**Поняття архітектури програмного забезпечення**

[Поняття архітектури програмного забезпечення](http://www.lib.mdpu.org.ua/e-book/web/Lec10.html#_Toc283495491)   
[Принципи проектування](http://www.lib.mdpu.org.ua/e-book/web/Lec10.html#_Toc283495494)   
[Якість архітектури](http://www.lib.mdpu.org.ua/e-book/web/Lec10.html#_Toc283495495)   
[Ознаки загниваючого проекту](http://www.lib.mdpu.org.ua/e-book/web/Lec10.html#_Toc283495496)   
[Як зробити кращу архітектуру](http://www.lib.mdpu.org.ua/e-book/web/Lec10.html#_Toc283495497)   
[Принципи гарної ахрітектури](http://www.lib.mdpu.org.ua/e-book/web/Lec10.html#_Toc283495498)   
[Повторне використання коду](http://www.lib.mdpu.org.ua/e-book/web/Lec10.html#_Toc283495506)   
[Framework-системи](http://www.lib.mdpu.org.ua/e-book/web/Lec10.html#_Toc283495507)   
[CMF і CMS](http://www.lib.mdpu.org.ua/e-book/web/Lec10.html#_Toc283495509)   
[Архітектура CMF-системи](http://www.lib.mdpu.org.ua/e-book/web/Lec10.html#_Toc283495510)   
[Маленькі бібліотеки](http://www.lib.mdpu.org.ua/e-book/web/Lec10.html#_Toc283495511)   
[Використання функцій](http://www.lib.mdpu.org.ua/e-book/web/Lec10.html#_Toc283495514)   
[Патерни проектування](http://www.lib.mdpu.org.ua/e-book/web/Lec10.html#_Toc283495515)   
[Загальні відомості](http://www.lib.mdpu.org.ua/e-book/web/Lec10.html#_Toc283495516)   
[Огляд патернів](http://www.lib.mdpu.org.ua/e-book/web/Lec10.html#_Toc283495517)   
[Патерни проектування класів / об'єктів](http://www.lib.mdpu.org.ua/e-book/web/Lec10.html#_Toc283495518)   
[Архітектурні системні патерни](http://www.lib.mdpu.org.ua/e-book/web/Lec10.html#_Toc283495519)   
[Патерни управління](http://www.lib.mdpu.org.ua/e-book/web/Lec10.html#_Toc283495520)   
[Патерни, призначені для представлення даних у Web](http://www.lib.mdpu.org.ua/e-book/web/Lec10.html#_Toc283495521)   
[Патерни інтеграції корпоративних інформаційних систем](http://www.lib.mdpu.org.ua/e-book/web/Lec10.html#_Toc283495522)   
[Model-Viewer-Controller](http://www.lib.mdpu.org.ua/e-book/web/Lec10.html#_Toc283495523)   
[Схема архітектури MVC](http://www.lib.mdpu.org.ua/e-book/web/Lec10.html#_Toc283495524)

**Поняття архітектури програмного забезпечення**

**Архітектура програмного забезпечення** (англ. software architecture) - це структура програми або обчислювальної системи, яка включає програмні компоненти, видимі зовні властивості цих компонентів, а також відносини між ними. Цей термін стосується також документування архітектури програмного забезпечення. Документування архітектури ПЗ спрощує процес комунікації між зацікавленими особами (англ. stakeholders), дозволяє зафіксувати прийняті на ранніх етапах проектування рішення про високорівневий дизайн системи і дозволяє використовувати компоненти цього дизайну і шаблони повторно в інших проектах.  
  
**Огляд**

   Область комп'ютерних наук з моменту свого утворення зіткнулася з проблемами, пов'язаними зі складністю програмних систем. Раніше проблеми складності вирішувалися розробниками шляхом правильного вибору структур даних, розробки алгоритмів та застосування концепції розмежування повноважень. Хоча термін "архітектура програмного забезпечення" є відносно новим для індустрії розробки ПЗ, фундаментальні принципи цієї області невпорядковано застосовувалися піонерами розробки ПЗ починаючи з середини 1980-х. Перші спроби усвідомити і пояснити програмну архітектуру системи були повні неточностей і страждали від нестачі організованості, часто це була просто діаграма з блоків, сполучених лініями. У 1990-ті роки спостерігається спроба визначити і систематизувати основні аспекти даної дисципліни. Початковий набір шаблонів проектування, стилів дизайну, передового досвіду (best-practices), мов опису та формальна логіка були розроблені протягом цього часу.

   Основною ідеєю дисципліни програмної архітектури є ідея зниження складності системи шляхом абстракції і розмежування повноважень. На сьогоднішній день до цих пір немає згоди щодо чіткого визначення терміна "архітектура програмного забезпечення".

   Будучи в справжній момент свого розвитку дисципліною без чітких правил про "правильний" шлях створення системи, проектування архітектури ПЗ все ще є сумішшю науки і мистецтва. Аспект "мистецтва" полягає в тому, що будь-яка комерційна система має на увазі наявність застосування або місії. Те, які ключові цілі має система, описується за допомогою сценаріїв як нефункціональні вимоги до системи, також відомі як атрибути якості, що визначають, як буде вести себе система. Атрибути якості системи включають в себе **відмовостійкість**, збереження зворотної сумісності, розширюваність, надійність, придатність до сервісного обслуговування (maintainability), доступність, безпека, зручність використання, а також інші якості. З точки зору користувача програмної архітектури, програмна архітектура дає напрям для руху і вирішення завдань, пов'язаних зі спеціальністю кожного такого користувача, наприклад, зацікавленої особи, розробника ПЗ, групи підтримки ПЗ, фахівця із супроводу ПЗ, фахівця з розгортання ПЗ, тестера, а також кінцевих користувачів. У цьому сенсі архітектура програмного забезпечення насправді об'єднує різні точки зору на систему. Той факт, що ці декілька різних точок зору можуть бути об'єднані в архітектурі програмного забезпечення є аргументом на захист необхідності та доцільності створення архітектури ПЗ ще до етапу розробки ПЗ.  
  
**Підсумок**

**Архітектура** - це принцип організації компонентів усередині системи: їх кількість, якість, інтерфейси і протоколи взаємодії.

   Що залежить від архітектури? Від неї залежить ціна на підтримку і розробку нових фіч, трудовитрати на побудову цілої системи з використанням даної архітектури. Тобто формально від архітектури залежить найважливіший параметр розробки - собівартість. А побічно ще і можливість повторного використання коду, а разом з ним і зменшення трудовитрат на кожну подальшу розробку.

   Добре, ми з'ясували що архітектура це дуже важливий аспект розробки. Але що ж це таке? У контексті PHP5 додатки з упором в  **парадигму** - це ієрархія класів, інтерфейси та схеми їх взаємодії.

   Вибір або створення архітектури залежить від конкретних завдань. Наприклад, наскільки універсальним планується додаток, які модулі повинні бути присутніми, яка запланована навантаження на ресурс.  
  
**Принципи проектування**   
  
**Якість архітектури**   
  
**Ознаки загниваючого проекту**

   Отже, будь-який програмний код має взаємозалежності одних частин від інших. Класи вимагають наявності інших класів, одні функції викликають інші і т.д. У міру зростання будь-якого проекту взаємозалежностей стає все більше і більше. Вимоги до проекту змінюються, розробники іноді вносять швидкі й не завжди вдалі рішення. Якщо залежностями грамотно не управляти, то проект неминуче почне загнивати. Код стає складніше розуміти, він частіше ламається, стає менш гнучким і важким для повторного використання. У результаті швидкість розробки падає, проект чинить опір змінам, і ось вже серед розробників звучать заклики «Давайте все переробимо! Наступного разу ми зробимо супер-архітектуру ». Ось найбільш поширені ознаки поганого або загниваючого в плані коду проекту:

**Закритість** (rigid) - система відчайдушно чинить опір змінам, неможливо сказати, скільки займе реалізація тієї або іншої функціональності, тому що зміни, швидше за все, торкнуться багатьох компонентів системи. Через це вносити зміни стає занадто дорого, тому що вони вимагають багато часу.

    Нестійкість, **крихкість** (fragile) - система ламається в непередбачених місцях, хоча зміни, були проведені до цього, зламані компоненти явно не зачіпали.

**Нерухомість** або **монолітність** (not reusable) - система побудована таким чином і характер залежностей такий, що використовувати будь-які компоненти окремо від інших не представляється можливим.

**В'язкість** (high viscosity) - код проекту такий, що зробити що-небудь неправильно набагато простіше, ніж зробити щось правильно.

**Невиправдані** **повторення** (high code duplication) - розмір проекту набагато більший, ніж він міг би бути, якби абстракції застосовувалися частіше.

**Надмірна** **складність** (overcomplicated design) - проект містить рішення, користь від яких неочевидна, вони приховують реальну суть системи, ускладнюючи її розуміння і розвиток.

   Майже будь-який більш-менш досвідчений розробник може пригадати приклад коду, який відповідав хоча б одному за цією ознакою.  
  
**Як зробити кращу архітектуру**

   За довгі роки розумні люди виробили деякі основоположні принципи ООП, дотримання яких дозволяє створювати кращу архітектуру:

**Висока** **зчепленість** **коду** (High Cohesion) - код, відповідальний за яку-небудь одну функціональність, повинен бути зосереджений в одному місці.

**Низька** **зв'язаність** **коду** (Low Coupling) - класи повинні мати мінімальну залежність від інших класів.

**Вказуй**, **а** **не** **питай** (Tell, Don't Ask) - класи містять дані і методи для оперування цими даними. Класи не повинні цікавитися даними з інших класів.

**Не розмовляй з незнайомцями** (Don't talk to strangers) - класи повинні знати тільки про своїх безпосередніх сусідів. Чим менше знає клас про існування інших класів або інтерфейсів - тим більш стійкий код.

    Всі ці рекомендації спрямовані на те, щоб постаратися розвести класи по сторонах, зосередити сильні взаємозв'язки в одному місці і провести чіткі розмежувальні лінії в коді.

    Але ці принципи надто розпливчасті, тому з'явився якийсь набір більш чітких правил, якими слід керуватися при формуванні архітектури.

**Принцип персональної відповідальності** (Single Responsibility Principle) - клас володіє тільки 1 відповідальністю, тому існує тільки 1 причина, що приводить до його зміни.

**Принцип відкриття-закриття** (Open-Closed Principle) - класи повинні бути відкриті для розширень, але закриті для модифікацій. Здається, що це неможливо, однак варто згадати шаблон проектування Strategy і стає більш-менш зрозуміло.

**Принцип підстановки**Ліскоу (Liskov Substitution Principle) - дочірні класи можна використовувати через інтерфейси базових класів без знання про те, що це саме дочірній клас. Інакше - дочірній клас не повинен заперечувати поведінку батьківського класу і повинна бути можливість використовувати дочірній клас скрізь, де використовувався батьківський клас.

**Принцип інверсії залежностей** (Dependency Inversion Principle) - всередині системи стоять на основі абстракцій. Модулі верхнього рівня не залежать від модулів нижнього рівня. Абстракції не залежать від подробиць.

