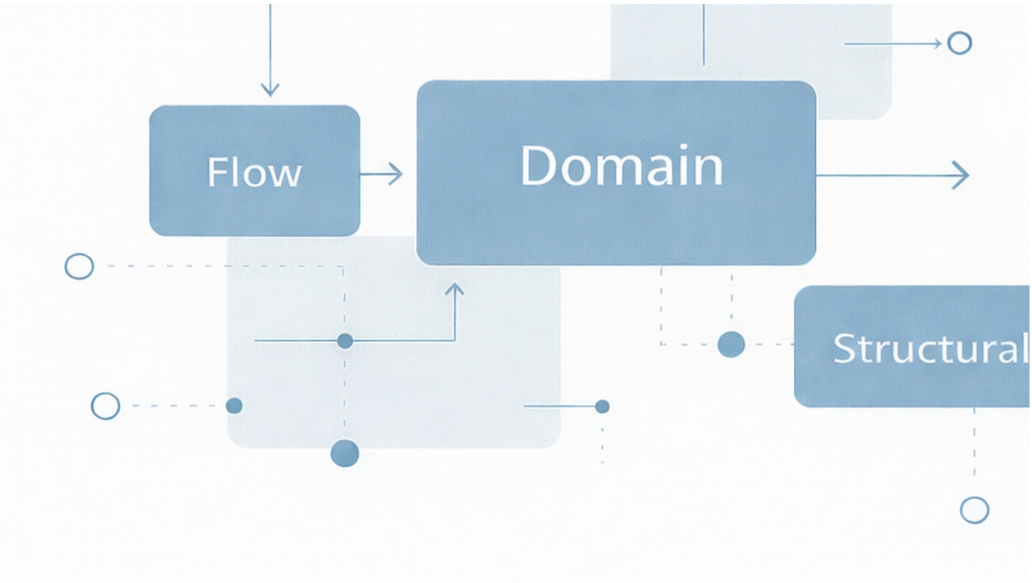
ΑΡΧΙΤΕΚΤΟΝΙΚΩΝ MODEL & UML DIAGRAM



Συγγραφέας: Κώστας Μεσσήνης

Έτος 2026

Έκδοση: Ελληνικά

ΑΡΧΙΤΕΚΤΟΝΙΚΩΝ MODEL & UML DIAGRAM

Συγγραφέας Κώστας Μεσσήνης

Έτος 2026

Identifier (UUID):
urn:uuid:b99aae83-7d61-47ed-8ef4-567feaba9a74
Edition: EL v1.0
Build: 2026-02-27
Πρώτη έκδοση: 2026
My web: https://github.com/messinisk/software_architecture_MODEL_UML

ΠΡΟΛΟΓΟΣ

Το παρόν έργο αποτελεί συστηματική μελέτη και σύνθεση διαφορετικών προσεγγίσεων αρχιτεκτονικής λογισμικού, με σκοπό την παροχή ενός ενιαίου, ολοκληρωμένου και πρακτικά εφαρμόσιμου πλαισίου τόσο για μελέτη όσο και για πραγματική ανάπτυξη συστημάτων. Σε μια εποχή όπου η ποικιλία των τεχνολογιών και των προτύπων καθιστά συχνά δύσκολη τη σύγκριση και την επιλογή αρχιτεκτονικής, η ανάγκη για καθαρή ταξινόμηση, ενιαίο μοντέλο κατανόησης και αντικειμενικά κριτήρια επιλογής είναι περισσότερο επιτακτική από ποτέ.

Η προσέγγιση που ακολουθήθηκε βασίστηκε σε δύο άξονες:

1. Στην ενοποίηση των κυριότερων αρχιτεκτονικών στυλ σε τέσσερις θεμελιώδεις οικογένειες (Structural, Flow-Based, State-Oriented, Domain-Centric).
2. Στην αξιοποίηση UML ως ενιαίας “γλώσσας” μοντελοποίησης, η οποία επιτρέπει καθαρή απεικόνιση της δομής, της συμπεριφοράς και των μηχανισμών κάθε αρχιτεκτονικής.

Ο στόχος του κεφαλαίου δεν είναι μόνο περιγραφικός· επιδιώκει να λειτουργήσει ως πρακτικός οδηγός. Προσφέρει decision trees, πίνακες αξιολόγησης, δείκτες ωριμότητας και ρίσκου, καθώς και παραδείγματα UML για κάθε αρχιτεκτονική οικογένεια. Με αυτόν τον τρόπο, ο αναγνώστης εξοπλίζεται με μια πλήρη μεθοδολογία που μπορεί να εφαρμοστεί στην πράξη, ανεξάρτητα από το μέγεθος ή την πολυπλοκότητα του έργου λογισμικού.

Το παρόν κεφάλαιο μπορεί να αποτελέσει σημείο αναφοράς για φοιτητές, μηχανικούς λογισμικού, αρχιτέκτονες συστημάτων και επαγγελματίες που επιθυμούν μια καθαρή, συνεκτική και ρεαλιστική θεώρηση της αρχιτεκτονικής λογισμικού μέσα από το πρίσμα της UML και των σύγχρονων πρακτικών ανάπτυξης.

Πίνακας περιεχομένων

Ευρετήριο Εικόνες

Ευρετήριο Εικόνες

1.ΕΙΣΑΓΩΓΗ

Με τη συγκεντρωτική αυτή προσέγγιση, το κεφάλαιο παρέχει στον αναγνώστη μία πλήρη εικόνα των διαθέσιμων αρχιτεκτονικών επιλογών, καθώς και των κριτηρίων που καθορίζουν τη βέλτιστη εφαρμογή τους σε πραγματικά έργα λογισμικού.

1.1.Ρόλος της Αρχιτεκτονικής Λογισμικού

Η αρχιτεκτονική λογισμικού αποτελεί θεμελιώδη πτυχή του σχεδιασμού σύγχρονων συστημάτων και επηρεάζει άμεσα την ποιότητα, την επεκτασιμότητα και τη συντηρησιμότητα τους. Σε ένα περιβάλλον όπου οι απαιτήσεις των εφαρμογών αυξάνονται και διαφοροποιούνται συνεχώς, η σωστή επιλογή και τεκμηρίωση της αρχιτεκτονικής καθίσταται κρίσιμη για την επιτυχία ενός έργου. Το παρόν κεφάλαιο επιδιώκει να προσφέρει μια ολοκληρωμένη και συστηματική κατανόηση των σημαντικότερων αρχιτεκτονικών προσεγγίσεων, παρουσιάζοντας ένα ενιαίο πλαίσιο ανάλυσης, σύγκρισης και επιλογής.

1.2.Ενοποίηση Αρχιτεκτονικών Προτύπων

Αντί της παραδοσιακής κατακερματισμένης παρουσίασης των αρχιτεκτονικών στυλ, η παρούσα μελέτη ενοποιεί τα κυριότερα μοντέλα σε τέσσερις θεμελιώδεις οικογένειες: Structural, Flow-Based, State-Oriented και Domain-Centric (DDD). Η ταξινόμηση αυτή επιτρέπει την αποτύπωση της ουσίας κάθε αρχιτεκτονικής, ανεξάρτητα από συγκεκριμένες τεχνολογίες, frameworks ή patterns (όπως MVC, Clean Architecture, Event-Driven, Pipeline, CQRS ή DDD). Με αυτόν τον τρόπο, τα διαφορετικά στυλ παρατίθενται σε ένα κοινό εννοιολογικό πλαίσιο, διευκολύνοντας τη συγκριτική ανάλυση τόσο σε θεωρητικό όσο και σε πρακτικό επίπεδο.

1.3.UML ως Καθολικό Εργαλείο Μοντελοποίησης

Το κεφάλαιο παρουσιάζει επίσης τον τρόπο με τον οποίο η UML μοντελοποίηση λειτουργεί ως ενιαίο μέσο περιγραφής αρχιτεκτονικών δομών, ροών και συμπεριφορών. Μέσα από Class Diagrams, State Machines, Activity Diagrams, Sequence Diagrams και Context Maps, αποτυπώνονται οι βασικές διαφορές και τα σημεία σύγκλισης των αρχιτεκτονικών προσεγγίσεων. Η UML χρησιμεύει εδώ όχι μόνο ως τεκμηριωτικό εργαλείο, αλλά και ως μηχανισμός κατανόησης και επικοινωνίας μεταξύ αναλυτών, αρχιτεκτόνων και ομάδων ανάπτυξης.

1.4.To Unified Architecture Selection Framework

Τέλος, εισάγεται το Unified Architecture Selection Framework, το οποίο συνδυάζει decision matrices, risk–maturity indicators και decision trees για την αντικειμενική αξιολόγηση και επιλογή της καταλληλότερης αρχιτεκτονικής σε κάθε περίπτωση. Το πλαίσιο αυτό αποτελεί πρακτικό οδηγό που βοηθά στη λήψη τεκμηριωμένων αποφάσεων, λαμβάνοντας υπόψη τόσο την πολυπλοκότητα του domain όσο και τις λειτουργικές απαιτήσεις του συστήματος.

2.ΘΕΜΕΛΙΩΔΕΙΣ ΟΙΚΟΓΕΝΕΙΕΣ ΑΡΧΙΤΕΚΤΟΝΙΚΩΝ

2.1.Structural / Layered Architecture

Χαρακτηριστική για εφαρμογές με καθαρή διάκριση επιπέδων (UI, Application, Domain, Infrastructure).

Παραδείγματα:

- Κλασικές web εφαρμογές (Django, ASP.NET, MVC)
- Desktop εργαλεία (γραφείου, ERP-lite, CAD βοηθητικά modules)
- Mobile εφαρμογές βασισμένες σε MVC/MVVM

Η προσέγγιση αυτή απλοποιεί την ανάπτυξη και προσφέρει υψηλή συντηρησιμότητα για καθημερινά επιχειρησιακά έργα.

2.2.Flow-Based / Pipeline Architecture

Βέλτιστη όταν τα δεδομένα περνούν από διαδοχικά στάδια μετασχηματισμού ή επεξεργασίας. Παραδείγματα:

- ETL συστήματα και data engineering pipelines
- Machine Learning model pipelines (preprocessing → training → evaluation)
- Video & audio processing flows
- Αναλυτές κώδικα (compilers, static analyzers)

Η αρχιτεκτονική αυτή παρέχει υψηλή απόδοση και προβλεψιμότητα στις ροές.

2.3 State-Oriented / Event-Driven Architecture

Εφαρμόζεται σε συστήματα όπου το γεγονός (event) προκαλεί αλλαγή κατάστασης. Παραδείγματα:

- Παιχνίδια (game loop, FSM-based rules)
- IoT και συστήματα αισθητήρων
- Microservices με message brokers (Kafka, RabbitMQ)
- Reactive εφαρμογές πραγματικού χρόνου.

Είναι ιδανική για αλληλεπιδραστικά, ασύγχρονα ή δυναμικά περιβάλλοντα.

2.4 Domain-Centric Architecture (DDD)

Εστιάζει στην αποτύπωση της επιχειρησιακής λογικής και των εννοιών του domain. Παραδείγματα:

- Banking και χρηματοοικονομικά συστήματα
- E-commerce engines
- Ασφαλιστικά, λογιστικά και ERP συστήματα
- Σύνθετες εφαρμογές B2B με πολλαπλά subdomains Προσφέρει εξαιρετική ακρίβεια μοντελοποίησης και μακροπρόθεσμη βιωσιμότητα.

