МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

А.І. Безверхий, В.Г. Вербицький

ПРОФЕСІЙНА ПРАКТИКА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Навчально-методичний посібник
для здобувачів ступеня вищої освіти бакалавра
спеціальності «Інженерія програмного забезпечення»
освітньо-професійної програми «Програмне забезпечення систем»

Затверджено
Вченою радою ЗНУ
протокол № від

Запоріжжя
2020

УДК 621.396.218
Б 575

Безверхий А.І., Вербицький В.Г. Професійна практика інженерії програмного забезпечення : навчально-методичний посібник для здобувачів ступеня вищої освіти бакалавра спеціальності «Інженерія програмного забезпечення» освітньо-професійної програми «Програмне забезпечення систем». Запоріжжя : ЗНУ, 2020. 138 с.

В навчально-методичному посібнику подано в систематизованому вигляді програмний матеріал дисципліни «Професійна практика інженерії програмного забезпечення». Викладено огляд гнучких технологій розробки програмних застосунків, розглянуті фреймворки організації такої розробки на практиці та рекомендації з їх використання Для формування необхідних навичок запропонована командна розробка з елементами проектного навчання.

Наведено питання до самоконтролю, зразки завдань тестового контролю знань та список рекомендованої літератури.

Для здобувачів ступеня вищої освіти бакалавра спеціальності 121 «Інженерія програмного забезпечення» освітньо-професійної програми «Програмне забезпечення систем».

Рецензент

*М.Ю. Пазюк***,** доктор технічних наук, професор, завідувач кафедри автоматизації управління технологічними процесами Запорізького національного університету

Відповідальний за випуск

В.Г. Вербицький, доктор фізико-математичних наук, професор, завідувач кафедри програмного забезпечення автоматизованих систем

ЗМІСТ

ПЕРЕДМОВА

Зміст діяльності фахівців з інженерії програмного забезпечення (англ. Software engineering) полягає у розробці та супроводженні: прикладного програмного забезпечення, комп'ютерних систем та мереж, корпоративних систем, систем підтримки прийняття рішень, автоматизованих систем управління, інтелектуальних систем, програмних продуктів для бізнесу, мультимедійного програмного забезпечення, баз даних та знань, програмних систем діагностики та сертифікації, програмних засобів захисту інформації у комп'ютерних системах та мережах. Цей широкий спектр діяльності забезпечується цілим рядом дисциплін професійної підготовки студентів спеціальності 121 Інженерія програмного забезпечення. До них належить і «Професійна практика інженерії програмного забезпечення».

Метою викладання цієї навчальної дисципліни є засвоєння сучасних інженерних принципів створення надійного, якісного програмного забезпечення, що задовольняє пропонованим до нього вимогам; формування у студентів розуміння необхідності застосування даних принципів програмної інженерії. У даній дисципліні висвітлюються етичні, правові питання, які виникають під час професійної практики програмної інженерії та процесів, пов'язаних із ним. Також увага приділяється гнучкій розробці програмного забезпечення Agile у вигляді Scrum-реалізації. За словами Кена Швабера [1], Scrum — це не методологія, це фреймворк. А це означає, що Scrum не дає готових рецептів, що робити в тих або інших випадках – це визначається практичною проектною діяльністю. Тому, головним у цьому курсі є отримання студентом практичного досвіду командної розробки шляхом участі у розробці реальних програмних проектів.

Основними завданнями дисципліни «Професійна практика інженерії програмного забезпечення» є: оволодіння студентами історією комп'ютингу і програмної інженерії, принципами професійної діяльності і етики програмної інженерії, громадськими зобов'язання і зобов'язання з охорони довкілля , захистом інтелектуальної власності і іншого законодавства, значущого для діяльності по програмній інженерії, а також отримання навичок командної розробки програмного продукту із застосуванням гнучкої технології розробки програмного забезпечення.

У результаті вивчення навчальної дисципліни студент повинен знати:

- основи історії інженерії, комп'ютингу і програмній інженерії;
- розуміти роль стандартів і що визначають стандарти сукупностей знань в інженерії і програмній інженерії;
- усвідомлювати потребу в постійному підвищенні своєї кваліфікації як інженера, так і інженера по програмному забезпеченню;

- важливість безлічі різних співтовариств, значущих для програмної інженерії на регіональному рівні, в країні, а також на міжнародній арені.

Після вивчення навчальної дисципліни студент повинен вміти:

- приймати виважені рішення, при зіткненні з етичними дилемами, посилаючись як на загальні моральні принципи, так і на етичні кодекси інженерії, комп'ютингу і програмній інженерії;
- приділяти увагу безпеці, захищеності і правам людини в інженерії і управлінні ухваленням рішень;
- роз'яснювати і застосовувати законодавство, яке зачіпає програмну інженерію, включаючи законодавство по авторському праву, патентам і іншим аспектам інтелектуальної власності;
- описувати ефект, який роблять рішення в програмній інженерії на суспільство, економіку, соціальне середовище, їх замовників, партнерів.

Згідно з вимогами освітньо-професійної програми студенти повинні досягти таких компетентностей:

ЗК01. Здатність до абстрактного мислення, аналізу та синтезу.

ЗК05. Здатність вчитися і оволодівати сучасними знаннями.

ЗК07. Здатність працювати в команді.

ЗК08. Здатність діяти на основі етичних міркувань.

ЗК10. Здатність діяти соціально відповідально та свідомо.

ЗК11. Здатність реалізувати свої права і обов'язки як члена суспільства, усвідомлювати цінності громадянського вільного демократичного) суспільства та необхідність його сталого розвитку, верховенства права, прав і свобод людини і громадянина в Україні.

ФК14. Здатність брати участь у проектуванні програмного забезпечення, включаючи проведення моделювання (формальний опис) його структури, поведінки та процесів функціонування.

ФК17. Здатність дотримуватися специфікацій, стандартів, правил і рекомендацій в професійній галузі при реалізації процесів життєвого циклу.

РОЗДІЛ 1 ЕТИЧНІ, ПРАВОВІ ЗАСАДИ ТА СТАНДАРТИ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Визначення інженерії програмного забезпечення

У 1958 всесвітньо відомий статистик Джон Т'юкі (John Tukey) вперше ввів термін **Software** - програмне забезпечення. У 1972 році IEEE випустив перший номер журналу «Transactions on **Software Engineering**» - "Праці з Інженерії програмного забезпечення".

Інженерія програмного забезпечення (Software Engineering) є галуззю інформатики, яка вивчає питання побудови комп'ютерних програм, відображає закономірності розвитку програмування, узагальнює досвід програмування у вигляді комплексу знань і правил регламентації інженерної діяльності розробників програмного забезпечення.