**Принцип відділення інтерфейсу** (Interface Segregation Principle) - клієнти не повинні потрапляти в залежність від методів, якими вони не користуються. Клієнти визначають, які інтерфейси їм потрібні.  
   Тепер докладніше.  
  
**Принципи гарної ахрітектури**   
  
**Принцип відкриття-закриття (Open Close Principle або OCP)**

   Програмні суті такі як класи, модулі та функції повинні бути відкриті для розширення, але закриті для змін.

   Ви можете обговорювати його, коли пишете ваші класи, щоб бути впевненими в тому, що коли вам буде потрібно розширити поведінку, ви не повинні будете змінювати клас, але можете розширювати його. Подібний же принцип застосуємо для модулів, пакетів і бібліотек. Якщо у вас є бібліотека, що складається з множини класів, то є багато причин для того, щоб ви вважали за краще розширення замість зміни коду, який вже написаний (заради зворотної сумісності, повернення до попереднього тестування і т.д.). Це причина, по якій ми повинні бути впевнені, що наші модулі слідують Принципу відкриття-закриття. По відношенню до класів Принцип відкриття-закриття може бути гарантовано корисний за рахунок використання Абстрактних Класів і конкретних класів для реалізації їх поведінки. Це буде змушувати мати Конкретні Класи, що розширюють Абстрактні Класи замість їх зміни. Деякі приватні випадки цього принципу є Шаблонний Патерн і Стратегічний Патерн (Template Pattern and Strategy Pattern).  
  
**Принцип Заміщення Ліскоу (Liskov's Substitution Principle)**

   Похідні типи повинні бути здатні повністю заміщатися їх базовими типами.

   Цей принцип є всього лише розширенням Принципу відкриття-закриття в умовах поведінки, що означає, що ми повинні бути впевнені, що нові похідні класи є розширенням базових класів без зміни їх поведінки. Нові похідні класи повинні бути здатні замінювати базові класи без будь-яких змін у коді. Принцип Заміщення Ліскоу був введений на 1987 Conference on Object Oriented Programming Systems Languages and Applications, in Data abstraction and hierarchy.  
  
**Принцип Єдиної Відповідальності (Single Responsibility Principle)**

   Клас повинен мати тільки одну причину для зміни.

   У цьому контексті відповідальність розглядається як єдина причина для зміни. Цей принцип стверджує, що якщо ми маємо 2 причини для зміни класу, то ми повинні розділити функціональність на 2 класи. Кожен клас повинен мати тільки одну відповідальність, і в майбутньому, якщо нам буде потрібно зробити одну зміну ми будемо робити це в класі, який утримує цю одну відповідальність. Коли нам потрібно робити зміни в класі, який має більше відповідальностей, зміна може вплинути на іншу функціональність класів.

   Принцип Єдиної Відповідальності був введений Tom DeMarco в його книзі «Structured Analysis and Systems Specification, 1979». Роберт Мартін переробив цю концепцію і визначив, що відповідальність є причиною для зміни.  
  
**Принцип Відділення Інтерфейсу (Interface Segregation Principle)**

   Клієнти не повинні бути залежними від інтерфейсів, які вони не використовують.

   Цей принцип вчить нас піклуватися про те, як ми пишемо наші інтерфейси. Коли ми пишемо інтерфейси, ми повинні подбати про додавання тільки тих методів, які там повинні бути. Якщо ми додаємо методи, які не повинні бути там, тоді класи, що реалізують інтерфейс будуть повинні реалізовувати зайві методи так само як і інші методи. Наприклад, якщо ми створюємо інтерфейс, званий Worker (Робочий) і додаємо метод lunch break (обідня перерва), тоді всі workers (робітники) будуть мати реалізацію цього зайвого методу. А що якщо робітник виявився роботом?

   Інтерфейси містять методи, які не є специфічними для них, такі методи призводять до того, що інтерфейси називають забрудненими або жирними. Ми повинні уникати створення таких інтерфейсів.  
  
**Принцип інверсії залежностей**

   (Dependency Inversion Principle) - залежності всередині системи будуються на основі абстракцій. Модулі верхнього рівня не залежать від модулів нижнього рівня. Абстракції не залежать від подробиць.

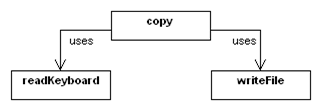
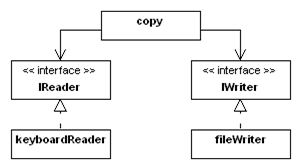
   Даний принцип дуже важливий і гідний докладного розгляду.  
  
**Принцип інверсії залежностей в деталях**

   У протиставленні поганому дизайну, гарний дизайн архітектури повинен бути гнучким, стійким, і пристосованим до повторного використання. Чим нижче взаємозв'язок компонентів додатка один з одним, тим вища гнучкість і мобільність всієї програми в цілому. Програми, що характеризуються високим коефіцієнтом мобільності, дозволяють застосовувати свої компоненти знову і знову для вирішення однотипних задач. Це веде до зниження дублювання коду. Такі програми складаються з великого набору досить дрібних компонентів, кожен з яких виконує малу частину роботи, але виконує її якісно. Дрібні компоненти набагато простіше тестувати, реалізовувати і супроводжувати.

   Якщо ви дотримуєтеся принцип інверсії залежностей, то ваш код більш пристосований до змін і менше залежить від контексту виконання. Вірно і зворотне твердження. Якщо ваш додаток є хорошим прикладом вдалого дизайну архітектури, то він, в тій чи іншій мірі, дотримується принципу інверсії залежностей.  
  
**Суть принципу**   
   1. Модулі верхнього рівня не повинні залежати від модулів нижнього рівня. Обидва типи модулів повинні залежати від абстракцій;  
   2. Абстракція не повинна залежати від реалізації. Реалізація повинна залежати від абстракції.

   Традиційні методи розробки (наприклад, процедурне програмування) мають тенденцію до створення коду, в якому високорівневі модулі, як раз, залежать від низькорівневих. Це відбувається через те, що одна з цілей цих методів розробки - визначення ієрархії підпрограм, а отже і ієрархії викликів усередині модулів (високорівневі модулі викликають низькорівневі). Саме це є причиною низької гнучкості і закостенілості дизайну. При правильному використанні, ГО методики дозволяють обійти це обмеження.

   Розглянемо приклад програми, яка копіює в файл дані, введені з клавіатури.

   У нас є три модулі (у даному випадку це функції). Один модуль (іноді його називають сервіс) відповідає за читання з клавіатури. Другий - за виведення у файл. А третій високорівневий модуль об'єднує два низькорівневих модуля з метою організації їх роботи.  
  
[](http://wiki.agiledev.ru/lib/exe/detail.php?id=ooad:dependency_injection&cache=cache&media=ooad:no_injection.png)   
**Наш модуль copy може виглядати приблизно так.**  
   while (($ data = readKeyboard ())! == false)  
   {  
    writeFile (". / filename", $ data);  
   }  
   Низькорівневі модулі readKeyboard і writeFile володіють високою гнучкістю. Ми легко можемо використовувати їх у контексті відмінному від функції copy. Проте сама функція «copy» не може бути повторно використана в іншому контексті. Наприклад, для відправки даних з файлу системного оброблювача логів.  
   Використовуючи принцип інверсії залежностей, можна зробити модуль copy незалежним від об'єктів джерела і призначення даних. Для цього необхідно виробити абстракції для цих об'єктів, і зробити модулі залежними від цих абстракцій, а не один від одного.  
  
[](http://wiki.agiledev.ru/lib/exe/detail.php?id=ooad:dependency_injection&cache=cache&media=ooad:injection.png)   
   interface IReader  
   {  
   public function read ();  
   }  
   
   interface IWriter  
   {  
    public function write ($ data);  
   }  
   Модуль **copy** повинен покладатися тільки на вироблені абстракції і не робити ніяких припущень з приводу індивідуальних особливостей об'єктів вводу / виводу.  
   while (($ data = $ reader-> read ())! == false)  
   {  
    $ Writer-> write ($ data);  
   }  
   Приблизно таким чином виглядає використання нашого модуля користувачем.  
   $ Copier = new copier ();  
   
   / / Копіювання даних з клавіатури в файл  
   $ Copier-> run (new keyboardReader (), new fileWriter ('. / Filename'));  
   
   / / Надсилання даних з файлу системного оброблювача логів  
   $ Copier-> run (new fileReader ('. / Filename'), new syslogWriter ());  
   Тепер модуль copy можна використовувати в різних контекстах копіювання. Зміна поведінки модуля-копіювальника досягається шляхом асоціації його з об'єктами інших класів (але які залежать від тих же абстракцій).  
   Незважаючи на простоту виконаних нами дій ми отримали дуже важливий результат. Тепер наш код володіє наступними якостями:  
• модуль може бути використаний для копіювання даних в контексті відмінному від даного;  
• ми можемо додавати нові пристрої введення / виведення не змінюючи при цьому модуль copy.  
    Таким чином, знизилася крихкість коду, підвищилася його мобільність і гнучкість.  
  
**Повторне використання коду**

**Повторне використання коду** (англ. code reuse) - методологія проектування комп'ютерних та інших систем, що полягає в тому, що система (комп'ютерна програма, програмний модуль) частково або повністю повинна складатися з частин, написаних раніше компонентів і / або частин іншої системи. Повторне використання - основна методологія, яка застосовується для скорочення трудовитрат при розробці складних систем.

   Найпоширеніший випадок повторного використання коду - бібліотеки програм. Бібліотеки надають загальну досить універсальну функціональність, яка покриває обрану предметну область. Приклади: Бібліотека функцій для роботи з комплексними числами, бібліотека функцій для роботи з 3D-графікою, бібліотека для використання протоколу TCP / IP, бібліотека для роботи з базами даних. Розробники нової програми можуть використовувати існуючі бібліотеки для вирішення своїх завдань.

   Повторне використання коду за межами одного проекту практично неможливо, якщо у вас немає розробленого проектного каркаса [framework]. У різних проектах різні набори сервісів, що і ускладнює повторне використання об'єкта.

   Розробка проектного каркаса віднімає багато сил і часу. Але навіть якщо з якихось причин ви не створили собі подібної системи, існує декілька прийомів заохочення повторного використання коду.  
  
**Framework-системи**   
  
**Вступ**

   Що таке framework-система? Навіщо вона Вам потрібна? У чому вона Вам може допомогти? А в чому ні? У цьому документі я спробую дати відповіді на ці питання.

   Інтернет технології за останні десять років зробили дуже великий крок вперед, ставши полігоном для ведення бізнесу та електронної комерції. Розвиток глобальної мережі спричинило за собою розвиток інтернет-додатків. Якщо раніше сайти були не більше ніж «оголошеннями на заборі», то тепер це повноцінні програми, здатні виконувати завдання по автоматизації збору даних, обробки даних та надання інформації.

