УДК [004.057.5+331.55+37.06]::[378::004.4]
Стрюк А. М.
Криворізький національний університет, Кривий Ріг, Україна
ORCID ID 0000-0001-9240-1976

ФОРМУВАННЯ ЗДАТНОСТІ МАЙБУТНІХ ІНЖЕНЕРІВ-ПРОГРАМІСТІВ ДО ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

DOI 10.14308/ite000755

Стаття присвячена одній із компетентнісних складових мобільно орієнтованого середовища професійно-практичної підготовки майбутніх фахівців з інженерії програмного забезпечення (ІПЗ). Показано, що введення стандарту вищої освіти за спеціальністю 121 «Інженерія програмного забезпечення» для першого (бакалаврського) рівня вищої освіти породило низку проблем забезпечення якості підготовки, пов'язаних насамперед із низьким рівнем деталізації компетентностей і програмних результатів навчання. Шляхом розв'язання поставлених проблем є детальне проєктування системи професійних компетентностей майбутніх фахівців з ІПЗ.

У статті розглянуто підходи до формування однієї з найважливіших спеціальних професійних компетентностей майбутніх інженерів-програмістів – К14 (здатність брати участь у проектуванні програмного забезпечення, включаючи проведення моделювання (формальний опис) його структури, поведінки та процесів функціонування). На основі історико-генетичного огляду практики навчання проєктування програмного забезпечення майбутніх фахівців з ІПЗ в США, Канаді, Великій Британії, Австралії, Новій Зеландії та Сингапурі сформульовано рекомендації з вибору форм організації навчання, добору змісту навчання, способів діяльності студентів та викладачів із проєктування програмного забезпечення, інструментарію моделювання та проєктування, оцінювання рівня сформованості відповідної компетентності. Розглянуто приклад організації навчання проєктування в умовах, наближених до виробничих – студійного навчання. Показано проблеми переходу від архітектурного до детального проєктування та реалізації проєкту.

Перспективи подальшого розвитку дослідження полягають в обґрунтуванні третьої (після інженерії вимог та проєктної інженерії) інженерної складової ІПЗ – конструювання програмного забезпечення.

Ключові слова: *інженерія програмного забезпечення, проєктування програмного забезпечення, професійна підготовка фахівців з інженерії програмного забезпечення, здатність майбутніх інженерів-програмістів до проєктування програмного забезпечення*

Вступ. Затвердження у 2018 році стандарту вищої освіти за спеціальністю 121 «Інженерія програмного забезпечення» для першого (бакалаврського) рівня вищої освіти [22] у світлі нової процедури акредитації освітніх програм породило дві основні проблеми, розв'язання яких було покладено на гарантів освітніх програм:

1. Стандарт надає надзвичайно високий ступінь свободи у трактуванні змісту компетентностей та програмних результатів навчання, що призводить до появи суттєво різних та негармонізованих освітніх програм і планів, оцінка яких виконується експертами НАЗЯВО за також розмитими критеріями якості. При цьому аналогічні

зарубіжні стандарти (університетські, державні та світові) мають високий рівень деталізації складових компетентностей та критеріїв оцінювання їх сформованості – так само, як у США, Австралії та Японії критерії якості освітніх програм є деталізованими для певної галузі освіти, а не застосовуваними до всіх. Відсутність чітких критеріїв оцінювання якості, специфічних для галузі знань, породжує ризики порушення принципу академічної доброчесності при їх оцінюванні.

2. Висока швидкість зміни змісту технологічної складової підготовки фахівців з інженерії програмного забезпечення поряд із намаганням укладачів освітніх програм якщо не випередити, то хоча б відповісти на вимоги виробництва, призводить до інструментального ухилу в навчанні інженерії програмного забезпечення – аж до нівелювання найважливіших для сталого професійного розвитку інженера-програміста загальних компетентностей. Такий ухил призводить до нерозрізненості інженерії програмного забезпечення від інших спеціальностей галузі знань 12 «Інформаційні технології» та професійної дезорієнтації вступників та студентів за нею.

Розв’язання поставлених проблем вимагає системного опанування світового досвіду підготовки фахівців з інженерії програмного забезпечення (ІПЗ) у його генезисі [24; 23] та відповідного проєктування системи професійних компетентностей (змісту, показників сформованості та засобів діагностики), як загальних [13], так і спеціальних.

Серед спеціальних компетентностей фахівця з ІПЗ на особливу увагу заслуговують ті з них, що відображають «сутність та дух» ІПЗ – специфіку професійної діяльності інженерів-програмістів, що не може бути набута у процесі навчання за іншими спеціальностями галузі знань 12 «Інформаційні технології». Так, рекомендації до розробки навчальних програм для бакалаврів ІПЗ визначають компетентність із пошуку компромісів, сутність якої полягає в узгодженні суперечливих цілей проєкту, пошуку прийнятних компромісів за обмежень вартості, часу, знань, наявних систем і організацій: «Студенти повинні займатися проблемами, які приводять їх до суперечливих і мінливих вимог. ... Компоненти навчальної програми повинні спрямовувати на такі проблеми з метою забезпечення високоякісних функціональних і нефункціональних вимог та розробку якісного програмного забезпечення» [14, с. 21].

Пошук компромісів та узгодження протиріч є традиційною діяльністю з інженерного проєктування (дизайну), що є не обов’язковою для всіх фахівців з інформаційних технологій, проте є ключовою для інженерів з програмного забезпечення. У стандарті [22] їй відповідає спеціальна компетентність К14 – «здатність брати участь у проектуванні програмного забезпечення, включаючи проведення моделювання (формальний опис) його структури, поведінки та процесів функціонування». Формування здатності до проєктування програмного забезпечення визначальне в тому, чи дійсно є випускник освітньої програми з ІПЗ інженером-програмістом.

Огляд освітніх програм з ІПЗ, доступних на сайтах ЗВО України, показує, що навчання проєктування програмного забезпечення виконується за трьома основними напрямами:

1) штучне введення елементів проєктування до різних курсів у тісному зв’язку з інструментальними засобами на прикладі розв’язання типових спрощених задач певної галузі;

2) спонтанне навчання елементів проєктування під час виробничої практики на реальних задачах;

3) навчання засобу моделювання програмного забезпечення (зазвичай, UML).

Лише у незначній кількості освітніх програм спостерігаються спроби врахування світового досвіду організації навчання проєктування на основі синергетичного об’єднання академічних та промислових форм і методів навчання.

Тому **метою** нашої статті є історико-генетичний огляд практики навчання проєктування програмного забезпечення майбутніх фахівців з ІПЗ.

Навчання проєктування програмного забезпечення: зарубіжний досвід

Виокремлену у стандарті [22] спеціальну компетентність К14 (здатність брати участь у проектуванні програмного забезпечення, включаючи проведення моделювання (формальний опис) його структури, поведінки та процесів функціонування) В. М. Пелевін співвідносить зі здатністю та готовністю до здійснення системотехнічного проєктування, що включає проєктування системного, прикладного ПЗ та мереж ЕОМ [21, с. 14].

Моделювання та аналіз можна вважати основними поняттями будь-якої інженерної дисципліни, оскільки вони є важливими для документування та оцінки проєктних рішень і альтернатив [14, с. 31]. Проєктування програмного забезпечення стосується питань, методів, стратегій, способів подання та шаблонів, що використовуються для визначення способу реалізації компонента або системи [14, с.32].

Д. Керрінгтон (David Carrington) [2] визначає, що проєктування ПЗ – це етап розробки ПЗ, під час якого специфікація перетворюється в структуру, придатну для реалізації. Навчання проєктуванню повинно охоплювати як об'єкт проєктування, так і процес, за допомогою якого цей об'єкт створюється [2, с. 547].

Ч. Ху (Chenglie Hu) [6] ставить низку питань стосовно того, чим є проєктування програмного забезпечення:

1. *Чи є проєктування програмного забезпечення інженерною діяльністю?* «Наприклад, проєктування мосту повинно здійснюватись відповідно до фізичних законів, але при проєктуванні програмного забезпечення немає законів, яких слід дотримуватися, принаймні теоретично. Архітектура моста добре помітна неозброєним оком, тоді як програмне забезпечення – абстрактна сутність, над якою ми працюємо лише з різними його відображеннями. На практиці інженерне проєктування має передувати офіційному будівництву мосту, тоді як будівництво програмного забезпечення може відбуватися без чіткого проєкту. Однак найсерйозніша відмінність полягає в тому, що після початку будівництва зміни в специфікації інженерного продукту можуть бути недопустимі, тоді як зміни вимог до програмного забезпечення, як правило, очікуються і дійсно можуть відбутися в будь-який час протягом життєвого циклу програмного забезпечення» [6, с. 63].