3.ΧΑΡΤΗΣ ΑΠΟΦΑΣΗΣ & ΚΡΙΤΗΡΙΑ ΕΠΙΛΟΓΗΣ ΑΡΧΙΤΕΚΤΟΝΙΚΗΣ

Η επιλογή αρχιτεκτονικής λογισμικού αποτελεί κρίσιμο στάδιο στη σχεδίαση συστημάτων και επηρεάζει άμεσα τη συντηρησιμότητα, την επεκτασιμότητα και τη συνολική διάρκεια ζωής του έργου. Παρά την πληθώρα υπαρχουσών προσεγγίσεων (Layered, Clean Architecture, MVC, Pipeline, Event-Driven, DDD κ.ά.), όλες μπορούν να ταξινομηθούν σε τέσσερις βασικές οικογένειες αρχιτεκτονικών προτύπων: Structural (Layered), Flow-Based (Pipeline), State-Oriented (Event-Driven), και Domain-Centric (DDD). Κάθε οικογένεια διαθέτει συγκεκριμένα πλεονεκτήματα και περιορισμούς, οι οποίοι καθορίζουν την καταλληλότητά της για διαφορετικούς τύπους εφαρμογών.

3.1.Αναλυτική Παραμετροποίηση Κριτηρίων

-Πολυπλοκότητα του Domain

Το κυριότερο κριτήριο αποτελεί ο βαθμός πολυπλοκότητας του επιχειρησιακού ή λογικού domain.

Για ***υψηλής πολυπλοκότητας*** συστήματα, όπου οι κανόνες, οι μεταβάσεις καταστάσεων και οι έννοιες του domain είναι κρίσιμες, προκρίνεται το ***Domain-Centric (DDD)***, το οποίο προσφέρει εννοιολογική οργάνωση και ισχυρά μοντέλα (Aggregates, Entities, Value Objects).

Για ***συστήματα με σχεδόν στατικά δεδομένα*** ή απλό CRUD, αρκεί η ***Structural/Layered*** προσέγγιση.

-είδος ροής εκτέλεσης

Συστήματα που βασίζονται σε ροές επεξεργασίας ή μετασχηματισμούς δεδομένων ευνοούν αρχιτεκτονικές τύπου ***Flow-Based/Pipeline***, όπου κάθε στάδιο αποτελεί διακριτή λειτουργική μονάδα. Αντίθετα, εφαρμογές που αντιδρούν σε εξωτερικά γεγονότα, σήματα ή μηνύματα απαιτούν ***State-Oriented/Event-Driven*** αρχιτεκτονική με σαφείς μηχανισμούς χειρισμού συμβάντων.

-Μέγεθος και Ωριμότητα της Ομάδας

Αρχιτεκτονικές με ισχυρή τυποποίηση (π.χ. Clean Architecture ή DDD) απαιτούν μεγαλύτερη εμπειρία και οργανωμένη ομάδα. Για μικρές ομάδες ή έργα με αυστηρό χρόνο παράδοσης προτιμώνται μοντέλα μικρότερης πολυπλοκότητας, όπως MVC ή απλή Layered προσέγγιση.

-Απαιτήσεις Κλιμάκωσης και Απόδοσης

Συστήματα με υψηλό φορτίο, ασύγχρονη επεξεργασία ή ανάγκη οριζόντιας κλιμάκωσης τείνουν να επιλέγουν ***Event-Driven*** μοντέλα, τα οποία διαχειρίζονται φυσικά υψηλό concurrency. Αντίθετα, εφαρμογές γραφείου ή desktop λογισμικό δεν ωφελούνται από τέτοια πολυπλοκότητα.

-Testability και Συντηρησιμότητα

Αρχιτεκτονικές με σαφή διαχωρισμό ευθυνών (Layered, Clean, DDD) επιτρέπουν υψηλότερη testability. Pipeline και Event-Driven μοντέλα έχουν αυξημένη πολυπλοκότητα κατά τη δοκιμή, αλλά καλύτερη συμπεριφορά σε μεγάλης κλίμακας συστήματα.

-Κόστος Υλοποίησης έναντι Μακροπρόθεσμου Οφέλους

Οι πιο προηγμένες αρχιτεκτονικές προσεγγίσεις έχουν μεγαλύτερο αρχικό κόστος. Ωστόσο, για μακροπρόθεσμα έργα, η επένδυση σε Domain-Centric ή Event-Driven αρχιτεκτονική μειώνει δραματικά το τελικό κόστος συντήρησης και αλλαγών. Για βραχυπρόθεσμα έργα, απλές Layered ή MVC αρχιτεκτονικές είναι επαρκείς και αποδοτικές.

3.2.DECISION MATRIX

Αρχιτεκτονικό Κριτήριο → Προτεινόμενο Αρχιτεκτονικό Στυλ

Βαθμολογία: ✓ = Ιδανική επιλογή, ● = Καλή επιλογή, – = Όχι κατάλληλη επιλογή

Πίνακας 1: Decision Matrix

Κριτήριο / Συνθήκη	Structural	Flow-Based	State-Oriented	Domain-Centric (DDD)
Απλό Domain / CRUD	✓✓✓	–	–	–
Μεσαίας πολυπλοκότητας επιχειρησιακοί κανόνες	✓✓	–	●	●
Σύνθετο ή κρίσιμο Domain (π.χ. οικονομικά, ERP)	●	–	●	✓✓✓
Συστήματα που βασίζονται σε κατάσταση (FSM, loops)	–	–	✓✓✓	●
Ροές επεξεργασίας (ETL, ML pipelines)	–	✓✓✓	–	–
Εφαρμογές πραγματικού χρόνου και γεγονότων	–	–	✓✓✓	●
Γρήγορη ανάπτυξη / μικρή ομάδα	✓✓✓	✓	–	–
Υψηλή κλιμάκωση (cloud, distributed)	✓✓	●	✓✓✓	✓✓
Ευκολία testing	✓✓	✓	●	✓✓✓
Μακροπρόθεσμη συντηρησιμότητα	✓✓	●	✓✓	✓✓✓
Απόδοση / throughput	✓	✓✓✓	✓✓	●

Ξεκινάμε από τον Πίνακα 1: τον Decision Matrix.
Εδώ, κάθε γραμμή είναι μια συνθήκη, και κάθε στήλη μια οικογένεια αρχιτεκτονικής.

- Structural → κυρίαρχη επιλογή για τις περισσότερες εφαρμογές.
- Flow-Based → η κορυφαία λύση για επεξεργασία δεδομένων.
- State-Oriented → μοναδικό για παιχνίδια, reactive, event-driven.
- DDD → αξεπέραστο σε σύνθετα domains.
  Τα σύμβολα δείχνουν "πόσο ταιριάζει" η κάθε οικογένεια.
- Σε απλό domain ή CRUD εφαρμογές, όπως admin panels και βασικές φόρμες, η Structural προσέγγιση υπερέχει ξεκάθαρα. Είναι η πιο γρήγορη, η πιο ευθεία, και η πιο προβλέψιμη.
- Όταν ανεβαίνουμε σε μεσαίας πολυπλοκότητας επιχειρησιακούς κανόνες, η Structural παραμένει καλή, αλλά αρχίζουν να φαίνονται και οι εναλλακτικές:
  η State-Oriented και η Domain-Centric μπορούν να υποστηρίξουν πιο απαιτητική λογική, ειδικά όταν υπάρχουν γεγονότα ή ισχυροί κανόνες.
- Στο σύνθετο ή κρίσιμο domain, όπως οικονομικά συστήματα ή ERP, εμφανίζεται καθαρά ο ρόλος του DDD:
  εκεί το domain δεν είναι "λίγη λογική", είναι το ίδιο το προϊόν.
  Η State-Oriented μπορεί επίσης να σταθεί σε τέτοια περιβάλλοντα, ειδικά αν η συμπεριφορά ορίζεται από καταστάσεις και συμβάντα.
- Για συστήματα που είναι καθαρά κατάσταση, όπως FSM, loops, ή event-driven state transitions, η State-Oriented προσέγγιση είναι η φυσική επιλογή.
- Αντίστοιχα, όταν μιλάμε για ροές επεξεργασίας, όπως ETL ή ML pipelines, τότε το Flow-Based κυριαρχεί:
  σπάει την εργασία σε στάδια και μετασχηματισμούς, με καθαρή σειρά εκτέλεσης.
- Σε real-time και event-based εφαρμογές, ξανά βλέπουμε την υπεροχή του State-Oriented.
  Η Structural εδώ δεν είναι "λάθος", απλώς είναι συνήθως ανεπαρκής για τη δυναμική φύση των γεγονότων.
- Στο κομμάτι της γρήγορης ανάπτυξης και μικρής ομάδας, η Structural είναι το πιο φιλικό περιβάλλον.
  Το Flow-Based βοηθά σε συγκεκριμένες περιπτώσεις, αλλά δεν είναι η πρώτη επιλογή για γενικές εφαρμογές.
- Για υψηλή κλιμάκωση, cloud και distributed, η State-Oriented και το DDD προσφέρουν πιο στιβαρά μοντέλα.
  Η Structural μπορεί να κλιμακώσει, αλλά συχνά "σπάει" σε συντηρησιμότητα όταν μεγαλώσει πολύ το domain.
- Στην ευκολία testing, το DDD εμφανίζεται πολύ ισχυρό, επειδή οι κανόνες συγκεντρώνονται σε καθαρές δομές και invariants.
  Η Structural επίσης τεστάρεται καλά, ενώ State-Oriented θέλει προσοχή λόγω πολλών μεταβάσεων.
- Τέλος, στην απόδοση, το Flow-Based είναι πολύ δυνατό σε throughput, ειδικά σε pipelines, ενώ State-Oriented επίσης αποδίδει καλά σε async flows.

3.3 DECISION TREE (PlantUML)

«Στην Εικόνα 1 βλέπουμε ένα δέντρο αποφάσεων για την επιλογή αρχιτεκτονικής.
Η λογική είναι απλή: κάνουμε διαδοχικές ερωτήσεις, και ανάλογα με την απάντηση καταλήγουμε στην πιο κατάλληλη οικογένεια.

Πρώτη ερώτηση: “Το domain είναι απλό ή πρόκειται για CRUD;”
Αν η απάντηση είναι ***ναι***, τότε επιλέγουμε ***`Structural Architecture`***. Δηλαδή μια κλασική δομική προσέγγιση, με layers, controllers, services και repositories. Και η διαδικασία σταματά εδώ.

Αν όμως η απάντηση είναι ***όχι***, τότε περνάμε στη δεύτερη ερώτηση:
“Υπάρχει flow-based επεξεργασία δεδομένων;”
Δηλαδή το σύστημα λειτουργεί σαν ροή σταδίων—φόρτωση, καθαρισμός, μετασχηματισμοί, υπολογισμοί;
Αν η απάντηση είναι ***ναι***, τότε οδηγούμαστε σε ***`Flow-Based / Pipeline`***. Δηλαδή σχεδιασμό σε στάδια, με καθαρή σειρά εκτέλεσης.

