ДОДАТОК А

Слайди презентації

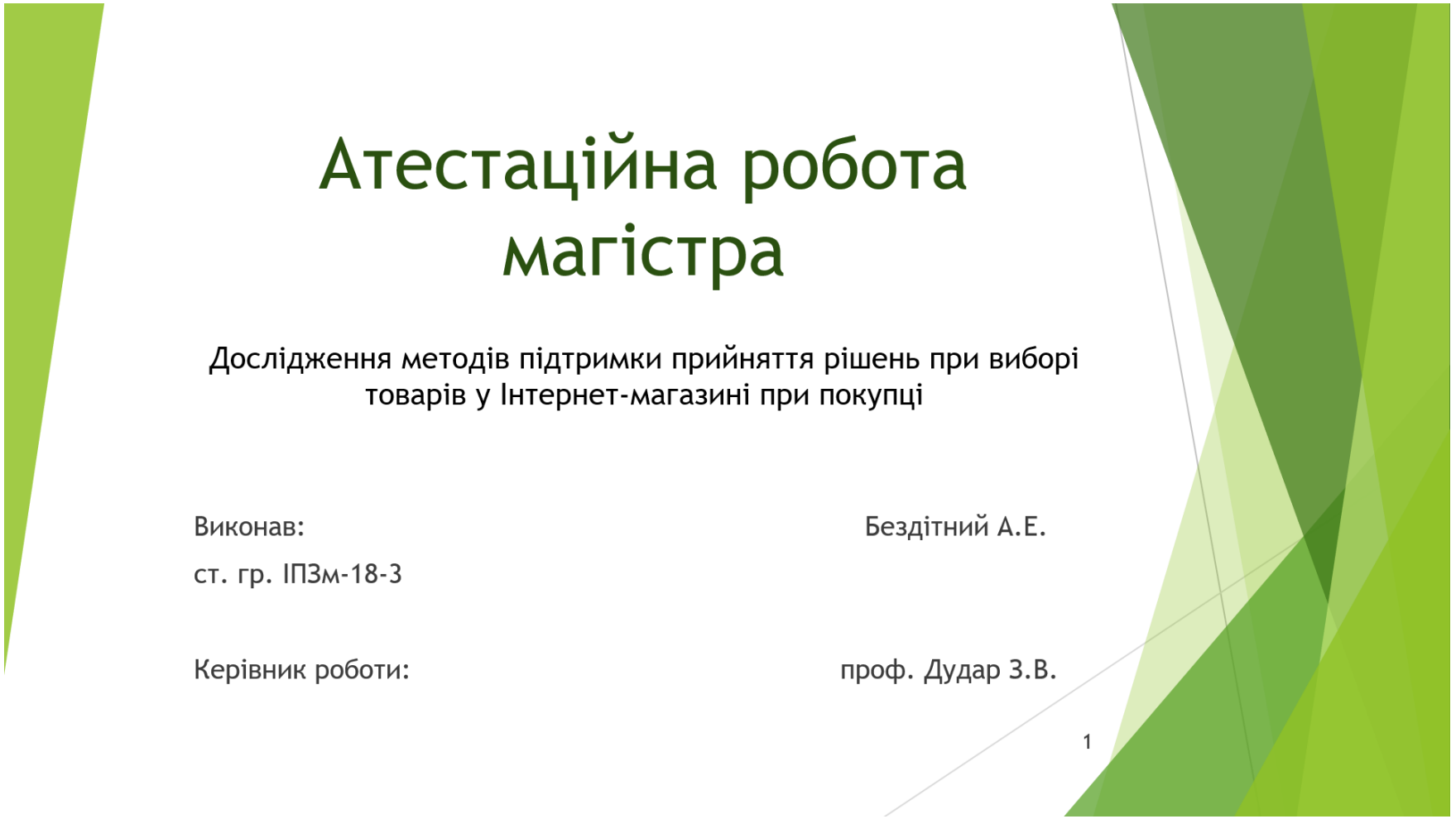


Рисунок А.1 – Титульний слайд

Рисунок А.2 – Слайд «Аналіз існуючих рішень»

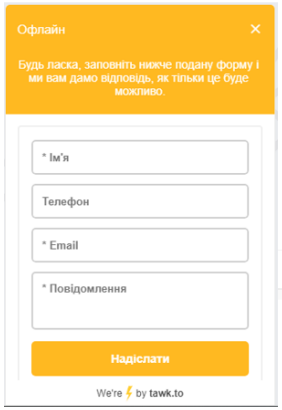
Аналіз існуючих рішень

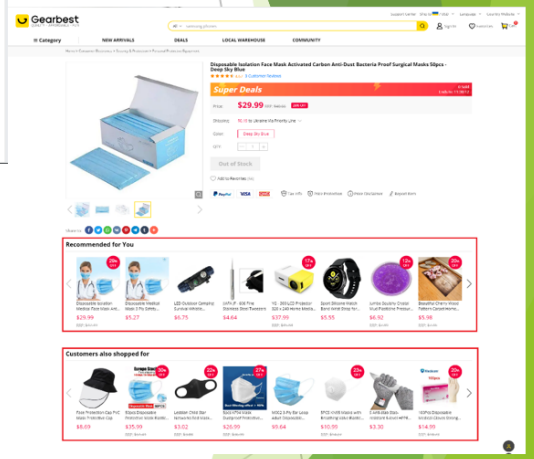
Переваги:

- зручний спосіб відображення інформації;
- можливість втручання консультанта;
- низька ресурсоємкість.

Недоліки:

- не враховуються атрибути товарів;
- відсутність можливості врахування суб’єктивності вибору продуктів;
- жорстка прив’язка до конкретного Інтернет-магазину.