   Перед сучасними web-розробниками постає дуже широкий спектр завдань. Це ефективна робота з реляційними базами даних, зберігання і обробка даних у форматі XML, побудова гнучких систем відображення інформації та багато іншого. Така множина завдань робить старі методи розробки додатків вкрай неефективними. Це призводить до думки про необхідність наявності спеціального інструментарію для web-розробника, який допоможе йому у вирішенні часто виникаючих проблем і завдань.  
  
*Отже, що таке framework?*

    Коли людина вирішує якесь завдання багато разів поспіль, він вчиться вирішувати її швидко і ефективно! З точки зору web-програмування, framework-система (CMF-система) це платформа, що дозволяє вирішувати завдання, які постійно виникають при створенні інтернет-додатків. Не варто думати, що CMF-система - це просто набір модулів для вирішення різнотипних завдань, яких в Інтернеті безліч. Framework-система це щось більше. Це:  
• Термінологія, яка дозволяє розробникам говорити лаконічно про дуже складні речі;  
• Набір архітектурних стандартів, які система накладає на інтернет-додатки. Це знімає з розробників необхідність придумувати все з нуля і дозволяє більш ефективно використовувати код повторно;  
• Модулі для вирішення завдань «першої необхідності», що дозволяють почати розробку з порожнього місця, не винаходячи нічого свого.

   Framework-система для web-розробника відіграє таку ж роль, як саквояж з інструментами для монтажника. Навіть якщо монтажник зможе виконати свою роботу без свого саквояжа, він витратить більше часу, а якість виконаної роботи буде на порядок нижче. Аналогічна ситуація спостерігається в процесі створення інтернет-додатків.  
  
**CMF і CMS**

   Розглядаючи поняття framework-системи, не можна обійти стороною, поняття системи управління контентом. Дуже часто поняття CMF (Content Management Framework) плутають з поняттям CMS (Content Management System). Це невірно, тому що це принципово різні речі.

**CMF-системи** не можна порівнювати з CMS-системами! Це головне правило, яке дуже часто порушують розробники при обговоренні питань пов'язаних з розробкою та використанням CMF-систем. CMF і CMS різні поняття, незважаючи на їх схожість.

**CMS-система** - це набір модулів для швидкого створення сайтів. На відміну від CMF, CMS-система - це є завершений продукт, який орієнтований, в першу чергу, не на програмістів, а на користувачів не знайомих з премудростями створення інтернет-додатків. CMS-система (дуже часто її називають «движок сайту») дозволяє за лічені години створити сайт або портал який складається з обмеженого набору готових модулів (новини, гостьова книга, форум). В більшості своїй, CMS-системи створюються без урахування їх подальшого зростання. Підсумок - відсутність жорсткої внутрішньої архітектури системи. Це істотно ускладнює процес супроводження проекту.

   Якщо вам достатньо можливостей CMS-системи, то, швидше за все, Ви будете задоволені. Однак якщо перед Вами постане питання про зміну дизайну або розширення можливостей програми, то, в більшості випадків, Вам доведеться вдатися за допомогою до кваліфікованих програмістів. І, можливо, навіть їм буде не просто розібратися в цій CMS-системі. Прочитавши наступний параграф, Ви зрозумієте, чому в цьому абзаці так багато «можливо» і «швидше за все».

    Вищесказане відноситься до «чистих» CMS-систем. Тобто до CMS-систем, які написані з нуля на порожньому місці. На щастя, ніхто не заважає використовувати вигоди обох типів систем. Останнім часом в Інтернеті почали з'являтися CMF / CMS-системи. Ці системи являють собою CMS-систему, створену на фундаменті framework'а. Вигоди очевидні. Детермінована внутрішня архітектура, яка в більшості випадків документована і розвинені механізми абстракції, які не залежать від CMS-утворюючих модулів. Супроводжувати проект, створений на основі CMF / CMS-системи, на порядок простіше, ніж проект, створений на основі «чистої» CMS-системи. Це пояснюється тим, що в першому випадку, при створенні CMS-системи, програмістам доводиться виконувати ряд вимог, які диктує framework. Завдяки цьому CMS-система набуває яскраво виражену архітектуру, як і всі додатки створюються за допомогою CMF-системи.

   Якщо CMS-система являє собою закінчений продукт, то CMF-система - це набір інструментів, за допомогою яких, можна створити абсолютно будь-який продукт, зокрема і CMS-систему. Так як framework-система - це набір інструментів, то для її використання потрібні програмісти, які можуть з цими інструментами працювати. З цим пов'язаний ще один момент, характерний для CMF - навчання персоналу для роботи з CMF-системою.

   Продукти CMF-системи (додатки, написані на її основі) відрізняються індивідуальністю та високим рівнем адаптації до конкретної ситуації, тому що вони є програмними рішеннями, призначеними для вирішення конкретного кола завдань у конкретному контексті. За допомогою CMF можна створювати будь-які інтернет-додатки, починаючи гостьовими книгами, закінчуючи інтернет-магазинами і веб-сервісами.

   Маючи фахівців, які знають архітектуру використовуваної CMF-системи, стає можливим, відносно легко, розширювати можливості системи, проводити аудити безпеки і т.д.  
  
**Архітектура CMF-системи**

   У попередніх розділах вже було сказано дуже багато про архітектуру. Вам може здатися, що архітектурні стандарти абсолютно не потрібні. Але необхідно розуміти, що подібна «диктатура» не має на меті обмеження програміста у прийнятті рішень. Навпаки це зроблено для забезпечення максимальної гнучкості архітектури та можливості її безболісної зміни. Безумовно, в жертву таким якостям доводиться приносити простоту та прозорість системи.

   Питання архітектури дуже складні за своєю суттю, і навіть багато фахівців не можуть сказати за п'ять хвилин нічого зрозумілого із приводу якихось конкретних рішень. Але, незважаючи на таку велику контекстну залежність архітектури від типу додатка, існують добре вивчені і перевірені варіанти вирішення архітектурних проблем. Ці варіанти рішення носять назву «шаблони проектування» (Design Patterns). У тому чи іншому вигляді шаблони проектування можуть бути застосовані у всіх додатках. Виявлення оптимальних реалізацій шаблонів становить невід'ємну частину роботи над framework-системою (я б сказав, що справжня framework-система просякнута духом патернів проектування).

   Шаблони проектування існують для всіх основних типів завдань, що виконуються CMF-системою. Рішення цих завдань вимагають продуманої стандартизації (зрозуміло, в рамках проекту). Таких завдань декілька:  
• Обробка запиту ІС 1);  
• Організація предметної області ІВ;  
• Організація подання ІС;  
• Організація допоміжних підсистем;

   Завдання, які я виділив, занадто умовні, що б вважати їх формальним списком завдань framework-системи. Цей список наведений, що б Ви могли зрозуміти, в яких напрямках розробники концентрують свої зусилля.  
  
**Обробка запиту**

   Підсистема обробки запиту зіставляє запит клієнта з дією, що виконується системою. Запити до системи можуть бути досить «різношерстими». Вони відрізняються як по вигляду, так і за смисловим навантаженням. Це залежить від типу додатка. Самі механізми зіставлення і їх дії можуть змінюватися під час супроводження проекту. Ці вимоги диктують розробникам CMF-системи необхідність створення зручного механізму аналізу та обробки запитів. У разі якщо розробники справляються зі своїм завданням, то додатки, побудовані на базі їх framework-системи, будуть красивими і легко запам'ятовуються адресами кшталт «http://www.server.com/news/2005-02-03» замість «http : / / www.server.com/index.php?module=news&action=show&date=2005-02-03 ». Безумовно, краса запиту не єдине якість, якого домагаються розробники. Гнучкі механізми зіставлення запиту з дією грають дуже важливу роль, так  це дуже зміна частина системи.  
  
**Організація предметної області**

   У кожної інформаційної системи є предметна область. Це набір термінів, об'єктів і правил, якими оперує додаток. Організація предметної області, одна з найскладніших завдань, яка сьогодні стоїть перед розробниками. У переважній більшості випадків функціонування предметної області забезпечують реляційні бази даних і об'єктно-орієнтовані технології відображення. Реляційний і об'єктно-орієнтований підхід геніальні окремо. Проте їх композиція, при невмілому поводженні, перетворює архітектуру інформаційної системи в купу мотлоху, розібратися в якій буде важко навіть досвідченому фахівцеві.  
  
**Організація подання**

**Уявлення** - це підсистема відображення даних. З її допомогою логіка предметної області відокремлюється від логіки відображення даних. Уявлення - це найстабільніша частина інформаційної системи. Відображення даних може мінятися дуже часто, на відміну від самих даних і методів їх обробки. Тому framework-система повинна надати зручні та гнучкі механізми роботи з логікою відображення. Для вирішення цього завдання використовуються шаблонні системи , чиє завдання полягає у відділенні логіки відображення і укладання її в окремі файли (шаблони відображення), які можна редагувати окремо від усіх інших частин системи. Завдяки цьому, роботу над проектом можна ефективно розпаралелити (Організація предметної області → програміст + адміністратор БД, Організація подання → верстальник + дизайнер).  
  
**Організація допоміжних підсистем**

   Під допоміжними підсистемами мається на увазі набір архітектурних рішень, покликаних полегшити працю програміста. Сюди можна віднести реалізації патернів загального призначення, які безпосередньо не відносяться до інших підсистем. Зокрема до допоміжних підсистем відносяться такі поняття як резолверів, хендла, різний реєстр (и), спостерігачі і т.д. Ці речі можуть бути використані в будь-якій іншій підсистемі для вирішення виникаючих проблем.

   Наприклад, патерн singletone (одиночка) може бути використаний для підтримки декількох інстанцій об'єкта в одиничному екземплярі. Це завдання носить суто допоміжний характер і не може бути віднесено безпосередньо до рівня бізнес-логіки або будь-якого іншого.

   Проте не варто недооцінювати важливість прийняття рішень по відношенню до цієї підсистемі. Від того наскільки зручними та ефективними будуть реалізації допоміжних патернів, залежить те, наскільки зручно буде програмувати інші підсистеми, і наскільки ефективно вони будуть працювати. Код, написаний у цій підсистемі, багато в чому визначає код, який буде писати програміст, що користується цим framework'ом.  
  
**Маленькі бібліотеки**

   Один з ворогів повторного використання коду - той факт, що люди не складають з свого коду бібліотеки. Клас багаторазового використання може бути похований у директорії однієї з програм і може ніколи не випробувати хвилюючого почуття реінкарнації в новому проекті. І тільки тому, що програміст не захотів винести цей клас (або класи) у бібліотеку.

   Одна з причин трагедії: люди не люблять маленькі бібліотеки. Є в маленьких бібліотеках щось таке, що люди вважають неправильним. Придушіть в собі це почуття. Комп'ютеру абсолютно все одно, скільки у вас бібліотек.

   Якщо ви написали код, який можна використовувати повторно, але він не вписується в вашу бібліотеку, створіть нову.   
  
**Тримайте свою базу бібліотек [репозиторій]**

   Більшість компаній не має ніякого поняття, який код у них є. І більшість програмістів до цих пір не повідомляють про те, що вони зробили і не цікавляться тим, що вже написано. Репозиторії покликані змінити ситуацію на краще.

