УДК 004

ШВИДКА АБО ЯКІСНА РОЗРОБКА?

Т.В. Панченко

Київський національний університет імені Тараса Шевченка,
вул. Володимирська 60, Київ, Україна,
тел. 259-0519, e-mail: t.panchenko@infosoft.ua

Розглянуто технології швидкої розробки програмних систем, проаналізовано їх переваги та недоліки. Акцентовано важливість докладного вивчення і адекватного застосування. Показано, що швидкість розробки може бути не досягнена у випадку недбалого застосування технології, при цьому якість може бути значно втрачено. Стверджується, що якість програмного продукту важливіша за швидкість його створення.

The Rapid Application Development technologies were considered, their advantages and disadvantages were analysed. The importance of careful study and appropriate usage accentuated. It is shown that speed of development can be not achieved in the case of negligent usage of technology; herewith the quality can be significantly lost. It is argued that the quality of software is more important than the speed of its development.

Проблематика

У сучасному процесі розробки програмних систем надзвичайно багато уваги приділяється «технологіям», які дозволяють спростити, пришвидшити та, в решті решт, здешевити цей процес. Так, зокрема, термін «швидка розробка програм» (RAD) давно і міцно увійшов у вжиток в галузі програмування, а без використання фреймворків (Framework), шаблонів проектування (Design Patterns) та готових універсальних компонентів не мислима розробка великої системи. Але чи дійсно всі «технології» завжди допомагають зробити процес розробки простішим, а результат – тобто, програмний продукт – більш якісним, чи потрібно ще дещо, аби розробляти дійсно ефективні, надійні, та підтримувані (іншими словами – *якісні*) програми?

В процесі навчання студентів розробці програмних систем (в першу чергу для Web), а також в ході роботи над рядом проектів (також для Web), автор набув певного експериментально підтвердженого досвіду щодо застосування різних технологій розробки (Web Application Frameworks та ін.). Основні тези систематизовано та викладено у даній роботі. Спочатку проаналізовано окремі аспекти і особливості більш конкретних технологій (зокрема, застосування фреймворків та ін.), далі подано більш загальні висновки щодо технологій і їх застосування в цілому (в тому числі – навчання технологіям).

Фреймворки і швидкодія

Одна з перших характеристик будь-якої програмної системи, особливо – для Web, яка повинна цікавити власника (замовника, користувача і т. і.) – це її швидкодія, яка може виражатися у пропускній здатності системи (кількість операцій або опрацьованих запитів за одиницю часу, на певному апаратному забезпеченні, тощо) або іншою подібною (важливо: вимірюваною) величиною. Так, від швидкодії Web-системи залежить:

- кількість користувачів, які одночасно можуть працювати з нею (наприклад, кількість одночасних відвідувачів сайту),
- сатисфакція користувачів від взаємодії (прийнятність часу відповіді сайту на запити відвідувача суттєво впливає на те, чи користувач продовжить відвідування цього сайту і стане замовником, що платить, чи залишить його і, можливо, перейде до конкурентів та скористається їх послугами),
- як наслідок двох попередніх факторів – отримуваний від користувачів прибуток сайту, як через рекламну модель монетизації, так і прямий – від замовлень відвідувачів, що стали клієнтами.

Якщо співставити час відповіді системи на запит користувача з урахуванням навантаження (кількість одночасних відвідувачів) з очікуваннями користувача, можна спрогнозувати частку відмов користувачів та коефіцієнт конверсії у покупців – адже, дійсно, повільна робота ресурсу часто стає причиною відмови від подальшої співпраці [1].

Згадаємо, що фреймворк – це певний набір класів, які з одного боку дають можливість спростити розробку і зменшити код програми, беручи на себе суттєву частину «рутинної, однотипної» роботи, з іншого боку, насичують код часу виконання (run-time) – таким чином, реальний виконуваний код зазвичай стає значно складнішим, насиченим різними перевірками («засторогами») та іншою «стандартною» функціональністю. В результаті прості дії «обростають» великою кількістю коду, який, звичайно ж, уповільнює опрацювання запитів за тих же умов і при однаковому навантаженні.

