1. Програмна інженерія. Визначення. Історія.

Инженерия ПЗ — это системный подход к анализу, проектированию, оценке, реализации, тестированию, обслуживанию и модернизации программного обеспечения, то есть применение инженерии к разработке программного обеспечения.

Як відомо, перші комп'ютери, що працювали під управлінням Програм**,** які зберігаються в пам'яті, з'явилися у 40 - 50-х рр. XX ст. Разом з ними постало нове завдання, суть якого полягає у створенні програм, і процес, спрямований на її вирішення - програмування**.** Тому подальший розвиток обчислювальної техніки пов'язаний не тільки з удосконаленням комп'ютерів і їх розповсюдженням, але й з розвитком програмування.

Майже одночасно з появою комп'ютерів відбулося розділення розробників програм на два типи - прикладних і системних програмістів.

До першого типу (прикладні програмісти) увійшли фахівці з прикладних галузей (доменів) - математики, фізики, економіки, освіти**;** технологій. Вони писали програми мовами високого рівня *(Cobol, Fortran)* для вирішення тих завдань, що виникають у галузях. Їх діяльність називалася прикладним програмуванням.

До другого типу (системні програмісти) увійшли фахівці, від яких не вимагалося знань доменів, оскільки вони займалися автоматизацією процесів розробки програм. Системні програмісти за звичай писали програми в машинному коді або мовою АСЕМБЛЕР. Їх діяльність називалася системним програмуванням. Сукупність прикладних і системних програм називається програмним забезпеченням.

У 60 - 70-х рр. XX ст. були створені високопродуктивні обчислювальні машини (швидкістю близько 1 млн., опер./с БЕСМ-6 в СРСР і UNIVAC в США). З їх появою виникла можливість вирішення великих і складних завдань. Це, своєю чергою, потребувало розробки великих програм (від 100 тис. до 1 мли. рядків). Великі програми спричинили проблеми, пов'язані з їх створенням і вико рисканням.

Із збільшенням продуктивності, кількості обчислювальних машин і розширенням сфери їх застосування, з'явилися програми двох типів. Програми першого типу створювалися і продавалися разом з машинами (транслятори, операційні системи, бібліотеки підпрограм). Програми другого типу створювалися на замовлення і призначалися для вирішення завдань з різних предметних галузей. Таким чином, з'явився замовник - організація, яка ставила завдання, призначала терміни, виділяла бюджет і оплачувала роботу. У зв'язку з цим дуже швидко з'явилось завдання - супровід Про грами і проблема - непорозуміння між розробником і замовником.

У 60-х рр. XX ст. унаслідок розповсюдження застосування комп'ютерів, зросла роль і утвердилась важливість програмного забезпечення. Цьому сприяла поява значної кількості проектів програмного забезпечення, що характеризувалися такими аспектами:

- наявністю замовника або ринкової ніші;
- великими розмірами і витратами;
- жорсткими вимогами до процесів реалізації і результатів;
- мілітаризацією.

Контекст, у якому розроблялося і використовувалося програмне забезпечення, сприяв особливому стану програмного забезпечення в обчислювальній техніці і характеризувався такими чинниками:

- розроблялося дуже велике за обсягом програмне забезпечення, характерним представником якою була тоді операційна система OS360 для ЕОМ серії IBM. Це програмне забезпечення містило більше 500 тис. операторів і розроблялося значним, для того часу, колективом розробників (близько 1000). Досвід цього проекту був узагальнений і став відомий;

- програмне забезпечення вирішувало настільки серйозні зав дання, що виникла проблема з його супроводом, суть якого зводилася не тільки до виправлення помилок, допущених при розробці, але й до модифікації програмного забезпечення у зв'язку із зміною вимог замовника або середовища, у якому експлуатувалося програмне забезпечення, або бажанням розробника продовжити експлуатацію, шляхом випуску вдосконалених версій;

- часті зриви термінів розробки і перевищення бюджету потре бували не тільки нового підходу до організації процесу розробки, але і нових методів і засобів, що забезпечують обґрунтований розрахунок параметрів проекту, які характеризували фінансування, терміни, об'єми програмного забезпечення, кількісний і якісний склад колективу розробників. Існуюча і широко використовувана одиниця вимірювання - «людино-місяць» не працювала на таких масштабних проектах;