   В ідеальному світі програміст міг би зайти на сайт, подивитися по каталогу або пошуком знайти потрібний пакет бібліотек і закачати собі. Якщо ви можете налагодити таку систему, в якій програмісти на добровільній основі будуть підтримувати базу вихідників - це прекрасно. Якщо ви заведете бібліотекаря, який відстежує коефіцієнт повторного використання, то це просто розкішно.

   Інший спосіб - *автоматична генерація сховища з початкових кодів*. Досягається подібний ефект через використання стандартних заголовків для класів, методів, бібліотек і різних підсистем. Такі заголовки служать водночас технічним керівництвом і пунктами в списку репозиторія.  
  
**Файли, що включаються**

   Можливість повторного використання існуючого коду є дуже важливим, тому що може зберегти час і гроші і сприяти узгодженості. Припустимо, що сайт Web містить текстове меню, яке повторюється на кожній сторінці. Замість повторного кодування меню буде значно легше закодувати його один раз і динамічно включати вміст меню на кожну з окремих сторінок Web. Це можна зробити за допомогою так званого серверного включення файлу.

   Файли, що включаються можуть містити будь-який код XHTML або PHP і зазвичай зберігаються з розширенням. Inc, хоча можна використовувати також розширення. Php,. Txt, або. Htm. Вміст файлу, що включається кодується один раз і включається в будь-яку необхідну кількість сторінок PHP. Якщо під файлом, що включається робиться зміна, то оновлення автоматично відбивається на всіх сторінках PHP, які посилаються на цей файл.

   Нижче показаний приклад типового файлу, що включається, що містить інформацію про заголовок сторінки.  
  
   Header.inc  
   <h3> Welcome to WebBooks.Com </ h3>  
   Цей приклад показує файл, що включається з ім'ям header.inc. Файл містить текст "Welcome to WebBooks.Com", оточений тегом XHTML <h3>. Він створює заголовок третього рівня, який можна тепер включати на всі сторінки, які складають сайт WebBooks. Після створення файлу, що включається, його можна включити в сторінку PHP за допомогою однієї з наступних функцій:  
   require (ім'я\_файлу) - включає і перевіряє вказаний файл  
   include (ім'я\_файлу) - інший спосіб підключення файлів  
  
   У наступному прикладі файл header.inc включається в існуючу сторінку PHP:  
   home.php  
   <? Php  
   require ('header.inc');  
   echo "<p> This is the WebBooks site ...</ p>";  
   ?>  
  
   Функція require () викликає файл header.inc і перевіряє вміст файлу. Вміст потім виводиться, так ніби воно було частиною сторінки home.php. У цьому прикладі функція require () кодується вгорі сторінки, так як вона містить інформацію заголовка. Оператор require () можна, однак, включити в будь-якому місці документа PHP. Розташування функції require () визначає, де буде виводитися вміст файлу в контексті сторінки PHP.  
   Welcome to WebBooks.Com  
   This is the WebBooks site ...

   Важливо відзначити, що при використанні файлів, що включаються, які містять конфіденційну інформацію, таку, як паролі або інформацію про користувача, файли повинні зберігатися з використанням розширення. Php, а не. Inc або іншого нестандартного розширення. Файли, які застосовують нестандартні розширення файлів, можуть завантажуватися з сервера Web, а їх вміст можна переглядати як звичайний текст. Використання розширення. Php гарантує, що клієнт не зможе побачити вихідний код, сервер поверне тільки код XHTML.  
  
**Використання функцій**   
   Функції використовуються для розбиття великих блоків коду на менші, більш керовані одиниці. Міститься всередині функції код виконує певне завдання і повертає значення. PHP містить два типи функцій - визначені користувачем (або створені програмістом) і внутрішні (вбудовані функції), які є частиною визначення мови PHP. Цей розділ присвячений створенню та застосуванню певних функцій користувача.  
   Певні функції користувача створюються за допомогою ключового слова function. Вони особливо корисні у великих програмах PHP, так як можуть містити блоки коду, які можуть викликатися чи використовуватися в програмі, що дозволяє уникнути повторного переписування коду. Далі представлений приклад простої визначеної користувачем функції PHP:  
   function AddNumbers ($ num1, $ num2)  
   {

   echo "Це приклад функції PHP. Вона обчислює суму двох чисел і повертає  
   результат, що викликається у програмі ";

return $ num1 + $ num2;

}

    Певні функції користувача можуть викликатися в будь-якому місці блоку коду PHP. У PHP функція виконується при використанні в коді її імені. Після виклику функція отримує всі передані їй значення у формі параметрів, виконує певні завдання і повертає значення, що викликає програма. Простий приклад показаний нижче.  
   <? Php  
   function AddNumbers ($ num1, $ num2)

{  
   return $ num1 + $ num2;  
   }  
   echo "Сума 5 і 2 дорівнює". AddNumbers (5,2);  
   ?>  
   Проте певна на початку функція AddNumbers () викликається тільки пізніше в програмі. Виклик функції відбувається в операторі echo. Виводиться рядок "Сума 5 і 2 дорівнює". Ім'я функції з'єднується з рядком виведення, викликаючи тим самим функцію. Для функції передається два параметри - 5 і 2. Вони присвоюються параметрами функції $ num1 і $ num2. Параметри складаються, і викликається оператор return, щоб "повернути" значення або суму двох чисел в те місце в блоці коду PHP, який спочатку викликав функцію. Висновок результату показаний нижче:  
   Сума 5 і 2 дорівнює 7  
   Імена функцій слідують тим же правилам, що і змінні у PHP. Допустимі імена можуть починатися з букви або підкреслення, після чого може слідувати будь-які літери, цифри або підкреслення.  
  
**Патерни проектування**   
  
**Загальні відомості**

   В якості основи проектування інформаційних систем застосовуються "типові рішення" або "шаблони проектування" (Patterns).

**Шаблони проектування** (патерн, design pattern) - це багато разів застосовувана архітектурна конструкція, що надає рішення для загальної проблеми проектування в рамках конкретного контексту й описує значимість цього рішення.

   Патерн не є закінченим зразком проекту, який може бути прямо перетворений в код, скоріше це опис або зразок для того, як вирішити завдання, таким чином, щоб це можна було використовувати в різних ситуаціях. Об'єктно-орієнтовані шаблони часто показують відносини і взаємодії між класами або об'єктами, без визначення того, які кінцеві класи чи об'єкти додатки будуть використовуватися. Алгоритми не розглядаються як шаблони, так як вони вирішують завдання обчислення, а не проектування.

   У 1970-і роки архітектор Крістофер Олександр склав набір шаблонів проектування. В області архітектури ця ідея не отримала такого розвитку, як пізніше в області програмної розробки. Згідно з визначенням Крістофера Олександра: "Кожне типове рішення описує якусь повторювану проблему і ключ до її розгадки, причому таким чином, що ви можете користуватися цим ключем багаторазово, жодного разу не прийшовши до одного й того ж результату" .

   У 1987 році Кент Бек і Вард Каннігем взяли ідеї Олександра та розробили шаблони відповідно до розробки програмного забезпечення для розробки графічних оболонок мовою Smalltalk.

   У 1988 році Ерік Гамма почав писати докторську дисертацію при Цюріхському університеті про загальну переносимість цієї методики на розробку програм.

   У 1989-1991 роках Джеймс Коплін трудився над розробкою ідіом для програмування на C + + та опублікував у 1991 році книгу Advanced C + + Idioms. У цьому ж році Ерік Гамма закінчує свою докторську дисертацію і переїжджає до США, де у співробітництві з Річардом Хелмом, Ральфом Джонсоном і Джоном Вліссідсом публікує книгу Design Patterns - Elements of Reusable Object-Oriented Software. У цій книзі описані 23 шаблона проектування. Також команда авторів цієї книги відома громадськості під назвою Банда чотирьох (Gang of Four, часто скорочується до GoF). Саме ця книга стала причиною зростання популярності шаблонів проектування.

   Іншим видатним діячем у галузі проектування програмних систем, який підтримав використання патернів, є Мартін Фаулер, який написав книгу "Архітектура корпоративних програмних додатків" (Patterns of Enterprise Application Architecture). Як зазначив Мартін Фаулер у своїй книзі "збираючись скористатися типовими рішеннями, не забувайте, що вони тільки відправна точка, а не пункт призначення".

   У книзі Крейга Лармана "Застосування UML і шаблонів проектування" описано 9 шаблонів GRASP (General Responsibility Assignment Software Patterns, загальні зразки розподілу обов'язків) - патернів, використовуваних в об'єктно-орієнтованому проектуванні для вирішення спільних завдань за призначенням обов'язків класам і об'єктам. Кожен з них допомагає розв'язати певну проблему, що виникає при об'єктно-орієнтованому аналізі, і яка виникає практично в будь-якому проекті з розробки програмного забезпечення.

   Головна користь кожного окремого шаблону полягає в тому, що він описує рішення цілого класу абстрактних проблем. Також той факт, що кожен шаблон має своє ім'я, полегшує дискусію про абстрактні структури даних (ADT) між розробниками, так як вони можуть посилатися на відомі шаблони. Таким чином, за рахунок шаблонів проводиться уніфікація термінології, назв модулів і елементів проекту.

   Правильно сформульований шаблон проектування дозволяє, відшукавши вдале рішення, користуватися ним знову і знову.

   Однак іноді шаблони консервують громіздку і малоефективну систему понять, розроблену вузькою групою. Коли кількість шаблонів зростає, перевищуючи критичну складність, виконавці починають ігнорувати шаблони і всю систему, з ними пов'язану. Нерідко шаблонами замінюється відсутність або недоліки документації, яка складна програмному середовищі.

   Є думка, що сліпе застосування шаблонів з довідника, без осмислення причин і передумов виділення кожного окремого шаблону, уповільнює професійне зростання програміста. Люди, які дотримуються цієї думки, вважають, що знайомитися зі списками шаблонів треба тоді, коли "доріс" до них в професійному плані - і не раніше. Хороший критерій потрібного ступеня професіоналізму - виділення шаблонів самостійно, на підставі власного досвіду. При цьому, зрозуміло, знайомство з теорією, пов'язаної з шаблонами, корисно на будь-якому рівні професіоналізму і направляє розвиток програміста в правильну сторону. Сумніву піддається тільки використання шаблонів "за довідником".