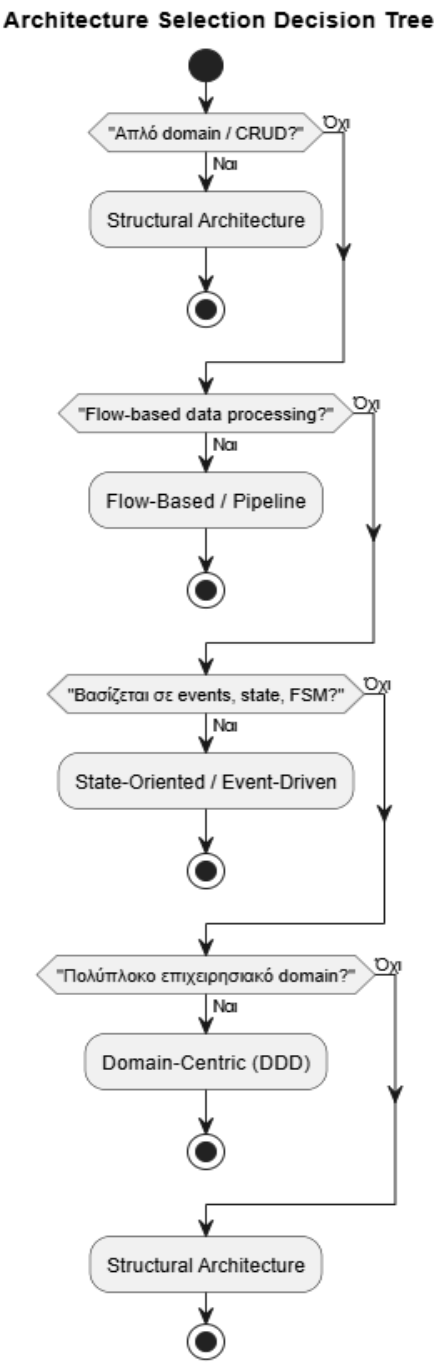
Αν και εδώ η απάντηση είναι ***όχι***, πάμε στην τρίτη ερώτηση:
“Το σύστημα βασίζεται σε events, κατάσταση, ή FSM;”
Με άλλα λόγια, το σύστημα αλλάζει συμπεριφορά ανάλογα με την κατάσταση, και κινείται μέσω γεγονότων και μεταβάσεων.
Αν η απάντηση είναι ***ναι***, τότε επιλέγουμε ***`State-Oriented / Event-Driven`***.

Αν ούτε αυτό ισχύει, προχωράμε στην τέταρτη ερώτηση: “Το επιχειρησιακό domain είναι πολύπλοκο;”
Αν η απάντηση είναι ***ναι***, τότε η επιλογή είναι ***`Domain-Centric, δηλαδή DDD`***: εστίαση σε aggregates, entities, value objects και κανόνες του domain.

Και τέλος, αν ακόμα κι εδώ η απάντηση είναι ***όχι***, το δέντρο επιστρέφει σε μια ασφαλή προεπιλογή: ***`Structural Architecture`***.

Με μία φράση: το δέντρο αυτό λέει—διάλεξε αρχιτεκτονική με βάση το τι είναι το σύστημα στην ουσία του:
δομή, ροή, κατάσταση, ή σύνθετο domain.»

Εικόνα 1: Γράφημα decision tree



3.4 Συμπέρασμα

Η επιλογή αρχιτεκτονικής δεν είναι ζήτημα προσκόλλησης σε ένα συγκεκριμένο πρότυπο, αλλά αποτέλεσμα συστηματικής αξιολόγησης των ιδιαιτεροτήτων του συστήματος, του βαθμού πολυπλοκότητας του domain, των απαιτήσεων κλιμάκωσης, των χρονικών περιορισμών και της εμπειρίας της ομάδας. Οι τέσσερις θεμελιώδεις οικογένειες αρχιτεκτονικών παρέχουν ένα σαφές πλαίσιο λήψης απόφασης και επιτρέπουν στον μηχανικό λογισμικού να επιλέξει την πιο κατάλληλη προσέγγιση για ένα βιώσιμο, επεκτάσιμο και κατανοητό σύστημα.

4.Συγκριτική Ανάλυση UML Templates

Οι τέσσερις οικογένειες αρχιτεκτονικών προτύπων — Structural, Flow-Based, State-Oriented και Domain-Centric — διαφοροποιούνται όχι μόνο ως προς το μοντέλο εφαρμογής, αλλά και ως προς τον τρόπο που απεικονίζονται μέσω UML. Η σύγκριση πρέπει να εξετάζει τρεις διαστάσεις: ***δομική μοντελοποίηση***, ***διαγραμματική αναπαράσταση***, και ***συμπεριφορική απεικόνιση***.

4.1 Δομική Προσέγγιση (Structural UML)

Στο επίπεδο δομής, το UML περιγράφει πώς οργανώνονται τα συστατικά του συστήματος:

- Structural Architecture → Κλασικό Class Diagram με πακέτα (UI, Application, Domain, Infrastructure).
- Flow-Based Architecture → Χρήση Component Diagrams ή Activity Diagrams με διαδοχικά βήματα.
- State-Oriented Architecture → Έμφαση σε State Machine Diagrams, Event Models και Transition Tables.
- Domain-Centric (DDD) → Πλούσιο Class Diagram με Aggregates, Entities, Value Objects και Domain Services.

4.2 Διαγραμματική Προσέγγιση (Diagram-Level View)

Οι κατηγορίες διαφέρουν ως προς την κύρια μορφή UML που τις εκφράζει:

• Structural → Class & Package Diagrams

• Flow-Based → Activity, Sequence & Data Flow Diagrams

• State-Oriented → State Machine, Sequence, Event Diagrams

• Domain-Centric → Class, Context Map, Interaction Diagrams

4.3 Συμπεριφορική Προσέγγιση (Behavioral UML)

Η συμπεριφορά του συστήματος αποτυπώνεται διαφορετικά:

• Structural → Sequence Diagrams για Use Cases

• Flow-Based → Activity Diagrams με pipeline ροή

• State-Oriented → State Diagrams και Event-driven lifecycles

• Domain-Centric → Event Storming-like Sequence Diagrams & Aggregate Invariants

4.4 Συγκεντρωτικός Πίνακας UML Χρήσης

Οικογένεια	***Δομή UML***	***Κύρια Διαγράμματα***	***Συμπεριφορά***
Structural	*Layers, components*	*Class, Package*	*Use-case flows*
Flow-Based	*Stages, transforms*	*Activity, Pipeline*	*Sequential processing*
State-Oriented	*States, events, handlers*	*FSM, Sequence*	*Event-driven transitions*
Domain-Centric	*Aggregates & Entities*	*Class, Context Map*	*Domain rules & invariants*

Πίνακας 2: Συγκεντρωτικός Πίνακας UML

Ο Πίνακας 2 είναι σαν “λεξικό αντιστοίχισης” μεταξύ οικογένειας και UML.
Η Structural οικογένεια ευνοεί Layers και Components.
Τα βασικά διαγράμματα είναι Class και Package, και η συμπεριφορά συνήθως παρουσιάζεται ως use-case flows.
Η Flow-Based οικογένεια είναι “στάδια και μετασχηματισμοί”.
Το UML εδώ ταιριάζει με Activity diagrams και pipeline οπτικοποίηση. Η συμπεριφορά είναι σχεδόν πάντα σειριακή επεξεργασία.
Η State-Oriented βασίζεται σε states, events και handlers.
Κυρίαρχα διαγράμματα: FSM και Sequence, γιατί θες να δείξεις μεταβάσεις και αλληλουχίες γεγονότων.
Το DDD δίνει έμφαση σε Aggregates και Entities.
Εδώ έχουμε Class diagrams, αλλά επίσης και Context Map, γιατί μιλάμε για όρια, υποdomains και συσχετίσεις μεταξύ bounded contexts.

5.Παραδείγματα UML ανά Οικογένεια.

5.1.Structural UML Example

«Στην Εικόνα 2 βλέπουμε ένα τυπικό layered μοντέλο, με καθαρό διαχωρισμό σε τέσσερα πακέτα.Πάνω απ' όλα είναι το ***UI***, όπου υπάρχει ο ***DashboardController***. Αυτός είναι το σημείο εισόδου: δέχεται αιτήματα, χειρίζεται αλληλεπιδράσεις και κατευθύνει τη ροή.

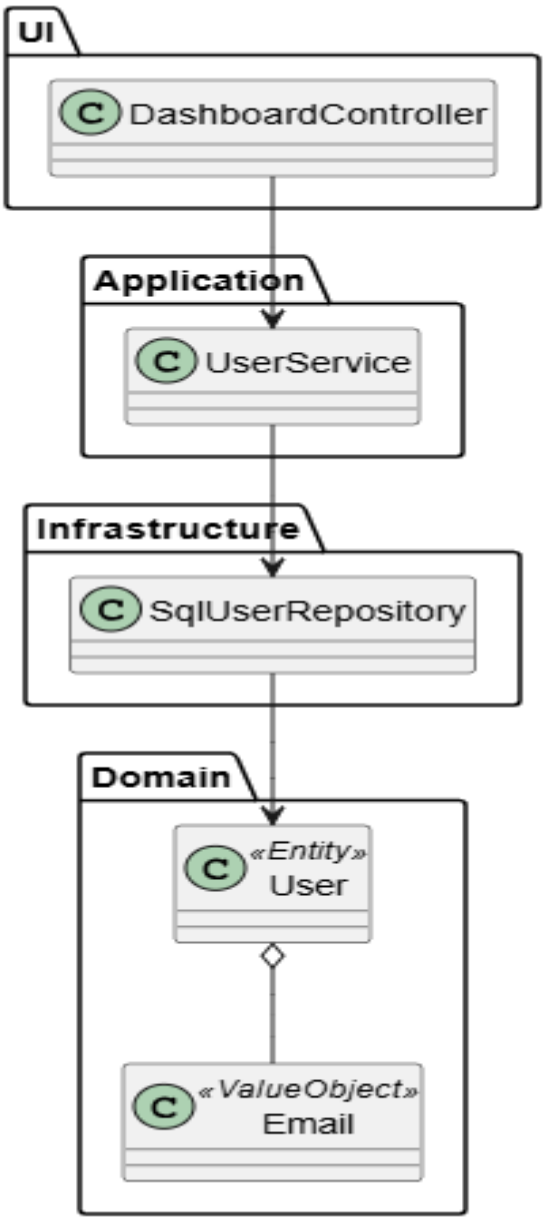
Κάτω από το UI είναι το ***Application layer***, με τον ***UserService***. Εδώ ζει η εφαρμοστική λογική: ο συντονισμός των ενεργειών, οι use cases, και η οργάνωση της εργασίας.

Στη συνέχεια υπάρχει το ***Domain layer***, όπου βλέπουμε τον ***User*** ως Entity και το ***Email*** ως Value Object. Αυτό σημαίνει ότι η έννοια "Χρήστης" έχει ταυτότητα και κύκλο ζωής, ενώ το "Email" είναι μια τιμή που πρέπει να είναι έγκυρη και συνεπής, χωρίς δική της ταυτότητα.

Τέλος, στο ***Infrastructure layer***, έχουμε το ***SqlUserRepository***.

Εκεί υλοποιείται η πρόσβαση στη βάση δεδομένων και η αποθήκευση/ανάκτηση των entities. Οι βελάκια δείχνουν τις εξαρτήσεις: ο Controller καλεί τον Service, ο Service καλεί το Repository, και το Repository χειρίζεται τα Domain αντικείμενα. Και μέσα στο Domain, ο User "περιέχει" ένα Email ως value object. Είναι ένα καθαρό παράδειγμα Structural/Layered αρχιτεκτονικής:

UI για παρουσίαση, Application για use cases, Domain για μοντέλο και κανόνες, Infrastructure για τεχνικές υλοποιήσεις.»



Εικόνα 2: Γράφημα UI

5.2.Flow-Based UML Example

«Στην Εικόνα 3 έχουμε ένα κλασικό pipeline.

Η λογική εδώ είναι σειριακή: κάθε βήμα παίρνει είσοδο, παράγει έξοδο, και περνά στο επόμενο.