У цьому визначенні виділимо два основних аспекти:

1. Інженерію програмного забезпечення можна розглядати як інженерну дисципліну, в якій інженери застосовують **теоретичні ідеї, методи і засоби розробки програмного забезпечення**, створюють продукти відповідно до стандартів, що регламентують процеси їх проектування та розробки.
2. Інженерія програмного забезпечення описує **методи управління програмним проєктом, його якістю та ризиками.** Застосування таких методів дозволяє досягти високої якості програмних продуктів.

Стандарт всесвітньої організації – Інституту інженерів з електротехніки і електроніки (IEEE) визначає суть інженерії програмного забезпечення як застосування систематизованого, наукового і розрахованого підходу до створення, функціонування і супроводу програмного забезпечення.

На відміну від науки, мета якої - отримання знань, для Software Engineering знання - це спосіб отримання деякого прибутку.

Необхідність поєднання інженерних підходів з розробкою програмного забезпечення виникла у кінці 60-тих років минулого століття у наслідок ускладнення і збільшення об'ємів програмного забезпечення. Ці причини викликали необхідність інженерних підходів для створення технологій для колективної розробки програмного забезпечення.

Таким чином розробку програмних систем можна вважати інженерною діяльністю, але вона має деякі відмінності від традиційної інженерії:

- гілки інженерії мають високу ступінь спеціалізації, а в програмній інженерії **спеціалізація торкнулася тільки окремих областей** (наприклад, операційні системи, транслятори, редактори і т. п.);

- програмування об'єктів **ґрунтується на стандартах**, за допомогою яких відображаються типові вимоги замовників, тобто типізація об'єктів і артефактів в сфері програмування;
- технічні рішення класифіковані і каталогізовані, а в програмній інженерії кожна нова розробка - **це нова проблема**, для реалізації якої встановлюють аналогію з раніше розробленими системами.
- **робота в команді - одна з найважливіших навичок**, яку прагнуть роботодавці, вона занесена Світовим економічним форумом у 10 навичок, необхідних для досягнення успіху в 2020, 2022, 2025 роках.

1.2 Професійна діяльність в галузі інженерії програмного забезпечення

У світі і в Україні створені професійні об'єднання і асоціації з програмної інженерії. Найвідоміші з них представлені далі.

1. Асоціація обчислювальної техніки (*Association for Computing Machinery)*

Рік створення 1947.**Розташування** США, Нью-Йорк.

Сайти www.acm.org, www.utacm.org

Найперша і найбільша асоціація з обчислювальної техніки. Об'єднує близько 83 000 фахівців. Штаб-квартира знаходиться в Нью-Йорку.

Щорічно асоціація присуджує Премію Тьюринга і премію імені Грейс Мюррей Хоппер. У число нагород також входять Премія Геделя, Премія Дейкстри, Премія Кнута, Премія Гордона Белла і Премія Паріса Канеллакіса.

ACM складається з більш ніж 170 регіональних відділень і 35 спеціальних тематичних груп (англ. Special interest group, SIG), в рамках яких і ведеться основна діяльність.

Деякі тематичні групи:

- SIGACT [en] - теорія алгоритмів і обчислень,
- SIGAI [en] - штучний інтелект,
- SIGCHI [en] - людино-машинне взаємодія,
- SIGCOMM [en] - передача даних,
- SIGGRAPH - комп'ютерна графіка,
- SIGKDD [en] - Data Science,
- SIGMM [en] - мультимедіа,
- SIGMOD [en] - управління даними,
- SIGOPS - операційні системи,
- SIGPLAN [en] - мови програмування,
- SIGSOFT [en] - розробка програмного забезпечення,
- SIGWEB [en] - веб-технології.

Додатково є більше 500 вузівських відділень. Перший студентський філал організації був створений в 1961 році в Університеті Луїзіани в Лафайетті.

2. **Інститут інженерів з електротехніки і електроніки** — **IEEE** (англ. *Institute of Electrical and Electronics Engineers*) (I triple E — «Ай тріпл і») — міжнародна некомерційна асоціація спеціалістів в області техніки, світовий лідер в області розробки стандартів з радіоелектроніки і електротехніки та програмування.

3. **Асоціація «Інформаційні технології України».**

У квітні 2004 року українські IT-компанії - «Miratech», «SoftLine», «Mirasoft», «ProFIX», «Ukrsoft», «SoftServe» прийняли рішення про створення Асоціації **«Інформаційні технології України».** Компанії-засновники Асоціації є провідними українськими розробниками програмного забезпечення, які накопичили успішний досвід в області продажу та ліцензування послуг на розвинених ринках Північної Америки та Європи.

Метою створення Асоціації є консолідація зусиль по просуванню на зовнішніх ринках конкурентоспроможної продукції українських компаній, злиття наукового і виробничого потенціалу України для створення програмного забезпечення, яке відповідає міжнародним стандартам, а також умов для подальшого розвитку галузі.

Сайт: http://itukraine.org.ua/

Кодекс етики та професійної діяльності в області програмної інженерії (версія 5.2)

Рекомендований ACM / IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices і спільно схвалений ACM і IEEE-CS в якості стандарту навчання і роботи в області програмної інженерії.

Комп'ютери грають центральну і все зростаючу роль в торгівлі, промисловості, управлінні, медицині, освіті, дозвіллі та в житті суспільства в цілому. Програмні інженери - це ті, хто вносить свій внесок, або безпосередньо, або через навчання, в аналіз, розробку специфікацій, проектування, реалізацію, сертифікацію, підтримку і тестування програмних систем. Граючи важливу роль в розробці програмних систем, програмні інженери мають значні можливості творити добро або завдавати зло, дозволяти іншим творити добро або завдавати зло, або впливати на тих, хто творить добро або заподіює зло. Щоб забезпечити, наскільки це можливо, що їх зусилля будуть використані в благих цілях, програмні інженери повинні неухильно перетворювати програмну інженерію в корисну і шановану професію. Для цього програмісти повинні твердо дотримуватися наступного Кодексу професійної етики. Кодекс містить вісім Принципів, які впливають на лінію поведінки і вибір рішення програмними інженерами, включаючи практиків, викладачів, менеджерів і вище керівництво, а також учнів і студентів. Принципи визначають етику

відносин між окремими інженерами, групами і організаціями, а також пов'язані з цим зобов'язання. У кожен Принцип включені ілюстрації деяких зобов'язань, що накладаються цими відносинами. Ці зобов'язання ґрунтуються на гуманності професії програмного інженера, особливої уваги по відношенню до людей, на яких впливає діяльність програмних інженерів, і унікальності цієї діяльності. Кодекс проголошує ці зобов'язання для всіх, хто відносить себе до програмних інженерам або збирається їм стати. Окремі частини Кодексу не можуть бути використані ізольовано від інших для виправдання упущень і проступків. Перелік Принципів та Положень далеко не вичерпаний. Положення не можуть розглядатися як розділяють професійна поведінка на прийнятне і неприйнятне в усіх реальних ситуаціях. Кодекс не є простим етичним алгоритмом, генеруючим етичні рішення. У деяких ситуаціях стандарти можуть суперечити один одному або іншим стандартам. Такі ситуації вимагають від програмного інженера дій відповідно до духу Кодексу професійної етики в залежності від конкретних обставин.