Таким чином, фреймворк – при застосуванні в чистому вигляді, прискорюючи процес розробки (design time) – уповільнює роботу (run time) результуючої системи. Механізми кешування (caching), які покликані повернути швидкодію складним системам, ускладнюють код та зменшують гнучкість останнього до внесення змін. Водночас відомо, що швидкість реакції на зміни в бізнес-процесах (тобто адаптація підтримуючих про-

грамних систем під нові потреби ринку, власників або замовників) є важливою для ефективного функціонування бізнесу взагалі – а, отже, код має залишатись досить простим для забезпечення можливості його адаптації та оптимізації.

Фреймворки, архітектура та гнучкість до змін

Фреймворки, звісно, не захищають від помилок проектування архітектури програмної системи, але також немає гарантії, що вони допоможуть швидко виправити знайдені неточності архітектури. Так, фреймворки структурують код системи (читай – нав'язують спосіб організації і, навіть, мислення, адже англ. framework – дослівно – «робота в рамках», про це – далі), але при цьому часто зменшують гнучкість до змін. Це означає, що при чергових змінах бізнес-процесу в системі може виявитись, що внесення відповідних коректив зачіпає значну частину коду – і не тому, що система погано спроектована, а саме завдяки «структуруванню». Так, навіть при додаванні полів у реляційній або об'єктній моделі предметної області системи, зміни треба вносити щонайменше у 4 різних файли для MVC-фреймворку:

- структура даних, подання даних (scheme);
- модель (model);
- форми подачі інформації (view);
- реактивні обробники подій (controller).

Код, який розробник сам структурував, скажімо, при класичному об'єктно-орієнтованому підході (ООП), при користуванні фреймворком додатково має бути організований (структурований, розбитий на частини) згідно визначених правил. Це може зробити код менш цілісним і складніше організованим, порівняно з тим же ООП. В цьому випадку внесення змін, тобто підтримка програмної системи, може бути ускладнена і вимагати більше зусиль за рахунок додаткової «структуризації», зокрема адаптація існуючої системи під конкретні потреби або для конкретної ситуації вимагатиме більше часу.

Об'єктно-реляційне відображення

Об'єктно-реляційне відображення (ORM) покликано з одного боку спростити взаємодію з відокремленим компонентом більшості систем – сховищем даних (базою даних, БД), а з іншого – глибше інтегрувати код цієї взаємодії і операції над об'єктами БД власне у код системи, а також інкапсулювати технічні та рутинні операції (додавання, оновлення в таблицях БД тощо). В результаті, з одного боку, отримується більш природній (для розробника системи, а не фахівця БД) спосіб (код) взаємодії, а з іншого – завдяки суттєвій інкапсуляції – непідконтрольний та слабо-керований (автогенерований) код.

В першу чергу мова тут іде про ситуації масових вибірок даних з БД. У випадках маніпулювання окремими записами таблиць (об'єктами на рівні ORM) цей підхід є, дійсно, зручним і накладними затратами підходу (додаткові об'єкти в пам'яті, довший ланцюжок виконання операцій, більше коду і часу для виконання тих самих дій) часто можна знехтувати, бо вони не такі відчутні. Якщо ж мова йде про масові вибірки, до того ж за складними умовами, то ефективність результуючого коду (в першу чергу з міркувань його швидкодії) в цьому випадку явно погіршується:

- через ORM складно і неприродно керувати використанням індексів (врешті все одно робота зведеться до запитів до БД), так само як і виконувати більш тонку оптимізацію запитів;
- ORM за задумом відводить і відділяє програміста від специфіки БД, позбавляючи цим самим можливості виконувати оптимізацію (у тому числі «низькорівневу») і керувати виконанням запиту.