2

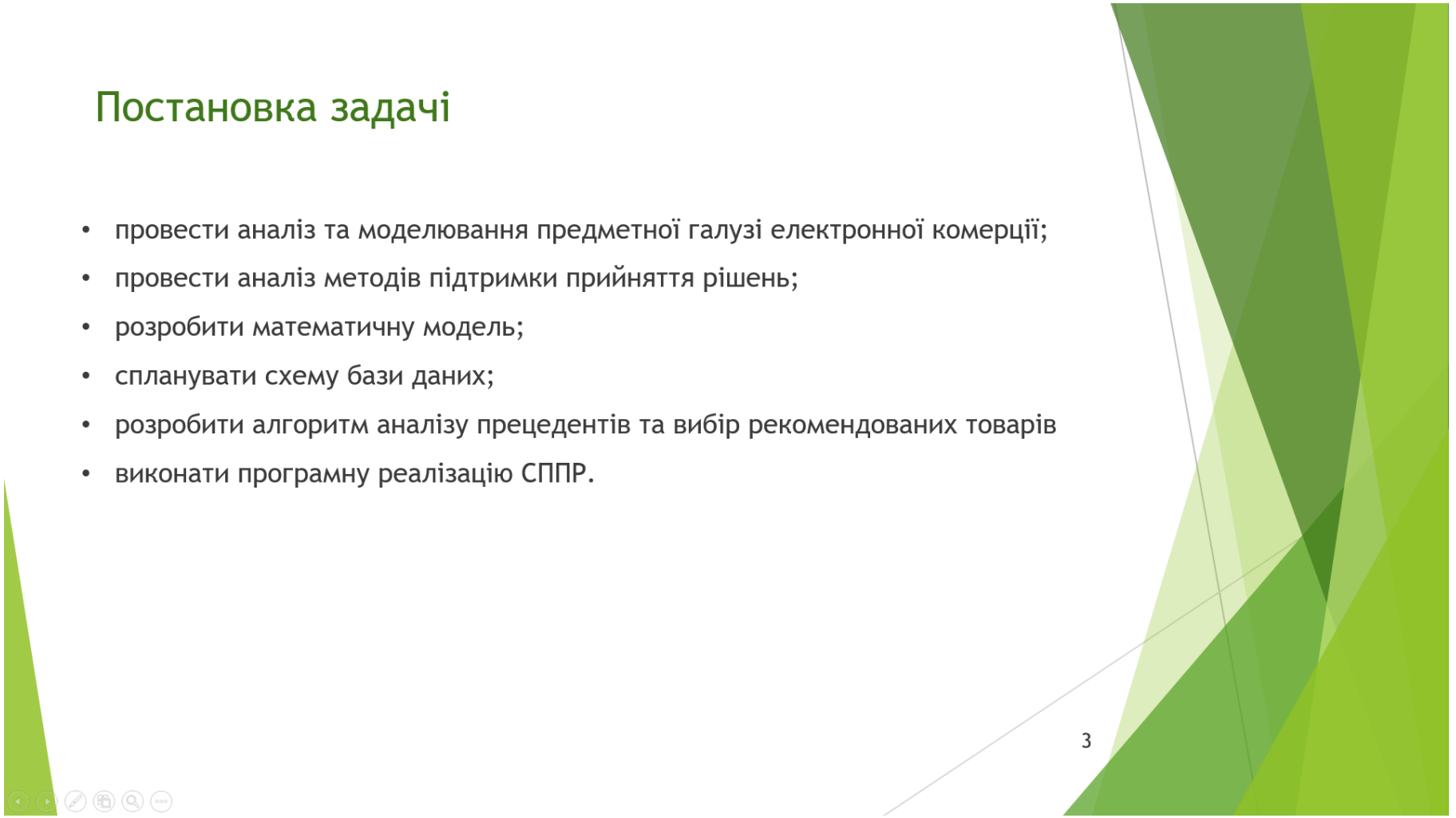


Рисунок А.3 – Слайд «Постановка задачі»