Слід вдумливо використовувати основні положення етики, а не сліпо покладатися на її докладні вказівки. Ці Принципи спонукають програмних інженерів усвідомити, на кого впливає виконувана ними робота; розібратися, чи належать вони і їхні колеги до оточуючих з належною повагою; взяти до уваги, як суспільство, будучи поінформованим належним чином, поставилося б до їхніх рішень; нарешті, оцінити, чи відповідають їхні професійні дії ідеалам програмної інженерії. У всіх цих оцінках турбота про добробут, безпеку і процвітання суспільства є первинною; тобто, «Інтереси суспільства» є центральними в даному Кодексі.

Динамічний і вимогливий контекст програмної інженерії вимагає кодексу, який можна пристосувати до нових ситуацій в міру їх появи. Однак навіть в такому загальному вигляді Кодекс забезпечує підтримку програмним інженерам і їх керівникам, які потребують правильному виборі дій в специфічних умовах, шляхом документування професійних етичних установок. Кодекс забезпечує етичну базу, до якої можу звертатися як окремі члени команд, так і команди в цілому. Кодекс дозволяє визначити дії, які етично недоречно вимагати від програмних інженерів або їх команд.

Даний кодекс призначений не тільки для оцінки спірних дій; він має також важливе освітнє значення. Оскільки в ньому виражене спільну думку щодо етичної сторони професії, він є засобом, що дозволяє довести до відома як суспільства, так і професіоналів етичні зобов'язання всіх програмних інженерів. Програмні інженери повинні прагнути до того, щоб зробити аналіз, розробку специфікацій, проектування, реалізацію, тестування і підтримку програмного забезпечення корисною та шанованою професією. Відповідно до їх високої відповідальності за добробут, безпеку і процвітання суспільства програмні інженери повинні твердо дотримуватися наступних восьми Принципів:

Принцип 1: СУСПІЛЬСТВО

Програмні інженери повинні діяти неухильно в інтересах суспільства. Зокрема, програмні інженери повинні:

1. Нести повну відповідальність за свою роботу.

2. Обмежувати інтереси програмних інженерів, роботодавців, клієнтів і користувачів користю для суспільства в цілому.

3. Схвалювати програмне забезпечення лише в разі, якщо вони твердо переконані в тому, що воно безпечне, відповідає специфікаціям, пройшло відповідне тестування і не загрожує якості життя, чи не порушує приватність і не шкодить навколишньому середовищу. Результат роботи повинен безумовно служити на благо суспільству.

4. Доводити до відома уповноважених осіб та організацій дійсну або потенційну небезпеку для користувачів, суспільства або довкілля, яка, на їхню думку, пов'язана з використанням програмного забезпечення або супутньою йому документації.

5. Брати участь в роботі над проблемами, що викликають тривогу в суспільстві, що стосуються програмного забезпечення, його інсталяції, розвитку, підтримки або документування.

6. Бути чесним і не допускати брехні у всіх висловлюваннях, особливо публічних, по відношенню до програмного забезпечення або пов'язаних з ним документації, методик та інструментів.

7.Звертати увагу на проблеми, пов'язані з фізичними вадами, розподілом ресурсів, економічної відсталості і іншими факторами, здатними обмежити доступ до користування програмним забезпеченням.

8. Бути готовим добровільно використовувати свою професійну майстерність для загального блага і сприяти поширенню знань про свою професію.

Принцип 2: КЛІЄНТ І РОБОТОДАВЕЦЬ

Програмні інженери повинні діяти відповідно до інтересів клієнта і

роботодавця, якщо вони не суперечать інтересам суспільства. Зокрема, програмні інженери повинні:

1. Надавати послуги в межах своєї компетентності, бути чесними і не приховувати обмеженості свого освіти і досвіду.

2. Чи не використовувати програмне забезпечення, отримане або свідомо нелегальним, або неетичним шляхом.

3. Користуватися власністю клієнта або роботодавця тільки належним чином і з їх відома.

4. Переконатися, що всі вони вживали документи, які повинні бути затверджені, дійсно затверджені уповноваженою особою.

5. Зберігати в таємниці будь-яку конфіденційну інформацію, отриману під час виконання професійних обов'язків, якщо це не суперечить інтересам суспільства і законодавству.

6. Ідентифікувати, документувати, збирати факти і негайно сповіщати клієнта або роботодавця, якщо, на їхню думку, проект близький до провалу, виявляється надто дорогим, порушує закон про інтелектуальну власність або може спричинити інші проблеми

7. Ідентифікувати, документувати і доповідати роботодавцю або клієнту про соціальні проблеми, пов'язаних з програмної і супутньої документації, про які їм стало відомо.

8. Не приймати пропозицій побічної роботи, яка може завдати шкоди роботі, що виконується для основного роботодавця.

9. Чи не діяти проти інтересів роботодавця або клієнта, за винятком випадків, коли це суперечить вищим етичних міркувань; в цьому випадку слід інформувати роботодавця або інша уповноважена особа про ці міркування.

Принцип 3: ПРОДУКТ

Програмні інженери повинні забезпечувати відповідність якості своїх продуктів і їх модифікацій найвищим можливим професійним стандартам. Зокрема, програмні інженери повинні:

1. Прагнути до високої якості, прийнятною вартістю і розумним термінів виконання проектів, доводячи істотні альтернативи до відома роботодавця і клієнта, заручившись їх згодою з вибором, а також ставлячи користувачів і суспільство до відома про них.

2. Забезпечити адекватність і досяжність цілей і спрямованості для всіх проектів, над якими вони працюють або мають намір працювати.

3. Виявляти, визначати та вживати заходів щодо проблем, пов'язаних з проектом, над яким вони працюють, і які мають відношення до етики, економіці, культурі, законності і навколишньому середовищу.