   Шаблони можуть пропагувати погані стилі розробки додатків, і часто сліпо застосовуються.  
   **Шаблони проектування класифікують наступним чином:**  
    • Патерни проектування класів / об'єктів  
    o Структурні патерни проектування класів / об'єктів

* Адаптер (Adapter) - GoF
* Декоратор (Decorator) або Оболонка (Wrapper) - GoF
* Заступник (Proxy) або Сурогат (Surrogate) - GoF
* Інформаційний експерт (Information Expert) - GRASP
* Компонувальник (Composite) - GoF
* Міст (Bridge), Handle (описувач) або Тіло (Body) - GoF
* Низька зв'язаність (Low Coupling) - GRASP
* Пристосуванець (Flyweight) - GoF
* Стійкий до змін (Protected Variations) - GRASP
* Фасад (Facade) – GoF

о Патерни проектування поведінки класів / об'єктів

* Інтерпретатор (Interpreter) - GoF
* Ітератор (Iterator) або Курсор (Cursor) - GoF
* Команда (Command), Дія (Action) або Транзакція (Транзакція) - GoF
* Спостерігач (Observer), Опублікувати - підписатися (Publish - Subscribe) або Delegation Event Model - GoF
* Не розмовляйте з невідомими (Don't talk to strangers) - GRASP
* Відвідувач (Visitor) - GoF
* Посередник (Mediator) - GoF
* Стан (State) - GoF
* Стратегія (Strategy) - GoF
* Зберігач (Memento) - GoF
* Ланцюжок обов'язків (Chain of Responsibility) - GoF
* Шаблонний метод (Template Method) - GoF
* Високе зачеплення (High Cohesion) - GRASP
* Контролер (Controller) - GRASP
* Поліморфізм (Polymorphism) - GRASP
* Штучний (Pure Fabrication) - GRASP
* Перенаправлення (Indirection) - GRASP

о Твірні патерни проектування

* Абстрактна фабрика (Abstract Factory, Factory), ін. назва Інструментарій (Kit) - GoF
* Одинак (Singleton) - GoF
* Прототип (Prototype) - GoF
* Творець примірників класу (Creator) - GRASP
* Будівельник (Builder) - GoF
* Фабричний метод (Factory Method) або Віртуальний конструктор (Virtual Constructor) - GoF

o Архітектурні системні патерни  
o Структурні патерни

* Репозиторій
* Клієнт / сервер
* об'єктно - орієнтований, Модель предметної області (Domain Model), модуль таблиці (Data Mapper)
* Багаторівнева система (Layers) чи абстрактна машина
* Потоки даних (конвеєр або фільтр)

o Патерни управління

* Патерни централізованого управління
* Виклик - повернення (сценарій транзакції - окремий випадок)
* Диспетчер
* Патерни управління, засновані на подіях
* Передача повідомлень
* Керування перериваннями
* Патерни, що забезпечують взаємодію з базою даних
* Активний запис (Active Record)
* Одиниця роботи (Unit Of Work)
* Завантаження на вимогу (Lazy Load)
* Колекція об'єктів (Identity Map)
* Безліч записів (Record Set)
* Успадкування з однією таблицею (Single Table Inheritance)
* Успадкування з таблицями для кожного класу (Class Table Inheritance)
* Оптимістичне автономне блокування (Optimistic Offline Lock)
* Відображення з допомогою зовнішніх ключів
* Відображення з допомогою таблиці асоціацій (Association Table Mapping)
* Песимістичне автономне блокування (Pessimistic Offline Lock)
* Поле ідентифікації (Identity Field)
* Перетворювач даних (Data Mapper)
* Збереження сеансу на стороні клієнта (Client Session State)
* Збереження сеансу на стороні сервера (Server Session State)
* Шлюз запису даних (Row Data Gateway)
* Шлюз таблиці даних (Table Data Gateway)
* Патерни, призначені для представлення даних у Web
* Модель-представлення-контролера (Model View Controller)
* Контролер сторінок (Page Controller)
* Контролер запитів (Front Controller)
* Представлення за шаблоном (Template View)
* Представлення з перетворенням (Transform View)
* Двоетапне подання (Two Step View)
* Контролер додатка (Application Controller)

o Патерни інтеграції корпоративних інформаційних систем

* Структурні патерни інтеграції
* Взаємодія "точка - точка"
* Взаємодія "зірка" (інтегруюча середа)
* Змішаний спосіб взаємодії
* Патерни за методом інтеграції
* Інтеграція систем за даними (data-centric)
* Функціонально-центричний (function-centric) підхід
* Об'єктно-центричний (object-centric)
* Інтеграція на основі єдиної понятійної моделі предметної області (concept-centric)
* Патерни інтеграції за типом обміну даними
* Файловий обмін
* Загальна база даних
* Віддалений виклик процедур
* Обмін повідомленнями

Також на сьогоднішній день існує ряд інших шаблонів:  
• Carrier Rider Mapper, надання доступу до збереженої інформації;  
• аналітичні шаблони, описують основний підхід для складання вимог для програмного забезпечення (requirement analysis) до початку самого процесу програмної розробки;  
• комунікаційні шаблони, описують процес спілкування між окремими учасниками / співробітниками організації;  
• організаційні шаблони, описують організаційну ієрархію підприємства / фірми;  
• Анти-патерни (Anti-Design-Patterns) описують як не слід поступати при розробці програм, показуючи характерні помилки в дизайні і в реалізації;  
Розглянемо докладніше деякі з патернів проектування.  
Огляд патернів   
Патерни проектування класів / об'єктів   
*Структурні патерни (Structural)*  
До структурних патернів відносяться:  
• Адаптер (Adapter) - GoF;  
• Декоратор (Decorator) або Оболонка (Wrapper) - GoF;  
• Заступник (Proxy) або Сурогат (Surrogate) - GoF;  
• Інформаційний експерт (Information Expert) - GRASP;  
• Компонувальник (Composite) - GoF;  
• Міст (Bridge), Handle (описувач) або Тіло (Body) - GoF;  
• Низька зв'язаність (Low Coupling) - GRASP;  
• Пристосуванець (Flyweight) - GoF;  
• Стійкий до змін (Protected Variations) - GRASP;  
• Фасад (Facade) - GoF.  
Наведемо приклади 2-х  даних патернів (табл. 1).

|  |  |
| --- | --- |
| **Таблиця 1. Приклади структурних патернів класів/об'єктів** | |
| Компонувальник **(Composite) – Go** | |
| **Проблема** | Як обробляти групу або композицію структур об'єктів одночасно? |
| **Рішення** | Визначити класи для композитних і атомарних об'єктів таким чином, щоб вони реалізовували той самий інтерфейс. 04_10 |
| **Фасад (Facade) – Go** | |
| **Проблема** | Як забезпечити уніфікований інтерфейс із набором розрізнених реалізацій або інтерфейсів, наприклад, з підсистемою, якщо небажано високе зв'язування із цією підсистемою або реалізація підсистеми може змінитися? |
| **Рішення** | Визначити одну крапку взаємодії з підсистемою – фасадний об'єкт, що забезпечує загальний інтерфейс із підсистемою й покласти на нього обов'язок по взаємодії з її компонентами. Фасад – це зовнішній об'єкт, що забезпечує єдину крапку входу для служб підсистеми. Реалізація інших компонентів підсистеми закрита й не видна зовнішнім компонентам. Фасадний об'єкт забезпечує реалізацію патерна "Стійкий до змін" з погляду захисту від змін у реалізації підсистеми. 04_11 |

*Патерни проектування поведінки (Behavioral)*  
До поведінковим патернів належать:  
• Інтерпретатор (Interpreter) - GoF;  
• Ітератор (Iterator) або Курсор (Cursor) - GoF;  
• Команда (Command), Дія (Action) або Транзакція (Транзакція) - GoF;  
• Спостерігач (Observer), Опублікувати - підписатися (Publish - Subscribe) або Delegation Event Model - GoF;  
• Не розмовляйте з невідомими (Don't talk to strangers) - GRASP;  
• Відвідувач (Visitor) - GoF;  
• Посередник (Mediator) - GoF;  
• Стан (State) - GoF;  
• Стратегія (Strategy) - GoF;  
• Зберігач (Memento) - GoF;  
• Ланцюжок обов'язків (Chain of Responsibility) - GoF;  
• Шаблонний метод (Template Method) - GoF;  
• Високе зачеплення (High Cohesion) - GRASP;  
• Контролер (Controller) - GRASP;  
• Поліморфізм (Polymorphism) - GRASP;  
• Штучний (Pure Fabrication) - GRASP;  
• Перенаправлення (Indirection) - GRASP.  
Наведемо приклади 3-х  даних патернів (табл. 2).

|  |  |
| --- | --- |
| **Таблиця 2. Приклади поведінкових патернів класів/об'єктів** | |
| Ітератор (Iterator) або Курсор (Cursor) – GoF | |
| **Проблема** | Складений об'єкт, наприклад, список, повинен надавати доступ до своїх елементів (об'єктів), не розкриваючи їхню внутрішню структуру, причому перебирати список потрібно по-різному залежно від завдання. |
| **Рішення** | Створюється клас "Ітератор", який визначає інтерфейс для доступу і перебору елементів, "КонкретнийІтератор" реалізує інтерфейс класу "Ітератор" і стежить за поточною позицією при обході "Агрегат". "Агрегат" визначає інтерфейс для створення об'єкту - ітератора. "КонкретнийАгрегат" реалізує інтерфейс створення ітератора і повертає екземпляр класу "КонкретнийІтератор", "КонкретнийІтератор" відстежує поточний об'єкт в агрегаті і може обчислити наступний об'єкт при переборі. Даний патерн підтримує різні способи перебору агрегату.  04_12 |
| **Відвідувач (Visitor) – Go** | |
| **Проблема** | Над кожним об'єктом деякої структури виконується операція. Визначити нову операцію, не змінюючи класи об'єктів. |
| **Рішення** | Клієнт, що використовує даний патерн, повинен створити об'єкт класу "КонкретнийВідвідувач", а потім відвідати кожен елемент структури. "Відвідувач", оголошує операцію "Відвідати" для кожного класу "КонкретнийЕлемент" (ім'я та сигнатура даної операції ідентифікують клас, елемент якого відвідує "Відвідувач" - тобто, відвідувач може звертатися до елемента прямо). "КонкретнийВідвідувач" реалізує всі операції, оголошені в класі "Відвідувач". Кожна операція реалізує фрагмент алгоритму, визначеного для класу відповідного об'єкта в структурі. Клас "КонкретнийВідвідувач" надає контекст для цього алгоритму і зберігає його локальний стан. "Елемент" визначає операцію "Прийняти", яка приймає "Відвідувача" як аргумент, "КонкретнийЕлемент" реалізує операцію "Прийняти", яка приймає "Відвідувача" як аргумент. "СтруктураОб’єкта" може перерахувати свої аргументи і надати відвідувачеві високорівневий інтерфейс для відвідування своїх елементів. Даний патерн логічно використовувати, якщо в структурі присутні об'єкти багатьох класів з різними інтерфейсами, і необхідно виконати над ними операції, що залежать від конкретних класів, або якщо класи, встановлюють структуру об'єктів, змінюються рідко, але нові операції над цією структурою додаються часто. Даний патерн спрощує додавання нових операцій, об'єднує споріднені операції в класі "Відвідувач". У даному патерні утруднено додавання нових класів "КонкретнийЕлемент", оскільки потрібне оголошення нової абстрактної операції в класі "Відвідувач". 04_13 |
| **Стан (State) – Go** | |
| **Проблема** | Варіювати поводження об'єкта залежно від його внутрішнього стану |
| **Рішення** | Клас "Контекст" делегує залежать від стану запити поточному об'єкту "КонкретнийСтан" (зберігає екземпляр підкласу "КонкретнийСтан", яким визначається поточний стан), і визначає інтерфейс, що представляє інтерес для клієнтів. " КонкретнийСтан" реалізує поведінку, асоційовану з якимось станом об'єкта "Контекст". "Стан" визначає інтерфейс для інкапсуляції поведінки, асоційованого з конкретним екземпляром "Контексту". Даний патерн локалізує залежне від стану поведінку і ділить його на частини, що відповідають станам, переходи між станами стають явними. 04_14 |

