**ПРАКТИЧНЕ ЗАНЯТТЯ 5. ПОБУДОВА ДІАГРАМ ВЗАЇМОДІЇ ДЛЯ ВИЯВЛЕННЯ РОЗПОДІЛУ ОБОВ’ЯЗКІВ ОБ’ЄКТІВ.**

**Мета:** набути практичних навичок розподілу обов’язків між об’єктів програмної системи на основі базових принципів об’єктно орієнтованного моделювання (шаблони GARSP) за допомогою діаграм взаїмодії.

**5.1 Принципи розподілу обов’язків між об’єктів програмної системи (шаблони GARSP).**

Після створення основних документів та моделей (артефактів) початкової фази розробки програмної системи (Модель прецедентів, Словник термінів, Додаткова спеціікація, Системна діаграма послідовності, Модель предметної області) переходимо до фази розвитку, де в рамках дисципліни об'єктно-орієнтоване проектування будемо моделювати та проектувати програмну систему. Для створення діарами класів системи необхідно визначити, які операції будуть забеспечувати взаїмодію об'єктів системи для виконання ії фунціональності. Тобто треба здійснити розподіл обов'язків між взаємодіючими об'єктами, використовуючи різні принципи об'єктно-орієнтованого проектуваннявання, наприклад шаблони GRASP або GoF (Gang-ofFour).

Найчастіше проектування програмних об'єктів і великомасштабних компонентів описують в термінах обов'язків, ролей і кооперації. Це частина більш широкого підходу, що отримав назву проектування на основі обов'язків (Responsibllity-driven design- RDD). У RDD вважається, що програмні об'єкти мають певні обов'язки - абстракції, що реалізуються ними. В UML обов'язок (responsibllity) визначається як "Контракт або зобов'язання класифікатора". Обов'язки описують поведінку об'єкта в термінах його ролей. У загальному випадку можна виділити два типи обов'язків: ***знання*** (knowing) і ***дію*** (doing).

Обов'язки, які відносяться до дій об'єкта.

* Виконання деяких дій самим об'єктом, наприклад створення екземпляра або виконання обчислень.
* Ініціювання дій інших об'єктів.
* Управління діями інших об'єктів і їх координування.

Обов'язки, які пов'язані з знанням об'єкта.

* Наявність інформації про закриті інкапсульованих даних.
* Наявність інформації про пов'язаних об'єктах.
* Наявність інформації про наслідки або обчислюваних величинах.

Обов'язки присвоюються об'єктам в процесе об'єктно-орієнтованого проектування. Наприклад, можна сказати, що об'єкт *Firm* відповідає за створення примірника *Order* (дія) або що об'єкт *Appliance* відповідає за наявність інформації про битову техніку (знання).

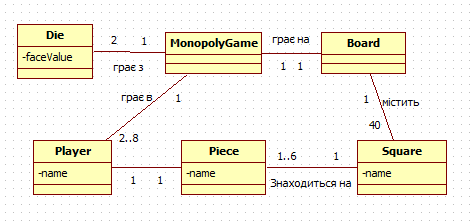
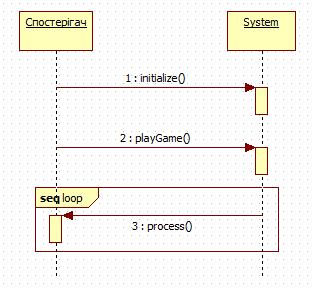
Обов'язки, що відносяться до розряду "знань", часто випливають з моделі предметної області, оскільки вона ілюструє атрибути і асоціації. Наприклад, якщо в моделі предметної області сутність *Order* містить атрибут *date*, то з метою зменшення розриву в уявленнях програмний клас *Order* теж повинен "знати" дату відповідного замовлення.

Рисунок 5.1 – Діаграми моделі предметної області та СДП для програмної системи *MonopolyGame.*

У RDD існує ідея коопераціі (collaboration). Обов'язки реалізуються за допомогою методів, що діють або окремо, або у взаємодії з іншими методами і об'єктами. Наприклад, для класу *Firm* можна визначити один або кілька методів призначення Майстра з діагностики (скажімо, метод *getMaster*). Для виконання цього обов'язку об'єкт *Firm* повинен взаємодіяти з об'єктами класу *Order*, в тому числі передавати повідомлення *getMasterActive* кожному об'єкту *Order* про необхідність надання відповідної інформації цими об'єктами.