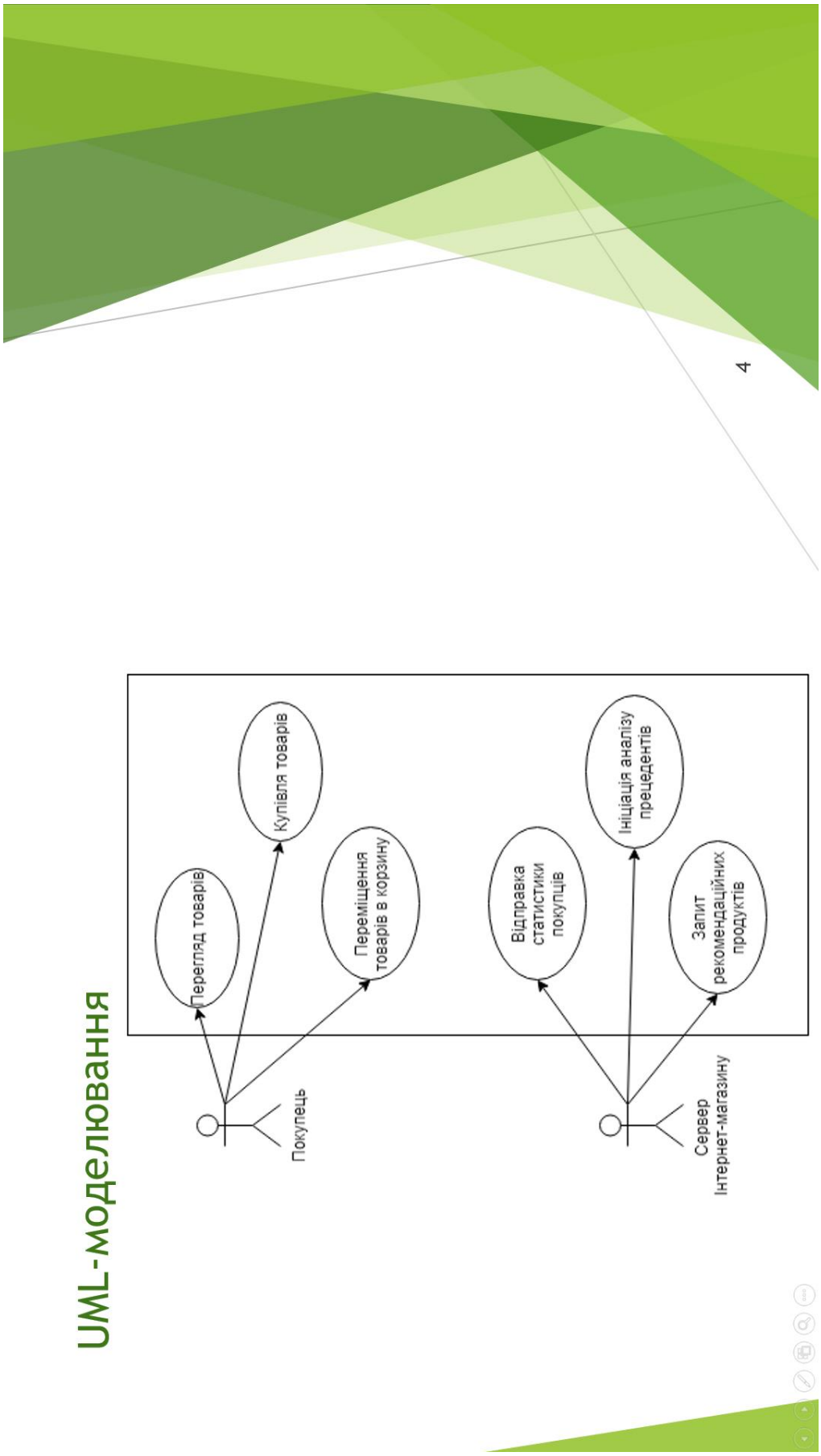


Рисунок А.4 – Слайд «UML-моделювання»

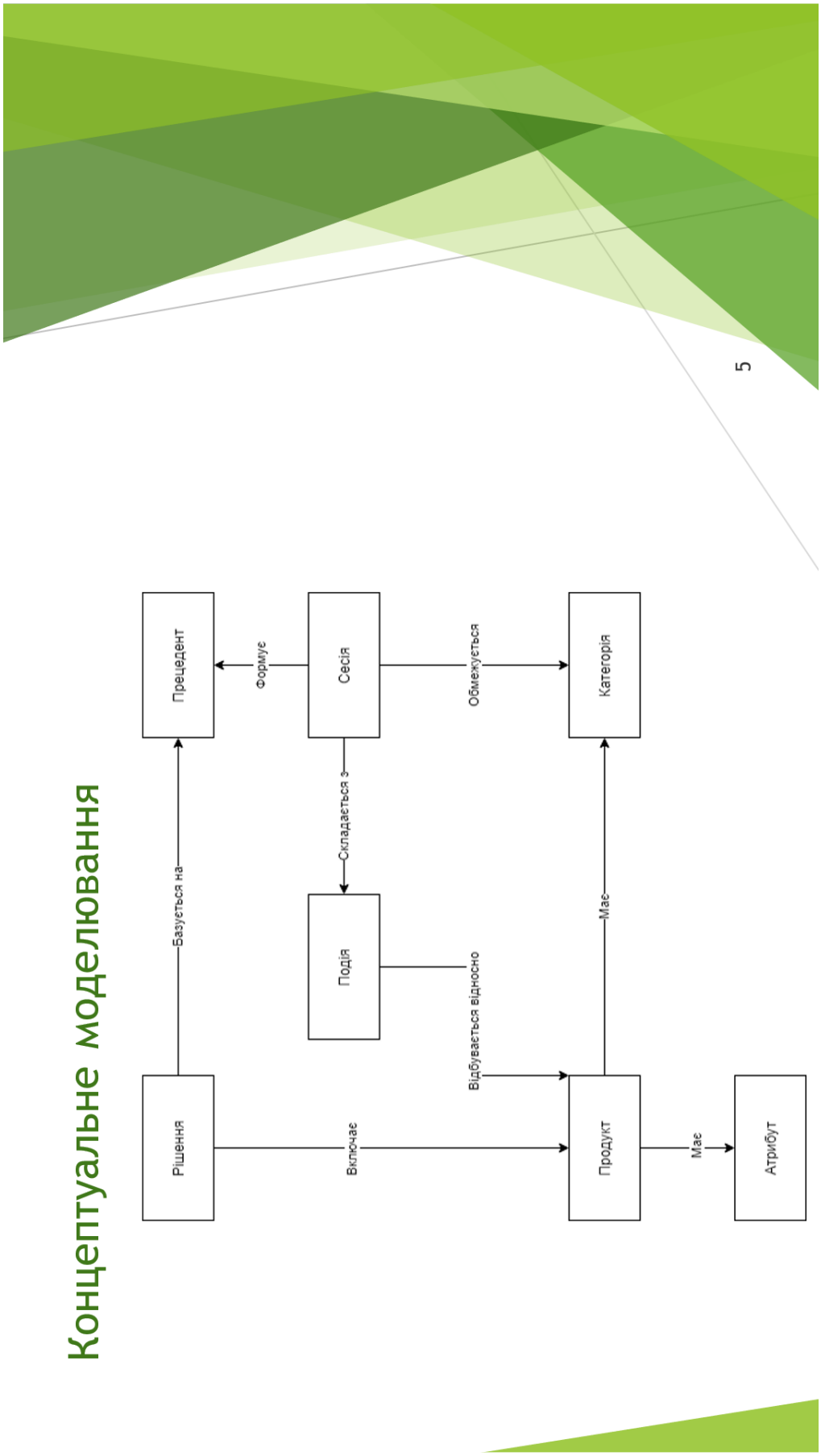


Рисунок А.5 – Слайд «Концептуальне моделювання»