**Твірні патерни проектування**  
*До породжувальних патернів належать*:  
• Абстрактна фабрика (Abstract Factory, Factory) - GoF;  
• Одинак (Singleton) - GoF;  
• Прототип (Prototype) - GoF;  
• Творець примірників класу (Creator) - GRASP;  
• Будівельник (Builder) - GoF;  
• Фабричний метод (Factory Method) або Віртуальний конструктор (Virtual Constructor) - GoF.  
Наведемо приклади 2-х  даних патернів (табл. 3) .

|  |  |
| --- | --- |
| **Таблиця 3. Приклади що породжують патерни класів/об'єктів** | |
| **Одинак (Singleton) – Go** | |
| **Проблема** | Який спеціальний клас повинен створювати "Абстрактну фабрику" і як одержати до неї доступ? Необхідний лише один екземпляр спеціального класу, різні об'єкти повинні звертатися до цього екземпляра через єдину точку доступу. |
| **Рішення** | Створити клас і визначити статичний метод класу, що повертає цей єдиний об'єкт. Розумніше створювати саме статичний екземпляр спеціального класу, а не оголосити необхідні методи статичними, оскільки при використанні методів екземпляра можна застосувати механізм спадкування й створювати підкласи. Статичні методи в мовах програмування не поліморфні й не допускають перекриття в похідних класах. Рішення на основі створення екземпляра є більше гнучким, оскільки згодом може знадобитися вже не єдиний екземпляр об'єкта. 04_15 |
| **Фабричний метод (Factory Method) або Віртуальний конструктор (Virtual Constructor) – Go** | |
| **Проблема** | Визначити інтерфейс для створення об'єкту, але залишити підкласам рішення про те, який клас інстанціювати, тобто, делегувати інстанціювання підкласами. |
| **Рішення** | Абстрактний клас "Творець" оголошує Фабричний Метод, який повертає об'єкт типу "Продукт" (абстрактний клас, що визначає інтерфейс об'єктів, що створюються фабричним методом). "Творець" також може визначити реалізацію за замовчуванням Фабричного Методу, який повертає "КонкретнийПродукт". "КонкретнийТворець" заміщає Фабричний Метод, який повертає об'єкт "КонкретнийПродукт". "Творець" "покладається" на свої підкласи у визначенні Фабричного Методу, що повертає об'єкт "КонкретнийПродукт". Даний патерн позбавляє проектувальника від необхідності вбудовувати в код залежать класів від програми. Однак при застосуванні даного патерну виникає додатковий рівень підкласів. 04_16 |

Архітектурні системні патерни   
*Структурні патерни*  
До структурних патернів належать:  
• Репозиторій;  
• Клієнт / сервер;  
• об'єктно - орієнтований, Модель предметної області (Domain Model), модуль таблиці (Data Mapper);  
• Багаторівнева система (Layers) чи абстрактна машина;  
• Потоки даних (конвеєр або фільтр).  
Наведемо приклад одного із даних патернів (табл. 4).

|  |  |
| --- | --- |
| Таблиця 4. Приклади стуктурних патернів архітектури | |
| **Багаторівнева система (Layers) або абстрактна машина** | |
| **Опис** | Відповідно до патерна "Багаторівнева система" структурні елементи системи організуються в окремі рівні з взаємопов'язаними обов'язками таким чином, щоб на нижньому рівні розташовувалися низькорівневі служби та служби загального призначення, а на більш високих - об'єкти рівня логіки додатка. При цьому взаємодія і зв'язування рівнів відбувається зверху вниз. Зв'язування об'єктів знизу вгору слід уникати. На малюнку показані типові рівні логічної архітектури системи. 04_17  Шар подання охоплює все, що має відношення до спілкування користувача з системою. До основних функцій шару подання відноситься відображення інформації й інтерпретація користувачем команд з перетворенням їх у відповідні операції в контексті домену (бізнес - логіка) і джерела даних. Джерело даних - підмножина функцій, що забезпечує взаємодію зі сторонніми системами, які виконуються. На відміну від архітектурного патерну "Клієнт - сервер", шари зовсім не обов'язково повинні розташовуватися на різних машинах. Багаторівнева система може бути розроблена покрокова (Ітеративний). Недоліками даного патерну є: • Зміна вихідного коду тягне за собою переробку всіх елементів системи, оскільки всі елементи системи тісно пов'язані один з одним. • Логіка програми тісно пов'язана з інтерфейсом користувача - важко міняти інтерфейс або принципи реалізації логіки. Через високу пов'язаність, роботу з реалізації системи складно розділити між розробниками і, крім того, складно модифікувати функції додатку або переходити на нові технології. |

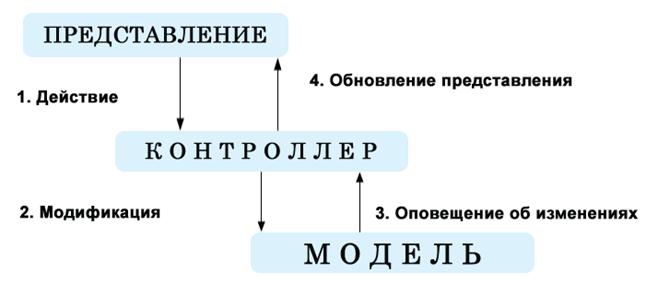
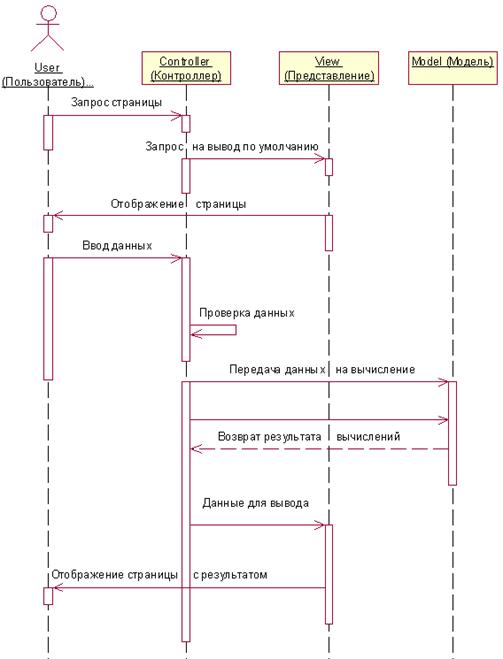
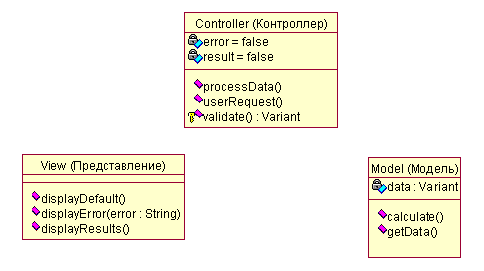
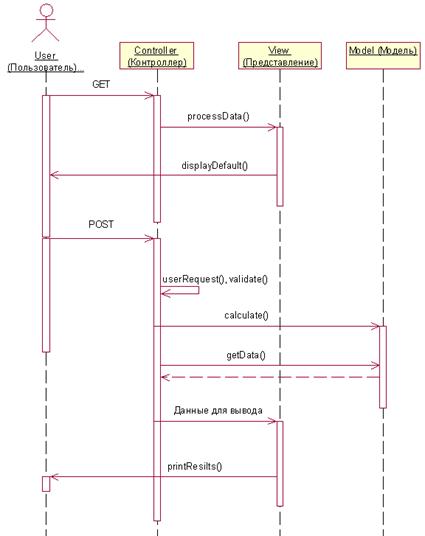
Патерни управління   
До патернів управління відносяться :  
•

* Патерни централізованого управління
* Виклик - повернення (сценарій транзакції - окремий випадок)
* Диспетчер
* Патерни управління, засновані на подіях
* Передача повідомлень
* Керування перериваннями
* Патерни, що забезпечують взаємодію з базою даних
* Активний запис (Active Record)
* Одиниця роботи (Unit Of Work)
* Завантаження на вимогу (Lazy Load)
* Колекція об'єктів (Identity Map)
* Безліч записів (Record Set)
* Успадкування з однією таблицею (Single Table Inheritance)
* Успадкування з таблицями для кожного класу (Class Table Inheritance)
* Оптимістичне автономне блокування (Optimistic Offline Lock)
* Відображення з допомогою зовнішніх ключів
* Відображення з допомогою таблиці асоціацій (Association Table Mapping)
* Песимістичне автономне блокування (Pessimistic Offline Lock)
* Поле ідентифікації (Identity Field)
* Перетворювач даних (Data Mapper)
* Збереження сеансу на стороні клієнта (Client Session State)
* Збереження сеансу на стороні сервера (Server Session State)
* Шлюз запису даних (Row Data Gateway)
* Шлюз таблиці даних (Table Data Gateway)

У даній лекції приклади патернів управління розглядатися не будуть.  
Патерни, призначені для представлення даних у Web   
До патернів, призначених для представлення даних у Web, відносяться:  
• Модель-представлення-контролера (Model View Controller);  
• Контролер сторінок (Page Controller);  
• Контролер запитів (Front Controller);  
• Представлення за шаблоном (Template View);  
• Представлення з перетворенням (Transform View);  
• Двоетапне подання (Two Step View);  
• Контролер додатка (Application Controller).  
Наведемо приклад 4-х  патернів (табл. 5) [18].