4. Гарантувати, що їх освіта, підготовка і досвід достатні для всіх проектів, над якими вони працюють або мають намір працювати.

5. Гарантувати, що у всіх проектах, над якими вони працюють або мають намір працювати, використовуються належні методики.

6. Працювати, слідуючи найбільш підходящим професійним стандартам і відступаючи від них лише в тих випадках, коли це виправдано з етичних або технічних причин.

7. Прагнути до повного розуміння специфікацій програмного забезпечення, над яким вони працюють.

8. Гарантувати, що специфікації на програмне забезпечення, над яким вони працюють, добре задокументовані, відповідають вимогам користувачів і затверджені належним чином.

9.Гарантувати реалістичність кількісних оцінок вартості, строків виконання, трудовитрат, якості і витрат за всіма проектами, над якими вони працюють або мають намір працювати, а також невизначеності цих оцінок.

10.Гарантувати адекватність тестування, налагодження та ревізій програмного забезпечення та супутньої документації, над якими вони працюють.

11.Гарантувати адекватність документації, включаючи виявлені проблеми і їх схвалені рішення, для всіх проектів, над якими вони працюють.

12.Розробляти програмне забезпечення і супутню документацію, ставлячись з повагою до приватності щодо тих, чиї інтереси зачіпає дане програмне забезпечення.

13.Використовувати тільки надійні дані, отримані прийнятними з точки зору моралі і закону засобами, і використовувати їх тільки належним чином.

14.Підтримувати цілісність даних, схильних до старіння і втрати актуальності.

15.Ставитися до всіх видів підтримки програмного забезпечення з тим же професіоналізмом, що і до нових розробок.

Принцип 4: ОЦІНКИ

Програмні інженери повинні підтримувати цілісність і незалежність своїх професійних оцінок. Зокрема, програмні інженери повинні:

1. Направляти всі технічні судження на службу людським цінностям.

2. Рекомендувати лише ті документи, які або розроблені під їх контролем, або ті, які знаходяться в області їх компетентності та з вмістом яких вони згодні.

3. Дотримуватися професійну об'єктивність по відношенню до програмного забезпечення або супутньою документації, які їх попросили оцінити.

4. Не брати участі у фінансових махінаціях, таких як підкуп, подвійна оплата та інші незаконні фінансові дії.

5.Розкривати всім зацікавленим сторонам конфлікти інтересів, яких неможливо уникнути розумними засобами.

6. Відмовлятися від участі в якості члена команди або радника в приватних, урядових чи професійних заходах, пов'язаних з програмним забезпеченням, через які може бути завдано потенційний збиток їх власним інтересам, інтересам їхніх роботодавців або клієнтів.

Принцип 5: МЕНЕДЖМЕНТ

Програмні інженери-менеджери та провідні співробітники повинні дотримуватися етичних підходів до управління розробкою і підтримкою програмного забезпечення і просувати ці підходи. Зокрема, керівники і провідні фахівці в області програмної інженерії повинні:

1. Гарантувати якісне управління всіма проектами, над якими вони працюють, включаючи ефективні процедури підвищення якості та зменшення ризику.

2. Гарантувати, що програмні інженери навчені стандартам перед тим, як мають намір дотримуватися їх.

3. Гарантувати, що програмні інженери знають політики і процедури роботодавця щодо захисту паролів, файлів і конфіденційної інформації, що стосується роботодавця чи інших осіб.

4. Розподіляти роботу тільки після з'ясування освіти і досвіду співробітника, з огляду на його бажання вдосконалювати свої освіту і досвід.

5. Гарантувати реалістичність кількісних оцінок вартості, строків виконання, трудовитрат, якості і прибутку по всіх проектах, над якими вони працюють або мають намір працювати, а також невизначеності цих оцінок.

6. Залучати до роботи програмних інженерів тільки після того, як їм надано повний і точний опис умов роботи.

7. Пропонувати справедливу винагороду за працю.

8. Чи не перешкоджати безпричинно призначенням співробітника на посаду, для якої він має відповідну кваліфікацію.

9. Гарантувати справедливе угоду щодо прав власності на будь-яке програмне забезпечення, технологію, дослідження, рукописи та іншу інтелектуальну власність, в яку програмний інженер вніс свій внесок.

10. Належним чином повідомляти про відповідальність за порушення політики роботодавця або даного Кодексу.

11. Чи не вимагати від програмного інженера нічого, що суперечить цьому Кодексу.

12. Чи не карати нікого, що виражає заклопотаність у зв'язку з етичними проблемами, пов'язаними з проектом.

Принцип 6: ПРОФЕСІЯ

Програмні інженери повинні піднімати престиж і репутацію своєї професії в інтересах суспільства. Зокрема, програмні інженери повинні:

1. Сприяти створенню в організації атмосфери, що сприяє етичної поведінки.

2. Поширювати знання в області програмної інженерії.

3. Розширювати знання в області програмної інженерії шляхом участі в професійних організаціях і зборах, а також своїми публікаціями.

4. Підтримувати інших колег, що прагнуть дотримуватися даного Кодексу.

5. Чи не ставити власні інтереси вище професійних інтересів, інтересів клієнта або роботодавця.

6. Підкорятися всім законам, що регулюють їх роботу, за винятком особливих ситуацій, коли це суперечить інтересам суспільства.

7. Бути точним в оцінках програмного забезпечення, над яким вони працюють, уникаючи не тільки свідомо брехливих обіцянок, а й обіцянок, які справедливо можуть бути сприйняті як спекулятивні, необгрунтовані, що вводять в оману, що збивають з пантелику або сумнівні.

8. Приймати відповідальність за виявлення, виправлення і оповіщення про помилки в програмному забезпеченні і пов'язаної з ним документації, над якими він працює.

9. Довести до відома клієнтів, роботодавців і керівництво про те, що програмні інженери слідують цьому Кодексу етики, і про наслідки цього.

10. Уникати організацій, які знаходяться в конфлікті з цим Кодексом.

11. Усвідомлювати, що порушення даного Кодексу несумісні з приналежністю до професійних програмним інженерам.

12. Висловлювати свою заклопотаність у разі істотного порушення даного Кодексу людям, причетним до цього, за винятком випадків, коли це неможливо, призводить до серйозних конфліктів або небезпечно.

13. Повідомляти про випадки істотного порушення даного Кодексу до відповідних інстанцій, якщо очевидно, що діалог з причетними до цього людьми неможливий, призводить до серйозних конфліктів або небезпечний.

Принцип 7: КОЛЕГИ

Програмні інженери повинні бути справедливі по відношенню до своїх колег, допомагати їм і підтримувати. Зокрема, програмні інженери повинні:

1. Закликати колег дотримуватися даного Кодексу.