**5.2 Шаблони проектування GRASP**

Принципи об'єктного проектування відображені в шаблонах проектування GRASP (General Responsibllity Assignment Software Pattems ), вивчення і застосування яких дозволить освоїти методичний підхід у розподілу обов’язків. Ці шаблони називають також шаблонами розподілу обов'язків (pattern of assigning responsibllities).

Всього до складу GRASP входить 9 шаблонів, але визначимо перших 6 найбільш важливих: Creator, Information Expert, Low Coupling, Controller, High Cohesion, Polymorphism, Pure Fabrication, Indirection, Protected Variations.

Шаблон ***Creator*** зводиться до наступного.

Назва Creator

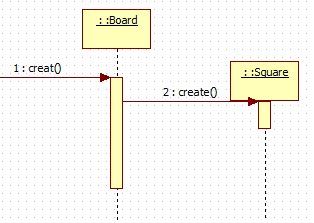
Проблема Хто відповідає за створення нового екземпляра деякого класу А?

Рішення (розглядається як порада) Призначити класу В обов'язок створювати екземпляри класу А, якщо виконується одна (або кілька) з наступних умов.

* Клас В містить (contains) або агрегує (aggregate) об'єкти А.
* Клас В записує (records) екземпляри об'єктів А.
* Клас В ахтівно використовує (closely uses) об'єкти А.
* Клас В володіє даними ініціалізації (has the initializing data) для
* об'єктів А.

Шаблони використовуються в процесі розподілу обов'язків. Розглянемо, як застосувати цей шаблон в реальній ситуації. По-перше, в даному випадку А і В є програмними об'єктами, а не об'єктами з моделі предметної області. Спочатку потрібно спробувати знайти програмний об'єкт В, що задовольняє перерахованим вище умовам. А що робити, якщо ніякі програмні класи ще не визначені? У цьому випадку необхідно звернутися до моделі предметної області.

Розглянемо використання шаблонів розподілу обов’язків на прикладі проектуванні програмної системи *MonopolyGame* рис. 5.1. Звернемося до моделі предметної області і зафіксуємо для себе, об'єкти *Square* містяться в об'єкті *Board*. Ця модель відображає концептуальний, а не програмний ракурс системи, проте її можна спроектувати на модель реалізації і припустити, що програмний об'єкт *Board* повинен містити програмні об'єкти *Square*. Тоді, згідно з шаблоном ***Creator***, об'єкт *Board* повинен створювати об'єкти *Square*. Об'єкти *Square* завжди агрегируются об'єктом *Board*, тому останній керує їх створенням і руйнуванням.



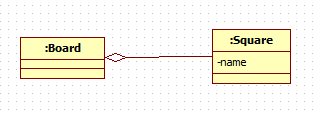


Рисунок 5.2 – Діаграми послідовності та класів , які відображають шаблон ***Creator***

Нагадаємо, що в контексті гнучкого проектування статичні і динамічні моделі створюються паралельно. Отже, це проектне рішення необхідно одночасно відобразити і на діаграмі класів і на діаграмі взаємодії (рис. 5.2). На рис. 5.2 видно, що при створенні об'єкта *Board* відразу ж створюються і об'єкти *Square*. Для спрощення на цій діаграмі не показаний цикл створення 40 примірників клітин.

Шаблон ***lnformation Expert*** зводиться до наступного.

Назва Information Expert

Проблема Який базовий принцип розподілу обов'язків між об'єктами?

Рішення Призначити цей обов'язок тому класу, який володіє достатньою інформацією для її виконання. Для виконання цього обов'язку об'єкт повинен володіти інформацією про власний стан, його оточення, успадкованої інформації і т.п. В данному випадку кожну конкретну клітину на полі може представляти об'єкт, що володіє інформацією про всі клітинах. На рис. 5.2 видно, що таким об'єктом є *Board*, котрий агрегує всі програмні об'єкти *Square*. Отже, об'єкт *Board* володіє достатньою інформацією для виконання цього обов'язку. На рис. 5.3 показаний результат застосування шаблону ***Expert*** в даному контексті.

Шаблон ***Low Coupling*** зводиться до наступного.

Назва Low Coupling

Проблема Як зменшити вплив внесених змін на інші об'єкти?

Рішення Мінімізувати ступінь зв'язування в процесі розподілу обов'язків.