|  |  |
| --- | --- |
| Таблиця 5. Приклади патернів, призначених для представлення даних у Web | |
| **Модель-подання-контролера (Model View Controller)** | |
| **Опис** | 04_18  Типове рішення модель-подання-контролера має на увазі виділення трьох окремих ролей. **Модель** - це об'єкт, що дає деяку інформацію про домен. У моделі немає візуального інтерфейсу, вона містить у собі всі дані і поведінку, не пов'язані з призначеним для користувача інтерфейсом. В об'єктно-орієнтованому контексті найбільш "чистою" формою моделі є об'єкт моделі предметної області. В якості моделі можна розглядати і сценарій транзакції, якщо він не містить в собі ніякої логіки, пов'язаної з призначеним для користувача інтерфейсом. Подібне визначення не дуже розширює поняття моделі, однак повністю відповідає розподілу ролей у розглянутому типовому рішенні. Представлення відображає вміст моделі засобами графічного інтерфейсу. Таким чином, якщо модель - це об'єкт покупця, відповідне подання може бути фреймом з купою елементів управління або HTML-сторінкою, заповненою інформацією про покупця. Функції подання полягають тільки у відображенні інформації на екрані. Всі зміни інформації обробляються третім "учасником" системи - контролером. Контролер одержує вхідні дані від користувача, виконує операції над моделлю і вказує подання на необхідність відповідного оновлення. У цьому плані графічний інтерфейс можна розглядати як сукупність подання та контролера. Говорячи про типове рішення модель-подання-контролер, не можна не підкреслити два основні типи поділу: відділення подання від моделі та відділення контролера від подання. Відділення подання від моделі - це один з фундаментальних принципів проектування програмного забезпечення. Наявність такого поділу дуже важливо з ряду причин: • Представлення і модель відносяться до абсолютно різних сфер програмування. • Користувачі хочуть, щоб, залежно від ситуації, одна і та ж інформація могла бути відображена різними способами. • Об'єкти, що не мають візуального інтерфейсу, набагато легше тестувати, ніж об'єкти з інтерфейсом. Ключовим моментом у відділенні подання від моделі є спрямування залежностей: представлення залежить від моделі, але модель не залежить від подання. Це означає, що зміна уявлення не вимагає зміни моделі. Відділення контролера від подання. Класичним прикладом необхідності такого поділу є підтримка редагованої і не редагованої поведінки. Цього можна досягти за наявності одного подання і двох контролерів (для двох варіантів використання), де контролери є стратегіями, використовуваними поданням. Тим часом на практиці в більшості систем кожному поданню відповідає тільки один контролер, тому поділ між ними не проводиться. Про це рішення згадали тільки з появою Web-інтерфейсів, де відділення контролера від подання виявилося надзвичайно корисним. . |
| **Контролер сторінок (Page Controller)** | |
| **Опис** | 04_19  В основі контролера сторінок лежить ідея створення компонентів, які будуть виконувати роль контролерів для кожної сторінки Веб-сайту. На практиці кількість контролерів не завжди в точності відповідає кількості сторінок, оскільки інколи при натисканні на посилання відкриваються сторінки з різним динамічним вмістом. Якщо говорити більш точно, контролер необхідний для кожної дії, під яким мається на увазі клацання на кнопці або гіперпосиланні. Контролер сторінок може бути реалізований у вигляді сценарію (сценарію CGI) або сторінки сервера (ASP, PHP, JSP і т.д.). Використання сторінки сервера зазвичай передбачає поєднання в одному файлі контролера сторінок та подання за шаблоном. Це добре для подання за шаблоном, але не дуже підходить для контролера сторінок, оскільки значно ускладнює правильне структурування цього компоненту. Дана проблема не настільки важлива, якщо сторінка застосовується тільки для простого відображення інформації. Тим не менш, якщо використання сторінки припускає наявність логіки, пов'язаної з отриманням даних користувача або вибором подання для відображення результатів, сторінка сервера може заповнитися кодом впровадженого сценарію. Щоб уникнути подібних проблем, можна скористатися допоміжним об'єктом (helper object). При отриманні запиту сторінка сервера викликає допоміжний об'єкт для обробки всієї наявної логіки. У залежності від ситуації, допоміжний об'єкт може повернути управління первісної сторінки сервера або ж звернутися до іншої сторінки сервера, щоб вона виступила як представлення. У цьому випадку обробником запитів є сторінка сервера, проте велика частина логіки контролера укладена у допоміжному об'єкті. Можливою альтернативою описаного підходу є реалізація обробника і контролера у вигляді сценарію. У цьому випадку під час вступу запиту веб-сервер передає управління сценарієм; сценарій виконує всі дії, покладені на контролер, після чого відображає отримані результати з допомогою потрібного подання. **Нижче перераховані основні обов'язки контролера сторінок.** • Проаналізувати адресу URL і витягти дані, введені користувачем у відповідні форми, щоб зібрати всі відомості, необхідні для виконання дії. • Створити об'єкти моделі і викликати їх методи, необхідні для обробки даних. • Визначити подання, яке повинно бути використано для відображення результатів, і передати йому необхідну інформацію, отриману від моделі. Контролер сторінок не обов'язково повинен являти собою єдиний клас, зате всі класи контролерів можуть використовувати одні й ті ж допоміжні об'єкти. Це особливо зручно, якщо веб-сервер повинен мати кілька обробників, що виконують аналогічні завдання. У цьому випадку використання допоміжного об'єкту дозволяє уникнути дублювання коду. При необхідності одні адреси URL можна обробляти за допомогою сторінок сервера, а інші - за допомогою сценаріїв. Ті адреси URL, у яких немає або майже немає логіки контролера, добре обробляти за допомогою сторінок сервера, тому що останні є простим механізмом, зручним для розуміння та внесення змін. Всі інші адреси, для обробки яких необхідна більш складна логіка, вимагають застосування сценаріїв. |
| **Контролер запитів (Front Controller)** | |
| **Опис** | 04_20  Контролер запитів обробляє всі запити, що надходять до Веб-сайту, і зазвичай складається з двох частин: Веб-обробника та ієрархії команд. Веб-обробник - це об'єкт, який виконує фактичне отримання POST або GET-запитів, що надійшли на Веб-сервер. Він витягає необхідну інформацію з адреси URL і вхідних даних запиту, після чого вирішує, яку дію необхідно ініціювати, і делегує його виконання відповідної команді. Веб-обробник зазвичай реалізується у вигляді класу, а не сторінки сервера, оскільки він не генерує жодних відгуків. Команди також є класами, а не сторінками сервера, більше того, їм не потрібно знати про наявність Веб-оточення, не дивлячись на те, що їм часто передається інформація з HTTP-запитів. У більшості випадків Веб-обробник - це досить проста програма, функції якої полягають у виборі потрібної команди. Вибір команди може відбуватися статично або динамічно. Статичний вибір команди - це проведення синтаксичного аналізу адреси URL і застосування умовної логіки, а динамічний - витяг деякого стандартного фрагмента адреси URL і динамічне створення екземпляра класу команди. Переваги статичного вибору команди перебувають у використанні явного коду, наявності перевірки часу компіляції і високої гнучкості можливих варіантів написання адрес URL. У свою чергу, використання динамічного підходу дозволяє додавати нові команди, не вимагаючи зміни Веб-обробника. При динамічному виборі команд ім'я класу команди можна помістити безпосередньо на адресу URL або скористатися файлом властивостей, який буде прив'язувати адреси URL до імен класів команд. Зрозуміло, це потребує створення додаткового файлу властивостей, однак дозволить легко і невимушено змінювати імена класів, не переглядаючи всі наявні на сервері Веб-сторінки. |
| **Подання за шаблоном (Template View)** | |
| **Опис** | 04_21  Основна ідея, що лежить в основі типового рішення уявлення за шаблоном, - вставка маркерів в текст готової статичної HTML-сторінки. При виклику сторінки для обслуговування запиту ці маркери будуть замінені результатами деяких обчислень (наприклад, результатами виконання запитів до бази даних). Подібна схема дозволяє створювати статичну частину сторінки за допомогою звичайних засобів, наприклад текстових редакторів, що працюють за принципом WYSIWYG, і не вимагає знання мов програмування. Для отримання динамічної інформації маркери звертаються до окремих програм. Подання за шаблоном використовується цілим рядом програмних засобів. Таким чином, завдання полягає не стільки в тому, щоб розробити дане рішення самому, скільки в тому, щоб навчитися його ефективно використовувати і познайомитися з можливими альтернативами. |

Патерни інтеграції корпоративних інформаційних систем   
**Структурні патерни інтеграції**  
До структурних патернів інтеграції належать :  
• Взаємодія "точка - точка";  
• Взаємодія "зірка" (інтегруюча середовище);  
• Змішаний спосіб взаємодії.  
У даній лекції приклади структурних патернів інтеграції розглядатися не будуть.  
**Патерни за методом інтеграції**  
До патернів за методом інтеграції належать:  
• Інтеграція систем за даними (data-centric);  
• Функціонально-центричний (function-centric) підхід;  
• Об'єктно-центричний (object-centric);  
• Інтеграція на основі єдиної понятійної моделі предметної області (concept-centric).  
**Патерни інтеграції за типом обміну даними**  
До патернів інтеграції за типом обміну даними відносяться :  
• Файловий обмін;  
• Загальна база даних;  
• Віддалений виклик процедур;  
• Обмін повідомленнями.  
У даній лекції приклади патернів інтеграції за типом обміну розглядатися не будуть.

Model-Viewer-Controller   
Стандартна схема архітектури «Модель-Вид-Контролер» зображена на наступному малюнку: (схема запозичена з книги «Ajax in action» видавничого дому «Вільямс»)  
Схема архітектури MVC   
  
Розберемо по пунктах дану схему.  
У шаблоні MVC, як випливає з назви, є три основних компоненти: Модель, Представлення, і Контролер.  
Представлення відповідає за відображення інформації, що надходить із системи або в систему.  
Модель є «суттю» системи і відповідає за безпосередні алгоритми, розрахунки тощо внутрішній устрій системи.  
Контролер є сполучною ланкою між «поданням» і «моделлю» системи, за допомогою якого і існує можливість зробити поділ між ними. Контролер отримує дані від користувача і передає їх в «модель». Крім того, він отримує повідомлення від моделі, і передає їх в «подання».  
Стосовно до інтернет-додатків існує думка, що частини контролера і подання об'єднані, тому що за відображення і одночасно за введення інформації відповідає браузер. З цим можна погодитися, а можна не погоджуватися і виділити  контролер в окрему частину, що ми і зробимо.  
Отже, домовимося:  
**Представлення.** Модуль виведення інформації. Це може бути шаблонізатор або що-небудь подібне, мета якого є тільки в поданні інформації у вигляді HTML на основі будь-яких готових даних.  
**Контролер**. Модуль управління введенням і виведенням даних. Даний модуль повинен стежити за переданими в систему даними (через форму, рядок запиту, cookie або будь-яким іншим способом) і на основі введених даних вирішити:  
• Передавати чи їх у модель  
• Вивести повідомлення про помилку і запросити повторне введення (змусити модуль уявлення оновити сторінку з урахуванням умов)  
Крім того, контролер зобов'язаний визначати тип даних, отриманих від моделі (чи є це готовий результат, відсутність  повідомлення про помилку) і передавати інформацію в модуль уявлення.  
**Модель.** Модуль, що відповідає за безпосередній розрахунок чого-небудь на основі отриманих від користувача даних. Результат, отриманий цим модулем, повинен бути переданий в контролер, і не повинен містити нічого, що відноситься до безпосереднього висновку (тобто має бути представлений у внутрішньому форматі програми).  
Досить складно з першого разу розібратися і зрозуміти. На це потрібен час і відповідний проект.  
Але насправді нічого надскладного в цьому немає.  
Уявімо собі як представлення будь-якого класу, який за допомогою шаблонізатора виводить результат чи повідомлення про помилку. На його вхід подається або масив з даними (об'єкт або що-небудь інше), або змінна, що містить текст з помилкою.  
В якості контролера буде виступати клас, що виробляє всі необхідні перевірки коректності даних і генерує повідомлення про помилки. Перевірку даних доцільно помістити саме в клас контролера, їх використовують досить часто. Як варіант, можна просто успадковувати клас контролера від більш загального класу, що реалізує перевірку вхідних даних за заданими правилами. Або, якщо так буде зручніше, включити в клас контролера клас або серію функцій перевірки даних.  
Цей же клас повинен передати дані, отримані в результаті роботи моделі, в клас представлення для виводу.  
Одними словами схему потоків даних у цій архітектурі пояснити складно, тому звернемося до мови UML і до діаграми послідовностей зокрема (незначні відступи від UML, прийняті в діаграмах, полягають в тому, що в деяких випадках разом з іменами сутностей або об'єктів, дані переказують в дужках).  
На цій діаграмі показана послідовність дій, а також послідовність переданих даних: від користувача, до користувача і між модулями.  
Схема відображає типовий процес виведення форми, заповнення її користувачем і повернення користувачеві результатів. Ніяких помилок в даному випадку не відбувається.  
**Діаграма послідовностей**  
  