2. Допомагати колегам в професійному зростанні.

3. Поважати роботу інших, але утримуватися від необґрунтованого довіри до неї.

4. Оглядати роботу інших об'єктивно, неупереджено, документуючи належним чином.

5. Прислухатися до думки, заклопотаності або скарг колег.

6. Допомагати колегам в освоєнні поточних робочих стандартів, включаючи політики і процедури захисту паролів, файлів та іншої конфіденційної інформації, а також заходів безпеки в цілому.

7. Чи не втручатися без необхідності в робочі справи колег; проте, щира турбота про інтереси роботодавця, клієнта або суспільства можуть змусити програмного інженера поставити під сумнів компетентність колеги.

8. У ситуаціях, що виходять за межі їх власної компетентності, питати думку інших професіоналів, компетентних в даній області.

Принцип 8: ОСОБИСТЕ ВІДПОВІДАЛЬНІСТЬ

Програмні інженери повинні постійно вчитися навичкам своєї професії і сприяти просуванню етичного підходу до своєї діяльності. Зокрема, програмні інженери повинні безперервно прагнути до наступного:

1. Поглиблювати свої знання в області аналізу, специфікації, проектування, розробки, підтримки і тестування програмного забезпечення та супутньої документації, а також управління процесом розробки.

2. Удосконалювати свої здібності до створення безпечного, надійного та функціонального якісного програмного забезпечення за розумною ціною і в розумні терміни.

3. Удосконалювати свої здібності до виробництва точної, інформативною, якісно написаної документації.

4. Удосконалювати знання програмного забезпечення та супутньої документації, над якою вони працюють, а також середовища, в якій вони будуть використовуватися.

5. Удосконалювати знання відповідних стандартів і законів, що регулюють програмне забезпечення і супутню документацію, над якими вони працюють.

6. Удосконалювати знання даного Кодексу, його інтерпретацію і використання у своїй роботі.

7. Не допускати несправедливого поводження з ким-небудь з причини не відносяться до справи упереджень.

8. Чи не підбурювати інших до дій, які порушують цей Кодекс.

9. Усвідомлювати, що приватне порушення даного Кодексу є несумісним з приналежністю до професійних програмним інженерам.

Етикет в Інтернеті. Основними користувачами мереж Інтернет спочатку були, в основному, працівники державних установ і наукових організацій. Порядок і способи використання Інтернету описувалися в інструкціях. Етикет використання мереж ґрунтувався на усталених в наукових колах нормах спілкування і обміну інформацією. З розвитком техніки і комунікацій в Інтернеті стало більше користувачів, що не відносяться ні до державних чиновників, ні до вчених мужів. Багато з них використовували Інтернет саме в тих цілях, для яких він створювався - для пошуку інформації. Для інших Інтернет став місцем задоволення своєї цікавості і особистих амбіцій. Поступово Інтернет із товариства чисто інформаційних мереж перетворюється в один з видів розваги, залишаючись при цьому, в першу чергу, джерелом інформації.

З розвитком міжнародної системи «електронних» грошей багато фірм висунули і реалізували концепцію продажу товарів через Інтернет. Тепер клієнт може переглянути і замовити товар, не виходячи з дому. У свою чергу доступність електронної пошти для користувачів дозволила оцінити її переваги в порівнянні з традиційними видами пошти. Відправлений лист може виявитися на іншому боці земної кулі за годину. Стало очевидним, що з'явилася необхідність в виробленні своїх норм етикету як для користувачів, так і для тих, хто їх обслуговує. У різних наукових установах, в електронних конференціях користувачів в процесі обговорення з'являються нові норми поведінки - нетікет (netiquette, від англійського net - «мережа» і французького etiquette - «етикет»). Обговорення цих норм триває і донині, хоча і з'явилися деякі засадничі правила. Зміст цих правил залежить від виду використання Інтернету.

Етикет в локальних комп'ютерних мережах. У багатьох організаціях правила поведінки користувача в локальних мережах встановлюються у формі інструкцій або офіційних правил. Чимало організацій, в яких будь-яких правил та інструкцій офіційно не існує, та й далеко не будь-яка інструкція може охопити всі питання етикету. В процесі розвитку локальних мереж вироблені деякі загальні правила (хоча застосування того чи іншого правила залежить від технічного оснащення мережі):

1. Не передавайте нікому ваше ім'я і пароль для входу в мережу: будь-які дії, вчинені в мережі під вашим ім'ям, потім можуть бути співвіднесені безпосередньо з вами;
2. Якщо ви залишаєте комп'ютер більш ніж на 10 хвилин, перед відходом припиніть виконання всіх програм з мережевою підтримкою (або пов'язаних з обміном даних по мережі) і закрийте їх (якщо це неможливо зробити в силу виконуваної завдання, то попередьте про цей факт свого адміністратора мережі);
3. Намагайтеся без необхідності не запускати кілька програм з мережевою підтримкою;
4. Перш ніж почати переміщення великого обсягу даних з іншого комп'ютера на свій або зі свого комп'ютера на інший комп'ютер в мережі, оцініть необхідність цього дії і можливість розбиття даних на окремі менші за обсягом пакети. Тільки в разі неможливості вирішення питання зазначеним чином вдавайтеся до переміщення всіх даних;
5. При наявності у вашого комп'ютера свого жорсткого диска віддавайте перевагу збереженню даних саме на ньому, а не на дисках загального користування (якщо таким не є диск вашого комп'ютера і якщо це правило не суперечить важливість справ);
6. Користуючись загальним (системним) поштовою скринькою, намагайтеся уникати поміщати туди дуже великі повідомлення;
7. Перед установкою на ваш комп'ютер нового програмного забезпечення з мережевою підтримкою або з можливим колективним використанням проконсультуйтеся з мережевим адміністратором і перевірте встановлюється програмне забезпечення на ліцензійну чистоту і на незараженість вірусами;
8. Стежте за тим, щоб працюючі у вас програми не завдавали шкоди будь-яким загальним (мережевим) ресурсів та ресурсів інших користувачів мережі.

Застосування колективного принтера накладає на членів локальної мережі певні додаткові правила:

1. Стежте, щоб не роздруковувалися зайві копії відправленого вами завдання;

2. Намагайтеся не роздруковувати що-небудь (документ і т. П.) Відразу після внесення кожного незначного зміни - багато програм дозволяють переглянути зразок можливої роздруківки на екрані;
3. Стежте за тим, щоб ваші роздруківки не скупчувалися у принтера - забирайте їх, по можливості, відразу після закінчення друку.