2. *Чи можна вважати саму комп'ютерну програму твором мистецтва?* «Ймовірно, це можливо, але не завжди. ... Причина в тому, що проєктувальник може, наприклад, через програмні обмеження різної природи порушити деякі принципи проєктування для розв'язання нагальних проблем. Проєктування – це баланс між розширюваністю, дотриманням принципів проєктування та урахуванням програмних обмежень, а отже, може бути елегантним та художньо привабливим» [6, с. 63].

3. *Чи можна проєктування програмного забезпечення визначити через проєктну діяльність?* «Проектна діяльність різноманітна; багато аспектів, безумовно, є технічними, але деякі можуть бути і соціальними, але жодні з них, ймовірно, не будуть стандартизовані для характеристики проекту або його реалізації» [6, с. 63].

4. *Чи можна визначити проєктування програмного забезпечення за допомогою артефактів чи етапів проєктування?* «Стандарт IEEE 1016-2009 для опису проєкту програмного забезпечення визначає зміст та процедуру його опису, ... однак спосіб опису проєкту ... може суттєво варіюватись ..., а коли професіонали не мають консенсусу щодо артефактів проєктування, вони, ймовірно, не матимуть консенсусу й щодо етапів проєктування. При цьому практики гнучких методів вірять, що

проєктування не лише високо ітеративне, але й емерджентне, моделі часто помилкові, і лише кодування, виконання тестів та рефакторинг коду відкривають правду про проєкт» [6, с. 63-64].

Ч. Ху пов'язує процес проєктування з проєктним (дизайнерським) мисленням (design thinking): «Професіонали давно підозрюють, що розробка програмного забезпечення цілком може бути когнітивним процесом, а суть проєктування – це швидкий процес моделювання та випробування певних рішень» [6, с. 64]. К. Л. Дим (Clive L. Dym), Е. М. Агогіно (Alice M. Agogino), О. Epic (Ozgur Eris), Д. Д. Фрей (Daniel D. Frey), Л. Дж. Ляйфер (Larry J. Leifer) перерахували розумові здібності, які часто асоціюються з гарним проєктним мисленням, серед чого: терпимість до неоднозначності, здатність стежити за загальною картиною, справлятися з невизначеністю, приймати рішення, мислити як частина команди та думати і спілкуватись кількома мовами проєктування [4, с. 104].

У результаті Ч. Ху визначає *проєктування програмного забезпечення* як систематичний розумовий процес, у якому проєктувальники генерують, оцінюють та конкретизують концепції програмної системи, структура та функції якої відповідають цілям клієнтів або потребам користувачів, задовольняючи при цьому певний набір обмежень [6, с. 64].

Процес проєктування водночас і надзвичайно креативний, і надзвичайно складний, тому не дивно, що по-справжньому видатних проєктувальників (дизайнерів) мало. Однак потреби в нових проєктах дуже високі, і, щоб їх задовільнити, у межах кожної архітектурної та інженерної галузі розроблено набори принципів, що надають можливість проєктувальникам із середніми здібностями створювати проєктні рішення прийнятної якості. Хоча ІПЗ значно молодша за інші інженерні галузі, за час її існування також накопичені певні знання про те, як створювати проєкти, і майбутні інженери-програмісти повинні оволодіти цими знаннями, тому навчання проєктування має бути невід'ємною частиною їхньої професійної підготовки [12].

«Виняткові люди народжуються з відчуттям того, як робити гарні проєкти. Більшість з нас повинні побачити приклади хороших проєктів, перш ніж ми зможемо самостійно придумати гарний проєкт. Тому ми навчаємо проєктуванню на прикладах, надаючи студентам ескізи проєктів, а потім є багато відгуків про їх спроби вдосконалити проєкт» [7, с. 33].

Р. М. Грем (Robert M. Graham) у статті 1970 року [5] пропонує методику навчання проєктування, за якої вводиться єдина ключова ідея та досліджуються її наслідки в середовищі, яке свідомо ігнорує питання ефективності. Після повного розвитку логічних наслідків цієї ідеї, додаткові ключові ідеї об'єднуються одна за одною та досліджуються їх наслідки. Далі розглядають ефективність та інші реальні обмеження, досліджуючи наслідки кожного з них. Цю процедуру продовжують, доки не буде досягнуто реалістичного проєкту. Крім того, для ілюстрації розробки проєкту використовується спеціально розроблений та задокументований кейс.

Д. М. Вайс (David M. Weiss) у [18] представляє більш пізній (середина 1980-х рр.) досвід навчання проєктування програмного забезпечення, інваріантного до застосовуваної методології проєктування. Його підхід передбачає два етапи: спочатку студенти знайомляться з принципами, що лежать в основі методології, та досвідом застосування цих принципів до невеликих чітко визначених задач. На другому етапі студентам пропонують реальну задачу, яку вони мають вирішити самостійно в змодельованому робочому середовищі, отримуючи вказівки лише з методологічних питань. Перший етап представлений у вигляді циклу лекцій. Другий – контролюється експертом з проєктування. У результаті першого етапу студенти отримують певне розуміння принципів, що лежать в основі методики проєктування, бачать зразки

застосування цих принципів та практикуються у їх застосуванні на навчальних прикладах. Після першого етапу студенти здатні використовувати методики проєктування під керівництвом досвідченого проєктувальника у процесі його майже щоденної взаємодії зі студентами. У результаті другого етапу студенти набувають достатньо досвіду з проєктування, щоб здійснювати його з меншими настановами, спілкуючись із досвідченим проєктувальником приблизно раз на тиждень [18, с. 1156].

На установній зустрічі проєкту між студентами розподіляються ролі, які вони будуть виконувати під час проєкту. Студенти знайомляться з послідовністю основних етапів проєкту, отримують перелік документів, які потрібно було пред'явити, та порядок їх підготовки. Також студентам надають короткі пам'ятки щодо процедури керування конфігурацією, опис відповідальності та складу команди з контролю якості. Кожен студент повинен розробити та задокументувати принаймні один модуль системи [18, с. 1157]. Д. М. Вайс відзначає, що студенти залишались невдоволеними тим, що були зосереджені на проєктуванні, але так і не розробили готовий продукт через складність проєктованої задачі (однак більш проста задача не надала б можливості продемонструвати та засвоїти всі необхідні прийоми та методи проєктування). Загалом, студенти відчували, що вони засвоїли методологію проєктування, але були розчаровані тим, що не створили жодного рядка коду. Щоб подолати це розчарування, можна запропонувати студентам продовжити роботу над реалізацією проєкта в наступному семестрі [18, с. 1158].

Д. М. Вайс надає такі рекомендації з організації навчання проєктування ПЗ [18, с. 1159]:

1) консультувати студентів має фахівець з проєктування, який може забезпечити неухильне дотримання методології;

2) задача з проєктування має бути достатньо складною, і студентам заборонено її переспрощувати;

3) студентів потрібно спонукати до прийняття власних проєктних рішень: жоден із викладачів не має втручатись у процес прийняття рішень;

4) розмір студентської групи має бути невеликим, щоб дозволити викладачам уважно стежити за успіхами кожного студента.

Е. Ламм (Ehud Lamm) [9] зазначає, що задачею курсу з проєктування програмного забезпечення є розвиток навичок проєктування програм через викладання основних концепцій інженерії програмного забезпечення, необхідних для вивчення та аналізу альтернативних проєктів програмного забезпечення.

Computing Curricula 2020 [3, с. 120] визначає такі компетенції, що належать до проєктування програмного забезпечення:

1. Здатність представити особам, що приймають рішення у бізнесі, архітектурно значущі вимоги з документа із їх специфікаціями.

2. Здатність оцінити та порівняти компроміси з альтернативних варіантів проєкту для задоволення функціональних і нефункціональних вимог та написати коротку пропозицію, у якій узагальнюються ключові висновки для клієнта.

3. Здатність розробити високорівневий проєкт конкретної підсистеми, зрозумілий для нефахівців із проєктування, ураховуючи архітектурні шаблони та шаблони проєктування.

4. Здатність розробити для клієнта високорівневий проєкт конкретної підсистеми, використовуючи принципи проєктування та наскрізні аспекти для задоволення функціональних та нефункціональних вимог.

5. Здатність до оцінки складників тестування якості програмного забезпечення при проєктуванні підсистем та модулів для розробника / виробника.

6. Здатність створювати документацію із проєктування програмного

забезпечення для подальшої ефективної роботи аналітиків, розробників, тестувальників, фахівців із підтримки тощо.

