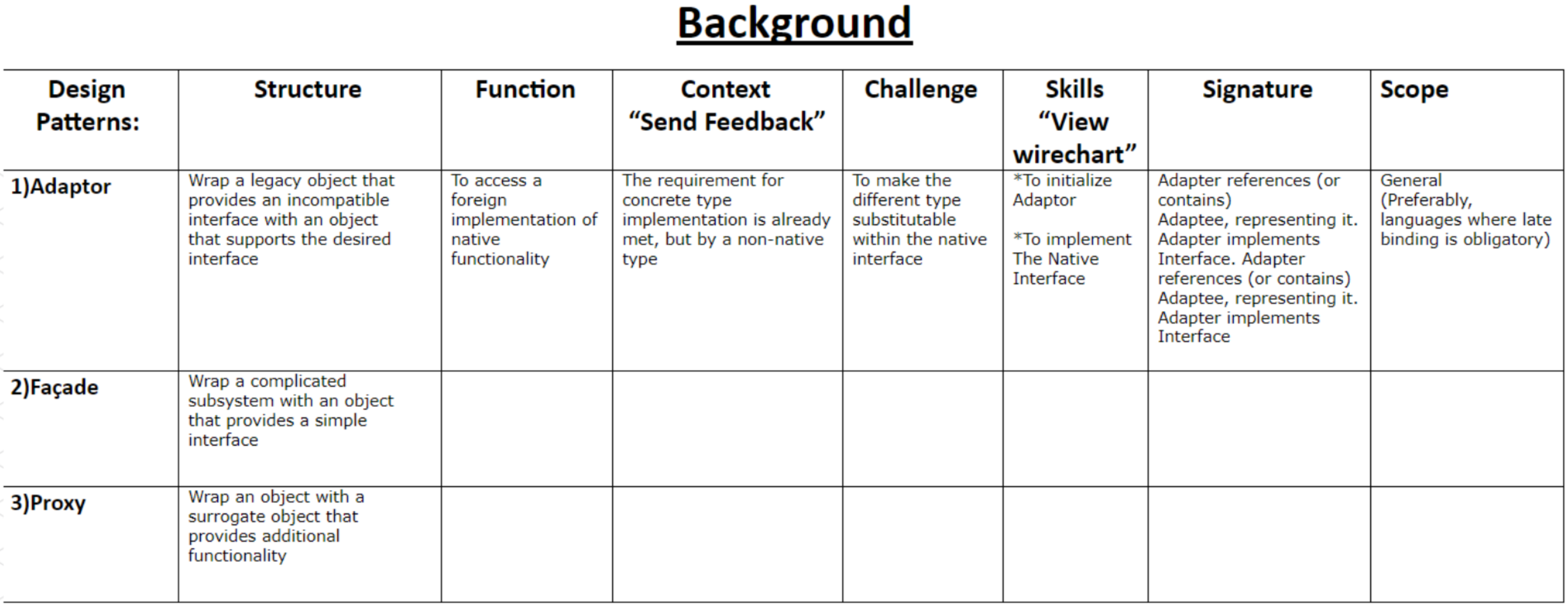
The create method is often declared statically.

roducts. Each
nt the Pizza
case means
a class") and
hat's the case.