Як видно з діаграми, звернення до моделі відбувається лише в разі надсилання користувачем вірних даних. На внутрішньому ж рівні додатку, модель відокремлена від подання та контролера. Контролер також відділений від моделі і уявлення, і його функція полягає в управлінні та перевірці.  
Тепер спробуємо скласти діаграму класів для більшої наочності.  
**Діаграма класів**  
  
Діаграма класів містить три класи, по одному для кожного компонента архітектури MVC. Для зручності, вони так і названі: Model, View, Controller.  
У поданні є три функції (хоча, цілком можливо обійтися тільки лише однією), які відповідають за відображення стану програми:  
displayDefault () - висновок форми за замовчуванням.  
displayError (error = false) - висновок форми з повідомленням про помилку,   
displayResults () - висновок результатів обчислень  
Контролер має не тільки методи, а й поля. З полями все просто: це помилка і результати обчислень. За замовчуванням їм задається значення false, що свідчить про те, що поки немає ні помилки, ні результатів.  
Три методи, присутні в контролері, служать для управління та перевірки. Метод для перевірки (validate ()) є необов'язковим, і цілком може бути відсутнім, якщо ніяких перевірок не потрібно.  
**Метод processData** () служить для виведення форми за замовчуванням, однак він включає також метод **userRequest** (), функціональність якого виконується лише в тому випадку, коли є введені користувачем дані. Саме метод **userRequest** () містить у собі функцію **validate** () (якщо дані не введені, отже, нема чого робити їх перевірку) і, крім того, повинен міститись виклик конструктора класу моделі.  
У моделі може міститися будь-яка кількість полів і методів. Однак два методи повинні бути обов'язковими (або навіть один. Як зручніше буде).  
**calculate** () - функція, яка виробляє основний розрахунок  
**getData** () - функція, яка повертає дані результату.  
Поділ функцій моделі швидше смислове. Цілком достатньо створити один метод, який буде і рахувати, і повертати результат.  
Повертаємося в метод **userRequest** () контролера. Після того, як у ньому був порахований результат і повернений в тому чи іншому вигляді, його можна сміливо віддавати на вхід функції **displayResults** () класу **View**. Однак зауважимо, що в принципі, можна віддавати на вхід уявлення і екземпляр класу моделі, якщо висновок зберігається в його полях, а їх багато (якщо лінь, так би мовити, створювати структуру, масив або ще якої-небудь об'ємний тип даних).  
Якщо функція validate () контролера виявила помилку, і встановила значення поля error в значення, відмінне від false, контролер сам викличе метод displayError () класу View.  
Тепер доречно навести ту ж саму діаграму послідовності, але замінивши в ній смислові значення, назвами функцій класів з відповідної діаграми.  
**Діаграма послідовностей**  
  
Отже, власне, у нас є три класи та алгоритм взаємодії між ними. Суть архітектурного шаблону MVC полягає в тому, щоб чітко розділити уявлення, управління і модель системи. Це дуже зручно, адже якщо що-небудь зміниться в одній з частин системи, інших частин ці зміни не торкнуться.  
Наприклад, у виставі ми можемо написати:  
/ / Це код на PHP  
public function displayDefault () {

echo "<p> Введіть ім'я:";  
echo "<input type='text' name='name' value=''>";

}  
А потім через місяць жахнутися, і перейти до використання шаблонізатора. Скажімо, smarty.  
Або, наприклад, в моделі змінити пару розрахункових формул. Або в контролері забрати пару обмежень, або змінити метод прийому-передачі даних. Якщо ж взяти до уваги принципи спадкування в ООП, то архітектура MVC стане ще зручніше. Скажімо, коли є дві форми, що виглядають однаково, але дещо відрізняються алгоритмами розрахунку.  
У висновку, хотілося б все ж таки привести деякий скелет коду на PHP для кращого засвоєння ідеї MVC.  
<?

/ \*\*  
\* Приклад реалізації MVC на PHP  
\*  
\*\* /

class Controller {

private $ error;  
private $ result;

function \_\_construct () {

$ This-> error = false;  
$ This-> result = false;

}

function processData () {

$ This-> userRequest ();

if ($ this-> error)  
View:: displayError ($ this-> error);  
else  
if ($ this-> result)  
View:: displayResults ($ this-> result);  
else  
View:: displayDefault ();

}

function userRequest () {

/ / Дані відправлені  
if (isset ($ \_POST ['send'])) {

$ This-> validate ();

if (! $ this-> error) {

/ / Основні обчислення  
$ Model = new Model ();  
$ Model-> calculate ($ \_POST ['name']);  
$ Result = $ model-> getData ();

/ / Перевірка на помилки в самій моделі  
if (! is\_array ($ result))  
$ This-> error = $ result;

else  
$ This-> result = $ result;

}  
}  
}

function validate () {

if (empty ($ \_POST ['name']))  
$ This-> error = 'Не введено ім'я!';  
else  
if (strlen (strval ($ \_POST ['name'])) <3)  
$ This-> error = 'Ім'я занадто коротке!';

}

} / / Class Controller

class View {

static function displayDefault () {

echo "<form method='POST' action=''>";  
echo "<p> Введіть ім'я:";  
echo "<input type='text' name='name' value=''>";  
echo "<input type='submit' name='send' value='Отправіть'>";  
echo "</ form>";

}

static function displayError ($ error) {

echo "<p> <b> Помилка: </ b> {$ error}";  
View:: displayDefault ();  
}

static function displayResults ($ results) {

echo "<p> <b> Результати: </ b>";  
echo "<p> Ваше ім'я <b>". $ results [0].  
"</ B> означає <i>". $ Results [1 ]."</ i> ";  
echo "<p> <a href ='".$\_ SERVER ['REQUEST\_URI'].  
"'> Дізнатися ще про одне імені </ a>";  
}

} / / Class View

class Model {

private $ data;

function \_\_construct () {

$ This-> data = false;

}

function calculate ($ name) {

$ This-> data [] = $ name;

$ Len = strlen ($ name);

if ($ len == 3)  
$ This-> data [] = 'стислість - сестра таланту';  
else  
if (($ len> 3) & & ($ len <6))  
$ This-> data [] = '... немає особливого значення';  
else  
$ This-> data [] = 'неймовірно багата фантазія батьків';  
}

function getData () {

if ($ this-> data)  
return $ this-> data;  
else  
return 'Обчислення не проведені!';

}

} / / Class Model

$ Controller = new Controller ();

$ Controller-> processData ();

?>  
Отже, ми отримали найпростішу MVC-систему. Виділимо позитивні і негативні сторони:  
**До мінусів можна віднести**  
• Збільшення обсягу коду  
• Необхідність дотримання заздалегідь заданого інтерфейсу  
• Для підтримки розробки потрібні більш кваліфіковані фахівці  
Остання вимога до нашого прикладу не відноситься, але для реальних систем воно досить актуально.  
**До плюсів віднесемо наступне:**  
• Безсумнівно більш гнучкий код  
• Можливість повторного використання кожної з трьох складових частин MVC  
• Безболісна заміна моделі (інші алгоритми розрахунку, способу зберігання даних і т.д.)  
• Досить просто перейти від одного подання, до іншого (від HTML до XML або JSON)  
Треба сказати, що код прикладу не ідеальний. У ньому є простори для рефакторингу (незважаючи на те, що він займає трохи більше ста рядків). Скажімо, у прикладі бере участь всього лише одна змінна, яка надходить від користувача (name), але що якщо їх буде багато?

**Model-View-Controller (MVC)** і **Model-View-Template (MVT)** — це два архітектурні патерни, які використовуються для організації коду у веб-додатках, але вони мають деякі відмінності в підході до обробки запитів та шаблонів.

**MVC (Model-View-Controller):**

• **Model**: Відповідає за дані програми. Модель взаємодіє з базою даних або іншими зовнішніми джерелами даних. У ній зберігаються всі бізнес-логіки та правила роботи з даними.

• **View**: Відповідає за відображення інформації користувачу. Це інтерфейс, з яким взаємодіє кінцевий користувач.

• **Controller**: Керує логікою програми. Він приймає запити від користувача, обробляє їх, звертається до моделі для отримання даних і передає їх до представлення (View) для візуалізації.

**Основні моменти:**

• **Controller** виконує основну роботу з обробки запитів та координації між **Model** і **View**.

• **View** у традиційній MVC — це готовий до виводу HTML-код, створений на основі даних, отриманих від контролера.

**MVT (Model-View-Template):**

Цей патерн використовується у фреймворку Django і є спрощеною версією MVC. Основні компоненти:

• **Model**: Як і в MVC, відповідає за дані програми і логіку доступу до бази даних.

• **View**: Відповідає за логіку програми. У Django це те, що обробляє запити та визначає, які дані потрібно передати в шаблони для візуалізації.

• **Template**: Шаблон використовується для створення HTML-коду. Це не просто статична сторінка, а динамічний файл, що використовує дані з моделі.

**Основні моменти:**

• **View** у MVT більше схоже на **Controller** у MVC, оскільки воно обробляє запити.

• **Template** замінює **View** у MVC і є місцем, де зберігаються HTML-шаблони для динамічного відображення даних.

**Основні відмінності між MVC і MVT:**

1. **Термінологія**: MVT має іншу назву для компонентів, але функціонально вони схожі.

• В MVT **View** — це логіка обробки запитів, а в MVC **View** — це інтерфейс, що відображає дані.

• У MVT використовується **Template** для формування HTML-сторінок, тоді як у MVC це робить **View**.

2. **Роль Controller/View**:

• В MVC **Controller** відповідає за обробку запитів і передачу даних до **View**.

• В MVT **View** виконує роль обробника запитів, а відображенням займаються **Templates**.

3. **Фреймворки**:

• MVC широко використовується в таких фреймворках, як Ruby on Rails, ASP.NET MVC, Spring MVC.

• MVT використовується у фреймворку Django, де цей патерн реалізований більш просто.

**Підсумок:**