Математична модель

$$X = < P_v, Y, D >,$$

де P_v - нещодавно переглянуті продукти покупцем, який потребує підтримки в прийнятті рішень

D - рішення прийняте системою (рекомендовані продукти),

Y - набір попередньо проаналізованих прецедентів на базі яких приймається рішення

$$Y =< S, A, C >$$

де S - сесія, в рамках якої користувач виконував певні дії

A - множина атрибутів товарів (критерії)

C - категорія товарів

Рисунок А.6 – Слайд «Математична модель»

Математична модель

$$\mathrm{A} = \mathrm{A}(t) = \{a_i\}_{i=1}^{n}$$

де A - множина атрибутів товару, котрі можуть змінюватися з часом

n - кількість атрибутів

$$S = S(t) = < E_v, E_c, V >$$

$$E_v = E_v(t), E_c = E_c(t).$$

$$E_v = \{e_{v_i}(t)\}_{i=1}^{n}, \text{ та } E_c = \{e_{c_i}(t)\}_{i=1}^{m},$$

де E_v - подія перегляду товару на сайті,

E_v - подія переміщення товару до кошика,

V - відвідувач магазину

Рисунок А.7 – Слайд «Математична модель»

Рисунок А.8 – Слайд «Математична модель»

Математична модель

$$D = < P_r, L >,$$

де D - рішення прийняте системою,

P_r - перелік рекомендації для покупки

L - коефіцієнт схожості з агрегованим набором критеріїв прецедентів

$$P_r = \{p_{r_i}\}_{i=1}^{n}$$

де p_{r_i} - рекомендований системою товар

n - кількість рекомендацій

8

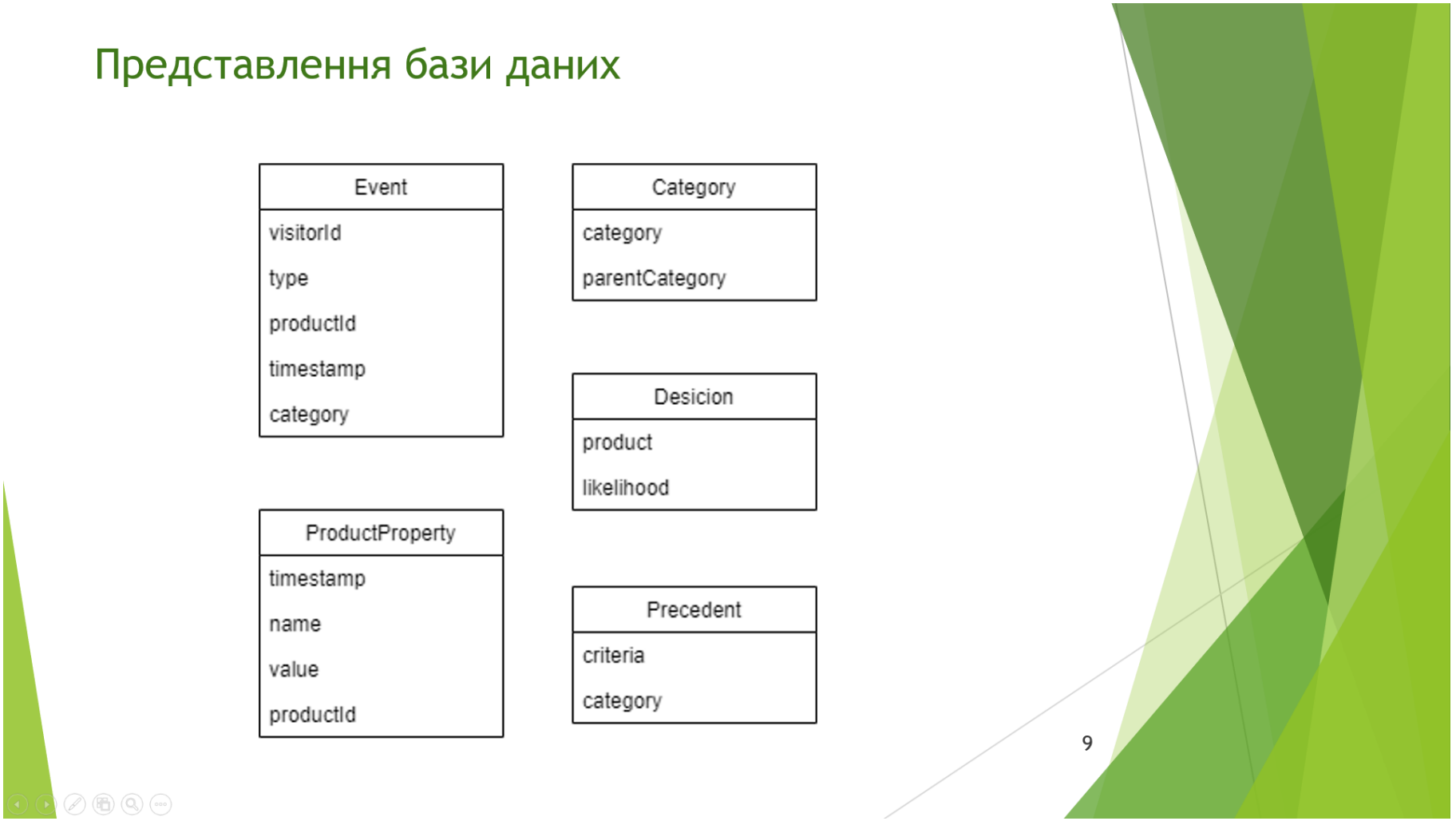


Рисунок А.9 – Слайд «Представлення бази даних»