Factory in book

Appendix

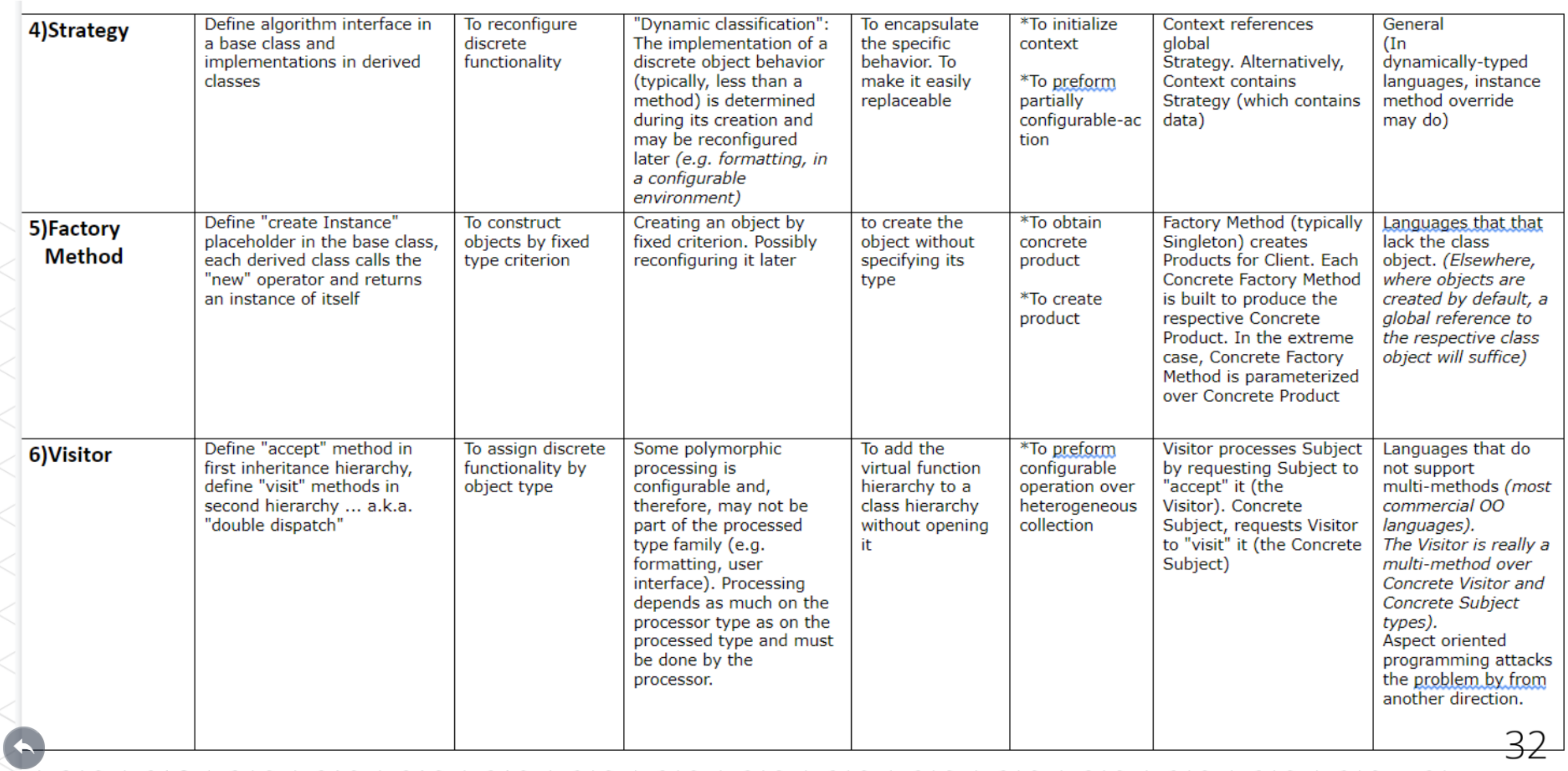
Appendix 1/7



Background

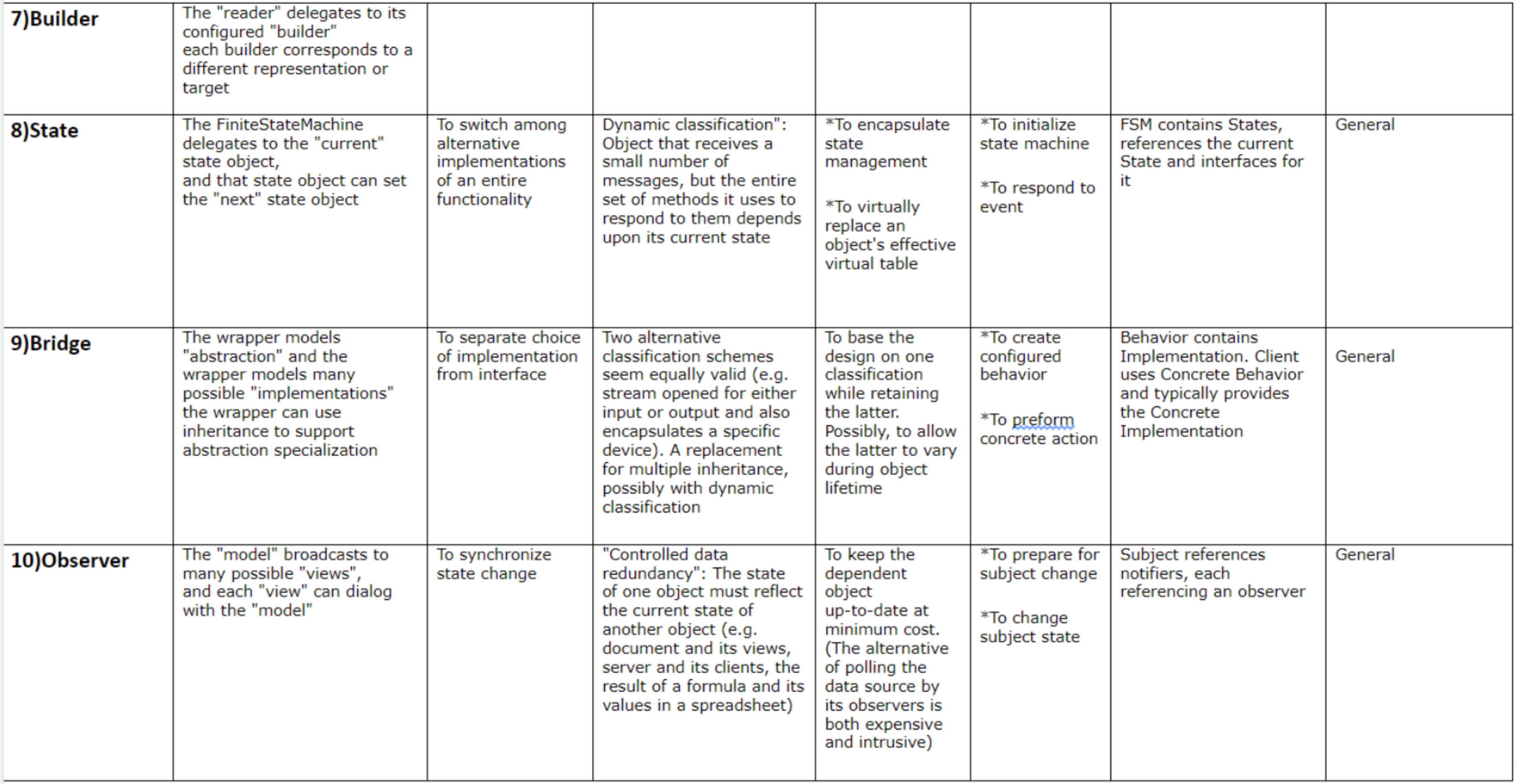
Design Patterns:	Structure	Function	Context “Send Feedback”	Challenge	Skills “View wirechart”	Signature	Scope
1)Adaptor	Wrap a legacy object that provides an incompatible interface with an object that supports the desired interface	To access a foreign implementation of native functionality	The requirement for concrete type implementation is already met, but by a non-native type	To make the different type substitutable within the native interface	*To initialize Adaptor *To implement The Native Interface	Adapter references (or contains) Adaptee, representing it. Adapter implements Interface. Adapter references (or contains) Adaptee, representing it. Adapter implements Interface	General (Preferably, languages where late binding is obligatory)
2)Façade	Wrap a complicated subsystem with an object that provides a simple interface						
3)Proxy	Wrap an object with a surrogate object that provides additional functionality						