Ядро знань щодо проєктування програмного забезпечення Ч. Ху [6] пропонує розділити на 2 категорії: знання про процес проєктування та знання про технології проєктування – до останніх він включає й шаблони проєктування. Знання про процес автор визначає як знання про загальновизнані етапи проєктування, а також парадигми або методики проєктування, що використовуються, зокрема, для створення артефактів проєктування [6, с. 64-65]. Аналіз вимог до програмного забезпечення є частиною процесу архітектурного проєктування, що охоплює функціональні та поведінкові аспекти архітектури. Архітектурний проєкт стосується не тільки того, що є системою, але і того, що робить система. Наступний етап проєктування – неархітектурне (детальне) проєктування – стосується модуляризації та деталізації інтерфейсів елементів проєкту, їх алгоритмів та процедур, а також типів даних, необхідних для підтримки архітектури та задоволення вимог. Неархітектурний проєкт визначає функціональність та структуру програмного забезпечення на високому рівні деталізації, проте недостатньому для реалізації. Процес проєктування також передбачає управління виробництвом проєктних артефактів для забезпечення правильної та точної реалізації дизайнерських ідей та рішень. Ч. Ху вважає, що процес проєктування не є завершеним, якщо проєкт не реалізований у коді [6, с. 66].

К. Пірс (Keith Pierce), Л. Денін (Linda Deneen), Г. Шут (Gary Shute) [12] вважають, що ключовим питанням навчання проєктування є раціональний вибір серед багатьох альтернативних рішень [12, с. 220]. «Нам здається, що викладання проєктування значно покращиться за ... [умов]: представлення альтернативних стратегій проєктування та альтернативних рішень, отриманих за допомогою цих стратегій; представлення критеріїв оцінки якості альтернатив та спонукання студентів до виконання завдань, під час яких вони могли б зробити вибір альтернативи та обґрунтувати його» [12, с. 220-221]. Методи проєктування автори розділяють на низькорівневі (застосовуються для проєктування малих модулів – вибір упорядкованими та невпорядкованими масивами, між рекурсією та ітерацією тощо) та високорівневі (застосовуються для прийняття рішень про організацію програмних модулів).

Й. Джа (Yanxia Jia), Й. Дао (Yonglei Tao) [8] вважають моделювання центральною складовою процесу розробки якісного програмного забезпечення. Викладаючи проєктування програмного забезпечення, викладачі мають робити наголос на створенні відповідної моделі для обмірковування проблеми та на використанні властивостей моделі для конструювання рішення [8, с. 702].

Автори [8] визначають такі ключові концепції:

– *моделювання* – процес створення абстрактного, графічного або математичного опису задачі проєктування, під час якого розробник замінює складну та детальну реальну ситуацію зрозумілою моделлю, що відображає сутність задачі;

– *еволюція моделі* в ітераційному процесі розробки має тенденцію до збереження форми представлення, тоді як *трансформація моделі* передбачає зміну точки зору, з якої розглядається задача проєктування, та зміну структури моделі проєктування;

– *рефакторинг коду* – це процес зміни коду комп'ютерної програми для того, щоб зберегти її здатність до розвитку, покращити її читабельність або спростити структуру, зберігаючи при цьому наявну функціональність. Навчання рефакторингу не лише дає студентам практичні навички програмування, а й допомагає їм зрозуміти найважливіші принципи ІПЗ;

– *шаблони проєктування* можуть допомогти розробникам вирішити певні задачі проєктування, а також покращити наявні проєкти.

Перепроєктування програмного забезпечення для кращого повторного використання чи набуття інших якостей є ілюстрацією трансформації моделі. Поетапна трансформація моделі вимагає від студентів постійної оцінки розриву між тим, що зроблено, і тим, що необхідно зробити. Така діяльність особливо корисна для формування у студентів здатності до оцінки наявного рішення та аналізу зисків і витрат [8, с. 705].

Шаблони проєктування програмного забезпечення часто використовують для вирішення поширеної задачі шляхом «загального» підходу до неї. Й. ван Нікерк (Johan van Niekerk), Л. Футчер (Lynn Futcher) [16] вказують, що шаблон проєктування надає концептуальну модель рішення з найкращих практик, яку водночас використовують розробники для створення конкретної реалізації їх задачі. Використання шаблонів проєктування надає низку переваг: 1) шаблон надає орієнтир щодо найкращої практики; 2) використання шаблону надає розробникам спільний «словник» для легкого та чіткого обговорення складних концепцій проєктування. Завдяки цим та іншим перевагам шаблони проєктування часто вивчають на курсах проєктування програмного забезпечення [16, с. 75].

Шаблон можна описати як «контекстне рішення задачі»:

– контекст – це повторювана ситуація, в якій застосовується шаблон;

– задача стосується мети, що має бути досягнута у цьому контексті, але вона також стосується будь-яких обмежень, що виникають у контексті;

– задача повинна бути повторюваною;

– рішення забезпечує загальний проєкт (основне рішення), яке визначає сутність рішення задачі з урахуванням даного контексту та обмежень [16, с. 77].

Шаблони проєктування надають проєктувальникам програмного забезпечення три основні переваги:

– по-перше, відомо, що рішення є надійним, оскільки воно перевірене часом;

– по-друге, переваги та недоліки шаблону відомі заздалегідь, і вони можуть бути враховані під час ескізного проєктування;

– по-третє, шаблони утворюють загальний словник, який може полегшити спілкування між різними зацікавленими сторонами [16, с. 78].

Ч. Вільямс (Chad Williams) та С. Курковскі (Stan Kurkovsky) [20] підкреслюють, що процес навчання застосування доцільних шаблонів проєктування програмного забезпечення можна зробити творчим шляхом перетворення самого процесу проєктування на конструктивістське навчальне середовище. Для цього вони пропонують не обмежувати тематику студентських проєктів та створювати умови для виходу студентів із зони комфорту шляхом застосування нових апаратних платформ та інтерфейсів до них. Проміжна оцінка проєкту виконувалась за рівнем доцільності та кількістю застосованих шаблонів проєктування, а заключна – за рівнем креативності всього проєкту.

Фреймворк можна визначити як напівзавершену програму, що містить певні фіксовані складові, спільні для всіх програм у проблемній області, поряд із певними змінними складовими, унікальними для кожної програми, створеної з нього. Більшість комерційного програмного забезпечення розробляється за допомогою фреймворків шляхом розширення та налаштування стандартних загальних функцій, які вони надають. З. Алі (Zoya Ali), Дж. Болінгер (Joseph Bolinger), М. Герольд (Michael Herold), Т. Лінч (Thomas Lynch), Дж. Раманатан (Jay Ramanathan), Р. Рамнат (Rajiv Ramnath) [1] вказують, що розробники, які володіють принципами об'єктно-орієнтованого проєктування, не застосовують при розробці програмного забезпечення певного фреймворку, зосереджуючись на тому, щоб просто «змусити його працювати». Адаптація до нового фреймворку може стати викликом для розробників-початківців

через те, що застосування шаблонів проєктування в новому фреймворку може призвести до поганого проєктування та неправильного використання фреймворку. Автори [1] пропонують триступеневий процес навчання проєктування з використанням фреймворків, спрямований на подолання вказаної проблеми:

1. Спочатку студентам пропонується спроєктувати їх програму за допомогою об'єктно-орієнтованого проєктування. Використовуючи формулювання задачі, студентів спонукають до її «об'єктно-орієнтованого обмірковування» та використання попереднього досвіду для створення об'єктів, класів, обов'язків, взаємозв'язків, методів та інших сутностей UML.

2. Далі студентам пропонується переробити програму за допомогою шаблонів проєктування. Одним із таких шаблонів є шаблон модель-представлення-контролер (MVC). Ідея цієї моделі полягає в тому, щоб ізолювати логіку домену програми від способу подання даних користувачеві (тобто інтерфейсу користувача програми), щоб ці два дуже важливі компоненти будь-якої програми могли бути розроблені, реалізовані та підтримувані окремо.

3. Нарешті студентам пропонується скоригувати використання шаблону проєктування відповідно до обраного фреймворку.

Для кращого розуміння фреймворку та безшовної інтеграції проєкту з фреймворком студентів необхідно навчити шаблонів проєктування, що диктуються фреймворком, та порівняти їх зі стандартним шаблоном проєктування. «Як природне доповнення до шаблонів проєктування, UML є мовою спілкування студентів та викладачів» [17, с. 42].

Викладання UML стимулює до застосування низхідного підходу до ІПЗ: спочатку студентів навчають виявленню вимог до програмного забезпечення, а потім перетворенню вимоги на архітектуру програмного забезпечення, низькорівневий дизайн та, нарешті, реалізацію. UML часто використовують як «lingua franca» на всіх етапах життєвого циклу, тому велику увагу приділено тому, щоб студенти створювали високоякісні, синтаксично та семантично правильні діаграми UML [19, с.19].