Як правило, в ORM є можливість виконати прямий запит SQL – в цьому випадку можна скористатись всіма можливостями SQL і, власне, БД. Але тоді ми не використовуємо переваги підходу ORM (універсальність нотації та інтеграція коду замість «змішування» технологій – коду і SQL-запитів у БД) і несемо накладні затрати у зв'язку з (беззмістовним і невиправданим у цьому випадку!) використанням складної технології.

Оскільки синтаксис ORM-запитів може суттєво відрізнятись від SQL, автор даної роботи неодноразово був свідком помилок, що викликані:

- неповним розумінням підходу ORM або його реалізаційних особливостей;
- специфікою синтаксичної нотації (наприклад, LINQ), і необхідністю «переключати» спосіб мислення у зв'язку з цим;
- неувагою розробника (в дрібницях), оскільки більше сподівань невиправдано покладається на механізми ORM («всемогутні» та «всевиправляючі»).

При цьому ті ж самі розробники швидше писали запити на SQL, а також допускали в них менше помилок.

Ще одне важливе зауваження щодо роботи з великими вибірками за допомогою ORM. Розробнику потрібно чітко розуміти, як саме виконується його запит:

- повністю транслюється у БД (трансформується у SQL);
- виконується в об'єктній моделі над окремими сутностями, лише дістаючи їх з БД;
- або ж має місце деякий змішаний варіант.

До того ж важливо знати, як саме подається результат запиту в ORM (у пам'яті). Це може бути:

- курсор по результату запиту у БД;
- створення об'єкту для кожного результату запиту;
- інший спосіб.

Тож, якщо запит повністю транслюється у БД, а результатом є курсор БД – можна сподіватись на високу швидкість роботи (виконання запиту і постобробку результату), в інших випадках швидкодія буде, скоріше за все, неприйнятною. Адже, якщо ORM намагається побудувати в оперативній пам'яті модель даних і реально маніпулювати даними через неї та/або перетворити кожен запис результату вибірки у відповідний типізований об'єкт (або сукупність об'єктів) – надто багато часу піде саме на ці накладні (непродуктивні) затрати, замість виконання корисної по суті роботи (за бізнес-процесом системи). Отже, в найгіршому випадку можливі як втрата швидкодії від застосування ORM, так і значні обсяги зайнятої пам'яті (тобто, недостатній об'єм для інших задач), і зменшення, як результат, ступеня паралелізму.

Додатково зауважимо, що на рівні прагматики (або навіть філософії) ми маємо факт «перетягування» класичних функцій БД з маніпуляції даними у не природне для цього середовище (на думку автора) – у код. Чи виявиться цей зсув акценту корисним – покаже час. На рівні ж архітектури програмної системи ми також часто маємо справу з дублюванням функціональності (ORM для взаємодії з окремими записами, прямі SQL запити для великих вибірок) – що є поганою практикою.

Коректність програмної системи

Одним з найважливіших питань щодо будь-якої програмної системи є питання її коректності (часткової, тотальної). Для деяких класів систем вимоги щодо доведення їх коректності навіть закріплені документально [2]. Відомі [2] класичні підходи до доведення коректності програмного забезпечення. Але всі вони так чи інакше спираються на той факт, що весь код, що підлягає перевірці, «виписаний явно», тобто його можна анотувати чи проводити над ним інші необхідні дії.

В тому чи іншому вигляді нехтувати цим питанням абсолютно неможливо: некоректна програмна система не може існувати довгий час, а тому застосування підходів до перевірки чи гарантування коректності (тестування або формальні, математичні, доведення) є необхідним. Причому, якщо за рахунок тестування досягається вибіркова коректність, то для математичної коректності потрібні більш складні теорії [2].