Appendix 2/7



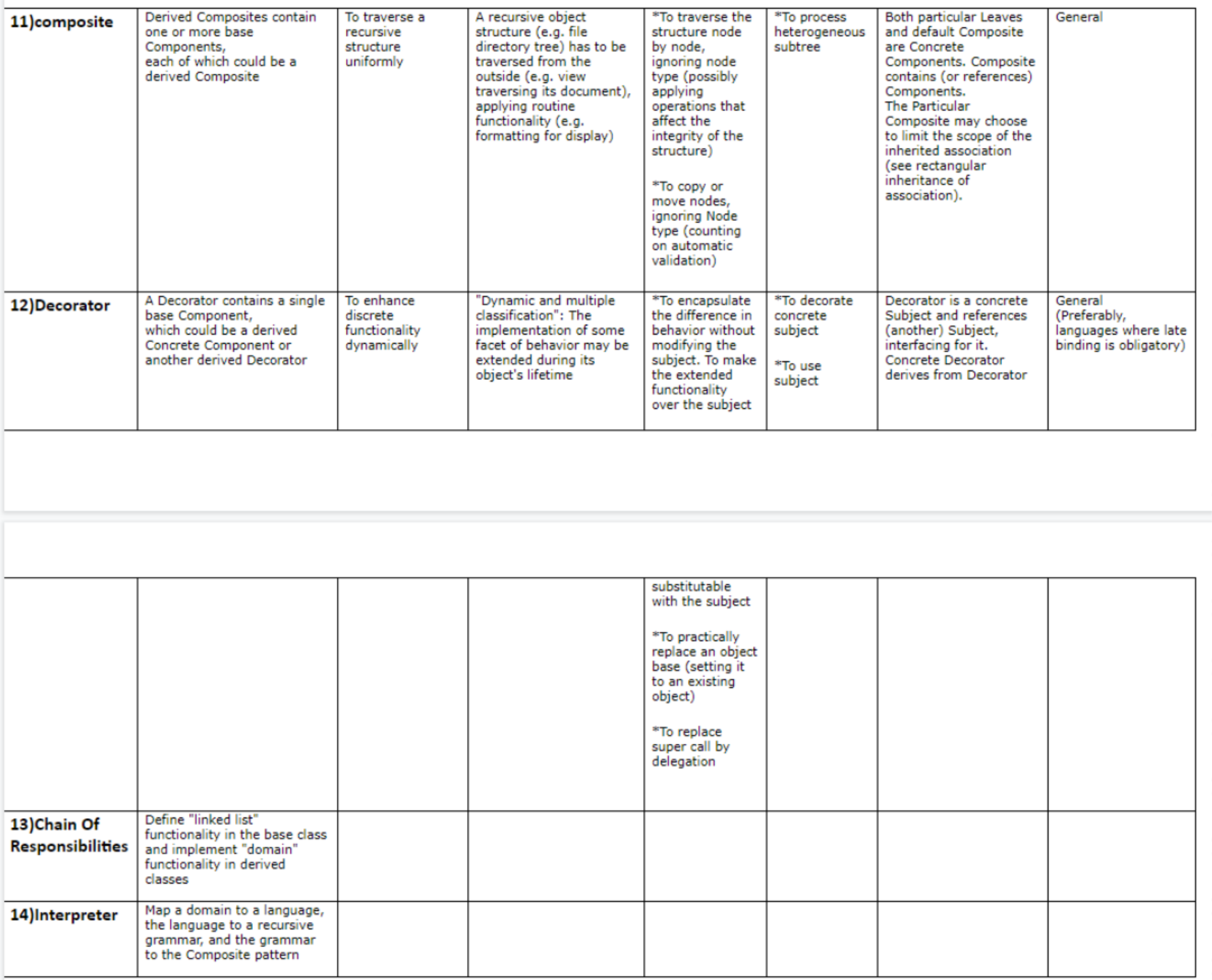
4)Strategy	Define algorithm interface in a base class and implementations in derived classes	To reconfigure discrete functionality	"Dynamic classification": The implementation of a discrete object behavior (typically, less than a method) is determined during its creation and may be reconfigured later *(e.g. formatting, in a configurable environment)*	To encapsulate the specific behavior. To make it easily replaceable	*To initialize context *To preform partially configurable-action	Context references global Strategy. Alternatively, Context contains Strategy (which contains data)	General (In dynamically-typed languages, instance method override may do)
5)Factory Method	Define "create Instance" placeholder in the base class, each derived class calls the "new" operator and returns an instance of itself	To construct objects by fixed type criterion	Creating an object by fixed criterion. Possibly reconfiguring it later	to create the object without specifying its type	*To obtain concrete product *To create product	Factory Method (typically Singleton) creates Products for Client. Each Concrete Factory Method is built to produce the respective Concrete Product. In the extreme case, Concrete Factory Method is parameterized over Concrete Product	Languages that that lack the class object. *(Elsewhere, where objects are created by default, a global reference to the respective class object will suffice)*
6)Visitor	Define "accept" method in first inheritance hierarchy, define "visit" methods in second hierarchy ... a.k.a. "double dispatch"	To assign discrete functionality by object type	Some polymorphic processing is configurable and, therefore, may not be part of the processed type family (e.g. formatting, user interface). Processing depends as much on the processor type as on the processed type and must be done by the processor.	To add the virtual function hierarchy to a class hierarchy without opening it	*To preform configurable operation over heterogeneous collection	Visitor processes Subject by requesting Subject to "accept" it (the Visitor). Concrete Subject, requests Visitor to "visit" it (the Concrete Subject)	Languages that do not support multi-methods *(most commercial OO languages)*. *The Visitor is really a multi-method over Concrete Visitor and Concrete Subject types)*. Aspect oriented programming attacks the problem by from another direction.