Автори [1] пропонують методику, яка надасть студентам можливість скористатися перевагами фреймворку в реалізації їх проєкту:

1. Перефразуйте задачу та визначте у цьому формулюванні всі іменники та дієслова. Іменники є кандидатами в об'єкти, класи та атрибути, а дієслова – обов'язками.

2. Об'єднайте виділені іменники в класи. Для цього може знадобитися відкидання нерелевантних або синонімічних іменників.

3. Об'єднайте виділені дієслова у класи, екземпляри та обов'язки.

4. Призначте обов'язки, визначивши необхідні методи для їх виконання.

5. Пройдіться за сценарієм, щоб переконатися, що кожен сценарій підтримується методами, та визначте взаємозв'язки між ними.

Проєктування за своєю суттю є міждисциплінарним предметом, на який суттєво впливає людський фактор, тому викладання проєктування – це не лише процес навчання самого проєктування, а й процес подальшого розвитку соціальних навичок студентів [6, с. 70]. С. Яржабек (Stanislaw Jarzabek) [7] зазначає, що саме командно зорієнтовані проєктні курси створюють можливість навчати принципів ІПЗ, коли їх застосування дійсно необхідне та вигідне, та пропонує таку їх класифікацію:

(1) виробнича практика: студенти працюють над реальними задачами у промисловому середовищі;

(2) проєктні курси, в яких студенти працюють над проблемами в різних предметних галузях під керівництвом викладачів, що є експертами у них. Іноді такі курси будуються на реальних задачах з промисловості;

(3) проєктні курси, на яких студенти навчаються передових принципів проєктування ПЗ та їх застосування у власних проєктах. Оскільки для забезпечення зворотного зв'язку зі студентами викладачам необхідно детально вивчити артефакти проєктування, такі проєкти повинні контролюватися викладачами, що спеціалізуються на ІПЗ та добре знають задачі, над якими працюють студенти;

(4) проєкти, розроблені з нуля, на відміну від проєктів, у яких студенти розширюють наявне програмне забезпечення;

(5) проєкти на основі певної програмної платформи, такої як .NET, JEE, сервісу, мобільного пристрою або Facebook.

Навички командної роботи, комунікації та письма можна розвивати в усіх зазначених проєктних курсах. Інші навички досить складно реалізувати в межах одного проєктного курсу. Наприклад, мета проєктних курсів (1) та (2) типів – показати студентам реальність нечітких, невизначених та мінливих вимог. Нечіткі вимоги та проєктування програмного забезпечення – це не тільки дві типові ознаки розробки програмного забезпечення, а й найбільш значні його проблеми, які складно викладати в межах одного курсу. Проєктні курси, які знайомлять студентів із нечіткими та мінливими вимогами, як правило, менш структуровані та суворі, ніж курси, які навчають студентів застосуванню принципів проєктування. У навчанні застосування принципів проєктування (проєктний курс (3) типу) студентам необхідно надавати зразки ескізів проєктів та багато детальних відгуків про їх початкові спроби вдосконалити проєкт. Для ефективного керівництва студентами керівники повинні бути добре знайомі з проблемною галуззю та проєктними рішеннями, що може бути досить складно у проєктних курсах (1) та (2) типів.

Студенти пишуть підсумковий звіт, у якому вони документують плани проєктів, процес розробки, архітектурні та детальні (неархітектурні) проєктні рішення. Кожній команді дається одна година на презентацію своєї роботи. Викладачі оцінюють студентів на основі якості проєктування, здатності оцінювати проєктні рішення та аргументувати свій вибір з огляду на заявлені атрибути якості – повторне використання, розширюваність та ефективність стратегії оцінки запитів [7, с. 37-38].

Навчання на прикладі з активним зворотним зв'язком – ефективний спосіб привчити студентів до проєктування у великому масштабі. Приклади, що надають студентам, можуть включати ескізи архітектури програмного забезпечення, специфікації API, а також ілюстрації того, як застосовувати методи проєктування. Деякі студенти чітко слідують конкретному прикладу для створення власних проєктних рішень, у той час, як інші вчаться на прикладах, але потім упроваджують інновації, експериментують з ідеями та пропонують власні варіанти методів проєктування. В обох випадках найважливіше, щоб студенти чітко розуміли сутність методики проєктування. Хоча лекції висвітлюють теорію принципів проєктування, спілкування між командами та викладачами веде до їх розуміння. На початку навчання студенти особливо часто помиляються і потребують багато відгуків та конструктивних дискусій для виконання правильного проєктування. [7, с. 38].

Д. А. Тамбуррі (Damian A. Tamburri), М. Разавіан (Maryam Razavian), П. Лаго (Patricia Lago) [15] помітили, що такий підхід допомагає студентам в опануванні основних проблем проєктування програмного забезпечення: а) відповідальних та раціональних проєктних рішень – студенти вчаться міркувати та дійсно відповідати за власні проєктні рішення; б) спільного проєктування – студенти можуть конструктивно дискутувати з командами-«опонентами», досягаючи глибшого розуміння ролі проєктувальника; в) ітеративного проєктування – студенти навчаються на помилках та рішеннях інших людей, активно переглядаючи власні проєкти; г) «соціального» проєктування – студенти вчаться ефективно працювати в команді та справлятися з

критичними відгуками, спричиненими різним досвідом та баченням (а отже, й надаючи відгуки з дуже різних точок зору) [15, с. 61-62].

Ж. Л. Мюртаг (Jeanne L. Murtagh), Дж. Е. Гамільтон-мол. (John A. Hamilton, Jr.) [10] описують організацію проєктно-орієнтованого навчання, визначаючи такі бажані результати навчання студентів:

1. Студенти повинні використовувати знання з попередніх інформатичних курсів для розробки помірно складних комп'ютерних проєктів, виходячи з чітких, послідовних та обґрунтовано повних вимог, представлених викладачем у проєктному завданні.

2. Студенти повинні розробити високорівневі (архітектурні) проєкти програм та їх інтерфейсів й отримати схвалення викладача, перш ніж переходити до детального проєктування. Студенти повинні продемонструвати, що всі вимоги з проєктного завдання були розподілені за певними частинами архітектурного проєктування.

3. Студенти повинні розробити детальний (неархітектурний) проєкт програмного забезпечення та всіх інтерфейсів і отримати схвалення викладача, перш ніж переходити до реалізації (тобто написання коду).

4. Студенти повинні розробити план випробувань для кожного проєкту. План випробувань повинен демонструвати, як будуть перевірені всі вимоги до програмного забезпечення, а також містити графік тестування програмного забезпечення.

5. Студенти повинні розробити документацію, яка показує, наскільки їхнє програмне забезпечення та план випробувань відповідають вимогам стандарту IEEE/EIA 12207 [10, с. 5.577.2-5.577.3].

«Існує цікава паралель між викладанням та навчанням, з одного боку, та шаблонами проєктування, з іншого. Шаблони проєктування – це не лише гарна практика: вони є кульмінацією випробуваних та перевірених методів проєктування програмного забезпечення, які надають такі бажані властивості, як гнучкість та повторне використання. У кількох випадках, безперечно, як й інші проєктувальники програмного забезпечення, ми розв'язували задачу проєктування, і лише пізніше дізнавалися, що насправді просто застосували певний шаблон. Це заспокоює, але справа в тому, що ми підсвідомо застосували те, що прийнято вважати гарною практикою. З викладанням ми часто робимо те саме; ми несвідомо використовуємо методику, коріння якої – в усталеній теорії навчання. Однак в обох випадках застосування перевірених методів важливо для отримання якісних результатів» [17, с.40].

Я. Уоррен (Ian Warren) [17] методично обґрунтував програмні результати навчання проєктування програмного забезпечення:

1. *Визначення та опис цілей проєктування програмного забезпечення*. Цілі проєктування включають коректність, надійність, гнучкість, багаторазове використання та ефективність. Студенти повинні враховувати, що програмне забезпечення має бути не лише правильним, але й задовольняти останні чотири нефункціональні цілі.

2. *Інтерпретація та конструювання UML-моделей програмного забезпечення*. UML, будучи стандартною галузевою нотацією, є очевидним вибором для навчання проєктування, тому студенти повинні вміти читати та записувати моделі мовою UML.

3. *Пояснення поняття шаблону проєктування та опис підмножини шаблонів.* Шаблони проєктування втілюють перевірені дизайнерські рішення. Студенти повинні зрозуміти, що саме використання шаблонів, а не намагання вирішити задачу «з нуля», є інженерним підходом.

4. *Застосування шаблонів з адаптацією їх до вирішення реальних задач.* Поінформованість про шаблони є важливою, але не замінює здатності застосовувати шаблон для вирішення конкретної задачі. Цей програмний результат стосується

глибших, функціональних знань.