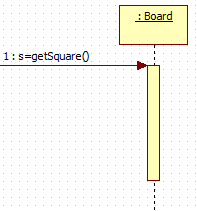
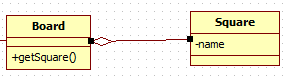


Рисунок 5.3 - Діаграми послідовності та класів , які відображають шаблон ***Expert.***

Ступінь зв'язування (coupling) - це міра пов'язаності одного елемента з іншими. Під зв'язуванням розуміється володіння інформацією або будь-яка залежність. Принцип Low Coupling використовується для оцінки існуючого проектного рішення або вибору рішення з кількох варіантів - за інших рівних умов слід віддавати перевагу проектному рішенню з більш низьким ступенем зв'язування. На рис. 5.1 видно, що об'єкт *Board* містить багато об'єктів *Square*. Чому не привласнити обов'язок знання об'єктів *Square* деякому класу *Dog*? Розглянемо ці альтернативи в термінах шаблону ***Low Coupling***. Якщо клас *Dog* містить метод *getSquare* (), значить, він повинен взаємодіяти з об'єктом *Board* для отримання інформації про колекцію об'єктів *Square*. Ймовірно, ці дані зберігаються в об'єкті-колекції *Мар*, за допомогою якого можна отримувати інформацію по ключу. Тоді об'єкт *Dog* зможе отримати доступ до цих даних по ключу *name* і повернути інформацію про конкретний об'єкті *Square*. Таким чином, ступень звязування стає вище той, яка зображена на рис. 5.2.

Чим корисний принцип мінімізації зв'язування? Або, іншими словами, чому доцільно мінімізувати ступінь зв'язування об'єктів і вплив змін одних об'єктів на інші? Справа в тому, що цей принцип дозволяє заощадити час, зусилля і зменшити кількість похибок при модіфікації програмної системи. В зв’язку з цим, можно підкреслити, що шаблон ***Expert*** підтримує принцип мінімізації зв’язування.

Як було зазначено раніше, згідно принципу Model-View-Separation, об'єкти інтерфейсу не повинні реалізовувати бізнес-логіку додатка, наприклад, обчислювати ходи користувача. Вони повинні делегувати запит (надіслати його іншому об'єкту) на рівень об'єктів предметної області. Шаблон ***Controller*** (контролер) дозволяє відповісти на таке просте питання: який об'єкт за межами рівня інтерфейсу користувача (UI) повинен отримувати повідомлення від рівня UI?

Шаблон ***Controller*** можна описати таким чином.

Назва Controller

Проблема Хто повинен відповідати за отримання та координацію виконання системних операцій, що надходять від рівня інтерфейсу користувача?

Рішення Присвоїти цей обов'язок класу, який задовольняє одній з наступних умов.

* Клас представляє всю систему в цілому, кореневой об'єкт, пристрій або важливу підсистему (зовнішній контролер).
* Клас є сценарій деякого прецеденту, в рамках якого виконується обробка цієї системної операції (контралер прецеденту або контролер сеансу).

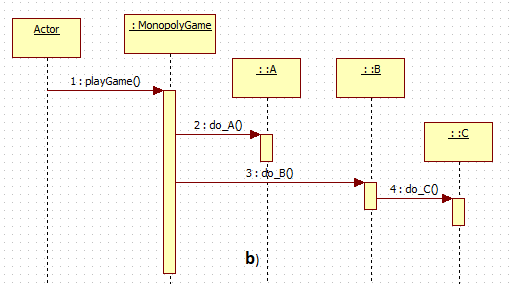
Розглянемо зазначені варіанти.

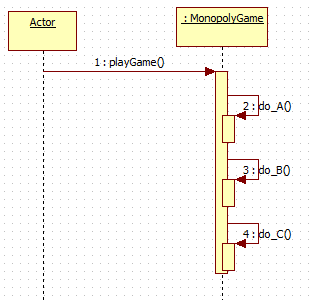
Варіант la. Клас представляє всю систему в цілому або кореневой об'єкт. В ролі такого об'єкта може виступати *MonopolyGame*.

Варіант lб. Клас представляє пристрій, на якому працює дана програмна система. Цей варіант відноситься до спеціалізованих апаратних засобів, наприклад, телефону або касового апарату (програмний клас *Phone* або *CachMachine*), і в даному випадку непридатний.

Варіант 2. Клас являє сценарій прецеденту або сеанс. Системна операція *playGame* виконується в рамках прецеденту *PlayMonopoly*. У ролі такого класу може виступати програмний клас *PlayMonopolyGameHandler* (при використанні цієї версії об'єктного рішення до імен класів додається суфікс *Handle*r або *Session*).