- досвід розробки програмного забезпечення, який нагромаджений за цей період, показав, що все рідше розроблялися принципово нові проекти. Лише 15% усіх проектів потребували розробки «з нуля». Нині 85% належать до проектів, що повторюються. Тому актуальним стає використання досвіду, нагромадженого в програмному забезпеченні До того ж, поступово стало зрозуміло, що має вирішуватися завдання використання досвіду не лише безпосередньо програмування, у вигляді частин програм, але і досвіду результатів виконання інших процесів, наприклад, проектування. Для вирішення цього завдання в 1984 р. були широко розгорнені роботи з дослідження програмного забезпечення в аспекті повторного використання *(reuse);*

- техніка програмування і процеси, що були ефективні в 50-х і ранніх 60-х рр. XX ст. («програмування в малому») для розробки невеликих програм малими колективами, стали неефективними при розробці великого за обсягом, складного програмного забезпечення, що складається з мільйонів рядків коду, та вимагає декількох років роботи сотень фахівців різних спеціальностей. Були потрібні нові технології, що почали вважати «програмуванням у великому».

Почали з'являтися нові процеси, що потребували певної організації (табл.1.1). Таким чином, склалася ситуація, яка призвела до кризи в програмному забезпеченні і необхідності пошуку шляхів виходу з неї, так званої «срібної кулі». Виходом з цієї ситуації стало обговорення на конференції NATO в 1968 р. нової дисципліни, яку назвали «інженерія програмного забезпечення» *(software engineering). Таблиця 1.1*

Періоди розвитку Об'єкти порівняння	1960±5 років програмування *"any-wich-way"*	1970±5 років «програмування в малому»	1980±5 ро-ків «програму-вання у ве-ликому»
Об'єкти	Маленькі програми	Алгоритми і програми	Системна структура
Дані	Неструктурована інформація	Структури даних і типи	Бази даних
Управління	Елементарне ро-зуміння діаграм управління	Програми виконуються і закін-чуються	Програми, що безпере-рвно вико-нуються
Простір етанів	Стан, що погано розуміється окре-мо від управління	Маленькі, прості	Великі, структуризовані
Організаційне управління	Немає	Індивідуальні зусилля	Колективні зусилля, супровід
Інструмент»	Асемблери	Компілятори, редактори, заван-тажувачі	Середовища, інтегровані інстру-менти

2. Які умови мають бути виконані для ефективного застосування Flyweight. Які фактори дозволяють знизити вимоги до пам'яті при використанні Flyweight.

Шаблон Легковаговик (Flyweight) - козак

Призначення

Використовується для ефективної підтримки (в першу чергу для зменшення затрат пам'яті) великої кількості дрібних об'єктів.

Опис

Шаблон Легковаговик (Flyweight) використовує загальнодоступний легкий об'єкт (flyweight, легковаговик), який одночасно може використовуватися у великій кількості контекстів. Стан цього об'єкта поділяється на внутрішній, що містить інформацію, незалежну від контексту, і зовнішній, який залежить або змінюється разом з контекстом легковаговика. Об'єкти клієнтів відповідають за передачу зовнішнього стану легковаговика, коли йому це необхідно.

Переваги

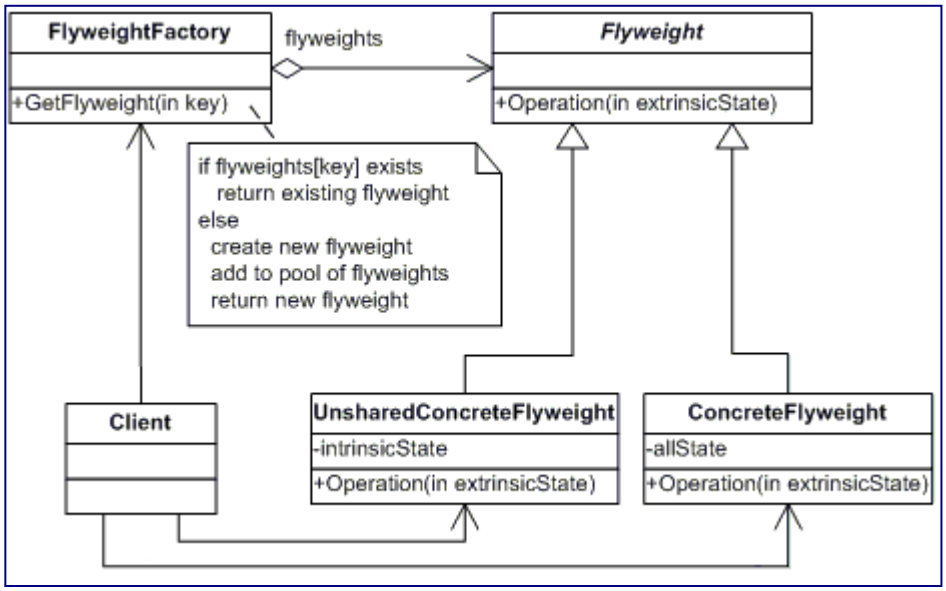
- Зменшує кількість об'єктів, що підлягають обробці.
- Зменшує вимоги до пам'яті.