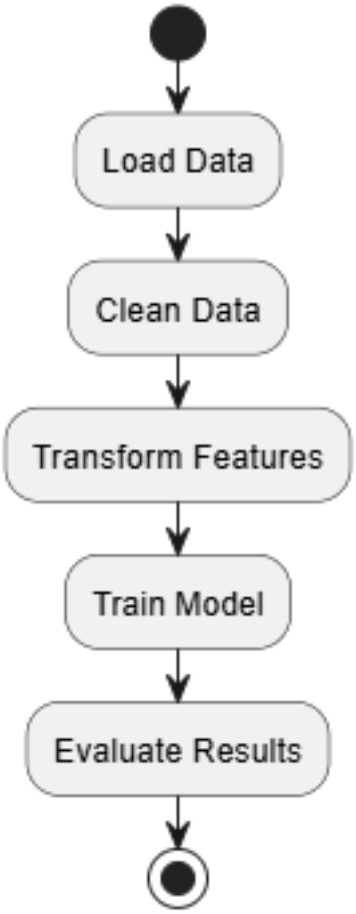
Ξεκινάμε με ***`Load Data`***: φόρτωση δεδομένων από αρχείο, βάση ή API.

Μετά ***`Clean Data`***: καθαρισμός—διόρθωση κενών τιμών, θόρυβος, outliers.

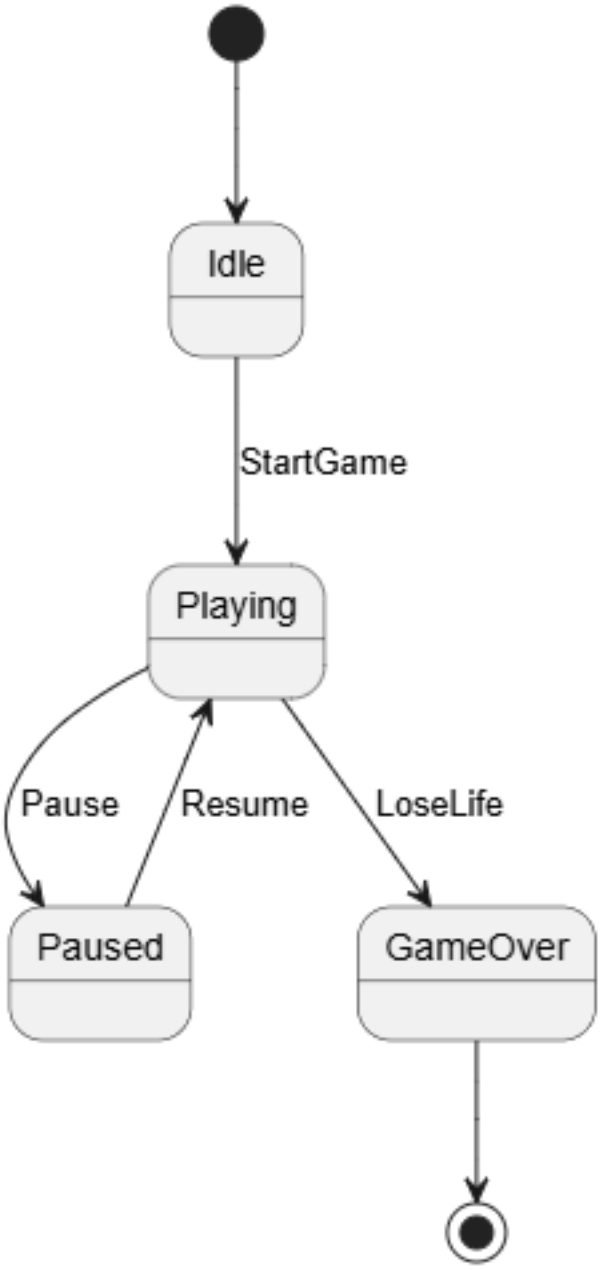
Και τέλος ***`Evaluate Results`***: αξιολόγηση—μετρικές, validation, σύγκριση αποτελεσμάτων.

Αυτό είναι Flow-Based σκέψη: το σύστημα ως αλυσίδα σταδίων, όπου η καθαρότητα της ροής είναι το κλειδί.»Έπειτα ***`Transform Features`***: μετασχηματισμοί χαρακτηριστικών—κωδικοποιήσεις, κανονικοποιήσεις, παράγωγα features.
Μετά περνάμε στο ***`Train Model`***: εκπαίδευση ενός μοντέλου.

Εικόνα 3: Pipeline



5.3.State-Oriented UML Example



Εικονα 4: State-Oriented

«Στην Εικόνα 4 βλέπουμε μια state machine για event-driven συμπεριφορά, τυπική σε παιχνίδια αλλά και σε reactive συστήματα.

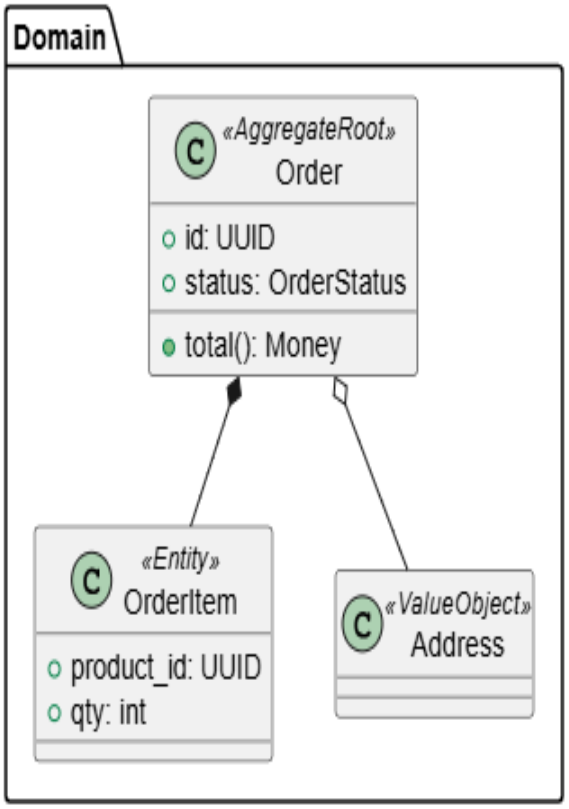
Το σύστημα ξεκινά από την αρχική κατάσταση και πηγαίνει στο ***`Idle`***.
Από εκεί, με το event ***`StartGame`***, μεταβαίνει στο ***`Playing`***.

Όταν συμβεί ***`Pause`***, μεταβαίνει στο ***`Paused`***.
Με το event ***`Resume`***, επιστρέφει ξανά στο ***`Playing`***.

Κατά τη διάρκεια του παιχνιδιού, αν συμβεί ***`LoseLife`***, τότε μεταβαίνει στο ***`GameOver`***.
Και από εκεί, η ροή καταλήγει στην τελική κατάσταση.

Το σημαντικό εδώ είναι ότι η λογική δεν είναι “αν-τότε” σκορπισμένη παντού.
Είναι συγκεντρωμένη σε καταστάσεις και μεταβάσεις: ποιο event επιτρέπεται πότε, και σε ποια νέα κατάσταση οδηγεί.»

5.4 Domain-Centric (DDD) UML Example



Εικόνα 5: Domain-Centric

«Στην Εικόνα 5 βλέπουμε ένα DDD παράδειγμα με ένα Order ως Aggregate Root.

Το Order έχει πεδία όπως id, status, και μια μέθοδο total() που επιστρέφει χρήματα.
Δίπλα του υπάρχει το OrderItem ως Entity, με product_id και qty.

*Η σχέση *Order -- OrderItem σημαίνει ισχυρή σύνθεση:*
τα OrderItems ανήκουν στο Order και δεν έχουν νόημα ανεξάρτητα από αυτό.

Υπάρχει επίσης το Address ως Value Object, συνδεδεμένο με πιο "ελαφριά" σχέση, γιατί είναι μια τιμή που περιγράφει διεύθυνση.

Το μήνυμα του διαγράμματος είναι καθαρό:
στο DDD, οργανώνουμε το domain γύρω από aggregates, ώστε οι κανόνες, τα invariants και οι αλλαγές να ελέγχονται από το Aggregate Root.
Και έτσι το domain παραμένει σταθερό, ακόμα κι όταν το σύστημα μεγαλώνει.»

6.UNIFIED ARCHITECTURE SELECTION FRAMEWORK

6.1.Πυρήνας Πλαισίου Απόφασης

– Τρία βασικά ερωτήματα

1. Τι κυριαρχεί στο σύστημα; Δεδομένα, γεγονότα ή επιχειρησιακή λογική;
2. Πόσο πολύπλοκο είναι το domain;
3. Ποιο είναι το αναμενόμενο μοντέλο εκτέλεσης; (CRUD, Pipeline, Event loop);

Αυτές οι τρεις ερωτήσεις οδηγούν σταθερά σε μία από τις τέσσερις οικογένειες.

6.2.Consolidated Decision Table

Πίνακας 3: Πόσο πολύπλοκο είναι το domain

Χαρακτηριστικό	Structural	Flow-Based	State-Oriented	Domain-Centric
CRUD εφαρμογές	✓✓✓	–	–	–
Business logic (μεσαίο)	✓✓	–	–	●
Σύνθετο domain	●	–	●	✓✓✓
Event loops / FSM	–	–	✓✓✓	●
Pipelines	–	✓✓✓	–	–
Real-time	–	●	✓✓✓	●
Οριζόντια κλιμάκωση	●	●	✓✓✓	✓✓
Time-to-market	✓✓✓	✓	–	–
Long-term maintainability	✓✓	●	✓✓	✓✓✓

Βαθμολογία: ✓ **= Ιδανική επιλογή,** ● **= Καλή επιλογή, – = Όχι κατάλληλη επιλογή**

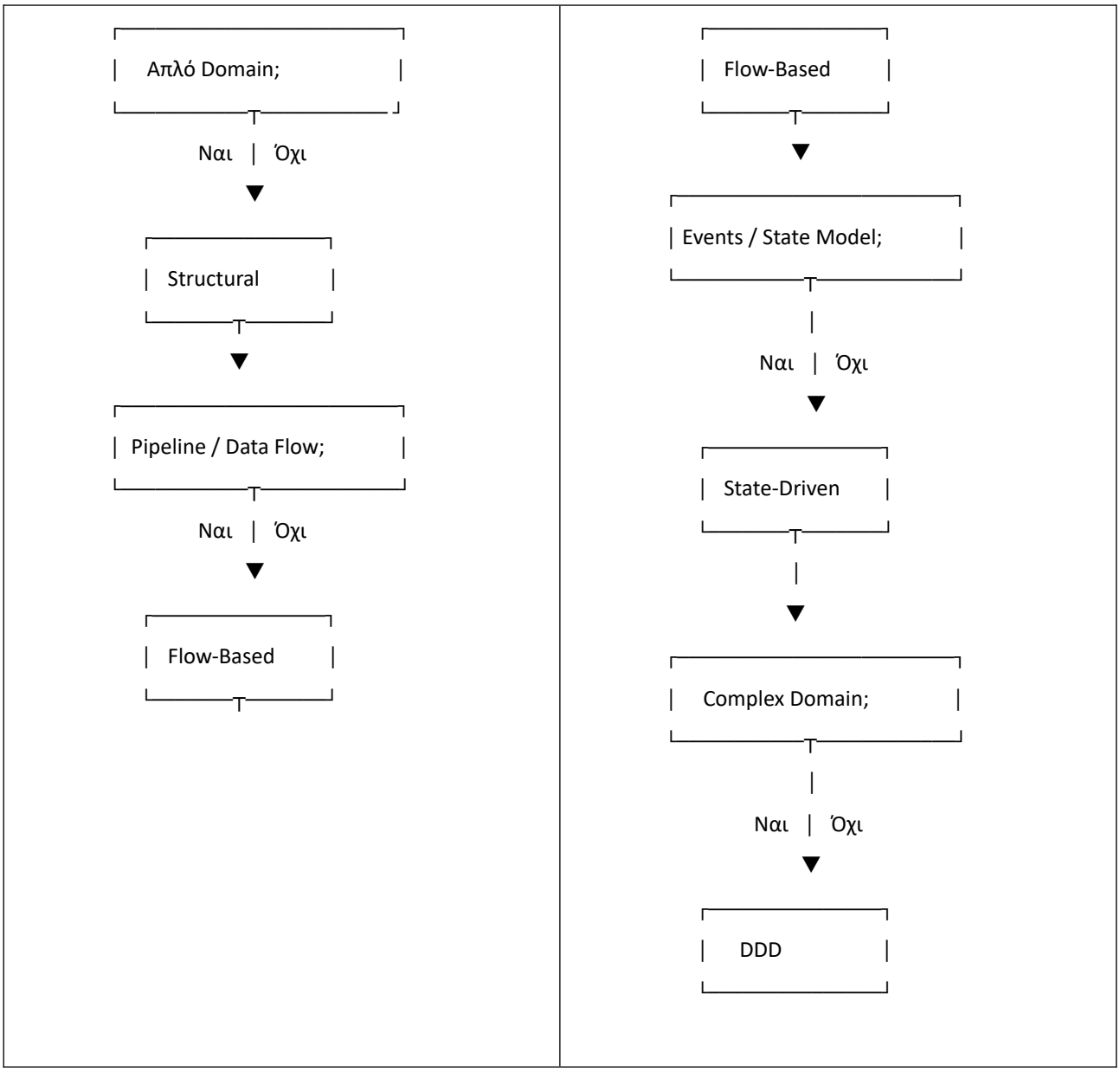
Ο Πίνακας 3 επαναλαμβάνει το βασικό μήνυμα με πιο “συμπυκνωμένη” ματιά:

CRUD → Structural.
Pipelines → Flow-Based.
Event loops και FSM → State-Oriented.
Σύνθετο domain → DDD.