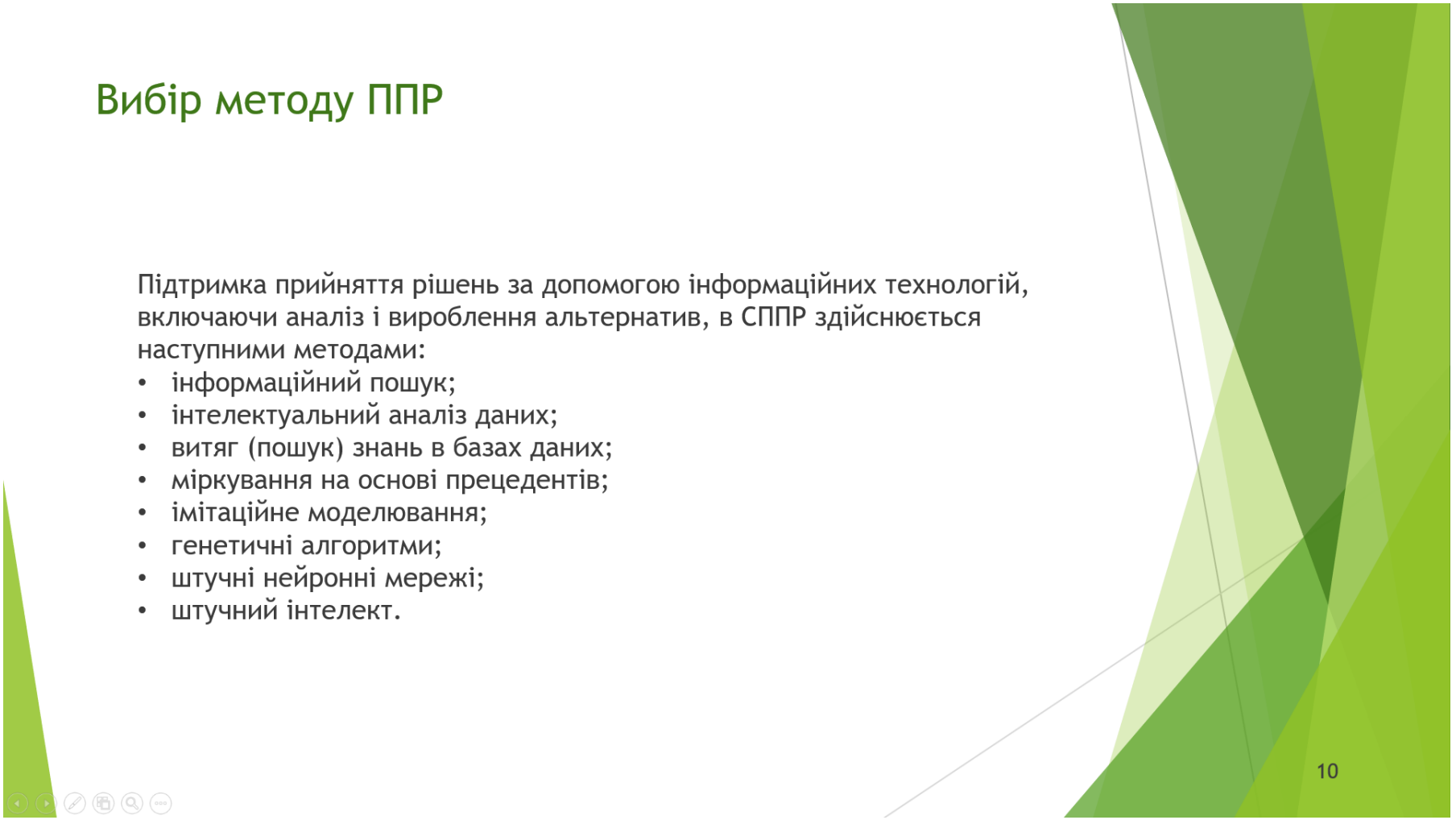
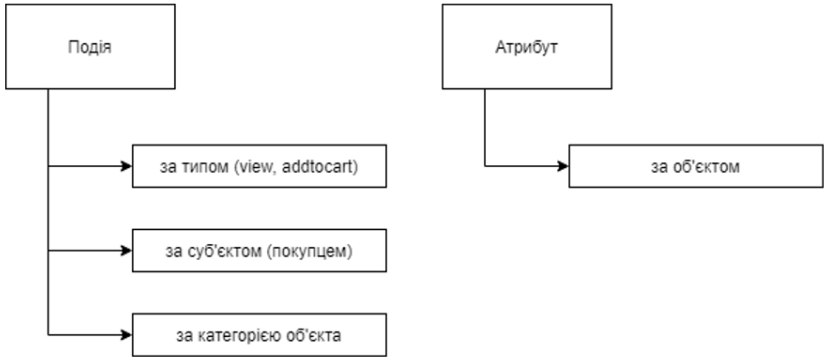


Рисунок А.10 – Слайд «Вибір методу ППР»

Рисунок А.11 – Слайд «Попередня обробка даних»

Попередня обробка даних

- Класифікація подій та атрибутів;
- Кластеризація для визначення сесій;
- Формування прецедентів



11

Рисунок А.12 – Слайд «Проблема нечіткості критеріїв»

Проблема нечіткості критеріїв

Вважатимемо відомою множину аналогів, тобто товарів, які підлягають аналізу $S = \{s_1, s_2, \dots, s_n\}$, тоді $C = \{c_1, c_2, \dots, c_m\}$ - множина кількісних та якісних критеріїв, за участі яких здійснюється оцінка альтернатив

Нехай $\mu^j(s_i)$ - число в діапазоні [0, 1], котре характеризує рівень оцінки варіанта $s_i \in S$ по критерію $c_j \in C$: чим більше число $\mu^j(x_i)$, тим вище оцінка варіанта по критерію $c_j \in C, i = \overline{1,n},\ j = \overline{1,m}$. Тоді критерій $c_j \in C$ можна представити у виді нечіткої множини $\widetilde{c_j}$, котра задана на універсальній множині таким чином:

$$\widetilde{c_j} = \{\frac{\mu^j(s_1)}{s_1}, \frac{\mu^j(s_2)}{s_2}, \dots, \frac{\mu^j(s_n)}{s_n}\}$$

де $\mu^j(s_i)$ - ступінь належності елемента s_i до нечіткої множини $\widetilde{c_j}$.

