UDC 004.05

O. Linevych, O. Lisovychenko

SOFTWARE EVOLUTION FROM SYSTEM PERSPECTIVE

Abstract: Currently, software evolution takes an unpredictable amount of time. The purpose of the article is to describe the study of 7 popular and 1 new development method and their impact on the rate of evolution. The new method is the most effective, as it collects the most information on the connections between data and processes.

Keywords: software, evolution, systems theory, development, task.

Introduction

One of the longest and most high-cost phases of the software lifespan is its evolution. It takes 85-90% of the software budget [1]. There are dozens of development methods to speed up evolution and to lower the cost. The methods helped to some degree by speeding up use case translation to a code but according to CHAOS company reports, for the last 10 years the IT market has stagnated, around 20% of the software fails each year and 50% is challenged because it’s still hard to evolve [2, 3].

Problem Formulation

To analyze the efficiency of the widespread development methods which speed up evolution, to deduce the causes of the slow system evolution, and to compare the widespread method efficiency to the New Universal method.

Problem Solutions

Amongst the widespread methods that were analyzed are categorization[4], DDD [5], TDD [6], BDD [7], API Design-First[8], and patterns.

These methods are usually used to extract different types of domain information from a use case.

Were implemented 35-40 use cases per system.

The extracted information is used to evolve a code system.

Evolution time depends on the amount and quality of the extracted information. The methods extracted such information about the domain.

Thus, were conducted experiments for 25 software systems (during the evolution phase) from 10 domains: banking, telecom, e-commerce, healthcare, prediction, recommendation, design, government land, real estate, and accounting.

The experiment course was:

- to document a planned time to implement a use case
- to extract information from a use case by a development method

- to document the extracted information
- to evolve a system using the information
- to document actual evolution time.

Table 1. **Domain Information Categories**

Name	**Categories**
Data	1. data specifications 2. data abstractions 3. data connections inside the data (data system) 4. data connections with the outside world (system context)
Process	1. process specifications 2. process abstractions 3. process connections inside the process (process system) 4. process connections outside the process(system context)

Abstraction Method

There are 3 categorisation subtypes:

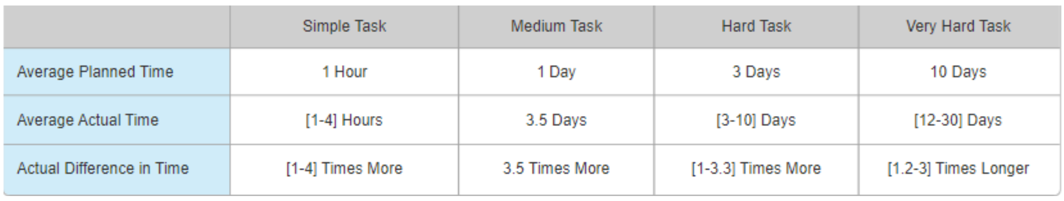
1. categorization
2. prototyping;
3. concept clustering.

The method is naturally used by the analysts and developers. The sense is to name, and find common properties of the data and processes, then translate them to the code.

These methods are used in combination with the next 5 methods.

When use cases were implemented using the only Abstraction method, then the software evolution speed had such indicators:

Table 2. **Abstraction Method**



	Simple Task	Medium Task	Hard Task	Very Hard Task
Average Planned Time	1 Hour	1 Day	3 Days	10 Days
Average Actual Time	[1-4] Hours	3.5 Days	[3-10] Days	[12-30] Days
Actual Difference in Time	[1-4] Times More	3.5 Times More	[1-3.3] Times More	[1.2-3] Times Longer

DDD Method

When use cases were implemented using the DDD method, the software evolution speed had such indicators:

The method helped to extract data specification, abstraction, and inside connection information but didn’t extract data outside and process inside/outside connections.