Και προσθέτει κάτι κρίσιμο:
όσο αυξάνεται η ανάγκη για οριζόντια κλιμάκωση και long-term maintainability, τόσο περισσότερο τείνουμε προς State-Oriented και DDD, γιατί οργανώνουν καλύτερα τη συμπεριφορά και τα όρια.

6.3.Graphical Architecture Selector

Πίνακας 4: Graphical Architecture



Ο Πίνακας 4 είναι πρακτικά ένα ***decision tree***.

Ρωτάς πρώτα: “Το domain είναι απλό;”
Αν ναι → πας Structural.

Αν όχι, ρωτάς: “Είναι pipeline ή data flow;”
Αν ναι → Flow-Based.

Αν όχι, ρωτάς: “Έχουμε events ή state model;”
Αν ναι → State-Driven.

Και τέλος: “Το domain είναι σύνθετο;”
Αν ναι → DDD.

Ουσιαστικά, αυτός ο πίνακας λέει:
διάλεξε αρχιτεκτονική με βάση το ***τι είναι το σύστημα στην ουσία του***:
δομή, ροή, κατάσταση, ή domain.

Το παραπάνω διάγραμμα δεν αποτελεί απλώς μια οπτική αναπαράσταση επιλογών, αλλά έναν ***συμπυκνωμένο χάρτη λήψης αρχιτεκτονικών αποφάσεων***. Η λειτουργία του είναι να βοηθήσει τον αναγνώστη να κατανοήσει γρήγορα *ποιο αρχιτεκτονικό μοντέλο ταιριάζει καλύτερα* σε ένα σύστημα, ξεκινώντας από τα βασικά χαρακτηριστικά του Domain και φτάνοντας μέχρι τη δομή του λογισμικού.

Με άλλα λόγια, το διάγραμμα δείχνει ξεκάθαρα ότι η αρχιτεκτονική ***δεν επιλέγεται αυθαίρετα***, αλλά ακολουθεί μια ***σταδιακή διαδικασία αξιολόγησης***:

1. Αν το Domain είναι απλό, προτιμούμε μια κλασική *Structural* προσέγγιση.

2. Αν η λογική βασίζεται σε καθαρές ροές δεδομένων, τότε ενδείκνυται η *Flow-Based* αρχιτεκτονική.

3. Αν η συμπεριφορά καθορίζεται από γεγονότα και καταστάσεις, η *State-Oriented* επιλογή είναι πιο κατάλληλη.

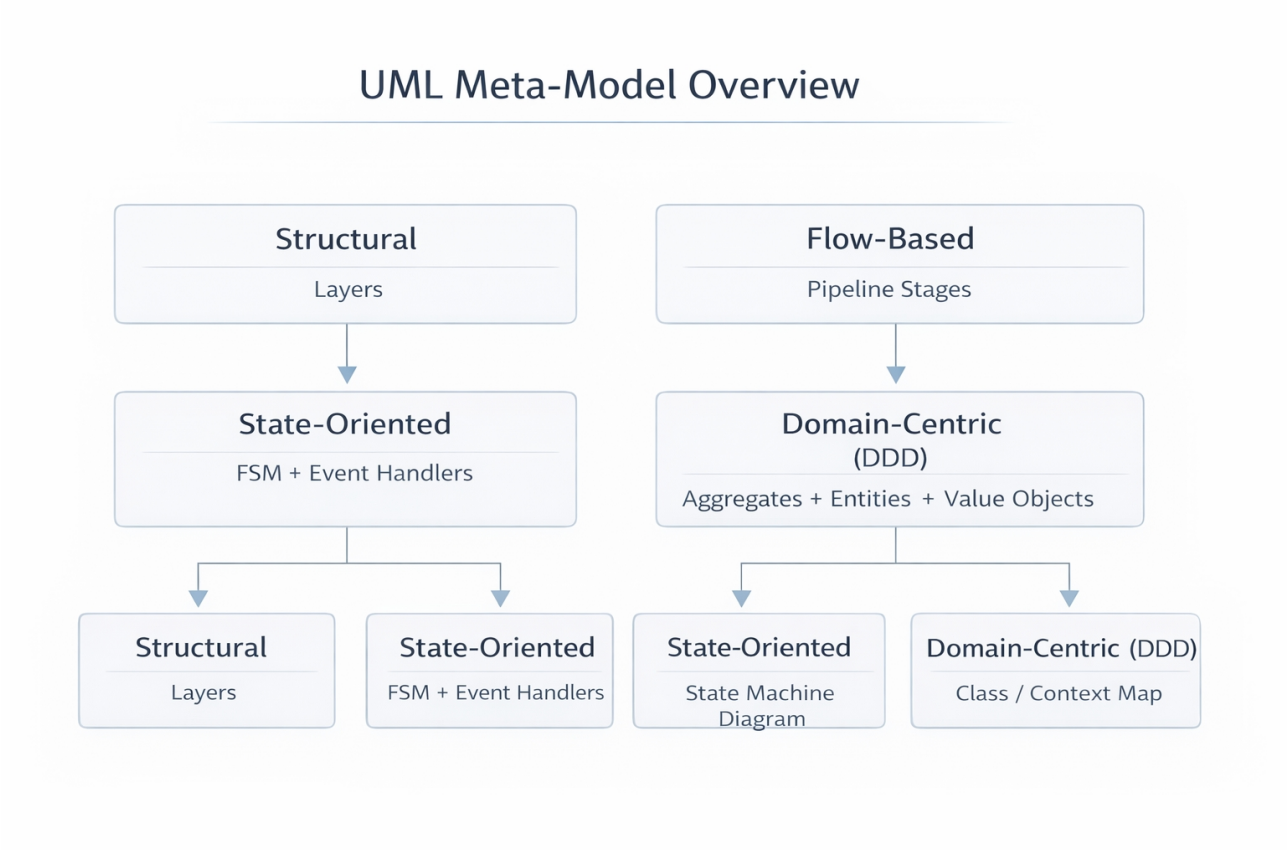
4. Αν το Domain είναι σύνθετο, εξελισσόμενο και απαιτεί αυστηρή απομόνωση εννοιών, τότε η επιλογή οδηγεί φυσικά στο *Domain-Driven Design (DDD)*.

Συνολικά, το διάγραμμα αυτό λειτουργεί ως ***εργαλείο προσανατολισμού***: προσφέρει ένα νοητικό πλαίσιο που βοηθά τον αναγνώστη να συνδέσει τις λειτουργικές ανάγκες του συστήματος με το κατάλληλο αρχιτεκτονικό στυλ. Έτσι, η τελική επιλογή δεν είναι θέμα προτίμησης, αλλά ***τεκμηριωμένης, μεθοδικής και ρεαλιστικής απόφασης***.

6.4.Executive Guide

- Χρησιμοποίησε Structural όταν το σύστημα είναι απλό, χρειάζεται γρήγορη ανάπτυξη και δεν υπάρχουν πολύπλοκοι κανόνες.
- Χρησιμοποίησε Flow-Based όταν η κύρια δραστηριότητα είναι επεξεργασία δεδομένων ή μετασχηματισμοί.
- Χρησιμοποίησε State-Oriented όταν το σύστημα λειτουργεί με γεγονότα, καταστάσεις ή μηχανές καταστάσεων.
- Χρησιμοποίησε DDD όταν το domain είναι περίπλοκο, κρίσιμο ή απαιτεί μέγιστη ορθότητα και συνέπεια.

6.5.UML Meta-Model (Σύντομη Επισκόπηση)



Εικόνα 5: UML Meta-Model

«Στην Εικόνα 6 βλέπουμε μια συνοπτική “επισκόπηση μετα-μοντέλου” για το UML, δηλαδή έναν χάρτη που συνδέει ***οικογένειες αρχιτεκτονικής*** με ***τι μοντελοποιούμε*** και ***ποια UML διαγράμματα ταιριάζουν***.

Στην κορυφή του διαγράμματος υπάρχει ο τίτλος: *UML Meta-Model Overview*. Και ακριβώς από κάτω, το σχήμα χωρίζεται σε δύο βασικούς άξονες, αριστερά και δεξιά.

Στην αριστερή πλευρά, στην πρώτη κάρτα, βλέπουμε την οικογένεια ***Structural*** με τη λέξη-κλειδί ***Layers***.

Δηλαδή, εδώ το σύστημα περιγράφεται ως επίπεδα: παρουσίαση, εφαρμογή, domain, υποδομή—ένα καθαρά δομικό μοντέλο.

Στη δεξιά πλευρά, στην αντίστοιχη κάρτα, εμφανίζεται η οικογένεια ***Flow-Based*** με τη λέξη-κλειδί ***Pipeline Stages***.

Εδώ το σύστημα μοντελοποιείται ως ροή σταδίων: βήμα πρώτο, βήμα δεύτερο, μετασχηματισμοί, επεξεργασία—μια αλληλουχία εργασιών.

Τώρα, το διάγραμμα δείχνει δύο καθοδικές κινήσεις, με βέλη.

Από το ***Structural*** (Layers), ένα βέλος οδηγεί προς την οικογένεια ***State-Oriented***, με περιγραφή ***FSM + Event Handlers***. Το νόημα εδώ είναι ότι, όταν ένα δομικό σύστημα αρχίζει να αποκτά συμπεριφορά που εξαρτάται από κατάσταση—δηλαδή ροές που δεν είναι απλά “κλήση υπηρεσίας”, αλλά “μετάβαση από κατάσταση σε κατάσταση”—τότε περνάμε στο State-Oriented, όπου κεντρικά στοιχεία είναι οι μηχανές καταστάσεων και οι χειριστές γεγονότων.

Αντίστοιχα, από το ***Flow-Based*** (Pipeline Stages), ένα βέλος οδηγεί προς την οικογένεια ***Domain-Centric (DDD)***, με περιγραφή ***Aggregates + Entities + Value Objects***.