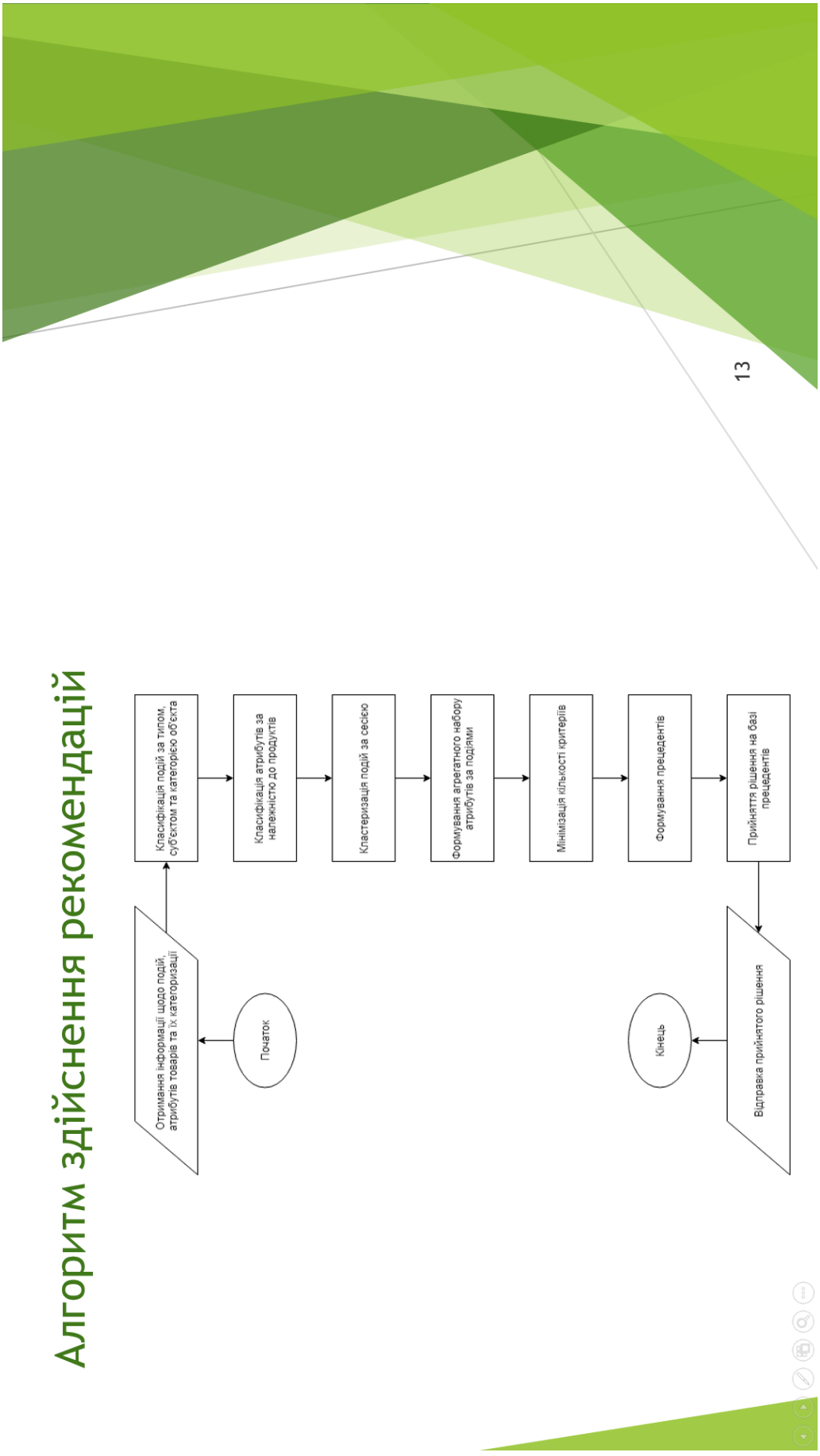


Рисунок А.13 – Слайд «Алгоритм здійснення рекомендацій»

API інтерфейс

- POST /data/event
- POST /data/product-property
- POST /data/category
- POST /analyze
- POST /recommend