Table 3. **DDD Method**

	Simple Task	Medium Task	Hard Task	Very Hard Task
Average Planned Time	1 Hour	1 Day	3 Days	10 Days
Average Actual Time	[1-4] Hours	2 Days	[3-9] Days	[12-30] Days
Actual Difference in Time	[1-4] Times More	2 Times More	[1-3] Times More	[1.2-3] Times Longer

TDD+BDD, API Design-First Methods

TDD is usually reinforced with BDD and 90% of the systems were used together. An exception was a start-up sports e-commerce system which used use cases and tests.

When use cases were implemented using the TDD and BDD methods, the software evolution speed had such indicators(API Design-First provided almost identical results because has a similar idea to identify use case aims):

Table 4. **TDD+BDD or API Design-First Methods**

	Simple Task	Medium Task	Hard Task	Very Hard Task
Average Planned Time	1 Hour	1 Day	3 Days	10 Days
Average Actual Time	[1-3] Hours	3.5 Days	[3-9] Days	[12-40) Days
Actual Difference in Time	[1-3] Times More	3.5 Times More	[1-3] Times More	[1.2-4] Times Longer

Pattern Method

This method helped to extract all data/process specifications, abstractions, and data/process inside connections if a pattern could be used. The pattern usage is limited to specific cases. There are 3 widespread pattern types: design[8], enterprise[9], and microservice[10]. The time to implement use cases using the pattern:

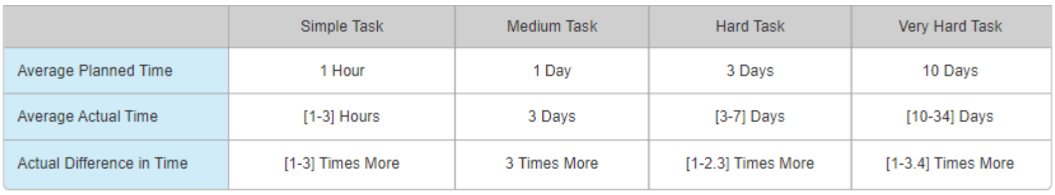
Table 5. **Pattern Method**

Pattern Method				
	Simple Task	Medium Task	Hard Task	Very Hard Task
Average Planned Time	1 Hour	1 Day	3 Days	10 Days
Average Actual Time	[1-3] Hours	3 Days	[3-8] Days	[12-35] Days
Actual Difference in Time	[1-3] Times More	3 Times More	[1-2.67] Times More	[1.2-4] Times More

Hybrid Method

The hybrid method united DDD, TDD, BDD, and pattern methods.

Table 6. **Hybrid Method**

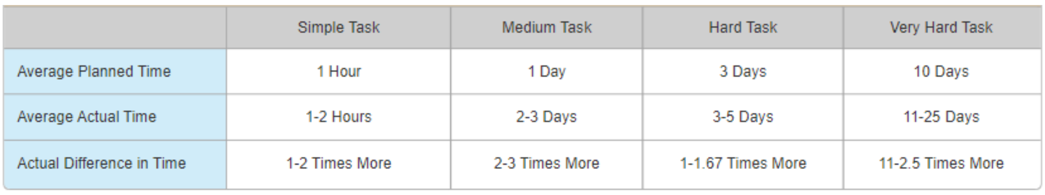


	Simple Task	Medium Task	Hard Task	Very Hard Task
Average Planned Time	1 Hour	1 Day	3 Days	10 Days
Average Actual Time	[1-3] Hours	3 Days	[3-7] Days	[10-34] Days
Actual Difference in Time	[1-3] Times More	3 Times More	[1-2.3] Times More	[1-3.4] Times More

RUP Method

When use cases were implemented using the only RUP method, then the software evolution speed had such indicators:

Table 7. **RUP Method**



	Simple Task	Medium Task	Hard Task	Very Hard Task
Average Planned Time	1 Hour	1 Day	3 Days	10 Days
Average Actual Time	1-2 Hours	2-3 Days	3-5 Days	11-25 Days
Actual Difference in Time	1-2 Times More	2-3 Times More	1-1.67 Times More	11-2.5 Times More

New Universal Method

This method is built on general system theory and evolutional software lifecycle described by R. Riordan[12], RDD, DDD, modified TDD, and categorization methods.