Якщо стоїть задача строгого математичного доведення коректності, то в більшості випадків застосування технологій швидкої розробки (фреймворки, готові компоненти тощо) ускладнюють її до неможливої або принаймні роблять дуже відносною (тобто доведення відносно коректної роботи фреймворка, компонентів і т.і.). У той же час спиратись на гарантовано коректну роботу цих складових системи – у строгому сенсі – не можна, оскільки у всіх таких компонентах та фреймворках час від часу знаходять та виправляють помилки, а *гарантій надійності* роботи не дає жоден з їх розробників. Випускаючи ж нову версію, строго, треба все доводити поновому, тобто задача стає практично нерозв'язною.

Якщо маємо справу з задачею тестування, то:

- по-перше, треба все перетестувати після виходу нових версій задіяних компонентів та/або фреймворків, не тільки на предмет сумісності, а і функціонування всієї системи загалом;
- а по-друге, яким би якісним і всеохоплюючим не було тестування, все одно воно не дасть гарантій коректності системи, адже є – принципово – вибірковим.

Задача гарантування коректності програмної системи завжди була і залишається складною і трудомісткою, але у випадку використання сторонніх компонентів для її побудови (зокрема, фреймворків) задача стає практично нерозв'язною у класичному смислі (повністю) ще і з тієї причини, що частина системи (знов-таки, ті ж самі компоненти та фреймворки) стає «чорною скринею», у яку немає способу «зазирнути» і стверджувати **напевно** будь-що відносно функціонування.

Розробка як композиція готових універсальних компонентів

Ще однією практикою є розробка програмних систем шляхом компонування готових (універсальних) компонентів. Це, беззаперечно, прогресивна практика, і розробляти все завжди «з нуля» ні економічно, ані ідеологічно (уникнення дублювання коду, повторне використання) не виправдано. Але дуже важливо не «сліпо» застосовувати компоненти, з яких начебто можна швидко «зібрати» потрібну функціональність, натомість усвідомлювати адекватність їх ужитку для тієї чи іншої ситуації. Наприклад, готові компоненти типу Grid («решітка») непогано підходять для редагування умовно-постійної інформації (таблиць «довідників», з відносно невеликою кількістю записів), у той час як для більш складних або більш динамічних даних (таблиць «фактів», з відносно великою кількістю записів) вони підходять гірше принаймні з трьох причин:

1) незручний вигляд для роботи, особливо, якщо з'являється горизонтальний scroll на екрані (тобто один запис вимагає забагато місця по ширині – наприклад, якщо в таблиці багато колонок) або одночасно видно мало записів (якщо кожен або деякі займають надто багато місця по висоті);

2) подача інформації щодо записів («фактів») у вигляді стандартизованої таблиці, як правило, є незручною для роботи у більшості прикладних областей (не є природним способом подачі інформації для цієї галузі,

натомість вимагається специфічна організація інформації при подачі – наприклад, картка клієнта або задачі проекту);

3) швидкодія може виявитись неприйнятно низькою (наприклад, якщо в таблиці багато колонок) за рахунок універсальності (а значить, внутрішньої складності) компоненту «решітка».

Так, зокрема, універсальність багатьох компонентів забезпечується через механізм рефлексії, що є достатньо повільним за рахунок перенесення багатьох перевірок з швидкого часу компіляції (compile-time) у повільну площину виконання (run-time).

Підсумовуючи, до компонент напряму відносяться ті самі проблеми, на які вказано вище:

1) швидкодія результуючих артефактів (модулів систем) – під питанням;
2) коректність функціонування (надійність) – важко показати;
3) внесення змін та підтримка працездатності за часом – можуть виявитись ускладненими.

Під останнім розуміються, з одного боку, вимушені внесення змін у компоненти з причин виправлення помилок або ж розвитку функціональності (випуск нових версій), а з іншого – пов'язані питання підтримки програмної системи, базованої на компонентах:

- при виправленні помилок у компонентах – повторне тестування на предмет коректного функціонування розробленої системи;
- при випуску нових версій – можливі суттєві внесення коректив до частин розробленої системи, якщо нові версії не мають зворотної сумісності (тобто не підтримують старі API, що цілком можливо у сучасному світі IT).