Застосуємо для програмної системи *MonopolyGame* варіант la, який використовується у випадку, коли в системі є лише кілька системних операцій. В якості контролера обираємо об’єкт *MonopolyGame,* який буде реалізовуватисистемнуоперацію *PlayMonopoly* (), тобто повідомлення від об’єкта вікна форми інтерфейсу користувача перетворюється в системну операцію *PlayMonopoly*.



Рисунок 5.4 – Приклад використання шаблона ***HighCohesion*** реалізації системної операції *playGame().*

У всіх випадках при застосуванні об’єктно-орієнтованного підхода для обробки зовнішніх подій використовуються контролери. Шаблон ***Controller*** забеспечує найбільш типичне проектне рішення для цього випадка. Контролер - це своєрідний вид інтерфейсу між рівнями предметної області та графічним представленням.

Щоб забезпечити можливість підтримки інформації про стан прецеденту, для обробки всіх системних подій в рамках одного прецеденту повинен використовуватися один і той же клас контролера. Така інформація може знадобитися, наприклад, для ідентифікації моменту порушення послідовності системних подій (наприклад, виконання операції makePayment перед виконанням операції endSale). Для різних прецедентів можна використовувати різні контролери.

Типовою помилкою при створенні контролерів є покладання на них занадто великого числа обов'язків. Зазвичай контролер повинен лише делегувати функції іншим об'єктам і координувати їх діяльність, а не виконувати ці дії самостійно.

Шаблон ***HighCohesion*** зводиться до наступного.

Назва HighCohesion

Проблема: як забезпечити сфокусованість обов'язків об'єктів, їх керованість і ясність та виконання принципу HighCohesion?

Рішення: розподіл обов'язків, що підтримує високий ступінь зачеплення.

У термінах об'єктно-орієнтованого проектування зачеплення (cohesion) (або, точніше, функціональне зачеплення) - це міра пов'язаності і сфокусованості обов'язків класу. Вважається, що елемент має високу ступенем зачеплення, якщо його обов'язки тісно пов'язані між собою і він не виконує непомірних обсягів роботи. У ролі таких елементів можуть виступати класи, підсистеми і т.п.

Цей принцип використовується для оцінки можливих альтернатив. Клас з низьким ступенем зачеплення виконує багато різнорідних функцій або незв'язаних між собою обов'язків. Такі класи створювати небажано, оскільки вони призводять до виникнення наступних проблем.

* Труднощі розуміння.
* Складнощі при повторному використанні.
* Складнощі підтримки.
* Ненадійність, постійна схильність до змін.

Неформально: зачеплення (cohesion) в контексті проектування програмних систем являє міру пов'язаності операцій одного програмного елемента і що виконується їм обсяг роботи.

У наведеному ліворуч проектному рішенні рис. 5.4 всю роботу системної операції *playGame()* виконує сам об'єкт *MonopolyGame*, а в ситуації, показаної праворуч, він делегує запит *playGame* іншим об’єктам та контролює його виконання, що є прикладом використання шаблону ***HighCohesion.***

Інші шаблони GARSP будуть розглянуті у дисципліні «Конструювання програмного забеспечення».

**5.3 Приклад розподілу обов’язків серед об’єктів програмної систему «RepairTech» прецедент «Оформлення замовлення».**

По перше, визначимо класи-контролери для реалізаціх системних операцій. Згідно шаблоном ***Controller***, можливі наступні варіанти:

* клас, який представляє всю систему в цілому ( *Register*, *FirmSystem),* пристрій або підсистему.
* клас, який представляє одержувача або штучний обробник всіх системних подій деякого сценарію прецеденту (*ProcessOrderHandle*, *ProcessOrderSession* ).

Вибір найбільш гідного контролера визначається ще такими факторами, як зачепленням і зв'язуванням. Про це більш детально розповідається нижче. На даній ітерації розробки прецедента створимо контролер, який буде представляти цій прецедент *ProcessOrderHandle* рис. 5.5 і відповідати за виконання системних операцій цього прецедента: *makeOrder*() - подія «створити замовлення», *getMaster*() – подія «призначити майстра», *changeStatus*() – подія «змінити статус», *saveOrder*() – подія «зберегти замовлення». Далі, якщо цей клас сильно збільшиться та перестане відповідати принципу***HighCohesion,*** його можливо замінимо на кілька контролерів.

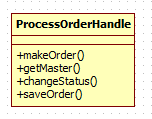


Рисунок 5.5 – Клас-Controller *ProcessOrderHandle* для програмної системи«RepairTech» прецедент «Оформлення замовлення».