Застосування

Шаблон Легковаговик можна використовувати коли:

- В програмі використовується велика кількість об'єктів.
- Затрати на збереження високі через велику кількість об'єктів.
- Більшість станів об'єктів можна зробити зовнішніми.
- Велика кількість груп об'єктів може бути замінена відносно малою кількістю загальнодоступних об'єктів, однократно видаливши зовнішній стан.
- Програма не залежить від ідентичності об'єктів. Оскільки об'єкти-легковаговики можуть використовуватися колективно, то тести на ідентичність будуть повертати значення "істина" ("true") для концептуально різних об'єктів.

Діаграма UML



Приклад реалізації

```
import java.util.*;

public enum FontEffect {
    BOLD, ITALIC, SUPERSCRIPT, SUBSCRIPT, STRIKETHROUGH
}

public final class FontData {
    /**
     * A weak hash map will drop unused references to FontData.
     * Values have to be wrapped in WeakReferences,
     * because value objects in weak hash map are held by strong references.
     */
    private static final WeakHashMap<FontData, WeakReference<FontData>> flyweightData =
        new WeakHashMap<FontData, WeakReference<FontData>>();
    private final int pointSize;
    private final String fontFace;
    private final Color color;
    private final Set<FontEffect> effects;

    private FontData(int pointSize, String fontFace, Color color, EnumSet<FontEffect> effects)
{
        this.pointSize = pointSize;
        this.fontFace = fontFace;
        this.color = color;
        this.effects = Collections.unmodifiableSet(effects);
    }

    public static FontData create(int pointSize, String fontFace, Color color,
        FontEffect... effects) {
        EnumSet<FontEffect> effectsSet = EnumSet.noneOf(FontEffect.class);
        effectsSet.addAll(Arrays.asList(effects));
        // We are unconcerned with object creation cost, we are reducing overall memory
consumption
        FontData data = new FontData(pointSize, fontFace, color, effectsSet);
        if (!flyweightData.containsKey(data)) {
            flyweightData.put(data, new WeakReference<FontData> (data));
        }
        // return the single immutable copy with the given values
        return flyweightData.get(data).get();
    }
```

```
    @Override
    public boolean equals(Object obj) {
        if (obj instanceof FontData) {
            if (obj == this) {
                return true;
            }
            FontData other = (FontData) obj;
            return other.pointSize == pointSize && other.fontFace.equals(fontFace)
                && other.color.equals(color) && other.effects.equals(effects);
        }
        return false;
    }

    @Override
    public int hashCode() {
        return (pointSize * 37 + effects.hashCode() * 13) * fontFace.hashCode();
    }

    // Getters for the font data, but no setters. FontData is immutable.



}
```

3. Реалізувати шаблон Builder. Забезпечити існування лише одного екземпляра кожного конкретного білдера.

Шаблон Builder

Будівник (англ. *Builder*) — шаблон проектування, відноситься до класу твірних шаблонів.

Призначення

Відокремлює конструювання складного об'єкта від його подання, таким чином у результаті одного й того ж процесу конструювання можуть бути отримані різні подання.

Мотивація

Застосування

Слід використовувати шаблон *Будівник* коли:

- алгоритм створення складного об'єкта не повинен залежати від того, з яких частин складається об'єкт та як вони стикуються поміж собою;
- процес конструювання повинен забезпечити різні подання об'єкта, що конструюється.