5. *Застосовувати нещодавно набуті та розвинені навички програмування.* Студенти-початківці мають хороші знання про основи ООП, але обмежені знання про бібліотеки класів Java та більш глибокі аспекти програмування. Цей програмний результат спрямований на озброєння студентів навичками реалізації програмних проєктів.

6. *Робота з відносно великими проєктами програмного забезпечення*. Оскільки досвід студентів у розробці програмного забезпечення, як правило, обмежений невеликими програмами, що включають кілька класів, ознайомлення студентів із великими проєктами – це хороша підготовка до роботи над заключним проєктом останнього року навчання та виробничої практики [17, с. 41-42].

Я. Уоррен виокремив методи, ефективні для навчання проєктуванню програмного забезпечення:

– *проблемні запитання виду «що буде, якщо ...?»*. Я. Уоррен наводить приклад використання таких запитань під час розгляду діаграм класів UML і, зокрема, відносин та множинних обмежень: «Представляючи діаграму класів, ми запитуємо студентів, що насправді означає обмеження множинності? Їм доводиться інтерпретувати діаграми та оцінювати власне розуміння, а не пасивно сприймати теорію. Коли ми бачимо, що студенти зрозуміли, ми можемо змінити обмеження; незначні зміни на діаграмі можуть значно вплинути на сутність моделі» [17, с. 43];

– *рольові ігри*: «При розгляді основних ідей об’єктно-орієнтованого підходу студенти діють як об’єкти та відтворюють сценарії, що показують, як між об’єктами формуються динамічні зв’язки та як опрацьовуються повідомлення... Рольова гра природно призводить до документування сценаріїв за допомогою UML – щойно студенти відпрацюють сценарій, вони документують його за допомогою діаграми взаємодії об’єктів UML. На основі початкової діаграми класів та діаграми взаємодії, створеної з рольової гри, ми можемо досліджувати питання добре сформованих моделей, де різні точки зору в кінцевому підсумку повинні бути взаємно узгодженими» [17, с. 43];

– *активне навчання*, від уведення діяльнісних фрагментів під час лекцій до сеансів інтерактивної та проблемно-орієнтованої діяльності як засобу набуття студентами функціональних знань щодо шаблонів проєктування [17, с. 48];

– *взаємонавчання* (peer-learning) проявляється в тому, що студенти мають офіційного партнера для навчання, з яким вони парно програмують та співпрацюють у вирішенні курсових завдань. Крім того, студенти працюють у малих групах над розв’язанням задач в аудиторії [17, с. 48].

Дж. Уіттл (Jon Whittle), К. Н. Булл (Christopher N. Bull), Дж. Лі (Jaejoon Lee), Дж. Котонья (Gerald Kotonya) [19] пропонують ще одну альтернативу традиційному навчанню – студійну освіту (studio-based education), що виводить на перше місце рефлексивну практику як спосіб розвитку навичок проєктування. У студії студенти запрошуються до критичного обмірковування власного та чужого дизайну, для цього використовуються різноманітні методи, такі як наставництво, критика дизайну, взаємонаставництво та узгоджене оцінювання. Студійний курс зазвичай, але не завжди, викладається у приміщенні, призначеному для цієї мети, тобто в студії. Фізична студія розглядається як ключовий елемент, оскільки вона дозволяє студентам постійно демонструвати власну роботу, що з часом заохочує рефлексивну практику. Студія також заохочує часті та неформальні взаємодії між студентами, що призводить до взаємонавчання [19, с. 12].

У табл. 1 наведено порівняльну характеристику середовища студійного та традиційного навчання.

При реалізації студійного курсу в Університеті Ланкастеру автори [19] виявили, що за студійного навчання студенти майже не використовували формальне моделювання (моделювання з використанням формалізованого опису, такого як UML, де використовуються точні позначення та правильний синтаксис), застосовуючи натомість неформальне моделювання з використанням або спеціальної нотації (наприклад, малювання фігур на дошці (рис. 1), ручкою на папері чи за допомогою інструменту цифрового ескізування), або вільної інтерпретації визнаної мови моделювання (наприклад, малювання діаграми класів UML без дотримання правильного синтаксису) [19, с. 16]. Хоча студенти в студії мало займалися формальним моделюванням, вони регулярно придумували неформальні моделі. Сюди входять як моделі UML (як правило, варіанти використання та діаграми класів), ескізи (наприклад, архітектура високого рівня, ескізи дизайну інтерфейсу користувача), так і моделі на основі процесів (наприклад, діаграми вигоряння, які використовуються в гнучкій розробці) [19, с. 17].

Таблиця № 1

Порівняльна характеристика студійного та традиційного навчання (за [19, с. 15])

Аспект	**Студійний курс**	**Традиційний курс**	**Приклад: студійний курс Університету Ланкастера**
Фізичне середовище	Наявна фізична кімната – студія, відкрита та змінювана з метою створення різноманітних групових, індивідуальних та соціальних просторів	Стандартна лабораторія	Спеціальна лабораторія з цілодобовим доступом, яку обслуговують самі студенти
Управління студією	Правила використання простору не повинні бути обмежувальними	Лабораторія суворо контролюється університетом	Цілодобовий доступ; дозволено їжу / напої; студенти мають права адміністратора
Режими освіти	Педагогічний персонал відіграє скоріше тренерську / менторську роль, ніж викладацьку	Студентам надається перелік документації, що має бути виготовлена	Студентам пропонується виробити стільки документації, скільки їм потрібно – відсутні нормативи щодо того, які діаграми чи позначення використовувати
Поінформованість	Робота розміщується для огляду (як незавершений або кінцевий продукт) – це допомагає студентам побачити роботи один одного	–	Студенти використовують мобільні дошки в студії для демонстрації роботи з проєктування, які залишаються на весь час виконання проєкту
Критика	Для надання зворотного зв’язку та розвитку ідей використовується постійна критика (офіційна та неофіційна, групова та індивідуальна, взаємна та викладацька)	Надається лише як частина щотижневих зустрічей з керівником проєкту	Забезпечується на постійній основі з використанням різноманітних методів: індивідуальних та групових демонстрацій / презентацій, неформального навчання, взаємної критики,

			критики з боку зовнішніх експертів (наприклад, компаній) та колегіального оцінювання
Культура	Студійна культура має бути соціальною та сприяти поширенню досвіду і підтримці хорошої трудової етики	Відсутність виділеної лабораторії означає, що студенти, як правило, зустрічаються лише в заздалегідь обумовлений час	Студенти використовують студію як дім, що веде до випадкових взаємодій та відчуття належності до спільноти
Натхнення	Під час проєктування студентів слід заохочувати до креативності у їх проєктах та рішеннях	Специфікація проєкту визначена викладачем наперед	Студенти пропонують свої власні ідеї проєктів під час керованого креативного мозкового штурму

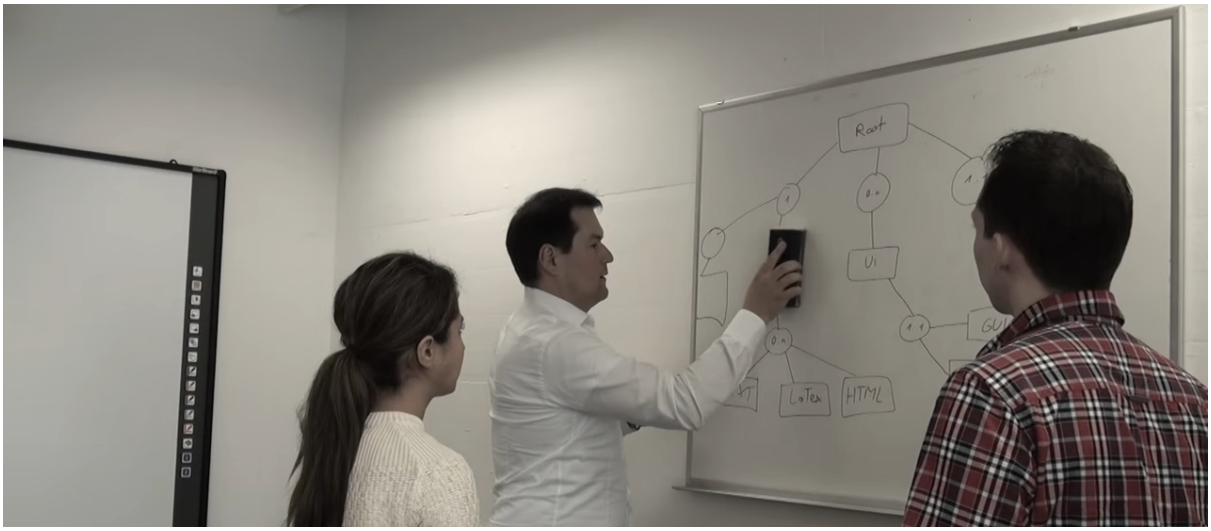

Рис. 1. Неформальне моделювання на дошці

За студійного підходу студенти займалися моделюванням стільки, скільки потрібно, і тоді, коли їм це було потрібно. Студенти не намагалися створити повністю відпрацьовані моделі для функції, яку вони збирались реалізувати – моделі використовувалися переважно для мозкового штурму та для продумування дизайну. Після того, як дизайн був продуманий достатньо для того, щоб дозволити команді перейти до наступного етапу розробки, моделі не змінювались, проте ними продовжували користуватись як артефактом проєктування: студенти, як правило, використовували їх або для нагадування про те, що вони робили, або як фоновий «шум», який певним чином допомагав їм працювати в групі [19, с. 18]. Такий підхід активно застосовується експертами з проєктування програмного забезпечення – так, М.Петре (Marian Petre) та А. ван дер Хук (André van der Hoek) у пораднику [11] використали неформальне моделювання для узагальнення власного досвіду проєктування (рис. 2).