Наступний крок – реалізувати системні операції класів рівня системи. Розглянемо, які класи будуть відповідати за реалізацію операції *makeOrder*() – створення об’єктів класу *Order*. Для цього будемо використовувати шаблон ***Creator***. Сутність будемо обирати застосовуя правила шаблона ***Creator***, згідно з яким обов'язок по створенню нових екземплярів делегується класу, який містить, агрегує або записує інформацію про створюваних клас. Проаналізувавши Модель предметної області, приходимо до висновку, що об’єкту, який відповідає за створення об’єктів *Order* немає. В цієї ролі можливо на даному етапі проектування використавувати об’єкт-контролер *ProcessOrderHandle,* який отримає від інтерфейсу користувача інформацію для ініціалізації об’єкта *Order* (рис. 5.5), тобто володіє даними ініціалізації. Якщо екземпляри *Order* будуть створюватися об'єктом *ProcessOrderHandle*, то об'єкт *ProcessOrderHandle* буде містити посилання на поточний екземпляр *Order*.

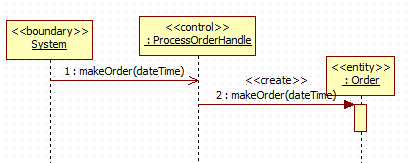


Рисунок 5.6 – Діаграма послідовності для створення об’єкта *Order.*

Реалізуємо системну операцію *getMaster.* Згідно з описом цієї опреції (стр. 74) нам необхідно перейти до стану, де реалізовано:

* створено список екземплярів *ListOrders*[] об’єктів *Order* *(створення екземпляру);*
* екземпляр *ListOrders*[i] об’єкта Order пов’язан з класом *Order* *(формування асоціації);*
* атрибуту *ord*.*id\_master* присвоєно значення *ListOrders*[*min*].*id\_master* *(модіфікація атрибуту)*
* атрибуту *ord*.*status* присвоєно значення *String* “diagnostic” *(модіфікація атрибуту).*

Для реалізації цієї операції будемо використовувати шаблон ***Information Expert***, тобто знайдемо об’єкти, які мають інформацію для виконання цієї операції.

Тепер виникає ключове питання: на основі якої моделі потрібно проаналізувати інформацію - Моделі предметної області або проектування? Модель предметної області ілюструє концептуальні класи з предметної області системи, а в Моделі проектування показані програмні класи. Відповідь на це питання зводиться до наступного.

1. Якщо в моделі проектування є відповідні класи, в першу чергу, слід використовувати її.
2. В іншому випадку потрібно звернутися до Моделі предметної області і постаратися уточнити її для полегшення створення відповідних програмних класів.

Інформацію про майстрів та кількість їх навантаження ми можемо отримати з об’єктів *Order,* дані про які має об’єкт *ProcessOrderHandle*, тобто опрецацію *getMaster()* буде реалізовувати цей об’єкт - визначати майстра з мінімальним навантаженням та передавати його *id\_master* у поточний об’єкт *Order.* Для цього нам потрібно визначити для кожного майстра його навантаження, тобто скільки замовлень розподілених на майстра мають статус «діагностика» і визначити майстра з мінімальним навантаженням (рис. 5.7*)*

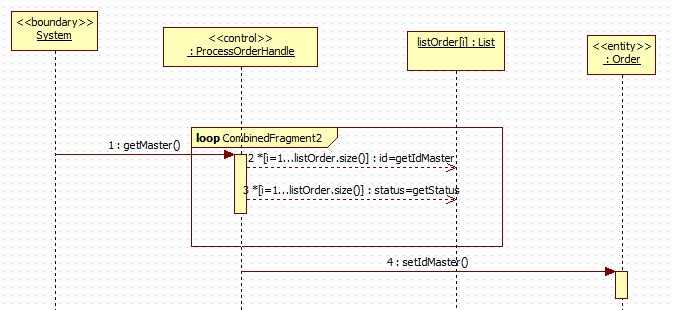


Рисунок 5.7 – Діаграма послідовності для операції *getMaster*().

Наступна системна операція *changeStatus*(), яка змінює статус замовлення. При створені нового замовлення можливо атрибут «статус» ініціювати за замовченням значенням «діагностика», але у бізнес процесі говориться про зміну статусу замовлення на різних етапах його виконання, тому необхідно створити операцію *changeStatus*(). Застосуємо знову шаблон ***Information Expert,*** експертом для зміни статусу може стати об’єкт *ProcessOrderHandle*, який отримає значення статусу з інтерфейсу користувача після визначення Майстра з діагностики рис.5.8.