У разі появи питань по використанню мережі або програм, що використовують мережеві ресурси, зверніться до адміністратора або скористайтеся відповідною документацією (при наявності такої)

1.3 Акредитація в інженерії програмного забезпечення

Основні акредитовані посади розробників програмного забезпечення

1 Frontend Developer,

2 .NET Developer,

3 ASP.NET MVC Developer,

4 Java Developer,

5 PHP Developer,

6 Python Developer,

7 Unity / Game Developer,

8 Quality Assurance,

9 iOS Developer,

10 Ruby Develope,

11 JavaScript Developer,

12 ASP.NET Core Developer,

13 C++ Developer,

14 Android Developer,

15 Angular Developer,

16 .NET Desktop Developer,

17 React Developer,

Посилання: http://gamedev.lviv.ua/?utm_source=Infopartners&utm_campaign=IPGD

https://www.youtube.com/watch?v=MoKkYb3h6Qo

Рисунок 1 – Сертифікація у Agile I Scrum

1.4 Стандарти програмного забезпечення

Основні стандарти з технологій розробки програмного забезпечення наступні:

• ISO / IEC 12207 - Information Technology - Software Life Cycle Processes - Процеси життєвого циклу програмних засобів. Стандарт містить визначення основних понять програмної інженерії (зокрема програмного продукту та життєвого циклу програмного продукту), структури життєвого циклу як сукупності процесів, детальний опис процесів життєвого циклу.

• SEI CMM - Capability Maturity Model (for Software) - модель зрілості процесів розробки програмного забезпечення. Стандарт відповідає на запитання: «Якими ознаками повинна володіти професійна організація з розробки ПО? ». Професіоналізм організації визначається через зрілість процесу, застосовуваного цією організацією. Виділяються п'ять рівнів зрілості процесу.

• ISO / IEC 15504 - Software Process Assessment - Оцінка і атестація зрілості процесів створення і супроводу ПЗ. Є розвитком і уточненням ISO 12207 і SEI CMM. Містить розширене по відношенню ISO 12207 кількість процесів життєвого циклу і 6 рівнів зрілості процесів. дається докладний опис схеми атестації процесів, на основі результатів якої може бути виконано оцінку зрілості процесів та надано рекомендації щодо їх удосконаленню.

• PMBOK - Project Management Body of Knowledge - Зведення знань з управління проектами. Містить описи складу знань у наступних 9 розділах (областях знань) управління проектами

• SWBOK - Software Engineering Body of Knowledge - Зведення знань з програмної інженерії - містить опису складу знань з 10 розділах (областям знань) програмної інженерії.

• ACM / IEEE CC2005 - Computing Curricula 2005 - Академічний освітній стандарт в галузі комп'ютерних наук. Виділено 4 основних розділи комп'ютерних наук: Computer science, Computer engineering, Software engineering і Information systems.

- CC2005 – Computing Curricula

- SE2014 – Software Engineering

Software Engineering 2014

Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering

A Volume of the Computing Curricula Series

23 February 2015

Joint Task Force on Computing Curricula
IEEE Computer Society
Association for Computing Machinery

- CS2013 – Computer Science
- CE2016 – Computer Engineering

- IT2017 – Information Technology
- CSEC2017 – Cybersecurity

• Професійний стандарт «Фахівець з розробки програмного забезпечення»

На ринку праці затребувані програмісти, що вміють працювати в команді, володіють інструментами колективної розробки програмного забезпечення.

У зв'язку з цим зростає значення професійних компетенцій колективної розробки програмного забезпечення, знання сучасних напрямів, методів і технологій розробки програмного забезпечення:

- розуміння обов'язків різних учасників команди по розробці програмного забезпечення: керівник розробки програмного забезпечення, керівник технічної групи (team leader), архітектор, програміст, тестувальник, дизайнер, верстальник, аналітик;
- володіння сучасними стратегіями і технологіями організації колективної розробки програмного забезпечення, включаючи системи управління версіями, процеси безперервної інтеграції, стандарти оформлення коду і методи інспекції коду;
- розуміння основних напрямів розвитку методів колективної розробки, їх відмінностей і доцільності застосування залежно від типу вирішуваних завдань і вимог організації;
- володіння гнучкими (Agile) методологіями розробки програмних продуктів.

Динаміка розвитку предметної області «розробка програмного забезпечення» настільки велика, що ринок вимагає постійної зміни кількості і якості знань та умінь від випускника.

Стандарт вищої освіти України: перший (бакалаврський) рівень, галузь знань–12 Інформаційні технології, спеціальність – **121 Інженерія програмного забезпечення** [19].

1.5 Правові норми програмного забезпечення

Знання правових норм відносно програмного забезпечення обов’язкове для розробників програмного забезпечення. Це відноситься перш за все до несанкціонованих утручань у роботу обчислювальної техніки та відповідальність за недозволене використання програмного забезпечення сторонніх розробників та ліцензійного програмне забезпечення [18]. Цим питанням присвячені окремі статті Кримінального кодексу України.

Стаття 361КК. Несанкціоноване втручання в роботу електронно-обчислювальних машин (комп'ютерів), автоматизованих систем, комп'ютерних мереж чи мереж електрозв'язку

1. Несанкціоноване втручання в роботу електронно-обчислювальних машин (комп'ютерів), автоматизованих систем, комп'ютерних мереж чи мереж електрозв'язку, що призвело до витоку, втрати, підробки, блокування інформації, спотворення процесу обробки інформації або до порушення встановленого порядку її маршрутизації,

- карається **штрафом від шестисот до тисячі неоподатковуваних мінімумів доходів громадян або обмеженням волі на строк від двох до п'яти років, або позбавленням волі на строк до трьох років, з позбавленням права обіймати певні посади чи займатися певною діяльністю на строк до двох років або без такого та з конфіскацією програмних та технічних засобів, за допомогою яких було вчинено несанкціоноване втручання, які є власністю винної особи.**

2. Ті самі дії, **вчинені повторно або за попередньою змовою групою осіб,** або якщо вони заподіяли значну шкоду, - караються позбавленням волі на строк **від трьох до шести років** з позбавленням права обіймати певні посади чи займатися певною діяльністю н**а строк до трьох років та з конфіскацією програмних та технічних засобів**, за допомогою яких було вчинено несанкціоноване втручання, які є власністю винної особи.

Примітка. Значною шкодою у статтях 361- 363-1, якщо вона полягає у заподіянні матеріальних збитків, вважається така шкода, яка **в сто і більше разів перевищує неоподатковуваний мінімум** доходів громадян.