The method can be used for any domain which can be described using the general system theory and, therefore is named as universal. The method is distinguished from other methods because of the system's perspective on a use case as a system of interrelated data and process subsystems.

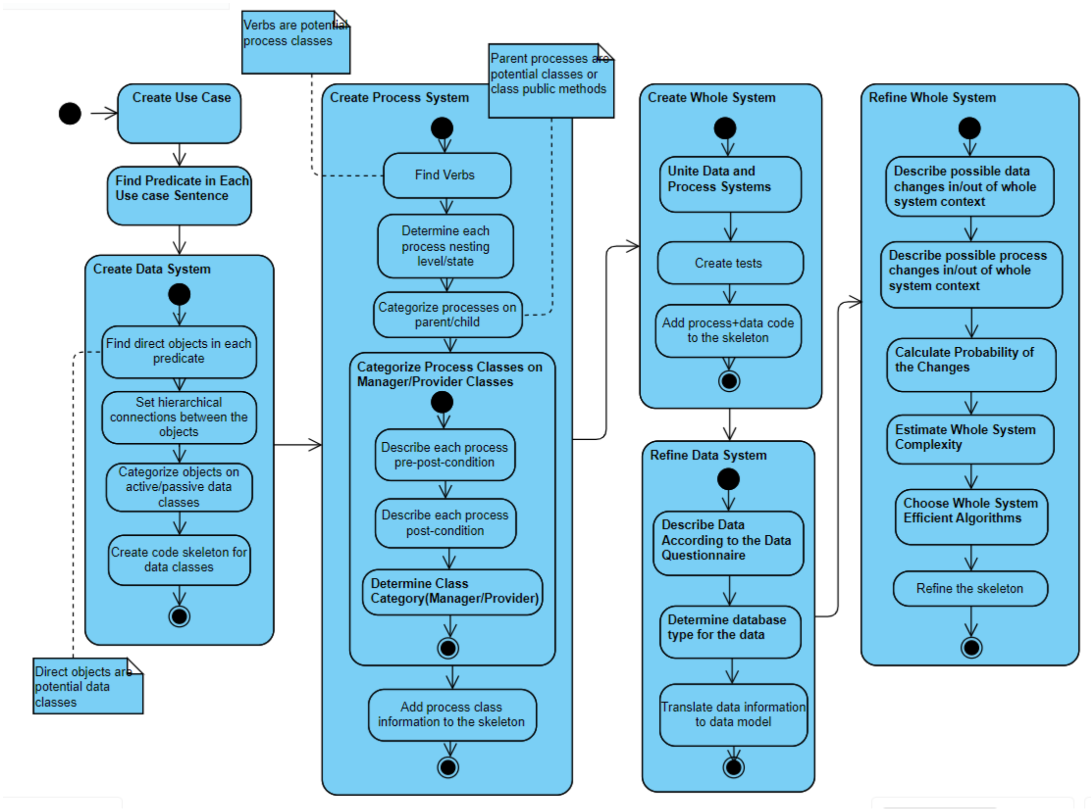
The sense is to translate a use case by sentence analysis and finding of data/process specifications, then data/process analysis of inside system structure, and then outside.

When use cases were implemented using the New Universal method, then the software evolution speed had such indicators.

Solution Analysis

The Abstraction method is one of the least stable: with time the ability to evolve the system degraded steeply because it wasn’t possible to predict how much time would be taken for a new use case implementation. Every use case was implemented with a different set of data/process abstractions/specifications depending on the developer's perspective and later (very)heavy tasks took more time(from instead 10 it took 30) to introduce changes.