Appendix 3/7



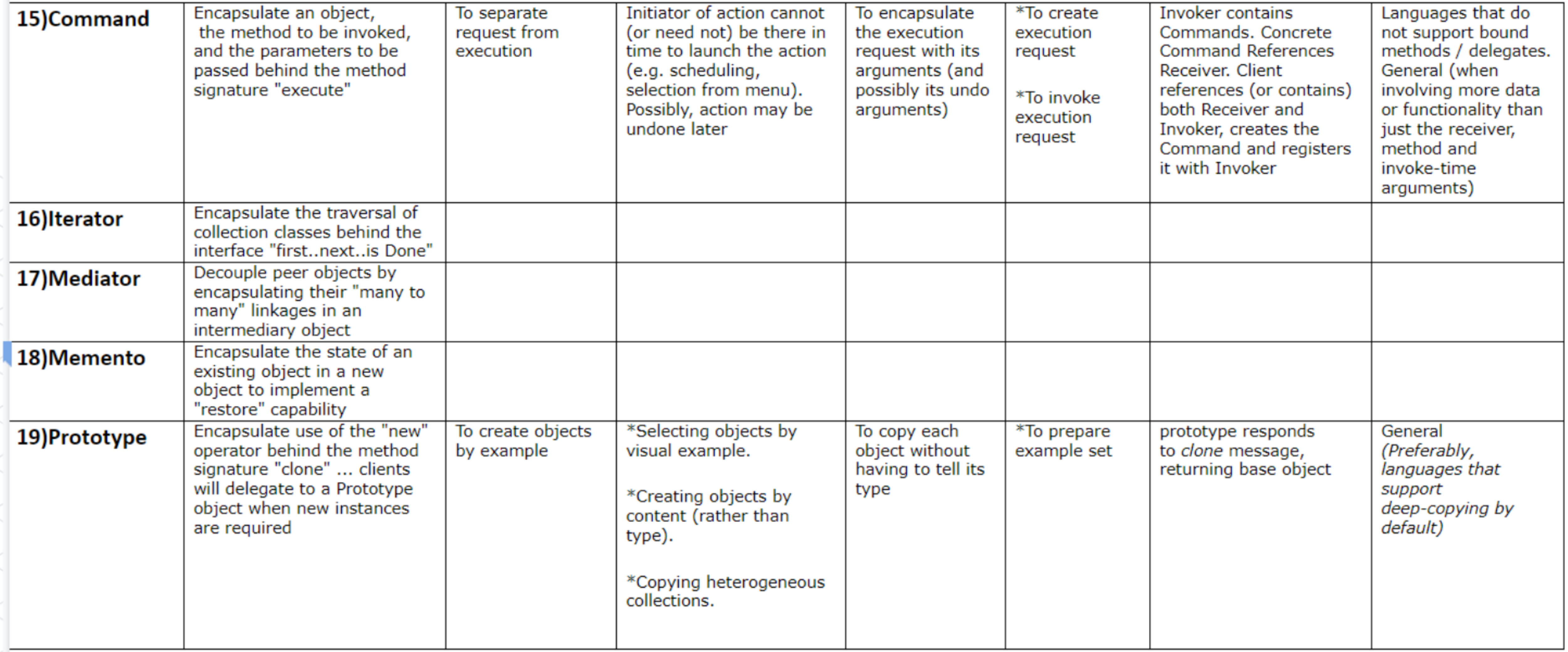
7)Builder	The "reader" delegates to its configured "builder" each builder corresponds to a different representation or target						
8)State	The FiniteStateMachine delegates to the "current" state object, and that state object can set the "next" state object	To switch among alternative implementations of an entire functionality	Dynamic classification": Object that receives a small number of messages, but the entire set of methods it uses to respond to them depends upon its current state	*To encapsulate state management *To virtually replace an object's effective virtual table	*To initialize state machine *To respond to event	FSM contains States, references the current State and interfaces for it	General
9)Bridge	The wrapper models "abstraction" and the wrapper models many possible "implementations" the wrapper can use inheritance to support abstraction specialization	To separate choice of implementation from interface	Two alternative classification schemes seem equally valid (e.g. stream opened for either input or output and also encapsulates a specific device). A replacement for multiple inheritance, possibly with dynamic classification	To base the design on one classification while retaining the latter. Possibly, to allow the latter to vary during object lifetime	*To create configured behavior *To preform concrete action	Behavior contains Implementation. Client uses Concrete Behavior and typically provides the Concrete Implementation	General
10)Observer	The "model" broadcasts to many possible "views", and each "view" can dialog with the "model"	To synchronize state change	"Controlled data redundancy": The state of one object must reflect the current state of another object (e.g. document and its views, server and its clients, the result of a formula and its values in a spreadsheet)	To keep the dependent object up-to-date at minimum cost. (The alternative of polling the data source by its observers is both expensive and intrusive)	*To prepare for subject change *To change subject state	Subject references notifiers, each referencing an observer	General