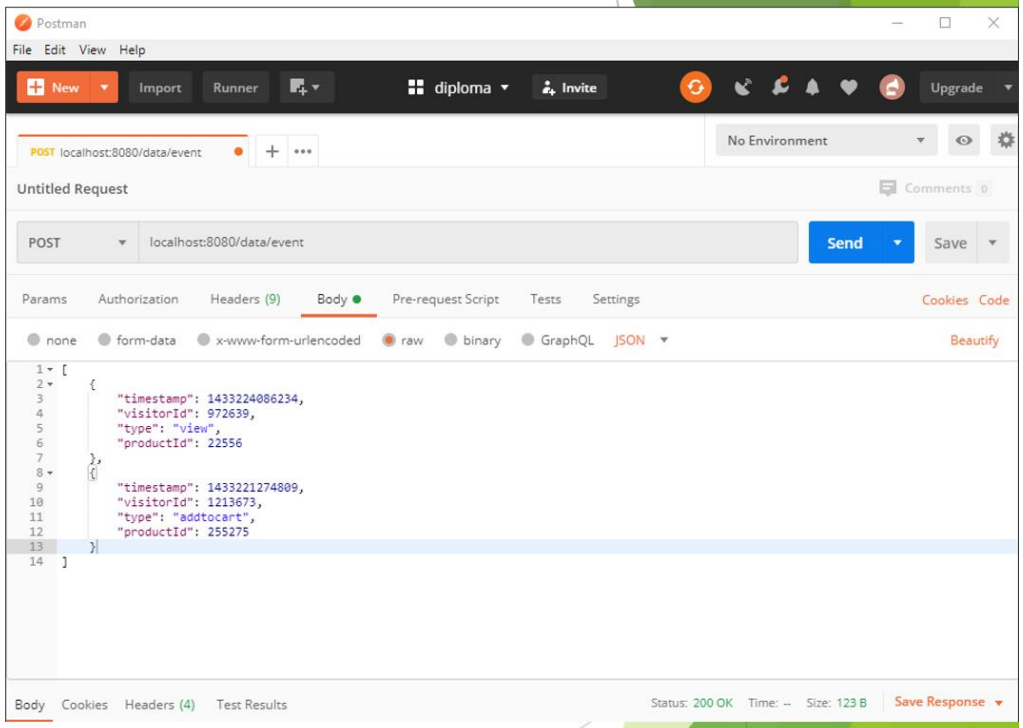


Рисунок А.14 – Слайд «API інтерфейс»

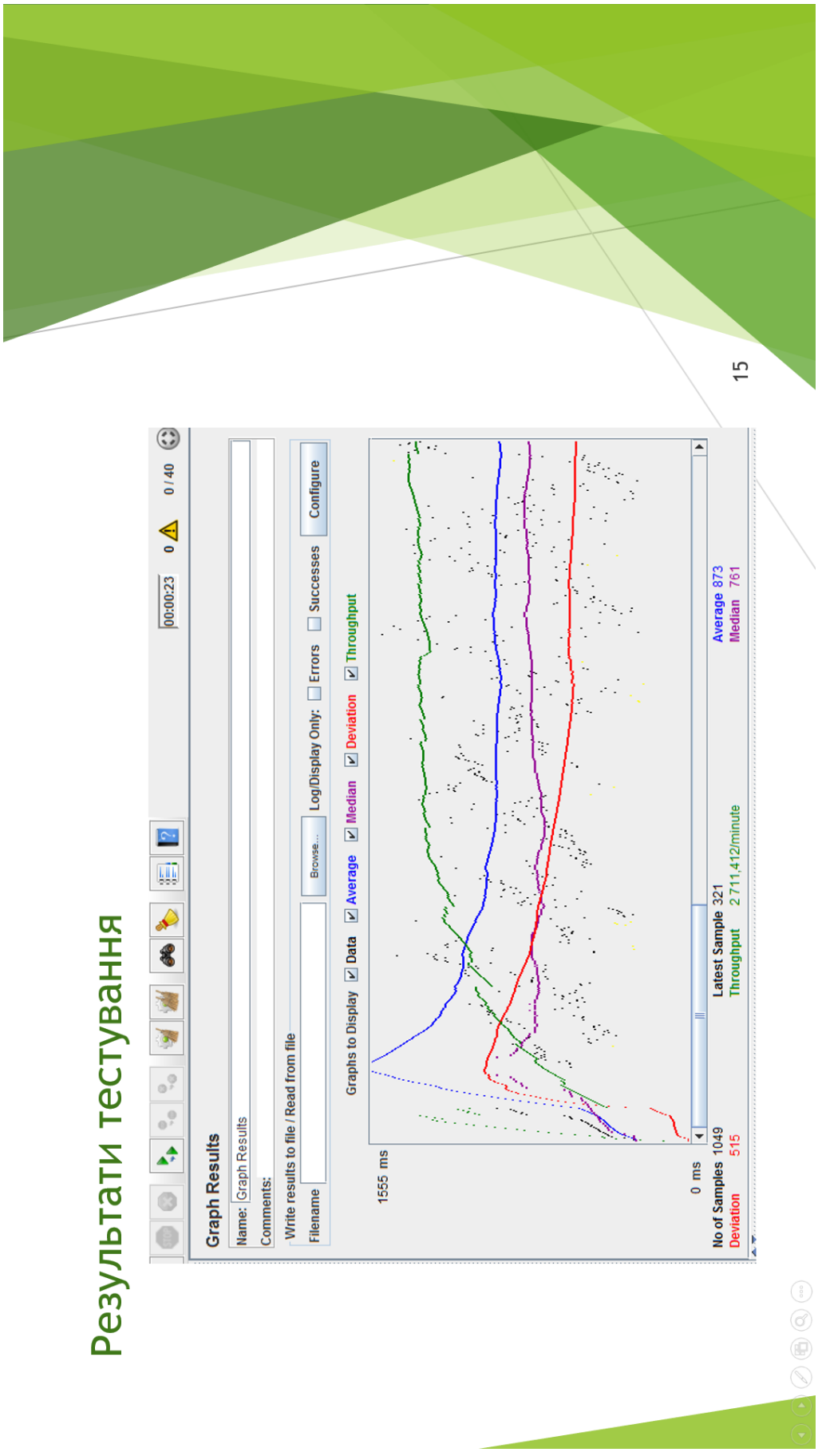


Рисунок А.15 – Слайд «Результати тестування»

Висновки

- Був проведений аналіз та моделювання предметної галузі
- Виконано аналіз методів підтримки прийняття рішень та запропоновано відповідні модифікації враховуючи особливості умов, в яких має працювати система
- Спроектовано математичну модель для представлення рекомендацій в сфері здійснення покупок в електронній комерції
- Розроблено алгоритм аналізу подій та формування прецедентів і подальшого прийняття рішень на їх основі
- Виконана програмна реалізація інформаційної системи

16

Рисунок А.17 – Слайд «Висновки»