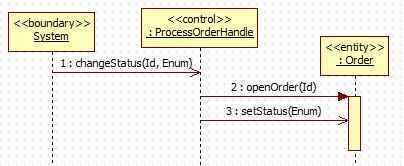
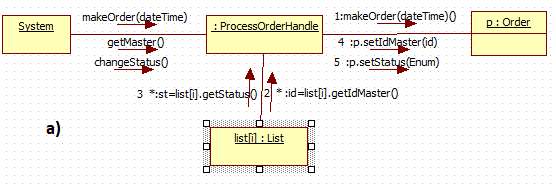


Рисунок 5.8 – Діаграма послідовності для реалізації операції *changeStatus*().

Таким чином, розглянуто обов’язкі різних об’єктів в реалізації сценарію прецеденту «Оформлення замовлення» на першій ітерації фази розвитку. Підсумуємо результати аналізу у вигляді діаграми кооперації рис. 5.9 a, яка більш компактна ніж діаграма послідовності, та діаграми класів рис. 5.9 б для цього прецеденту.



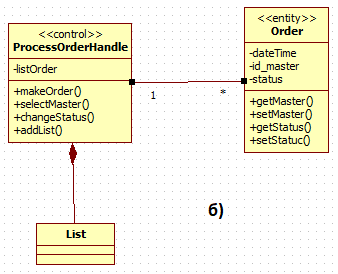


Рисунок 5.9 – a) діаграма кооперації, б) діаграма класів для прецедента «Оформлення замовлення»

На діаграмах не проаналізовані всі асоціації моделі предметної області на даній ітерації фази розвитку. Розглянемо використання шаблонів ***HighCohesion*** та ***Low Coupling*** в розподілу обов’язків.

Як було визначено вище, шаблон ***Low Coupling*** використовується для оцінки існуючого проектного рішення або вибору рішення з кількох варіантів - за інших рівних умов слід віддавати перевагу проектному рішенню з більш низьким ступенем зв'язування. Під зв'язуванням розуміється володіння інформацією або будь-яка залежність. Діаграма кооперації рис.5.9 показує, що на даній ітерації розподіл обов’язків між об’єктами прецеденту «Оформлення замовлення» немає альтернативного рішення і тому відповідає низькому зв'язуванню.

Шаблон ***HighCohesion*** , як було сказано вище, аналізує міру пов'язаності і сфокусованості обов'язків класу. Вважається, що елемент має високий ступень зачеплення, якщо його обов'язки тісно пов'язані між собою і він не виконує непомірних обсягів роботи. У ролі таких елементів можуть виступати класи, підсистеми і т. п. Клас з низьким ступенем зачеплення виконує багато різнорідних функцій або незв'язаних між собою обов'язків. Такі класи створювати небажано, оскільки вони призводять до виникнення різних проблем. Класи *ProcessOrderHandle* та *Order,* як показує діаграма класів рис. 5.9 б, мають високу ступень зачеплення, тому що всі операції пов’язані з поведінкою саме цих класів і кількість операцій невелика.

**Питання для самоконтролю**

1. В чому суть підходу RDD для визначення операцій, за допомогою яких об’єкти взаїмодіють ?
2. Що таке шаблони GARSP? Перерахуйте та опишить іх призначення.
3. Для виявлення яких обов’язків застосовується шаблон ***Creator***? Правила його використання. Наведить приклад використання цього шаблону.
4. Для виявлення яких обов’язків застосовується шаблон ***Information Expert***? Які правила його використання? Наведить приклад використання цього шаблону.
5. Для виявлення яких обов’язків застосовується шаблон ***Controller***? Які правила його використання? Наведить приклад використання цього шаблону.
6. Що таке принцип звязування в проектуванні програмних систем. Який шаблон реалізує цей принцип? Правила його використання.
7. Що таке принцип зачеплення в проектуванні програмних систем. Який шаблон реалізує цей принцип? Правила його використання.

**Завдання до лабораторної роботи** **№6**

1. Відповісти на питання для самоконтролю.
2. Проаналізувати розподіл обов’язків за допомогою шаблонів GARSP між об’єктів одного прецеденту.
3. Результат аналізу представити у вигляді діаграм послідовності, кооперації та класів для кожного шаблону та загалом для прецеденту).
4. Написати коди реалізації класів на C#.
5. Оформити протокол лабораторної роботи.