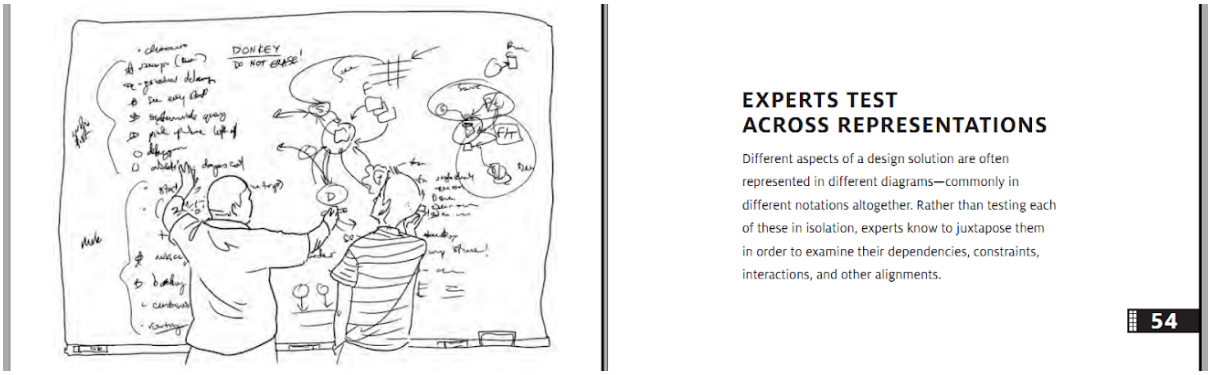

Рис. 2. Фрагмент порадника з проєктування М. Петре та А. ван дер Хука [11] у вигляді ескізного проєкту

Поєднання неформального та формального моделювання можливе за допомогою *гнучких засобів моделювання* (рис. 3), які після створення неформального ескізу надають можливість перейти до формального моделювання без зміни засобу [19, с. 17].

Автори студійного курсу [19] вважають, що студентів слід ознайомити із низкою методів проєктування, навчити застосувати ці методи на практиці та за допомогою рефлексивної практики заохотити їх дізнатися про позитивні та негативні сторони цих методів: «... можна стверджувати, що найважливіше, чого слід навчити, – це культура самої рефлексивної практики. Якщо студенти навчаться рефлексивним практикам, вони зможуть застосувати ці навички до будь-якого нового методу чи підходу, з якими вони зіткнуться у своїй майбутній професійній діяльності» [19, с. 20].

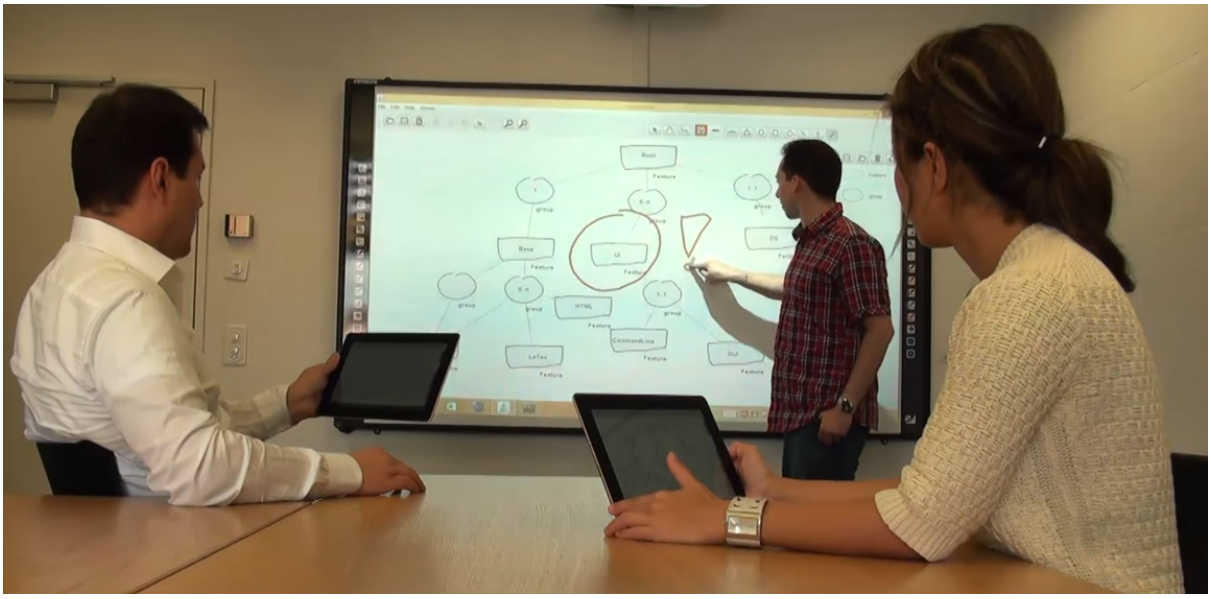

Рис. 3. Моделювання з використанням FlexiSketch Team

М. Петре та А. ван дер Хук [11, с. 148-158] рекомендують такі дії з рефлекції у процесі проєктування програмного забезпечення:

1. Не допускати при обговоренні проєкту відхилень на другорядні питання, регулярно перевіряти себе, «де я та що я роблю?»
2. На основі глибокого розуміння фундаментальних концепцій проєктування «тримати курс»: пам’ятати, які проєктні рішення вже прийняті, чому вони були саме такими та які ще рішення потрібно зробити.
3. Продумування того, що не проєктується: визначаючи та розглядаючи

обмеження проєктування, виявляти спроєктоване надмірно або недостатньо.

4. Періодично зупинятись для того, щоб подивитись на проєкт загалом, ставлячи собі питання, чи не змінились цілі клієнтів, сприйняття користувачів, сам ринок тощо.

5. Передбачити різні варіанти майбутнього, аналізуючи економічну доцільність для визначення того, куди вкластися – у методи, засоби, ресурси, альтернативи проєктування – для того, щоб зберегти зусилля в майбутньому.

У навчанні проєктування Ч. Ху рекомендує враховувати таке [6, с. 69-70]:

1. Технічна компетентність та пізнавальні можливості студентів можуть істотно вплинути на результати навчання проєктування. З цієї причини мета викладання проєктування програмного забезпечення не в тому, щоб зробити кожного студента дизайнером, а в тому, щоб студенти на практиці відчули, що може знадобитися для створення хорошого проєкту, одночасно набуваючи індивідуально досяжних навичок проєктування.

2. Для порівняно невеликих завдань проєктування керована тестами розробка надає можливість виражати ідеї моделювання безпосередньо у коді, більш ефективно оцінювати компроміси проєктування та допускати менше помилок при проєктуванні, ніж за використання діаграм.

3. Навчання «проєктування у малому» (написанню пов’язаних методів, доцільної інкапсуляції даних, використанню гарних назв методів тощо), імовірно, є набагато менш складним завданням, ніж вивчення проєктування загалом для дослідження структурної стійкості, яка вимагає проєктних компромісів щодо застосування знань про процес проєктування. Мінімальні базові здатності до проєктування (з відповідним рівнем проєктного мислення), які повинні здобути випускники, можуть включати здатність ефективно розпочинати проєктування, декомпозиціювати задачу, виконувати ітерації проєктування з обґрунтованими рішеннями щодо компромісів проєктування та реалізувати проєкт у коді.

4. Студенти не можуть ефективно здобути здатності до проєктування та навчитись проєктного мислення, якщо вони не розв’язують проблеми проєктування належної складності.

5. Незважаючи на відсутність універсальних формул або рецептів проєктування, аналіз даних (ідентифікація сутностей та їх відносин) та аналіз процесів (виявлення дій та логічних потоків) є критично важливими для розуміння, яку інформацію використовувати та як вона перетворюється.