Εδώ η ιδέα είναι ότι όταν οι ροές δεν είναι απλώς τεχνικές μετατροπές, αλλά κρύβουν επιχειρησιακό νόημα—κανόνες, οντότητες, ακεραιότητα—τότε η αρχιτεκτονική μετακινείται προς DDD: το domain γίνεται ο πυρήνας και οργανώνεται γύρω από aggregates, entities και value objects.

Στο κάτω μέρος του διαγράμματος, βλέπουμε τις “εκροές” μοντελοποίησης—δηλαδή τι είδους UML διαγράμματα ταιριάζουν ως κύρια εργαλεία, ανά κατεύθυνση.

Κάτω από το ***State-Oriented***, το σχήμα “σπάει” σε δύο επιλογές:
Πρώτον, ξαναβλέπουμε ***Structural — Layers***, που δηλώνει ότι ακόμη και σε state-driven συστήματα, συχνά κρατάμε μια δομική οργάνωση ως βάση.
Και δεύτερον, ***State-Oriented — FSM + Event Handlers***, που τονίζει πως το κλειδί παραμένει η μοντελοποίηση καταστάσεων και γεγονότων.

Κάτω από το ***Domain-Centric (DDD)***, το διάγραμμα επίσης σπάει σε δύο κατευθύνσεις:
Από τη μία, ***State-Oriented — State Machine Diagram***, δηλαδή όταν το domain εκφράζεται ισχυρά μέσα από μεταβάσεις κατάστασης—όπως σε workflows, εγκρίσεις, lifecycles.

Και από την άλλη, ***Domain-Centric (DDD) — Class / Context Map***, όπου η βασική UML γλώσσα είναι τα class diagrams για το μοντέλο και τα context maps για τα όρια, τα υποdomains και τις σχέσεις μεταξύ bounded contexts.

Με απλά λόγια, η Εικόνα 6 λέει το εξής:

Αν σκέφτεσαι το σύστημα ως ***δομή***, ξεκινάς από Structural και Layers. Αν σκέφτεσαι το σύστημα ως ***ροή***, ξεκινάς από Flow-Based και Pipeline Stages. Αν κυριαρχεί η ***κατάσταση και τα γεγονότα***, τότε χτίζεις State-Oriented και μοντελοποιείς με state machines και handlers. Και αν κυριαρχεί το ***επιχειρησιακό νόημα***, τότε πας Domain-Centric, δηλαδή DDD—με aggregates, entities, value objects, και χαρτογράφηση των ορίων με context maps.

Είναι, ουσιαστικά, ένας χάρτης:

από το “πώς βλέπεις το σύστημα”, στο “πώς το μοντελοποιείς σωστά σε UML”.»

7.UML PATTERNS & ANTI-PATTERNS ΑΝΑ ΟΙΚΟΓΕΝΕΙΑ

7.1.Structural / Layered Architecture

–Patterns

- Καθαρή ιεραρχία πακέτων: UI → Application → Domain → Infrastructure
- Use-case Sequence Diagrams με Controller → Service → Repository
- Interface segregation στα Service / Repository
- Dependency arrows πάντα προς μέσα (Domain)

– Anti-Patterns

- Cyclic dependencies μεταξύ UI ↔ Domain
- Υπερβολικά «χοντρές» classes με πολλαπλές ευθύνες
- Business logic στο UI (Controller bloating)
- Repositories που περιέχουν domain logic

7.2.Flow-Based / Pipeline Architecture

–Patterns

- Activity Diagrams που αναπαριστούν καθαρή ροή: Input → Step1 → Step2 → Output
- Class Diagrams με ***Stage*** components που εφαρμόζουν κοινό interface
- Χρήση Composition για pipeline assembly
- Sequence Diagrams που δείχνουν data passing μεταξύ σταδίων

–Anti-Patterns

- Branching logic μέσα σε κάθε Pipeline stage
- Αποθήκευση shared mutable state μεταξύ σταδίων
- «Μικτή» αρχιτεκτονική που εισάγει domain logic στα στάδια
- Παραβίαση της αρχής Single Responsibility στα components

7.3.State-Oriented / Event-Driven Architecture

–Patterns

- Καθαρά ***State Machine Diagrams*** με explicit transitions
- Event → Handler → StateChange ακολουθίες
- Sequence diagrams για events και asynchronous flows
- Διαχωρισμός μεταξύ Events, Commands, Handlers

–Anti-Patterns

- Handlers που τροποποιούν πολλαπλές καταστάσεις παράλληλα
- Events που «γνωρίζουν» business logic (fat events)
- Ασύμβατες μεταβάσεις κατάστασης (state leaks)
- Ελλιπής μοντελοποίηση της τελικής κατάστασης (no terminal states)

7.4.Domain-Centric Architecture (DDD)

–Patterns

- Καθαρή διάκριση ***Aggregate → Entity → Value Object***
- Invariants δηλωμένα στον Aggregate Root
- Context Maps για στρατηγική DDD μοντελοποίηση
- UML σχέσεις με composition όπου ταιριάζει (Aggregate consistency)
- Domain Events με σαφή ροή: Event → Handler → Side Effects

– Anti-Patterns

- Θεόρατοι Aggregates που περιλαμβάνουν πολλές οντότητες
- Entities που περιέχουν επιχειρησιακή λογική άλλου bounded context
- Value Objects που αλλάζουν κατάσταση (mutable VO)
- Διασταλμένες εξαρτήσεις Application → Domain αντί για αντίστροφη διαδρομή

7.5.Συγκεντρωτικός Πίνακας Patterns / Anti-Patterns

Πίνακας 5: Patterns - Anti-Patterns

Οικογένεια	Patterns	Anti-Patterns
Structural	Layers, Controllers, Services, Repositories	Cyclic deps, Fat controllers
Flow-Based	Stages, Activity Diagrams	Branching stages, Shared mutable state
State-Oriented	FSM, Events, Handlers	State leaks, Fat events
DDD	Aggregates, Entities, VOs, Context Maps	Overgrown aggregates, mutable VOs

Εδώ μπαίνουμε στο πρακτικό “τι να κάνεις” και “τι να αποφύγεις”. **Structural**: καλά patterns είναι Layers, Controllers, Services, Repositories. Αλλά πρόσεξε τα Anti-Patterns: κυκλικές εξαρτήσεις και “fat controllers” που καταπίνουν όλη τη λογική.

Flow-Based: patterns είναι Stages και Activity Diagrams. Anti-patterns: υπερβολικό branching στα στάδια και shared mutable state, που κάνει τη ροή απρόβλεπτη.

State-Oriented: patterns είναι FSM, Events και Handlers.Anti-patterns: state leaks — δηλαδή κατάσταση που “ξεφεύγει” από τον έλεγχο — και “fat events”, όπου ένα event γίνεται τεράστιο και μη διαχειρίσιμο.

DDD: patterns είναι Aggregates, Entities, Value Objects και Context Maps. Anti-patterns: overgrown aggregates που γίνονται “τέρας”, και mutable value objects που χαλάνε την έννοια της ακεραιότητας.

8.USE CASES & REAL PROJECTS MAPPING

8.1.Structural Architecture

-Use Cases

- Εφαρμογές γραφείου (UI + business logic + local DB)
- Web portals, admin panels, online booking
- Mobile εφαρμογές με MVC/MVVM
- REST APIs μικρής–μεσαίας πολυπλοκότητας

-Real Project Examples

- Django/Flask web apps
- ASP.NET MVC / Spring MVC
- Desktop ERP-lite εργαλεία

-UML Mapping

- Class + Package diagrams
- Sequence diagrams για Use Cases
- Repository + Service layers

8.2.Flow-Based Architecture

-Use Cases

- ETL workflows (extract-transform-load)
- Big Data processing (Spark pipelines)
- Machine Learning pipelines
- Speech/Video processing tools
- Compilers & static analyzers

-Real Project Examples

- Airflow DAGs
- Azure ML pipelines
- FFmpeg processing flows
- PySpark transformations

- UML Mapping

- Activity diagrams (best match)
- Component diagrams για Stages
- Data-flow sequences

8.3.State-Oriented / Event-Driven Architecture

-Use Cases

- Παιχνίδια (card games, strategy, platformers)
- IoT sensor networks
- Real-time trading engines
- Microservices μέσω Kafka / RabbitMQ
- Reactive συστήματα (Akka, Rx, WebSockets)

-Real Project Examples

- Unreal/Unity state machines
- Home automation event systems
- Kafka-based event-driven microservices
- Game engines rule subsystems

-UML Mapping

- State Machine Diagrams
- Event-flow Sequence diagrams
- Component + Event Handler diagrams

8.4.Domain-Centric (DDD) Architecture

-Use Cases

- Banking, payments, fintech
- Insurance, logistics, invoicing
- ERP, CRM, enterprise platforms
- E-commerce engines με πολλαπλά subdomains

-Real Project Examples

- CQRS/DDD microservice platforms
- Enterprise-level Spring Boot + DDD
- Large e-commerce backends (Order, Cart, Inventory BCs)

-UML Mapping

- Class diagrams με Aggregates
- Context Maps (bounded contexts)
- Domain event interaction diagrams

8.5 Συνολικός Πίνακας Αντιστοιχίσεων

Πίνακας 6: Πίνακας Αντιστοιχίσεων

Τύπος Έργου	Κατάλληλη Οικογένεια	Κυρία UML Μοντελοποίηση
Web app / admin panel	Structural	Class, Package, Sequence
Desktop εργαλείο	Structural	MVC/MVVM diagrams
ETL / ML pipeline	Flow-Based	Activity / Data Flow
Compiler / Analyzer	Flow-Based	Activity / Component
Game	State-Oriented	FSM, Event Diagrams
IoT / Reactive	State-Oriented	Event + State Machine
Enterprise business system	Domain-Centric	Aggregates / Context Maps
E-commerce platform	DDD	Aggregates, Events

Ο Πίνακας 6 είναι ο πιο “εφαρμοσμένος”:

Web app / admin panel → Structural, με class, package, sequence.
Desktop εργαλείο → Structural, συχνά με MVC ή MVVM.
ETL / ML pipeline → Flow-Based, activity και data flow.
Compiler / analyzer → επίσης Flow-Based, γιατί δουλεύει σαν στάδια ανάλυσης.
Game → State-Oriented, γιατί η λογική είναι καταστάσεις, loops, events.
IoT / reactive → State-Oriented, με events και state machines.
Enterprise business system → Domain-Centric, με aggregates και context maps.
E-commerce platform → DDD, γιατί έχει πλούσιο domain: παραγγελίες, πληρωμές, αποθήκη, κανόνες.

9.UNIFIED ARCHITECTURE COMPARISON CHART

9.1 Ενοποιημένο Συγκριτικό Διάγραμμα

Πίνακας 7: Συγκριτικό Πίνακας

Απόδοση	**Καλή**	**Πολύ υψηλή σε pipelines**	**Υψηλή σε async flows**	**Υψηλή αλλά όχι κορυφαία**
Συντηρησιμότητα	Υψηλή	Μεσαία	Μεσαία–Υψηλή	Πολύ υψηλή
Εφαρμοστικότητα	Web, Desktop, Mobile	Data engineering, ML	Games, IoT, microservices	Enterprise συστήματα
Ομοιογένεια UML	Πολύ σταθερή	Σταθερή	Ετερογενής (πολλοί τύποι UML)	Πολύ σταθερή
Κόστος Ανάπτυξης	Χαμηλό	Μεσαίο	Μεσαίο–Υψηλό	Υψηλό (αρχικά)
Καλύτερο σημείο χρήσης	CRUD & business apps	Processing pipelines	Real-time, async systems	Πολύπλοκο domain
Σχέση με DDD	Μερική	Αδύναμη	Μερική (μέσω events)	Πλήρης

- ***Structural*** = το «default» για απλές–μεσαίες εφαρμογές.
- ***Flow-Based*** = όταν το βασικό είναι οι μετασχηματισμοί δεδομένων.
- ***State-Oriented*** = όταν το σύστημα είναι ζωντανό, event-driven ή game-like.
- ***DDD*** = όταν κυριαρχεί η πολυπλοκότητα domain και οι επιχειρησιακοί κανόνες.