Стаття 361-2. Несанкціоновані збут або розповсюдження інформації з обмеженим доступом, яка зберігається в електронно- обчислювальних машинах (комп'ютерах), автоматизованих системах, комп'ютерних мережах або на носіях такої інформації

Несанкціоновані збут або розповсюдження інформації з обмеженим доступом, яка зберігається в електронно-обчислювальних машинах (комп’ютерах), автоматизованих системах, комп’ютерних мережах або на носіях такої інформації, створеної та захищеної відповідно до чинного законодавства, - караються штрафом **від п’ятисот до тисячі неоподатковуваних мінімумів доходів** громадян або **позбавленням волі на строк до двох років з конфіскацією програмних або технічних засобів**, за допомогою яких було здійснено несанкціоновані збут або розповсюдження інформації з обмеженим доступом, які є власністю винної особи.

Ті самі дії, вчинені **повторно або за попередньою змовою групою осіб,** або якщо вони заподіяли значну шкоду, - караються **позбавленням волі на строк від двох до п’яти років з конфіскацією програмних або технічних засобів**, за допомогою яких було здійснено несанкціоновані збут або розповсюдження інформації з обмеженим доступом, які є власністю винної особи.

(Кодекс доповнено статтею 3612 згідно із Законом України № 2289-IУ від 23 грудня 2004 р.)

Родовий об'єкт - інформаційна безпека, безпосередній - нормальне функціонування комп'ютерної інформації з обмеженим доступом.

Предмет злочину - інформація з обмеженим доступом, яка зберігається в електронно-обчислювальних машинах (комп'ютерах), автоматизованих системах, комп'ютерних мережах або носіях такої інформації, створеної та захищеної відповідно до чинного законодавства.

Комп'ютерна інформація (див. коментар до ст. 361 КК) з обмеженим доступом згідно зі ст. 30 Закону України «Про інформацію» від 2 жовтня 1992 р. за своїм правовим режимом поділяється на конфіденційну і таємну.

Конфіденційна інформація містить відомості, які перебувають у володінні, користуванні або розпорядженні окремих фізичних або юридичних осіб, поширюється за їх бажанням згідно з передбаченими ними умовами і має відповідний правовий статус. Режим доступу до конфіденційної інформації громадян та юридичних осіб визначають та встановлюють для неї систему способів захисту самостійно компетентні державні органи або інші власники інформації.

До таємної інформації належить інформація, що містить відомості, які становлять державну та іншу передбачену законом таємницю, розголошення якої завдає шкоди особі, суспільству і державі. Перелік відомостей, що становлять державну таємницю, визначається Законом України «Про державну таємницю» у редакції від 21 вересня 1999 р. (ВВРУ. - 1999. - № 49. - Ст. 428).

До іншої передбаченої законом таємниці, належить комерційна, банківська, лікарська таємниця, таємниця листування та ін. Правовий режим цих видів таємниць (інформації) регламентується спеціальними законами. Проте вказані види інформації виступають також і як предмети інших (самостійних) злочинів і збут або поширення такої інформації передбачено статтями 145, 168, 231, 232, 328 та ін. (за умови відсутності ознак злочинів проти основ національної безпеки України). Але якщо така інформація, яка зберігається в електронно-обчислювальних машинах (комп'ютерах), автоматизованих системах, комп'ютерних мережах або на носіях такої інформації не санкціонновано здобувається або розповсюджується - все вчинене повинно кваліфікуватися за сукупністю злочинів - за ст. 3612 і відповідною до ст. КК, яка встановлює відповідальність за збут чи розповсюдження конкретного виду інформації з обмеженим доступом (таємниці).

Інформація з обмеженим доступом як предмет злочину має зберігатися в електронно-обчислювальних машинах (комп'ютерах), автоматизованих системах чи комп'ютерних мережах. Інформація, яка зберігається в мережах електрозв'язку, до предмета даного злочину не належить.

Ознакою комп'ютерної інформації з обмеженим доступом є те, що вона повинна бути створена та захищена відповідно до чинного законодавства. При цьому в кожному випадку для з'ясування наявності цієї ознаки слід звернутися до відпо-

відних законів чи підзаконних нормативно-правових актів, в яких регламентується порядок створення і захисту такої інформації.

Про зміст поняття «носії комп'ютерної інформації» див. коментар до ст. 361.

Об'єктивна сторона злочину виражається у вчиненні несанкціонованого збуту або розповсюдження комп'ютерної інформації з обмеженим доступом, яка зберігається в електронно-обчислювальних машинах (комп'ютерах), автоматизованих системах, комп'ютерних мережах або на носіях такої інформації (див. коментар до ст. 3611).

Розглядуваний злочин (ч. 1 ст. 3612) є злочином з формальним складом і тому вважається закінченим з моменту вчинення суспільно небезпечних дій, зазначених у законі.

Несанкціоноване розповсюдження інформації з обмеженим доступом, яка зберігається в електронно-обчислювальних машинах (комп'ютерах), автоматизованих системах, комп'ютерних мережах або на носіях такої інформації, - це вчинення будь-яких дій, якими без згоди власника інформація з обмеженим доступом безпосередньо чи опосередковано надається іншим особам чи доводиться до їх відома, вводиться в обіг шляхом будь-якої, крім оплатної форми. Тут має місце передача права володіння такою інформацією іншим особам, а так само розголошення такої інформації.

Несанкціонований збут інформації з обмеженим доступом, яка зберігається в електронно-обчислювальних машинах (комп'ютерах), автоматизованих системах, комп'ютерних мережах або на носіях такої інформації, - це несанкціоноване розповсюдження такої інформації без згоди її власника на платній основі - шляхом купівлі- продажу, міни тощо.

Суб'єктивна сторона характеризується виною у формі прямого умислу.

Суб'єкт злочину - фізична осудна особа, яка досягла 16-річного віку.

У частині 2 ст. 3612 КК встановлена кримінальна відповідальність за ті самі дії, вчинені повторно або за попередньою змовою групою осіб, або якщо вони заподіяли значну шкоду (див. коментар до ст. 361).

1.6 Авторське право, майнові та немайнові права

Розробник програмного забезпечення повинен мати уяву про власні авторські права на розроблене програмне забезпечення та їх можливі порушення відносно програмного забезпечення сторонніх розробників.

Відповідно до ст.15 Закону "Про авторське право і суміжні права" до майнових прав автора відносяться:

-право на використання твору;

-права дозволяти або забороняти використовувати твір іншим особам.