Структура

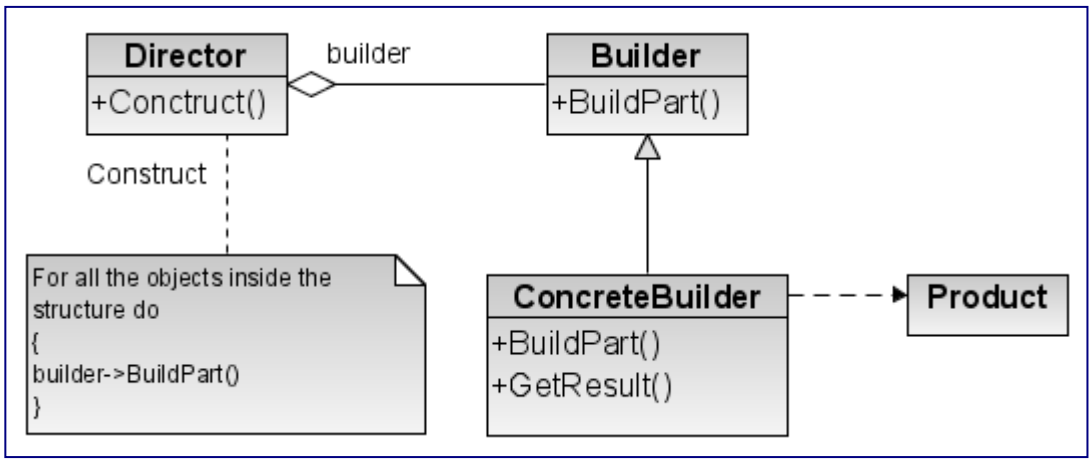
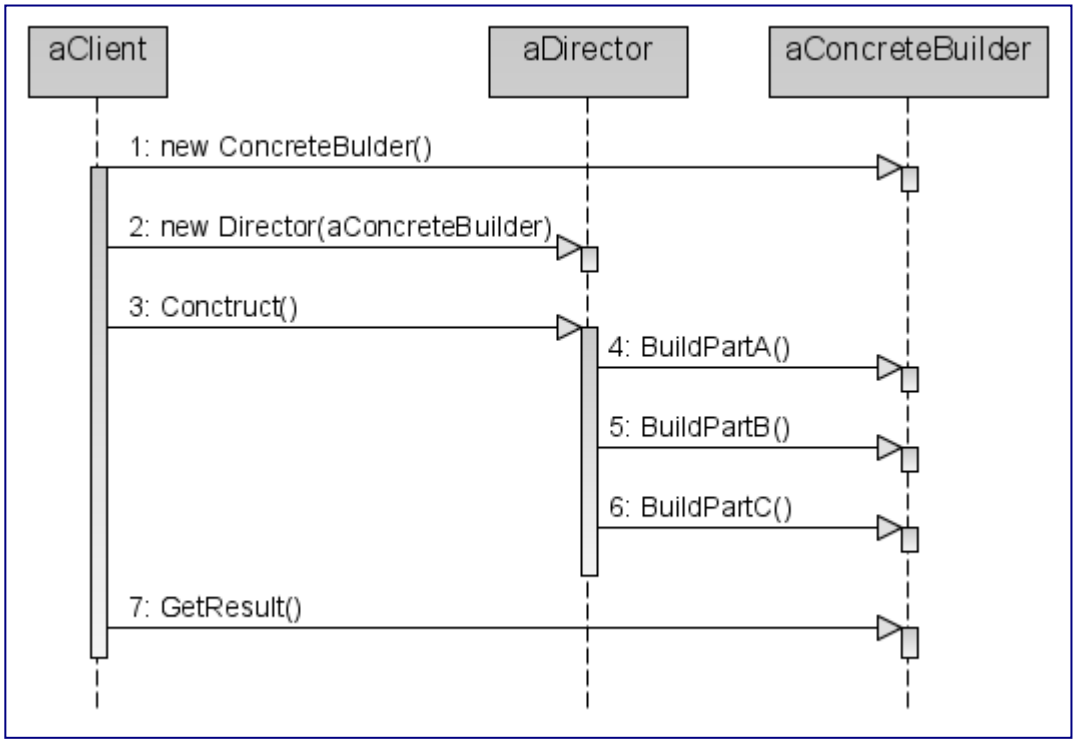


Рис. 15.2. UML діаграма, що описує структуру шаблону проектування *Будівник*

- Builder — будівник: визначає абстрактний інтерфейс для створення частин об'єкта *Product*;
- ConcreteBuilder — конкретний будівник: конструює та збирає докупи частини продукту шляхом реалізації інтерфейсу *Builder*; визначає подання, що створюється, та слідкує на ним; надає інтерфейс для доступу до продукту;
- Director — управитель: конструює об'єкт, користуючись інтерфейсом *Builder*;
- Product — продукт: подає складний конструйований об'єкт. *ConcreteBuilder* будує внутрішнє подання продукту та визначає процес його зборки; вносить класи, що визначають складені частини, у тому числі інтерфейси для зборки кінцевого результату з частин.