Appendix 4/7



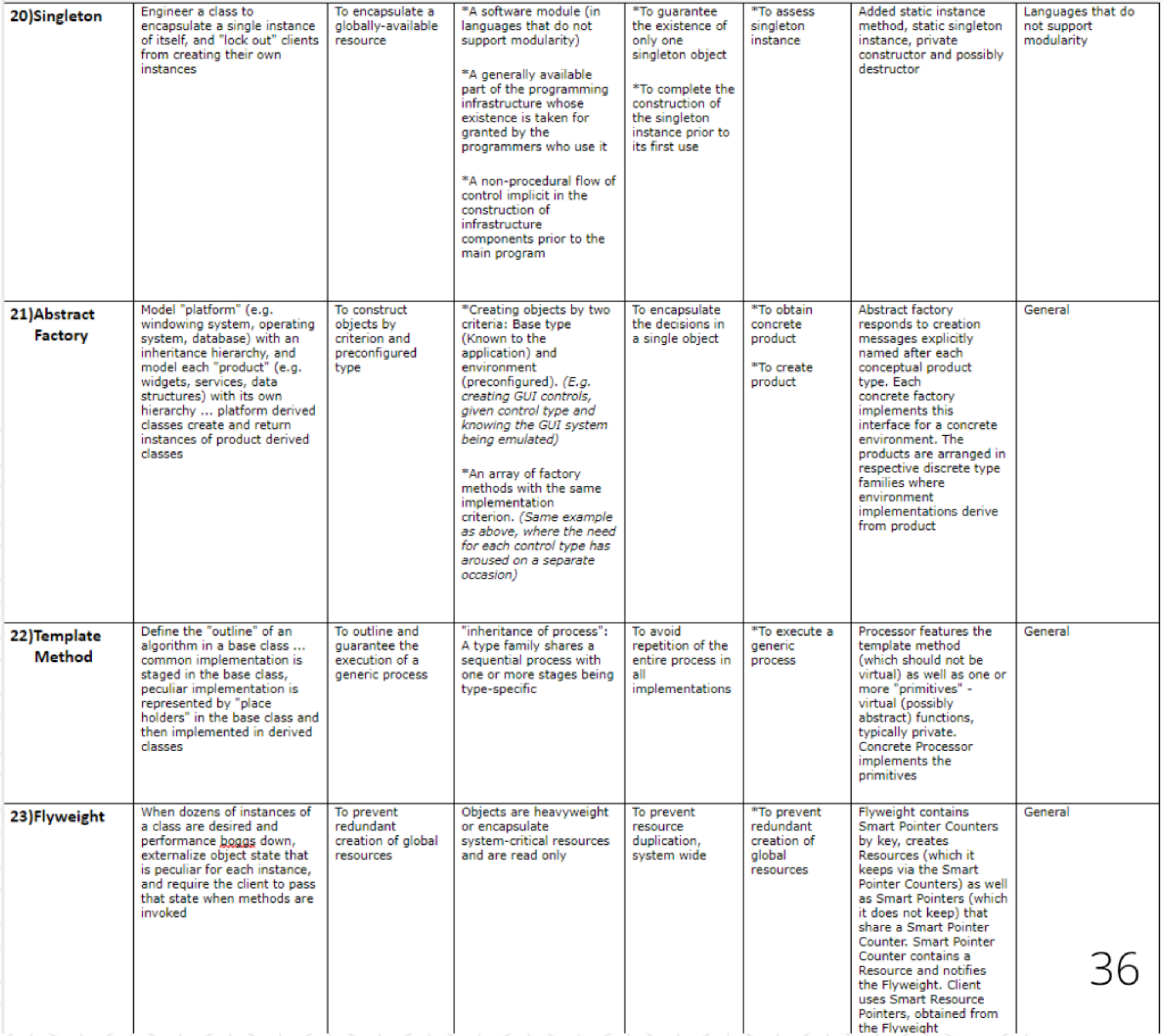
11)composite	Derived Composites contain one or more base Components, each of which could be a derived Composite	To traverse a recursive structure uniformly	A recursive object structure (e.g. file directory tree) has to be traversed from the outside (e.g. view traversing its document), applying routine functionality (e.g. formatting for display)	*To traverse the structure node by node, ignoring node type (possibly applying operations that affect the integrity of the structure) *To copy or move nodes, ignoring Node type (counting on automatic validation)	*To process heterogeneous subtree	Both particular Leaves and default Composite are Concrete Components. Composite contains (or references) Components. The Particular Composite may choose to limit the scope of the inherited association (see rectangular inheritance of association).	General
12)Decorator	A Decorator contains a single base Component, which could be a derived Concrete Component or another derived Decorator	To enhance discrete functionality dynamically	"Dynamic and multiple classification": The implementation of some facet of behavior may be extended during its object's lifetime	*To encapsulate the difference in behavior without modifying the subject. To make the extended functionality over the subject substitutable with the subject *To practically replace an object base (setting it to an existing object) *To replace super call by delegation	*To decorate concrete subject *To use subject	Decorator is a concrete Subject and references (another) Subject, interfacing for it. Concrete Decorator derives from Decorator	General (Preferably, languages where late binding is obligatory)
13)Chain Of Responsibilities	Define "linked list" functionality in the base class and implement "domain" functionality in derived classes						
14)Interpreter	Map a domain to a language, the language to a recursive grammar, and the grammar to the Composite pattern						