The DDD method helped to extract the same data and put it in the unified domain (terminology)form which was implemented by the developers. It sped up the next evolution cases instead of an average of 3.5 days it took 2 days for the medium tasks and hard tasks implementation took more often 5 days.



Verbs are potential process classes
Parent processes are potential classes or class public methods
Create Use Case
Find Predicate in Each Use case Sentence
Create Data System
Find direct objects in each predicate
Set hierarchical connections between the objects
Categorize objects on active/passive data classes
Create code skeleton for data classes
Direct objects are potential data classes
Create Process System
Find Verbs
Determine each process nesting level/state
Categorize processes on parent/child
Categorize Process Classes on Manager/Provider Classes
Describe each process pre-post-condition
Describe each process post-condition
Determine Class Category(Manager/Provider)
Add process class information to the skeleton
Create Whole System
Unite Data and Process Systems
Create tests
Add process+data code to the skeleton
Refine Data System
Describe Data According to the Data Questionnaire
Determine database type for the data
Translate data information to data model
Refine Whole System
Describe possible data changes in/out of whole system context
Describe possible process changes in/out of whole system context
Calculate Probability of the Changes
Estimate Whole System Complexity
Choose Whole System Efficient Algorithms
Refine the skeleton

Fig. 1. Activity Diagram of New Universal Method

***Table 8.* New Universal Method**

	Simple Task	Medium Task	Hard Task	Very Hard Task
Average Planned Time	1 Hour	1 Day	3 Days	10 Days
Average Actual Time	[1-2] Hours	4 Days	[3-4] Days	[10-20] Days
Actual Difference in Time	[1-2] Times More	4 Times More	[1-1.3] Times More	[1-2] Times More

However, DDD didn’t extract the data and process outside connections. This lack of information made the system non-integral with the existing system and complicated its structure. The structure complication leads to the fast degradation of the future evolution possibilities.

The TDD+BDD methods extracted the same data and process information as the Abstraction method and didn’t unify it according to the domain, but the evolution was fast at the beginning due to BDD use case descriptions and TDD aim orientation.

However, software evolution degraded the same because lacked the same data. Thus, heavy tasks were made slower with each new change and could be done in unplanned 40 days, not 10.

The Pattern method helped to extract more information, than DDD, TDD+BDD but it was limited by the number of limited situations where the patterns are efficient. The pattern method is one of the fastest methods to extract info and provides the most predictable system evolution because of the detailed documentation of previous experience.

The hybrid methods provided faster evolution and helped to extract more data, than previous methods, though without the existing system connection rules as other methods. The hybrid method helped to find data and process information in such a number of cases:

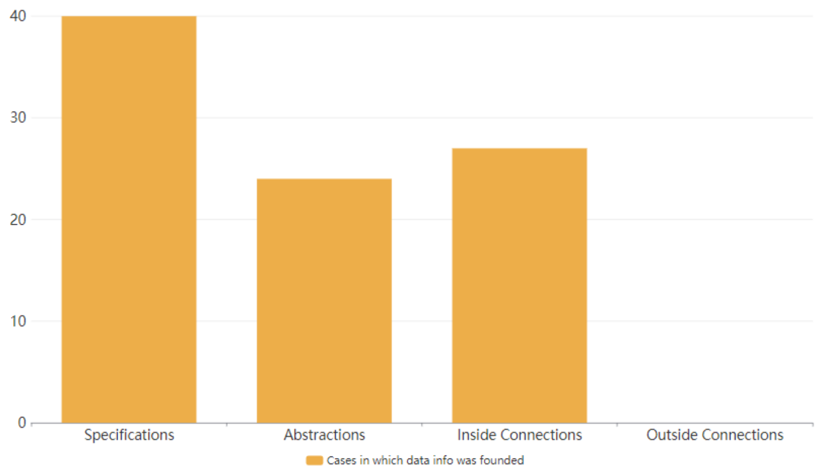


Fig. 2. Hybrid Method Data Info

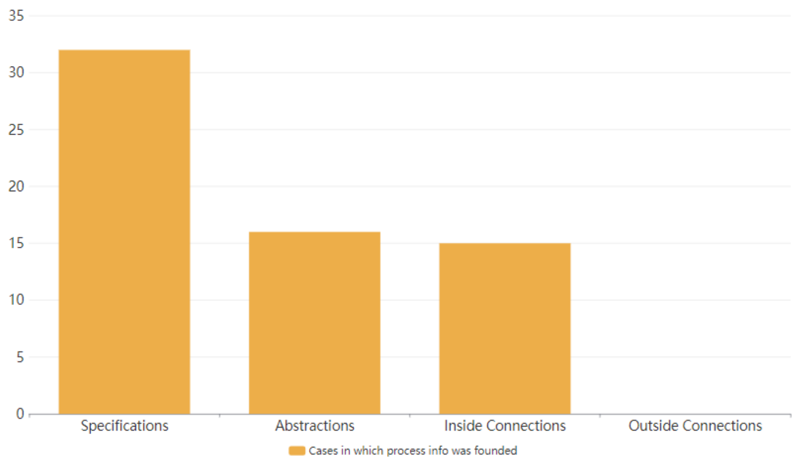


Fig. 3. Hybrid Method Extracted Process Info

The Hybrid Method provided almost identical results compared to the RUP method but the RUP method took more time to develop tasks at the beginning but degraded slower to an extent when very heavy tasks maximum took 25 days instead of 34.

The main problem is that even if combine all 5 widespread methods into the Hybrid method, it doesn’t help to extract data and process outside connections i.e. it’s not clear how to organize use case data and process abstractions that they could be built into the existing systems and predictably evolved in the future. The same if with RUP and API First methods.

To build a predictable system there’s a need to know system limitations of evolutions and rules. The New Universal method fulfils these requirements for medium tasks by minimizing task implementation to 9 hours and for very hard tasks that were implemented with a maximum of 18 days which is faster, than the Hybrid method which takes 22 days. Also, the method helps to minimize simple and hard task implementation to the planned time.

The New Universal method can find data/process connections because of general system theory [13] that clarifies domain structure including outside connections and the rules according to which social, biological, chemical, and other systems are automated by the software.

Conclusions

There’s a need for a method like the New Universal method which helps to extract and organize domain information considering data and process abstractions, data and process inside/outside connections.

Such a method would consider existing system rules and future domain challenges. Such a system with naturally built-in use cases will always be in a stable intermediate form and, thus can evolve naturally.

The natural evolution of the software is predictable because it is based on the fixed set of rules described in the general system theory.

REFERENCES

1. *Ogheneovo E.E.* On the Relationship between Software Complexity and Maintenance Costs / Journal of Computer and Communications 02(14), 2014.
2. *Alves L.M.* Longevity of risks in software development projects: a comparative analysis with an academic environment. 2021.
3. *Johnson J.* CHAOS Report Project Outcome Results 2020. URL: https://www.standishgroup.com/store/
4. *Booch G.* Object-Oriented Analysis and Design with Applications. Addison-Wesley Professional. 2007.
5. *Evans E.*. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional. 2003.
6. *Beck K.* Test Driven Development: By Example. Addison-Wesley Professional 2002.
7. *Ferguson J.* BDD in Action: Behavior-driven development for the whole software lifecycle. Manning Publications. 2014.
8. *Higginbotham J.* Principles of Web API Designing. Pearson Addison-Weasley. 2021.
9. *Gamma E.* Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional. 1994.

10. *Fowler M.* Enterprise Patterns. 2002.

11. *Richardson C.* Microservices Patterns: With examples in Java. Manning. 2018

12. *Riordan M.R.* Designing effective database systems. Addison-Wesley Microsoft Technology). 2005.

13. *Ludwig Von Bertalanffy.* General System Theory: Foundations, Development, Applications. George Braziller Inc. 1969.