Відносини

- *клієнт* створює об'єкт-управитель *Director* та конфігурує його потрібним об'єктом-будівником *Builder*;
- *управитель* повідомляє *будівника* про те, що потрібно побудувати наступну частину *продукту*;
- *будівник* оброблює запити управителя та додає нові частини до *продукту*;
- *клієнт* забирає продукт у *будівника*.




```
    /** "Product" */
class Pizza {
   private String dough = "";
   private String sauce = "";
   private String topping = "";

   public void setDough(String dough)     { this.dough = dough; }
   public void setSauce(String sauce)     { this.sauce = sauce; }
   public void setTopping(String topping) { this.topping = topping; }
}


/** "Abstract Builder" */
abstract class PizzaBuilder {
   protected Pizza pizza;

   public Pizza getPizza() { return pizza; }
   public void createNewPizzaProduct() { pizza = new Pizza(); }

   public abstract void buildDough();
   public abstract void buildSauce();
   public abstract void buildTopping();
}

/** "ConcreteBuilder" */
class HawaiianPizzaBuilder extends PizzaBuilder {
   public void buildDough()   { pizza.setDough("cross"); }
   public void buildSauce()   { pizza.setSauce("mild"); }
   public void buildTopping() { pizza.setTopping("ham+pineapple"); }
```

```
}

/** "ConcreteBuilder" */
class SpicyPizzaBuilder extends PizzaBuilder {
   public void buildDough()   { pizza.setDough("pan baked"); }
   public void buildSauce()   { pizza.setSauce("hot"); }
   public void buildTopping() { pizza.setTopping("pepperoni+salami"); }
}


/** "Director" */
class Waiter {
   private PizzaBuilder pizzaBuilder;

   public void setPizzaBuilder(PizzaBuilder pb) { pizzaBuilder = pb; }
   public Pizza getPizza() { return pizzaBuilder.getPizza(); }

   public void constructPizza() {
      pizzaBuilder.createNewPizzaProduct();
      pizzaBuilder.buildDough();
      pizzaBuilder.buildSauce();
      pizzaBuilder.buildTopping();
   }
}


/** A customer ordering a pizza. */
class BuilderExample {
   public static void main(String[] args) {
       Waiter waiter = new Waiter();
       PizzaBuilder hawaiianPizzaBuilder = new HawaiianPizzaBuilder();
       waiter.setPizzaBuilder(hawaiianPizzaBuilder);
       waiter.constructPizza();

       Pizza pizza = waiter.getPizza();
   }

       }
```

4. XML. Призначення. Структура. Приклад.

XML (*Extensible Markup Language*)-э те мова розмітки, що описує цілий клас об'єктів даних, називаних XML- документами. Ця мова використовується в якості засобу для опису грамматики інших мов і контролю за правильністю впорядкування документів. XML не містить ніяких тегів, призначених для розмітки, а просто визначає порядок їх створення. Таким чином, якщо, наприклад, ми вважаємо, що для позначення елемента *rose* у документі необхідно використовувати тег <*flower*>;, то XML дозволяє вільно використовувати обумовлений нами тег і ми можемо включати в документ фрагменти, подібні такому:

<flower>rose</flower>

Таким чином, у розробників з'являється унікальна можливість визначати власні команди, що дозволяють їм найбільш ефективно визначати дані, що зберігаються в документі. Автор документа створює його структуру, будує необхідні зв'язки між елементами, використовуючи ті команди, що задовольняють його вимогам і домагається такого типу розмітки, що необхідно йому для виконання операцій перегляду, пошуку, аналізу документа.

Ще одною з очевидних переваг XML є можливість використання її в якості універсальної мови запитів до сховищ інформації. Сьогодні в глибинах W3C знаходиться на розгляді робочий варіант стандарту XML-QL (або XQL), що, можливо, у майбутньому складе серйозну конкуренцію SQL. Крім того, XML-документи можуть виступати в якості унікального засобу збереження даних, що містить у собі одночасно засоби для розбору інформації й представлення її на стороні клієнта. У цій області одним із перспективних напрямків є інтеграція Java і XML - технологій, що дозволяє використовувати міць обох технологій при побудові машинно-незалежних додатків, що використовують, крім того, універсальний формат даних при обміні інформацією.