Прагматика. Зміщення акценту

Застосування технологій «швидкої розробки» (RAD) має ряд прагматичних (навіть філософських, світоглядних) побічних ефектів.

1. Систематизація коду (навіть ширше – декомпозиції задачі, процесу проектування і розробки, способу мислення) – одна з важливих властивостей фреймворків. Але чи дійсно лише систематизацією все обмежується? Так, зокрема, по відношенню до популярних сьогодні MVC (Model-View-Controller) або ORM (Object-Relational Mapping) фреймворків, можна стверджувати, що скоріше мова йде про обмеження (!) мислення і «затискання» ідей розробника у певні межі, «рамки», або ж «направлення» думок «по рейках», які прокладаються фреймворком. (В цьому ключі можна також говорити про обмеження свободи мислення розробника).

2. «Вбудовування цеглинок у стіну» замість творчості. Так, тенденції сьогодення – це відхід від ремесла і перехід то технології у програмуванні, перехід від мистецтва до застосування інструментарію. Але за цим переходом стоїть і більш глибинний перехід від мислення ідеями до володіння арсеналом засобів, від діяльності винахідника до інженерного ремесла.

3. Недостатня глибина розуміння, особливо – деталей реалізації. Як було показано вище, важливим є питання адекватності застосування тих чи інших підходів до розв'язання конкретних задач. Фреймворки, будучи складними технологіями (як і інші засоби RAD, і це не треба недооцінювати!), вимагають суттєвого часу, щоб розібратись у нюансах реалізації та застосування. Але ж сам підхід RAD, з іншого боку, підштовхує до «ранньої дії», коли за рахунок видимої простоти можна почати застосовувати технологію без глибокого розуміння, що приносить зворотній ефект: технологія стає неефективною і, будучи неадекватно застосована, лише погіршує як часові, так і якісні показники результату роботи – програмної системи.

4. Зміщення акценту від комплексного бачення задачі до «вписування» у фреймворк. Дійсно, замість концентрації на розв'язанні задачі по суті та пошуку ефективного алгоритму або методу для цієї мети, розробник вимушений (за умов обмеженого часу) шукати шлях реалізації і «вписуватись» в межі фреймворку (або іншої RAD-технології). Принцип «розділяй і володарюй», звичайно, корисний при розробці великих програмних систем, і розробникам корисно концентруватись на підзадачах для реалізації проекту в цілому – але (знов-таки, за рахунок складності технологій) часто розробники впадають у іншу крайність: «за деревами лісу не видно», коли цілісне бачення задачі, яке часто допомагає уникнути помилок, не досягається або ж ігнорується.

5. «Зав'язка» на фреймворк (або іншу RAD-технологію). Всі такі «зав'язки», як було зазначено вище, звужують поле для мислення і дій та виступають обмежуючими (у негативному смислі) факторами. Це – потенційно ще один «камінь спотикання» на шляху подальшого розвитку програмної системи (розширення її функціональності тощо). Так, розробник вимушений, зокрема, слідкувати за змінами фреймворка (виправленнями помилок, оновленнями для ліквідації виявлених вразливостей і т. д.) і вчасно вносити оновлення у розроблену систему – для забезпечення стабільності роботи останньої і можливості подальшого її розвитку. Але тут його спіткають принаймні дві суміжні складності:

- чергові зміни у фреймворку можуть порушити зворотну сумісність коду (API платформи), що ускладнить подальший розвиток системи і може привести до вибору: переписати (адаптувати) код системи, залишитись на попередній версії платформи (з усвідомленням ризиків) або відмовитись від подальшого розвитку системи взагалі;
- швидкоплинність процесів у сучасному світі (особливо – у IT), і, як наслідок, достатньо часті внесення змін у фреймворки (так само, як і у бізнес-процеси, які автоматизуються розроблюваними прикладними про-

грамними системами), залишають мало часу на докладне вивчення і чітке усвідомлення цих змін – тому результат оновлень (переходу на недостатньо глибоко досліджені нові версії) може відчутно вплинути на якість коду (всередині) і програмного продукту в цілому (ззовні).