ДОДАТОК В

Лістинг програмного коду

```
@RestController
public class DataLoaderResource {

    @Autowired
    private EventRepository eventRepository;
    @Autowired
    private CategoryRepository categoryRepository;
    @Autowired
    private ProductPropertyRepository productPropertyRepository;

    private MongoClient mongoClient = new MongoClient(new
MongoClientURI("mongodb://localhost:27017"));
    private MongoDatabase database = mongoClient.getDatabase("dss");

    @PostMapping("/data/category")
    public void loadCategoriesFromDataset() {
        Path categoriesPath =
Paths.get("src/main/resources/category_tree.csv");
        for (Category category : new CategoryCsvParser(categoriesPath)) {
            categoryRepository.save(category);
        }
    }

    @PostMapping("/data/events)
    public void loadEvents() {
        Path eventsPath = Paths.get("src/main/resources/events.csv");
        for (Event event : new EventCsvParser(eventsPath)) {
            eventRepository.save(event);
        }
    }
public void verifyEventsTimestampOrder() {
        Map<Long, Long> visitorLastTimestamp = new HashMap<>();
```

```
        Path eventsPath =
Paths.get("src/main/resources/sorted_events.csv");
        for (Event event : new EventCsvParser(eventsPath)) {
            Long previousTimestamp =
visitorLastTimestamp.put(event.getVisitorId(), event.getTimestamp());
            if (previousTimestamp != null && previousTimestamp >
event.getTimestamp()) {
                throw new RuntimeException("that's a problem");
            }
        }
    }

 public DBSCANClusterer(final double eps, final int minPts)
        throws NotPositiveException {
        this(eps, minPts, new EuclideanDistance());
    }

    /**
     * Creates a new instance of a DBSCANClusterer.
     *
     * @param eps maximum radius of the neighborhood to be considered
     * @param minPts minimum number of points needed for a cluster
     * @param measure the distance measure to use
     * @throws NotPositiveException if {@code eps < 0.0} or {@code minPts
< 0}
     */
    public DBSCANClusterer(final double eps, final int minPts, final
DistanceMeasure measure)
        throws NotPositiveException {
        super(measure);

        if (eps < 0.0d) {
            throw new NotPositiveException(eps);
        }
        if (minPts < 0) {
            throw new NotPositiveException(minPts);
        }
        this.eps = eps;
```

```
        this.minPts = minPts;
    }

    /**
     * Returns the maximum radius of the neighborhood to be considered.
     * @return maximum radius of the neighborhood
     */
    public double getEps() {
        return eps;
    }

    /**
     * Returns the minimum number of points needed for a cluster.
     * @return minimum number of points needed for a cluster
     */
    public int getMinPts() {
        return minPts;
    }

    /**
     * Performs DBSCAN cluster analysis.
     *
     * @param points the points to cluster
     * @return the list of clusters
     * @throws NullArgumentException if the data points are null
     */
    @Override
    public List<Cluster<T>> cluster(final Collection<T> points) throws
NullArgumentException {

        // sanity checks
        MathUtils.checkNotNull(points);

        final List<Cluster<T>> clusters = new ArrayList<Cluster<T>>();
        final Map<Clusterable, PointStatus> visited = new
HashMap<Clusterable, PointStatus>();

        for (final T point : points) {
```

```
            if (visited.get(point) != null) {
                continue;
            }
            final List<T> neighbors = getNeighbors(point, points);
            if (neighbors.size() >= minPts) {
                // DBSCAN does not care about center points
                final Cluster<T> cluster = new Cluster<T>();
                clusters.add(expandCluster(cluster, point, neighbors,
points, visited));
            } else {
                visited.put(point, PointStatus.NOISE);
            }
        }

        return clusters;
    }

    /**
     * Expands the cluster to include density-reachable items.
     *
     * @param cluster Cluster to expand
     * @param point Point to add to cluster
     * @param neighbors List of neighbors
     * @param points the data set
     * @param visited the set of already visited points
     * @return the expanded cluster
     */
    private Cluster<T> expandCluster(final Cluster<T> cluster,
                                     final T point,
                                     final List<T> neighbors,
                                     final Collection<T> points,
                                     final Map<Clusterable, PointStatus>
visited) {
        cluster.addPoint(point);
        visited.put(point, PointStatus.PART_OF_CLUSTER);

        List<T> seeds = new ArrayList<T>(neighbors);
        int index = 0;
```

```
            while (index < seeds.size()) {
                final T current = seeds.get(index);
                PointStatus pStatus = visited.get(current);
                // only check non-visited points
                if (pStatus == null) {
                    final List<T> currentNeighbors = getNeighbors(current,
points);
                    if (currentNeighbors.size() >= minPts) {
                        seeds = merge(seeds, currentNeighbors);
                    }
                }

                if (pStatus != PointStatus.PART_OF_CLUSTER) {
                    visited.put(current, PointStatus.PART_OF_CLUSTER);
                    cluster.addPoint(current);
                }

                index++;
            }
            return cluster;
        }

        /**
         * Returns a list of density-reachable neighbors of a {@code point}.
         *
         * @param point the point to look for
         * @param points possible neighbors
         * @return the List of neighbors
         */
        private List<T> getNeighbors(final T point, final Collection<T>
points) {
            final List<T> neighbors = new ArrayList<T>();
            for (final T neighbor : points) {
                if (point != neighbor && distance(neighbor, point) <= eps) {
                    neighbors.add(neighbor);
                }
            }
            return neighbors;
```

```
    }

    /**
     * Merges two lists together.
     *
     * @param one first list
     * @param two second list
     * @return merged lists
     */
    private List<T> merge(final List<T> one, final List<T> two) {
        final Set<T> oneSet = new HashSet<T>(one);
        for (T item : two) {
            if (!oneSet.contains(item)) {
                one.add(item);
            }
        }
        return one;
    }

@SpringBootApplication(exclude = {
        DataSourceAutoConfiguration.class,
        ApplicationAvailabilityAutoConfiguration.class,
        SpringApplicationAdminJmxAutoConfiguration.class
})
public class DssApplication {

    public static void main(String[] args) {
        SpringApplication.run(DssApplication.class, args);
    }
}
```