Εδώ βλέπουμε trade-offs.

Flow-Based δίνει ***πολύ υψηλή απόδοση*** σε pipelines, αλλά η συντηρησιμότητα είναι πιο “μεσαία”, γιατί οι ροές μπορούν να γίνουν σύνθετες.

Structural έχει υψηλή συντηρησιμότητα και χαμηλό κόστος ανάπτυξης — ειδικά στην αρχή.

State-Oriented είναι ισχυρό σε async και real-time, αλλά έχει κόστος σε complexity, γιατί οι μεταβάσεις θέλουν πειθαρχία.

DDD είναι “βασιλιάς” στη συντηρησιμότητα για σύνθετα domains, αλλά έχει υψηλότερο αρχικό κόστος: χρειάζεται μοντελοποίηση.

10. METHODOLOGY & WORKFLOW ΓΙΑ ΔΗΜΙΟΥΡΓΙΑ UML

Η δημιουργία UML διαγραμμάτων δεν είναι μια μηχανική διαδικασία παραγωγής εικόνων, αλλά μια συστηματική μεθοδολογία μοντελοποίησης που εξαρτάται από το αρχιτεκτονικό στυλ, τα χαρακτηριστικά του συστήματος και το επίπεδο αφαίρεσης στο οποίο εργάζεται η ομάδα.

Σε αυτή την ενότητα παρουσιάζεται η ενιαία μεθοδολογία και το workflow για τη δημιουργία UML σε κάθε μία από τις τέσσερις αρχιτεκτονικές οικογένειες: Structural, Flow-Based, State-Oriented και Domain-Centric (DDD).

Η δομή είναι κοινή για να διευκολύνει τη σύγκριση, αλλά κάθε workflow προσαρμόζεται στις απαιτήσεις του αντίστοιχου μοντέλου

10.1 Structural UML Workflow

(Layered, MVC, Clean Architecture, Hexagonal – Structural οικογένεια)

-Στόχος UML στη Structural αρχιτεκτονική

Να αποτυπώσει καθαρά:

- τις ***στιβάδες*** (layers),
- τις ***ευθύνες*** κάθε στρώματος,
- τις ***εξαρτήσεις*** μεταξύ modules,
- τις ***ροές κλήσεων*** από UI → Domain → Infrastructure.

-Προτεινόμενος UML συνδυασμός

- ***Package Diagram*** (για Layer boundaries)
- ***Class Diagram*** (Controllers, Services, Repositories)
- ***Sequence Diagram*** (Use-case level interactions)

-Workflow

1. ***Εντόπισε τα Layer Boundaries***
   – UI, Application, Domain, Infrastructure.
2. ***Κατάγραψε τα Components κάθε Layer***
   – Controllers, Services, Repositories, Entities.
3. ***Αποτύπωσε Dependencies μεταξύ Layers***
   – Μόνο επιτρεπόμενες ροές (π.χ. Clean Architecture: inward dependencies).
4. ***Σχεδίασε Class Diagrams για τις βασικές δομικές οντότητες.***
5. ***Οπτικοποίησε ένα Sequence Diagram για τα 2–3 κύρια Use Cases.***
6. ***Έλεγξε την αντιστοίχιση με το Code Structure.***

10.2 Flow-Based UML Workflow

(Pipeline, Batch Processing, ETL, Data Processing Systems)

-Στόχος UML στη Flow-Based αρχιτεκτονική

Να δείξει:

- τα ***στάδια επεξεργασίας*** (stages),
- τις ***ροές δεδομένων***,
- τους ***μετασχηματισμούς***,
- τους ***κανόνες εισόδου–εξόδου*** σε κάθε βήμα.

-Προτεινόμενος UML συνδυασμός

- ***Activity Diagram*** (Pipeline flow)
- ***Component Diagram*** (Stage components)
- ***Sequence Diagram*** (για data transformations)

-Workflow

1. ***Καταγραφή όλων των Pipeline Stages***
   – Stage1 → Stage2 → Stage3…
2. ***Ορισμός Data Contracts***
   – input/output schema for each stage.
3. ***Activity Diagram για τη βασική ροή δεδομένων.***
4. ***Component Diagram για δομικό επίπεδο***
   – ποια components υλοποιούν stages.
5. ***Sequence Diagram για πολύπλοκη αλληλεπίδραση***
   – ιδανικό για asynchronous transformations.
6. ***Έλεγχος consistency με πραγματική ροή στα δεδομένα.***

10.3 State-Oriented UML Workflow

(Event-Driven Systems, State Machines, Reactive Architectures)

-Στόχος UML στη State-Oriented αρχιτεκτονική

Να αποτυπώσει:

- ***καταστάσεις***,
- ***μεταβάσεις***,
- ***γεγονότα (events)***,
- ***handlers***,
- ***κανόνες ενεργοποίησης***.

-Προτεινόμενος UML συνδυασμός

- ***State Machine Diagram*** (κύριος οδηγός)
- ***Sequence Diagram*** (event → handler → state change)
- ***Activity/Flow Diagram*** (optional)

-Workflow

7. ***Αποτύπωσε όλες τις πιθανές καταστάσεις του συστήματος.***
8. ***Σύνδεσε Events → Transitions → Actions.***
9. ***Δημιούργησε State Machine Diagram***
   – είναι το "κέντρο" της αρχιτεκτονικής.
10. ***Πρόσθεσε Sequence Diagram για σημαντικά event-handling cycles.***
11. ***Συμπλήρωσε μία Event Table*** (event → new state).
12. ***Έλεγξε consistency με τους κανόνες business logic.***

10.4 Domain-Centric UML Workflow

(Domain Driven Design — Aggregates, Entities, Value Objects, Bounded Contexts)

-Στόχος UML στη DDD αρχιτεκτονική

Να μοντελοποιήσει:

- τα ***aggregates***,
- τις ***οντότητες & value objects***,
- τα ***domain services***,
- τα ***bounded contexts***,
- τις ***σχέσεις μεταξύ subdomains***.

-Προτεινόμενος UML συνδυασμός

- ***Class Diagram*** (Aggregates, Entities, VOs)
- ***Context Map*** (Bounded Contexts)
- ***Sequence Diagram*** (domain behaviors)
- ***Package Diagram*** (bounded context organization)

-Workflow

13. Ορισμός ***Subdomains*** & ***Bounded Contexts.***
14. Καταγραφή ***Aggregates***
    – Root Entity, Entities, Value Objects.
15. **Class Diagrams** για κάθε ***Aggregate.***
16. ***Context Map (Strategic Design)***
    – upstream/downstream σχέσεις, ACL, conformist, anti-corruption.
17. ***Sequence Diagram για domain behaviors***
    – *policies*, *invariants*, *transactional boundaries*.
18. Έλεγχος **invariants** → consistency across aggregates.

10.5 Unified UML Methodology

(Η ενοποιημένη μέθοδος για όλες τις οικογένειες)

Η ενιαία μεθοδολογία παραγωγής UML αποτελείται από 6 καθολικά βήματα που ισχύουν για όλες τις αρχιτεκτονικές:

Βήμα 1 — Identify the Purpose

Τι θέλεις να δείξεις;
Δομή; Ροή; Συμπεριφορά; Στρατηγική οριοθέτηση;

Βήμα 2 — Select the Appropriate UML Diagram Types

Η επιλογή UML δεν είναι αυθαίρετη.
Πρέπει να βασίζεται στον αρχιτεκτονικό τύπο:

- *Structural → Package/Class/Sequence*
- *Flow-Based → Activity/Component*
- *State-Oriented → State Machine/Sequence*
- *DDD → Class + Context Map + Sequence*

Βήμα 3 — Establish Boundaries

Πού τελειώνει το module;
Ποια είναι τα layers, stages ή contexts;

Βήμα 4 — Model the Core Mechanisms

Οι μηχανισμοί είναι:

- *interactions,*
- *transitions,*
- *pipelines,*
- *entity/value relationships.*

Βήμα 5 — Validate Consistency

Ερώτηση-κλειδί:
Το UML ταιριάζει με τον κώδικα ή περιγράφει ένα φανταστικό σύστημα;

Βήμα 6 — Version & Maintain

Κράτησε UML versioning μαζί με software versions.

11. Architecture Evaluation Metrics

*(Διαγράμματα Ωριμότητας & Εκτίμηση Ρίσκου)***

11.1 Metrics Ωριμότητας

–Modularity Index

Μετρά το βαθμό διαχωρισμού σε modules.

- *Υψηλό MI → Καλύτερη παράλληλη ανάπτυξη & συντήρηση*
- *Χαμηλό MI → Σφιχτές εξαρτήσεις, δύσκολη εξέλιξη*

Πίνακας 8: Ωριμότητας & Εκτίμηση Ρίσκου MI

Οικογένεια	***MI***
Structural	*Υψηλό*
Flow-Based	*Μεσαίο*
State-Oriented	*Μεσαίο–Υψηλό*
DDD	*Πολύ Υψηλό*

Στον Πίνακα 8, η ωριμότητα (MI) είναι υψηλή για Structural, μεσαία για Flow-Based, μεσαία–υψηλή για State-Oriented, και πολύ υψηλή για DDD — γιατί το DDD απαιτεί μεγαλύτερη πειθαρχία αλλά δίνει σταθερότητα.

–Complexity Surface

Προσδιορίζει πόσο «επιφάνεια πολυπλοκότητας» εκθέτει το σύστημα: APIs, events, use cases, κανόνες.

• Χαμηλό CS → απλό UI-driven σύστημα

• Υψηλό CS → επιχειρησιακή πολυπλοκότητα, πλούσιο domain

Οικογένεια	***CS***
Structural	*Χαμηλό–Μεσαίο*
Flow-Based	*Μεσαίο*
State-Oriented	*Μεσαίο–Υψηλό*
DDD	*Πολύ Υψηλό*

Πίνακας 9: Διαγράμματα Ωριμότητας & Εκτίμηση Ρίσκου CS

Στον Πίνακα 9 (CS), το κόστος πολυπλοκότητας ανεβαίνει όσο προχωράς προς State-Oriented και DDD.

– Change Reaction Time

Χρόνος αντίδρασης σε αλλαγή απαιτήσεων.

- *Low CRT → γρήγορη προσαρμογή*
- *High CRT → ακριβές αλλαγές, πολλά dependencies.*

Πίνακας 10: Διαγράμματα Ωριμότητας & Εκτίμηση Ρίσκου CRT

Οικογένεια	*CRT*
Structural	*Χαμηλό*
Flow-Based	*Μεσαίο*
State-Oriented	*Μεσαίο–Υψηλό*
DDD	*Χαμηλό (μετά την αρχική φάση)*

Στον Πίνακα 10 (CRT), η Structural έχει χαμηλό ρίσκο αλλαγών, Flow-Based μεσαίο, State-Oriented μεσαίο–υψηλό, ενώ το DDD γίνεται χαμηλό ρίσκο μετά την αρχική φάση, επειδή όταν "στήσεις" σωστά το domain, οι αλλαγές απορροφώνται καλύτερα.

11.2 Μετρικές Ρίσκου

–Risk of Misalignment

Πόσο πιθανό είναι η αρχιτεκτονική να μη συνάδει με το πραγματικό domain.