6. Ускладнення впровадження змін, управління змінами, та посилений постійний рефакторинг коду – як наслідок попереднього зауваження.

7. Сумарний час розробки може зрости (порівняно з часом без застосування RAD). Якщо скласти час навчання, відслідковування змін у технології і, власне, час програмування – чи дійсно розробка виявиться швидшою без втрати інших показників якості? – Досвід показує, що не завжди. Сумніви щодо пришвидшення особливо стосуються процесу підтримки розробленого програмного продукту та внесення необхідних змін (адаптацію під зміни автоматизованих бізнес-процесів), коли, до того ж, команда розробників може часто змінюватись (природно через довготривалість супроводження).

Навчання

Що стосується навчання сучасним інформаційним технологіям щодо розробки програмних систем, зокрема – RAD-технологіям, особливо – фреймворкам, треба враховувати ряд факторів:

- швидкоплинність – зміни у технології можуть наступити раніше, ніж будуть якісно опановані попередні стабільні версії, що потягне, скоріше за все, недосконале володіння технологією (аби «встигати за часом») або ж необхідність перенавчання – тобто переключення на вивчення оновленої версії без успішного оволодіння попередньою;
- треба вивчати технологію не відокремлено саму по собі, а у контексті, аби одразу розуміти межі застосування в цілому та її частин і уникати в подальшому неадекватних застосувань (приклади – див. вище), адже автор вважає, що це є найбільшою проблемою і основною причиною наступних помилок: потрібно чітко усвідомлювати зв'язки з іншими технологіями, співвідношення з ними і місце у процесі розробки, переваги і недоліки, а також галузь і умови застосування;
- потрібно виділяти достатньо часу для опановування технології, адже недостатньо глибоке вивчення на старті обов'язково призведе до надлишкових часових затрат потім, в процесі розробки, а у гіршому випадку – до неякісної розробки програмного продукту.

Стосовно інших сучасних підходів, наприклад, шаблонів проектування (Design Patterns) у ООП, також ситуація двояка:

- з одного боку, необхідно знати шаблони, щоб, як мінімум, «правильно читати» код інших розробників, а також, вони корисні для RAD, оскільки дозволяють швидко спроектувати і запрограмувати нетривіальні, але стандартні ситуації (шаблони «фабрика», «сінглтон» тощо);
- але, з іншого боку, деякі «шаблони» (або ширше – підходи) з часом стали переходити у розряд «поганих», або «анти-шаблонів» (anti-pattern), оскільки стали проявлятись мінуси у зв'язку зі зміною ситуації в IT (наприклад, глобальні змінні завжди вважались нормальною практикою – до появи паралелізму у широкому вжитку і введення лямбда-синтаксису, також глобальний стан – є нонсенс для концепції функціонального програмування, що останні роки знов набуває популярності).

Отже, висновок щодо навчання наступний: потрібно виділяти достатньо часу для засвоєння як самої технології (з прикладами успішного, правильного, а також некоректного, неадекватного застосування), так і її зв'язків, обмежень, переваг і недоліків, умов застосування.

«Швидка» розробка

Підсумовуючи, зазначимо, що «швидка» розробка зовсім не означає «нашвидкуруч», хоча слова мають спільний корінь. Реально ж, навіть сучасні методології типу «гнучкої розробки» (agile) підсвідомо спонукають до жертви іншими показниками заради швидкості випуску версій продукту. Якщо «швидка розробка» означає низьку якість результату (як часто і трапляється), то це – неправильне трактування, яке призводить до негараздів типу невідповідності програмного продукту вимогам або складності подальшої підтримки і розвитку функціональності.