Appendix 5/7



15)Command	Encapsulate an object, the method to be invoked, and the parameters to be passed behind the method signature "execute"	To separate request from execution	Initiator of action cannot (or need not) be there in time to launch the action (e.g. scheduling, selection from menu). Possibly, action may be undone later	To encapsulate the execution request with its arguments (and possibly its undo arguments)	*To create execution request *To invoke execution request	Invoker contains Commands. Concrete Command References Receiver. Client references (or contains) both Receiver and Invoker, creates the Command and registers it with Invoker	Languages that do not support bound methods / delegates. General (when involving more data or functionality than just the receiver, method and invoke-time arguments)
16)Iterator	Encapsulate the traversal of collection classes behind the interface "first..next..is Done"						
17)Mediator	Decouple peer objects by encapsulating their "many to many" linkages in an intermediary object						
18)Memento	Encapsulate the state of an existing object in a new object to implement a "restore" capability						
19)Prototype	Encapsulate use of the "new" operator behind the method signature "clone" ... clients will delegate to a Prototype object when new instances are required	To create objects by example	*Selecting objects by visual example. *Creating objects by content (rather than type). *Copying heterogeneous collections.	To copy each object without having to tell its type	*To prepare example set	prototype responds to *clone* message, returning base object	General *(Preferably, languages that support deep-copying by default)*