Πίνακας 11: Μετρικές Ρίσκου ROM

Οικογένεια	**RoM**
Structural	Μεσαίο
Flow-Based	Χαμηλό
State-Oriented	Μεσαίο
DDD	Χαμηλό (καλύτερη χαρτογράφηση domain)

Στον Πίνακα 11 (RoM), το Flow-Based έχει χαμηλό ρίσκο επειδή οι ροές είναι σαφείς, ενώ το DDD είναι επίσης χαμηλό όταν το domain έχει χαρτογραφηθεί σωστά.

–Fragility Under Change

Πόσο «σπάει» ο κώδικας όταν αλλάξει μια βασική έννοια.

Πίνακας 12: Μετρικές Ρίσκου FUC

Οικογένεια	**FUC**
Structural	Μεσαίο
Flow-Based	Χαμηλό
State-Oriented	Υψηλό (ευαίσθητο σε transition logic)
DDD	Πολύ Χαμηλό

Στον Πίνακα 12 (FUC), το State-Oriented έχει υψηλότερο ρίσκο, γιατί είναι ευαίσθητο στη λογική μεταβάσεων. Το DDD έχει πολύ χαμηλό ρίσκο εδώ, επειδή οι invariants και οι κανόνες "κλειδώνουν" τη συμπεριφορά.

– Over-Engineering Index

Πόσο εύκολα μπορεί να υπερεκπτυχθεί μια αρχιτεκτονική πέρα από τις ανάγκες.

Πίνακας 13: Μετρικές Ρίσκου OEI

Οικογένεια	**OEI**
Structural	Χαμηλό
Flow-Based	Χαμηλό
State-Oriented	Μεσαίο
DDD	Υψηλό (όταν το domain είναι απλό)

Και στον Πίνακα 13 (OEI), βλέπουμε μια ενδιαφέρουσα προειδοποίηση:

*το DDD έχει υψηλό ρίσκο **όταν το domain είναι απλό**, γιατί τότε απλά "υπερσχεδιάζεις".*

11.3 Visual Risk–Maturity Diagram

Ένα διάγραμμα που δείχνει την ισορροπία ωριμότητας–ρίσκου:

ity ↑

| DDD

| State-Oriented

| Structural

| Flow-Based

|

+----------→ Risk

Ερμηνεία:

- *Το DDD έχει κορυφαία ωριμότητα αλλά υψηλό αρχικό κόστος/ρίσκο υπερβολής.*
- *Structural έχει χαμηλό ρίσκο και σταθερή ωριμότητα.*
- *Flow-Based έχει χαμηλό ρίσκο αλλά περιορισμένη ωριμότητα (μόνο για data-driven έργα).*
- *State-Oriented είναι δυνατό αλλά ευαίσθητο στα λάθη.*

11.4 Κριτήρια Τελικής Αξιολόγησης

1. *Καταλληλότητα για το domain*
2. *Κόστος ανάπτυξης vs διάρκεια ζωής*
3. *Ευκολία αλλαγών*
4. *Ποιότητα και πληρότητα UML μοντελοποίησης*
5. *Κίνδυνος τεχνικού χρέους*
6. *Ρίσκο λάθους στη συμπεριφορική λογική*
7. *Προσδοκώμενος ρυθμός ανάπτυξης του συστήματος*

12.Conclusion

12.1 Συνολική Θεώρηση

Το κεφάλαιο αυτό παρουσίασε μια ενιαία δομική προσέγγιση για την κατανόηση, ταξινόμηση και αξιολόγηση αρχιτεκτονικών λογισμικού μέσω UML. Παρά την ποικιλία προτύπων που συναντώνται στη βιομηχανία (Layered, Clean, MVC, Pipeline, Event-Driven, DDD), όλες καταλήγουν σε τέσσερις θεμελιώδεις οικογένειες: Structural, Flow-Based, State-Oriented και Domain-Centric. Κάθε οικογένεια διαθέτει δικό της μοτίβο δομής, συμπεριφοράς και μοντελοποίησης, το οποίο εκφράζεται με συγκεκριμένους τύπους UML διαγραμμάτων.

12.2 Ενιαίο Πλαίσιο Απόφασης

Το Unified Architecture Selection Framework που αναπτύχθηκε προσφέρει έναν πρακτικό, αντικειμενικό και ρεαλιστικό μηχανισμό λήψης απόφασης. Με αξιοποίηση μετρικών ωριμότητας και ρίσκου, διαγραμμάτων σύγκρισης, decision trees και UML templates, παρέχεται μια πλήρης μεθοδολογία για τον αρχιτέκτονα λογισμικού ώστε να επιλέγει την πιο κατάλληλη αρχιτεκτονική με βάση τεκμηριωμένα κριτήρια. Αυτό επιτρέπει την ανάπτυξη συστημάτων πιο ανθεκτικών, επεκτάσιμων και αξιόπιστων.

12.3 Ο Ρόλος UML στην Αρχιτεκτονική Ανάλυση

Η χρήση UML επιτρέπει τη σύγκριση των αρχιτεκτονικών σε επίπεδο δομής (Class/Package Diagrams), ροής (Activity/Pipeline Diagrams), συμπεριφοράς (State Machines, Event Sequences) και, όπου απαιτείται, domain modeling (Aggregates, Context Maps). Με βάση τις αναλύσεις προηγούμενων σελίδων, έγινε φανερό ότι καμία αρχιτεκτονική δεν είναι «καλύτερη» από τις άλλες με απόλυτους όρους· η καταλληλότητα καθορίζεται από τον τύπο του έργου, την πολυπλοκότητα του domain και το μοντέλο εκτέλεσης.

13. Glossary of Architecture Terms

13.1 General Architecture Terms

Architecture Style (Αρχιτεκτονικό Στυλ):

Η γενική διάταξη ενός συστήματος, με βάση πρότυπα οργάνωσης κώδικα και επικοινωνίας.

Layered Architecture:

Αρχιτεκτονικό μοντέλο με διακριτά επίπεδα (UI, Application, Domain, Infrastructure).

Pipeline Architecture:

Μοντέλο όπου τα δεδομένα ρέουν σειριακά μέσα από διαδοχικά στάδια επεξεργασίας.

Event-Driven Architecture:

Σύστημα όπου οι λειτουργίες ενεργοποιούνται από γεγονότα αντί για synchronous κλήσεις.

Domain-Driven Design (DDD):

Προσέγγιση που τοποθετεί τη μοντελοποίηση του domain στο επίκεντρο της αρχιτεκτονικής.

13.2 UML Terms

Controller:

Τμήμα της εφαρμογής που λαμβάνει αιτήματα και τα δρομολογεί στα services.

Service Layer:

Επιχειρησιακή λογική που εκτελεί use cases.

Repository:

Αφηρημένη διεπαφή για πρόσβαση σε αποθήκευση δεδομένων.

Component Diagram:

UML διάγραμμα που απεικονίζει φυσικά ή λογικά υποσυστήματα.

Class Diagram:

Απεικόνιση αντικειμένων (κλάσεων), των σχέσεων τους και των ιδιοτήτων τους.

Sequence Diagram:

Απεικόνιση της χρονικής σειράς μεθόδων/αλληλεπιδράσεων.

Context Map:

Διάγραμμα που δείχνει σχέσεις μεταξύ bounded contexts σε DDD.

State Machine Diagram:

Απεικόνιση καταστάσεων και μεταβάσεων ενός συστήματος.

13.3 DDD Terms

Entity:

Αντικείμενο με ταυτότητα που διαρκεί στο χρόνο ανεξάρτητα από τις τιμές του.

Value Object:

Αμετάβλητο αντικείμενο που εκφράζει μία έννοια χωρίς ταυτότητα (π.χ. Money, Address).

Aggregate / Aggregate Root:

Ομαδοποίηση Entities & Value Objects που πρέπει να είναι συνεπή (consistency boundary).

Domain Service:

Λογική domain που δεν ανήκει φυσικά σε Entity ή Value Object.

Bounded Context:

Αυτόνομη εννοιολογική περιοχή του domain με δικούς της κανόνες και μοντέλα.

13.4 Event-Driven & Pipeline Terminology

Stage / Step:

Αυτόνομο τμήμα επεξεργασίας δεδομένων σε pipeline.

Data Flow:

Το μονοπάτι από το οποίο περνά ένα σύνολο δεδομένων σε pipeline.

Activity Diagram:

UML διάγραμμα που αναπαριστά βήματα ροής εργασίας.

Event:

Γεγονός που συμβαίνει και πυροδοτεί αντίδραση.

Event Handler:

Κώδικας που εκτελείται ως απάντηση σε event.

State Machine:

Μοντέλο που περιγράφει όλες τις δυνατές καταστάσεις ενός συστήματος και τις μεταβάσεις τους.

15. Bibliography / References

– Books(Βιβλία)

1. Bass, L., Clements, P., & Kazman, R.

*Software Architecture in Practice (**4th edition**).*

Addison-Wesley, 2021.

2. Evans, E.

Domain-Driven Design: Tackling Complexity in the Heart of Software.

Addison-Wesley, 2003.

3. Fowler, M.

Patterns of Enterprise Application Architecture.

Addison-Wesley, 2002.

4. Richards, M., & Ford, N.

Fundamentals of Software Architecture.

O'Reilly Media, 2020.

5. Newman, S.

Building Microservices (2nd edition).

O'Reilly Media, 2021.

6. Hohpe, G., & Woolf, B.

Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.

Addison-Wesley, 2004.

7. Gamma, E., Helm, R., Johnson, R., & Vlissides, J.

Design Patterns: Elements of Reusable Object-Oriented Software.

Addison-Wesley, 1994.

8. Vernon, V.

Implementing Domain-Driven Design.

Addison-Wesley, 2013.

9. Larman, C.

Applying UML and Patterns (3rd edition).

Prentice Hall, 2004.

10. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M.
Pattern-Oriented Software Architecture, Volume 1: A System of Patterns.
Wiley, 1996.

– Standards(Επίσημα Πρότυπα)

11. Object Management Group (OMG).
Unified Modeling Language (UML) Specification.
Version 2.5.1, 2017.
12. IEEE Standard 42010.
Systems and Software Engineering — Architecture Description.
IEEE, 2011.
13. ISO/IEC/IEEE 24765.
Systems and Software Engineering Vocabulary.
ISO, 2017.

– Official Documentation

14. PlantUML Official Documentation.
https://plantuml.com
15. UML Resource Page – OMG.
https://www.omg.org/spec/UML/
16. Arc42 Architecture Documentation Template.
https://arc42.org
17. C4 Model (Simon Brown).
https://c4model.com

– Research Papers(Επιστημονικά Άρθρα)

18. Kazman, R., Klein, M., & Clements, P.
ATAM: Method for Architecture Evaluation.
Carnegie Mellon University, SEI, 2000.

19. Taylor, R., Medvidovic, N., & Dashofy, E.
Software Architecture: Foundations, Theory, and Practice.
Wiley, 2009.
20. Nord, R. L., & Tomayko, J. E.
Software Architecture Lifecycle Practices.
Carnegie Mellon University, 2006.

– Authoritative Sources(Εγκυρες Online Πηγές)

21. Martin Fowler — Articles on Architecture & DDD
https://martinfowler.com/architecture/
22. Vaughn Vernon — DDD Resources
https://vaughnvernon.co/
23. Microsoft Architecture Center
https://learn.microsoft.com/en-us/azure/architecture/
24. ThoughtWorks Technology Radar
https://www.thoughtworks.com/radar
25. AWS Well-Architected Framework
https://aws.amazon.com/architecture/well-architected/

– Secondary Sources(Προαιρετικές Πηγές)

26. "Clean Architecture" — Robert C. Martin
https://www.cleancoder.com
27. Netflix Tech Blog — Architecture Posts
https://netflixtechblog.com/
28. Google Cloud Architecture Framework
https://cloud.google.com/architecture/framework