XML дозволяє також здійснювати контроль за коректністю даних, що зберігаються в документах, робити перевірки ієрархічних співвідношень усередині документа і встановлювати єдиний стандарт на структуру документів, умістом яких можуть бути самі різноманітні дані. Це означає, що його можна використовувати при побудові складних інформаційних систем, у котрих дуже важливим є питання обміну інформацією між різноманітними додатками, що працюють в одній системі. Створюючи структуру механізму обміну інформації на самому початку роботи над проектом, менеджер може позбути себе в майбутньому від багатьох проблем, пов'язаних із несумісністю використовуваних різноманітними компонентами системи форматів даних.

На основі XML уже сьогодні створені такі відомі спеціалізовані мови розмітки, як SMIL, CDF, MathML, XSL, і список робочих проектів нових мов, що знаходяться на розгляді W3C, постійно поповнюється.

Структура документа

Не обмежуючи автора яким-небудь фіксованим набором тегів, XML дозволяє йому вводити будь-які імена. Ця можливість є ключовою для активного маніпулювання даними.

Приклад для порівняння представлення списку імен і адрес на HTML і на XML.

От фрагмент HTML:

<H1>Editor Contacts</H1>
<H2>Ім'я: Джонатан Эйнджел</H2>
<P>Посада: старший редактор</P>
<P>Видання: Network Magazine</P>
<P>Вулиця і будинок: Гарісона, 600 </P>
<P>Місто: Сан-Франциско</P>
<P>Штат: Каліфорнія</P>
<P>Індекс: 94107</P>
<P>Електронна пошта:
jangel@mfi. com</P>

Теги розміщають дані на екрані, але нічого не повідомляють про їхню структуру.

У випадку XML той же самий фрагмент буде поданий у такий спосіб (і збережений у файлі EDITORS. XML).

```
<?xml version = '1.0' ?>
<?xml-stylesheet type='text/xsl' href='editors.xsl' ?>
<editor_contacts>
<editor>
<first_name>Jonatan</first_name>
<last_name>Andjel</last_name>
<title>chif editor</title>
<publication>Network
Magazine</publication>
<address>
<street>Garissona, 600 </street>
<city>San-Francisko</city>
<state>California</state>
<zip>94107</zip>
</address>
<e_mail>jangel@mfi.com</e_mail>
</editor>
</editor_contacts>
```

У XML теги не можуть накладатися, як у HTML, проте вони можуть бути вкладені один в одний. Насправді, вкладення навіть рекомендується як засіб створення ієрархії даних (підпорядковані або рівноправні відношення). Як очевидно з приведеного приклада, такі елементи, як <first_name> і <e_mail>, містять дані, у той час як інші (<address>) присутні тільки з метою структурування.

Теги початку і кінця елемента є основними використовуваними в XML розмітками, але ними справа не вичерпується. Наприклад, елементам можуть бути привласнені атрибути. Ця можливість аналогічна наявній в HTML, де, наприклад, елементу <table> може бути привласнений атрибут align=»center». У XML елемент може мати один або більше пов'язаних із ним атрибутів, причому при упорядкуванні документа ви можете видумати їх стільки, скільки побажнете, наприклад <publication topic=»networking» circulation=»controlled»>.

Документи XML можуть містити посилання на інші об'єкти. Посилання являють собою рядок, що починається з амперсанта і закінчується “;”. Ці посилання дозволяють, зокрема, вставити в

документ спеціальні символи. Посилання XML на об'єкти надають набагато більше можливостей, тому що вони можуть посилатися на визначені автором розділи тексту в тому ж самому або в іншому документі.

Наприклад, посилання на об'єкти дозволяють застосувати об'єктно-орієнтований підхід при створенні журнальної статті:

<article>
&introduction;
&body;
&sidebar;
&conclusion;
&resources;

</article>

Найпростіший XML- документ може виглядати так, як це показано в Прикладі

<?xml version="1.0"?>
<list_of_items>
<item id="1"><first/>Перший</item>
<item id="2">Другий <sub_item>підпункт 1</sub_item></item>
<item id="3">Третій</item>
<item id="4"><last/>Останній</item>
</list_of_items>

У XML існують відкриваючі, закриваючі і порожні теги (у HTML поняття порожнього тэга теж існує, але спеціального його позначення не потрібно).

Тіло документа XML складається з елементів розмітки (markup) і безпосередньо вмісту документа - даних (content). XML - теги призначені для визначення елементів документа, їхніх атрибутів і інших конструкцій мови.

Любий XML-документ повинний завжди починатися з інструкції <? xml? >, усередині якої також можна задавати номер версії мови, номер кодової сторінки й інші параметри, необхідні програмі-аналізатору в процесі розбору документа.