Питання адекватності застосованих технологій завжди треба тримати під контролем. Адже від цього, зокрема, залежить ефективність як процесу розробки, так і отримуваного результату (наприклад, щодо швидкодії розроблюваної системи).

Швидкість розробки за допомогою деякої технології (RAD, певний фреймворк тощо) треба оцінювати в комплексі, тобто час, необхідний на:

- вивчення технології;
- власне програмування з використанням технології;
- відслідковування та застосування змін у технології (включаючи повторні тестування) – залежатиме від частоти та масштабності таких змін;
- підготовка ще одного (або заміна) члена команди, що працюють над проектом.

В результаті такої комплексної оцінки може виявитись, що RAD є не такою вже й «швидкою» технологією розробки і вимагає більше часу на повний цикл розробки програмної системи, ніж відмова від застосування

цієї технології взагалі. Також слід враховувати час на підготовку фахівця, вартість цієї підготовки і подальшу «вартість» даного фахівця, а також – вартість перепідготовки («заміни») якісного спеціаліста.

Важливо також відзначити, що ніякі технології не замінять «голову на плечах». Якою б не була досконалою і універсальною технологія розробки, вона може стати інструментом успіху для фахівця або зброєю знищення самого ж проекту в руках «псевдо-фахівця» (у автора є багато прикладів підтвердження цієї тези). Сама по собі технологія не покращить ані структурованість проекту, ані якість коду – вона може лише допомогти організувати роботу над проектом та сам програмний продукт, а далі – реалізаційно – все одно все залежить від виконавця (програміста).

Тому, для розробки програмного забезпечення потрібно ставити за мету досягнення дійсно ефективності, надійності та підтримуваності (іншими словами – *якості*) програмної системи, а не лише швидкість досягнення результату (якої якості?). Теза з математики про те, що складність задачі (іншими словами – загальний час на розробку) нікуди не дівається сама по собі – вона може бути лише перекладена на підзадачі шляхом зведення до іншої або декомпозиції на дрібніші підзадачі, дійсна і для розробки програмного забезпечення. Отже, не слід сприймати технології як панацею, «швидка розробка» – просто ще один інструмент, який може принести успіх у випадку адекватного застосування.

Висновки

Підсумовуючи попередні розділи роботи, зафіксуємо наступні висновки щодо технологій RAD (фреймворки і т. д.):

1) їх необхідно вивчати, але у контексті, розуміючи рамки, переваги і недоліки;
2) технології необхідно адекватно (в усіх сенсах) застосовувати;
3) швидке навчання (без глибокого розуміння деталей) – лише шкодить (зокрема, проекту), отже потрібно суттєву увагу приділяти саме докладному, якісному, вивченню технологій («краще – менше, але краще», аби кількість не перекрила якість).

Також у процесі підготовки фахівців важливо більше уваги приділяти загальним, фундаментальним, підходам та принципам розробки програм, аніж конкретним технологіям.

Не завжди «технології» допомагають зробити процес розробки простішим, а результат (програмний продукт) – більш якісним. Для впевненої розробки дійсно ефективних, надійних, та підтримуваних (іншими словами – *якісних*) програм потрібна, в першу чергу, високоякісна підготовка фахівців, здатних та мотивованих до саморозвитку.

1. *Amit Singhal, Matt Cutts*. Using site speed in web search ranking. – Google Webmaster Central Blog. – 2010. (http://googlewebmastercentral.blogspot.com/2010/04/using-site-speed-in-web-search-ranking.html)
2. *Панченко Т.В*. Композиційні методи специфікації та верифікації програмних систем. Дис. … канд. фіз.-мат. наук / Київський національний університет імені Тараса Шевченка. – Київ, 2006. – 177 с.