6. Правильне документування проєкту – своєчасне, а не постфактум – є необхідним навиком, однак, чи повинні студенти використовувати UML або наскільки точно вони використовують мову для документування моделі, не так важливо. Студентів доцільно ознайомити з деякими класичними діаграмами, заохочуючи їх творче використання.

7. Для заключного оцінювання результатів навчання студентів доцільно поєднувати традиційні аудиторні випробування (щоб переконатися, що «вони це знають») з комплексними домашніми задачами проєктування, які повинні розв’язуватись окремо або командами з двох осіб із ретельним моніторингом та оцінкою не лише кінцевих продуктів проєктування, які виробляють студенти, а й якості виконання проєкту, задокументованої у звітах студентів.

Висновки та перспективи подальших досліджень

Узагальнення результатів дослідження надало можливість зробити такі висновки:

1. Проєктування програмного забезпечення є різновидом інженерного проєктування, у якому об’єднано дві складові – творча та системотехнічна.

Формування та розвиток інженерної творчості є технологією з опанування кращих зразків проєктів у процесі діяльності з проєктування, наближеної до виробничої. Результатом проєктування є оновлений та адаптований зразок (типовий проєкт, або шаблон проєктування) чи оригінальний новий дизайн (проєкт). У зв'язку з цим значний потенціал для розробки методики навчання проєктування програмного забезпечення майбутніх фахівців з ІПЗ наявний у суміжних дослідженнях як з інших галузей інженерії (зокрема, будівельної, механічної та комп'ютерної), так і з мистецтва (зокрема, архітектури, живопису).

2. У навчанні проєктування програмного забезпечення на особливу увагу заслуговує підтримка діяльності студентів із проєктування, теоретичні основи якого набуті за лекційною формою, у формі практичних занять, студій, курсових проєктів тощо із розв'язання реальних задач проєктування складного програмного забезпечення. Це висуває додаткову вимогу до викладачів проєктування – працювати на виробництві програмного забезпечення або бути тісно пов'язаними з його замовниками.

3. Моделювання програмного забезпечення відіграє ключову роль у його успішному проєктуванні. Водночас дотримання вимог використання точних формальних описів моделей у процесі проєктування не суттєво впливає на якість проєктування: зручність застосування точних формальних та гнучких неформальних моделей є порівняною. У зв'язку з цим доцільним є застосування гнучких методів та засобів моделювання, за яких побудова формальних діаграм UML виконується за неформальним ескізним проєктом.

4. Перехід від архітектурного проєктування на основі обраного шаблону до неархітектурного (детального) проєктування може потребувати адаптації шаблону до непроєктних реалій у вигляді інструментів конструювання програмного забезпечення, таких як фреймворку. Тому, незважаючи на те, що проєктування програмного забезпечення є можливим без його конструювання, доцільним результатом проєктування виступає принаймні сконструйований прототип програмного забезпечення: будучи частиною життєвого циклу програмного забезпечення, проєктування є рушійною силою його розвитку – воно постійно виконується аж до виведення програмного продукту з експлуатації.

5. Оцінювання сформованості компетентності з проєктування програмного забезпечення передбачає перевірку індивідуальних знань та командних умінь проєктування у процесі розв'язання комплексної задачі проєктування. Важливими для цього є артефакти проєктування – ескізні проєкти, формальні моделі, проєктна документація, створені прототипи тощо. У зв'язку з цим доцільною формою оцінювання є захист проєктів.

Перспективи подальшого розвитку цього дослідження полягають в обґрунтуванні третьої (після інженерії вимог та проєктної інженерії) інженерної складової ІПЗ – конструювання програмного забезпечення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Ali, Z., Bolinger, J., Herold, M., Lynch, T., Ramanathan, J., Ramnath, R. (2011). Teaching object-oriented software design within the context of software frameworks. *2011 Frontiers in Education Conference (FIE)*. New York : IEEE. https://doi.org/10.1109/fie.2011.6142889

2. Carrington, D. (1998). Teaching software design and testing. *FIE'98. 28th Annual Frontiers in Education Conference. Moving from "Teacher-Centered" to "Learner-Centered" Education. Conference Proceedings*. New York : IEEE. https://doi.org/10.1109/fie.1998.738732

3. IEEE Computer Society, Association for Computing Machinery (2020). *Computing*

Curricula 2020 : Paradigms for Global Computing Education, 205 p. (A Computing Curricula Series Report). https://www.acm.org/binaries/content/assets/education/curricula-recommendations/cc2020.pdf
4. Dym, C. L., Agogino, A. M., Eris, O., Frey, D. D., Leifer, L. J. (2005). Engineering Design Thinking, Teaching, and Learning. *Journal of Engineering Education*, 94(1), 103–120. https://doi.org/10.1002/j.2168-9830.2005.t
5. Graham, R. M. (1970). Teaching systems programming and software design. *ACM SIGCSE Bulletin*, 2(3), 56–60. https://doi.org/10.1145/873641.873652
6. Hu, C. (2013) The nature of software design and its teaching. *ACM Inroads*, 4(2), 62–72. https://doi.org/10.1145/2465085.2465103
7. Jarzabek, S. (2013). Teaching advanced software design in team-based project course. *2013 26th International Conference on Software Engineering Education and Training (CSEE&T)*. New York : IEEE. https://doi.org/10.1109/cseet.2013.6595234
8. Jia, Y., Tao, Y. (2009). Teaching Software Design Using a Case Study on Model Transformation. *2009 Sixth International Conference on Information Technology: New Generations*. New York : IEEE, 702–706. https://doi.org/10.1109/ITNG.2009.114
9. Lamm, E. (2003). Booch's Ada vs. Liskov's Java: Two Approaches to Teaching Software *Lecture Notes in Computer Science*, 2655, 102–112. https://doi.org/10.1007/3-540-44947-7_7
10. Murtagh, J. L., Hamilton Jr., J. A. (2000) Teaching A Real World Software Design Approach Within An Academic Environment. *2000 ASEE Annual Conference and Exposition: Engineering Education Beyond the Millenium, St. Louis, MO, 18 June 2000 - 21 June 2000*, 5.577.1–5.577.8. https://doi.org/10.18260/1-2—8739
11. Petre, M., van der Hoek, A. (2016). *Software Design Decoded: 66 Ways Experts Think*. Cambridge : The MIT Press, 184 p.
12. Pierce, K., Deneen, L., Shute, G. (1991). Teaching software design in the freshman year. *Lecture Notes in Computer Science*, 536, 219–231. https://doi.org/10.1007/bfb0024294
13. Semerikov, S., Striuk, A., Striuk, L., Striuk, M., Shalatska, H. (2020). Sustainability in Software Engineering Education: a case of general professional competencies. *E3S Web of Conferences*, 166, 10036. https://doi.org/10.1051/e3sconf/202016610036
14. Joint Task Force on Computing Curricula, IEEE Computer Society, Association for Computing Machinery (2015). *Software Engineering 2014 : Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*, 134 p. (A Volume of the Computing Curricula Series). https://computingcurricula.com/files/SE2014.pdf.
15. Tamburri, D. A., Razavian, M., Lago, P. (2013). Teaching Software Design with Social Engagement. *2013 26th International Conference on Software Engineering Education and Training (CSEE&T)*. New York : IEEE. https://doi.org/10.1109/CSEET.2013.6595237
16. van Niekerk, J., Futcher, L. (2015). The Use of Software Design Patterns to Teach Secure Software Design: An Integrated Approach. *IFIP Advances in Information and Communication Technology*, 453, 75–83. https://doi.org/10.1007/978-3-319-18500-2_7
17. Warren, I. (2005). Teaching Patterns and Software Design. *Conferences in Research and Practice in Information Technology Series*, 42, 39–49.
18. Weiss, D. M. (1987). Teaching a Software Design Methodology. *IEEE Transactions on Software Engineering*, SE-13(111), 1156–1163. https://doi.org/10.1109/tse.1987.232864
19. Whittle, J., Bull, C. N., Lee, J., Kotonya, G. (2014). Teaching in a Software Design Studio: Implications for Modeling Education. *CEUR Workshop Proceedings*, 1346, 12–21. http://ceur-ws.org/Vol-1346/edusymp2014_paper_1.pdf
20. Williams, C., Kurkovsky, S. (2017). Raspberry Pi creativity: A student-driven approach to teaching software design patterns. *2017 IEEE Frontiers in Education Conference (FIE)*. New York : IEEE. https://doi.org/10.1109/fie.2017.8190735