Appendix 6/7



20)Singleton	Engineer a class to encapsulate a single instance of itself, and "lock out" clients from creating their own instances	To encapsulate a globally-available resource	*A software module (in languages that do not support modularity) *A generally available part of the programming infrastructure whose existence is taken for granted by the programmers who use it *A non-procedural flow of control implicit in the construction of infrastructure components prior to the main program	*To guarantee the existence of only one singleton object *To complete the construction of the singleton instance prior to its first use	*To assess singleton instance	Added static instance method, static singleton instance, private constructor and possibly destructor	Languages that do not support modularity
21)Abstract Factory	Model "platform" (e.g. windowing system, operating system, database) with an inheritance hierarchy, and model each "product" (e.g. widgets, services, data structures) with its own hierarchy ... platform derived classes create and return instances of product derived classes	To construct objects by criterion and preconfigured type	*Creating objects by two criteria: Base type (Known to the application) and environment (preconfigured). *(E.g. creating GUI controls, given control type and knowing the GUI system being emulated)* *An array of factory methods with the same implementation criterion. *(Same example as above, where the need for each control type has aroused on a separate occasion)*	To encapsulate the decisions in a single object	*To obtain concrete product *To create product	Abstract factory responds to creation messages explicitly named after each conceptual product type. Each concrete factory implements this interface for a concrete environment. The products are arranged in respective discrete type families where environment implementations derive from product	General
22)Template Method	Define the "outline" of an algorithm in a base class ... common implementation is staged in the base class, peculiar implementation is represented by "place holders" in the base class and then implemented in derived classes	To outline and guarantee the execution of a generic process	"inheritance of process": A type family shares a sequential process with one or more stages being type-specific	To avoid repetition of the entire process in all implementations	*To execute a generic process	Processor features the template method (which should not be virtual) as well as one or more "primitives" - virtual (possibly abstract) functions, typically private. Concrete Processor implements the primitives	General
23)Flyweight	When dozens of instances of a class are desired and performance boggs down, externalize object state that is peculiar for each instance, and require the client to pass that state when methods are invoked	To prevent redundant creation of global resources	Objects are heavyweight or encapsulate system-critical resources and are read only	To prevent resource duplication, system wide	*To prevent redundant creation of global resources	Flyweight contains Smart Pointer Counters by key, creates Resources (which it keeps via the Smart Pointer Counters) as well as Smart Pointers (which it does not keep) that share a Smart Pointer Counter. Smart Pointer Counter contains a Resource and notifies the Flyweight. Client uses Smart Resource Pointers, obtained from the Flyweight	General

Appendix 7/7 - Resources

- GoF book - Design Patterns Elements of Reusable Object-Oriented Software (1995)
- Towards more accurate automatic recommendation of software design patterns Paper by Abeer Hamdy , Mohamed El Sayed (2018)
- Automatic Recommendation of Software Design Patterns: Text Retrieval Approach by Abeer Hamdy , Mohamed El Sayed (2018)
- Difference between design patterns - Scribd
- Design Pattern Quick Guide - Tutorialspoint
- “Design Patterns.” Refactoring.Guru, https://refactoring.guru/design-patterns. (2019)
- “Design Patterns and Refactoring.”, SourceMaking, https://sourcemaking.com/design_patterns. (2019)
- From user requirements to UML class diagram Hatem Herchi, Wahiba Ben Abdess (2012)
- Recommendation System for Design Patterns in Software Development: An DPR Overview Francis Palma, Hadi Farzin, Yann-Ga (2012)
- “A GQM-based Approach for Software Process Patterns Recommendation.” Meng, Zhangyuan, SEKE (2017)
- A Design Pattern Dictionary Version 2 of 8-Mar-04 by Avner Ben (2004)
- Automatic Transformation of User Stories into UML Use Case Diagrams using NLP Techniques Meryem Elallaouia, Khalid Nafilb (2018)
- Dynamically recommending design patterns S. Smith, D. R. Plante (2011)
- Software design patterns classification and selection using text categorization approach Shahid Hussain, Jacky Keung, Arif Ali Khan (2017)
- Automatic Transformation of User Stories into UML Use Case Diagrams using NLP Techniques Meryem Elallaouia, Khalid Nafilb (2012)
- A Survey on Design Pattern Detection Approaches Mohammed Ghazi Al-Obeidallah, Miltos Petridis, Stelios Kapetanakis (2016)
- "Design Pattern Detection using Machine Learning Techniques," A. Chaturvedi, A. Tiwari and S. Agarwal, 7th International Conference (2018)
- DESIGN PATTERN RECOGNITION Francesca Arcelli, Claudia Raibulet, Francesco Tisato (2004)
- Software design pattern mining using classification-based techniques by Ashish Kumar DWIVEDI , Anand TIRKEY, Santanu Kumar RATH (2018)
- Dwivedi, Ashish Kumar et al. “Software design pattern recognition using machine learning techniques.” IEEE Region 10 Conference (2016)
- Towards Machine Learning Based Design Pattern Recognition Sultan Alhusain and Simon Coupland, Maria Kavanagh, Robert John (2013)
- Design Pattern Support System: Help Making Decision in the Choice of Appropriate Pattern (2012)
- Methodological guideline to find suitable design patterns to implement adaptability Ime Pijnenborg (2016)