21. Пелевин, В. Н. (2010). *Формирование профессиональной компетентности будущих бакалавров по направлению «Информационные системы и технологии»* : автореф. дис. ... канд. пед. наук : 13.00.08 – теория и методика профессионального образования. Екатеринбург, 27 с.
22. Міністерство освіти і науки України (2018). *Про затвердження стандарту вищої освіти за спеціальністю 121 «Інженерія програмного забезпечення» для першого (бакалаврського) рівня вищої освіти* : наказ № 1166. https://mon.gov.ua/storage/app/media/vishcha-osvita/zatverdzeni%20standarty/12/21/121-inzheneriya-programnogo-zabezpechennya-bakalavr.pdf.
23. Striuk, A. (2018). Formation and development of software engineering as a knowledge area. *Journal of Information Technologies in Education (ITE)*, (37), 103-136. https://doi.org/10.14308/ite000684
24. Striuk, A. (2019). "Advanced course on software engineering" as the first model for training of software engineers. *Journal of Information Technologies in Education (ITE),* (40), 48-67. https://doi.org/10.14308/ite000702

REFERENCES (TRANSLATED AND TRANSLITERATED)

1. Ali, Z., Bolinger, J., Herold, M., Lynch, T., Ramanathan, J., Ramnath, R. (2011). Teaching object-oriented software design within the context of software frameworks. *2011 Frontiers in Education Conference (FIE)*. New York : IEEE. https://doi.org/10.1109/fie.2011.6142889
2. Carrington, D. (1998). Teaching software design and testing. *FIE'98. 28th Annual Frontiers in Education Conference. Moving from "Teacher-Centered" to "Learner-Centered" Education. Conference Proceedings*. New York : IEEE. https://doi.org/10.1109/fie.1998.738732
3. IEEE Computer Society, Association for Computing Machinery (2020). *Computing Curricula 2020 : Paradigms for Global Computing Education*, 205 p. (A Computing Curricula Series Report). https://www.acm.org/binaries/content/assets/education/curricula-recommendations/cc2020.pdf
4. Dym, C. L., Agogino, A. M., Eris, O., Frey, D. D., Leifer, L. J. (2005). Engineering Design Thinking, Teaching, and Learning. *Journal of Engineering Education*, 94(1), 103–120. https://doi.org/10.1002/j.2168-9830.2005.t
5. Graham, R. M. (1970). Teaching systems programming and software design. *ACM SIGCSE Bulletin*, 2(3), 56–60. https://doi.org/10.1145/873641.873652
6. Hu, C. (2013) The nature of software design and its teaching. *ACM Inroads*, 4(2), 62–72. https://doi.org/10.1145/2465085.2465103
7. Jarzabek, S. (2013). Teaching advanced software design in team-based project course. *2013 26th International Conference on Software Engineering Education and Training (CSEE&T)*. New York : IEEE. https://doi.org/10.1109/cseet.2013.6595234
8. Jia, Y., Tao, Y. (2009). Teaching Software Design Using a Case Study on Model Transformation. *2009 Sixth International Conference on Information Technology: New Generations*. New York : IEEE, 702–706. https://doi.org/10.1109/ITNG.2009.114
9. Lamm, E. (2003). Booch's Ada vs. Liskov's Java: Two Approaches to Teaching Software *Lecture Notes in Computer Science*, 2655, 102–112. https://doi.org/10.1007/3-540-44947-7_7
10. Murtagh, J. L., Hamilton Jr., J. A. (2000) Teaching A Real World Software Design Approach Within An Academic Environment. *2000 ASEE Annual Conference and Exposition: Engineering Education Beyond the Millenium, St. Louis, MO, 18 June 2000 - 21 June 2000*, 5.577.1–5.577.8. https://doi.org/10.18260/1-2—8739
11. Petre, M., van der Hoek, A. (2016). *Software Design Decoded: 66 Ways Experts*

Think. Cambridge : The MIT Press, 184 p.
12. Pierce, K., Deneen, L., Shute, G. (1991). Teaching software design in the freshman year. *Lecture Notes in Computer Science*, 536, 219–231. https://doi.org/10.1007/bfb0024294
13. Semerikov, S., Striuk, A., Striuk, L., Striuk, M., Shalatska, H. (2020). Sustainability in Software Engineering Education: a case of general professional competencies. *E3S Web of Conferences*, 166, 10036. https://doi.org/10.1051/e3sconf/202016610036
14. Joint Task Force on Computing Curricula, IEEE Computer Society, Association for Computing Machinery (2015). *Software Engineering 2014 : Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*, 134 p. (A Volume of the Computing Curricula Series). https://computingcurricula.com/files/SE2014.pdf.
15. Tamburri, D. A., Razavian, M., Lago, P. (2013). Teaching Software Design with Social Engagement. *2013 26th International Conference on Software Engineering Education and Training (CSEE&T)*. New York : IEEE. https://doi.org/10.1109/CSEET.2013.6595237
16. van Niekerk, J., Futcher, L. (2015). The Use of Software Design Patterns to Teach Secure Software Design: An Integrated Approach. *IFIP Advances in Information and Communication Technology*, 453, 75–83. https://doi.org/10.1007/978-3-319-18500-2_7
17. Warren, I. (2005). Teaching Patterns and Software Design. *Conferences in Research and Practice in Information Technology Series*, 42, 39–49.
18. Weiss, D. M. (1987). Teaching a Software Design Methodology. *IEEE Transactions on Software Engineering*, SE-13(111), 1156–1163. https://doi.org/10.1109/tse.1987.232864
19. Whittle, J., Bull, C. N., Lee, J., Kotonya, G. (2014). Teaching in a Software Design Studio: Implications for Modeling Education. *CEUR Workshop Proceedings*, 1346, 12–21. http://ceur-ws.org/Vol-1346/edusymp2014_paper_1.pdf
20. Williams, C., Kurkovsky, S. (2017). Raspberry Pi creativity: A student-driven approach to teaching software design patterns. *2017 IEEE Frontiers in Education Conference (FIE)*. New York : IEEE. https://doi.org/10.1109/fie.2017.8190735
21. Pelevin, V. N. (2010). *Formirovanie professionalnoi kompetentnosti budushchikh bakalavrov po napravleniiu «Informatcionnye sistemy i tekhnologii»* : avtoref. dis. ... kand. ped. nauk : 13.00.08 – teoriia i metodika professionalnogo obrazovaniia (*Formation of professional competence of future bachelors in the direction of "Information systems and technologies" :* authoref. dis. ... cand. ped. sciences : 13.00.08 – theory and methods of vocational education). Ekaterinburg, 27 s.
22. Ministerstvo osvity i nauky Ukrainy (2018). *Pro zatverdzhennia standartu vyshchoi osvity za spetsialnistiu 121 «Inzheneriia prohramnoho zabezpechennia» dlia pershoho (bakalavrskoho) rivnia vyshchoi osvity* : nakaz # 1166 (*On approval of the standard of higher education in the specialty 121 "Software Engineering" for the first (bachelor's) level of higher education* : order № 1166). https://mon.gov.ua/storage/app/media/vishcha-osvita/zatverdzeni%20standarty/12/21/121-inzheneriya-programnogo-zabezpechennya-bakalavr.pdf.
23. Striuk, A. (2018). Formation and development of software engineering as a knowledge area. *Journal of Information Technologies in Education (ITE)*, (37), 103-136. https://doi.org/10.14308/ite000684
24. Striuk, A. (2019). "Advanced course on software engineering" as the first model for training of software engineers. *Journal of Information Technologies in Education (ITE),* (40), 48-67. https://doi.org/10.14308/ite000702

Andrii Striuk
Kryvyi Rih National University, Kryvyi Rih, Ukraine

FORMATION OF SOFTWARE DESIGN SKILLS AMONG SOFTWARE ENGINEERING STUDENTS

The paper is devoted to the one of the competence components of a mobile-oriented environment for software engineering (SE) students. It is shown that the introduction of the higher education standard for SE bachelors has created a number of problems to ensure the quality of training, primarily related to low level of specification both for competencies and learning outcomes. The way to solve these problems is to design a detailed system of professional competencies for SE bachelors.

The paper considers approaches to the formation of the important special professional competency of future software engineers – K14 (ability to participate in software design, including modeling (formal description) of its structure, behavior and functioning processes). Based on a historical and genetic review of the software design training among SE students in the UK, USA, Canada, Australia, New Zealand and Singapore, recommendations for choosing of software design teaching techniques, selection of learning content, modeling and design tools, assessment of the level of formation of the relevant competence are developed. The example of the industrial-like software design training (studio training) is considered. The problems of transition from architectural to detailed design and project implementation are shown.

Prospects for further development of this study are to substantiate the third (after requirements engineering and design engineering) engineering component of SE – software constructing.

Key words: software engineering, software design, professional training of software engineering students, software design skills SE bachelors

Стаття надійшла до редакції 04.11.2021
The article was received 04 November 2021