Три статті Кримінальний кодексу України [18] :

ст.176 "Порушення авторського права і суміжних прав"

1. Незаконне відтворення, розповсюдження творів науки, літератури і мистецтва, комп'ютерних програм і баз даних, а так само незаконне відтворення, розповсюдження виконань, фонограм, відеограм і програм мовлення, їх незаконне тиражування та розповсюдження на аудіо- та відеокасетах, дискетах, інших носіях інформації, камкординг, кардшейрінг або інше умисне порушення авторського права і суміжних прав, а також фінансування таких дій, якщо це завдало матеріальної шкоди у значному розмірі, -

караються штрафом від двохсот до тисячі неоподатковуваних мінімумів доходів громадян або виправними роботами на строк до двох років, або позбавленням волі на той самий строк.

2. Ті самі дії, якщо вони вчинені повторно, або за попередньою змовою групою осіб, або завдали матеріальної шкоди у великому розмірі, -

караються штрафом від тисячі до двох тисяч неоподатковуваних мінімумів доходів громадян або виправними роботами на строк до двох років, або позбавленням волі на строк від двох до п'яти років.

3. Дії, передбачені частинами першою або другою цієї статті, вчинені службовою особою з використанням службового становища або організованою групою, або якщо вони завдали матеріальної шкоди в особливо великому розмірі, -

караються штрафом від двох тисяч до трьох тисяч неоподатковуваних мінімумів доходів громадян або позбавленням волі на строк від трьох до шести років, з позбавленням права обіймати певні посади чи займатися певною діяльністю на строк до трьох років або без такого.

Примітка. У статтях 176 та 177 цього Кодексу матеріальна шкода вважається завданою в значному розмірі, якщо її розмір у двадцять і більше разів перевищує неоподатковуваний мінімум доходів громадян, у великому розмірі - якщо її розмір у двісті і більше разів перевищує неоподатковуваний мінімум доходів громадян, а завданою в особливо великому розмірі - якщо її розмір у тисячу і більше разів перевищує неоподатковуваний мінімум доходів громадян.

ст.177 "Порушення прав на винаходи, корисну модель, промисловий зразок, топологію інтегральної мікросхеми, сорт рослин, раціоналізаторську пропозицію"

1. Незаконне використання винаходу, корисної моделі, промислового зразка, топографії інтегральної мікросхеми, сорту рослин, раціоналізаторської пропозиції, привласнення авторства на них, або інше умисне порушення права на ці об'єкти, якщо це завдало матеріальної шкоди у значному розмірі, -

караються штрафом від двохсот до тисячі неоподатковуваних мінімумів доходів громадян або виправними роботами на строк до двох років, або позбавленням волі на той самий строк.

2. Ті самі дії, якщо вони вчинені повторно, або за попередньою змовою групою осіб, або завдали матеріальної шкоди у великому розмірі, -
караються штрафом від тисячі до двох тисяч неоподатковуваних мінімумів доходів громадян або виправними роботами на строк до двох років, або позбавленням волі на строк від двох до п'яти років.
3. Дії, передбачені частинами першою або другою цієї статті, вчинені службовою особою з використанням службового становища або організованою групою, або якщо вони завдали матеріальної шкоди в особливо великому розмірі, -
караються штрафом від двох тисяч до трьох тисяч неоподатковуваних мінімумів доходів громадян або позбавленням волі на строк від трьох до шести років, з позбавленням права обіймати певні посади чи займатися певною діяльністю на строк до трьох років або без такого.

ст.229 "Незаконне використання знаку для товарів і послуг, фірмового найменування, кваліфікованого позначення походження товару"

1. Незаконне використання знаку для товарів і послуг, фірмового найменування, кваліфікованого зазначення походження товару, або інше умисне порушення права на ці об'єкти, якщо це завдало матеріальної шкоди у значному розмірі, -
караються штрафом від однієї тисячі до двох тисяч неоподатковуваних мінімумів доходів громадян.
2. Ті самі дії, якщо вони вчинені повторно, або за попередньою змовою групою осіб, або завдали матеріальної шкоди у великому розмірі, -
караються штрафом від трьох тисяч до десяти тисяч неоподатковуваних мінімумів доходів громадян.
3. Дії, передбачені частинами першою або другою цієї статті, вчинені службовою особою з використанням службового становища або організованою групою, або якщо вони завдали матеріальної шкоди в особливо великому розмірі, -
караються штрафом від десяти тисяч до п'ятнадцяти тисяч неоподатковуваних мінімумів доходів громадян з позбавленням права обіймати певні посади чи займатися певною діяльністю на строк до трьох років або без такого.
Примітка. Матеріальна шкода вважається завданою в значному розмірі, якщо її розмір у двадцять і більше разів перевищує неоподатковуваний мінімум доходів громадян, у великому розмірі - якщо її розмір у двісті і більше разів перевищує неоподатковуваний мінімум доходів громадян, а завданою в особливо великому розмірі - якщо її розмір у тисячу і більше разів перевищує неоподатковуваний мінімум доходів громадян.

Програмне забезпечення подібно іншим об'єктам інтелектуальної власності, таким як музичні та літературні твори, захищене від несанкціонованого копіювання законами про авторські права.

Закони про авторські права передбачають збереження за автором (видавцем) програмного забезпечення декількох виключних прав, одне з яких - право на виробництво копій програмного забезпечення.

Авторське право в мережі Інтернет. Поява нових технологічних можливостей привела до широкого використання об'єктів авторських і суміжних прав в Інтернеті.

На початку дане визначення самому терміну "Інтернет":

Інтернет - всесвітня інформаційна система загального доступу, яка логічно пов'язана глобальним адресним простором і базується на Інтернет-протоколі, визначеному міжнародними стандартами.

Самі правовідносини (об'єкти і суб'єкти авторського права) в мережі Інтернет дуже різноманітні. Перерахуємо хоч би деякі права суб'єктів (власників) прав з їх числа:

- авторські права провайдерів на комп'ютерні програми і бази даних, що реалізовують сам доступ до Інтернет або розміщення веб-сайтів на їх технічних майданчиках (серверах);
- авторські права виробників програмного забезпечення для цих серверів провайдерів;
- авторські права власників веб-сайтів на власне контент веб-сайту, його програмну частину і інші об'єкти авторського права, на нім розміщені - статті, зображення, музику, бази даних і так далі
- авторські права конкретних власників прав на об'єкти, розміщені на веб-сайтах, : комп'ютерні програми, музику, статті, зображення, бази даних і тому подібне, які дуже активно використовуються користувачами інтернету.

Твори в електронній формі, доступні в цифровій мережі, можуть бути сприйняті необмеженим кругом користувачів в будь-який час за бажанням кожного з них. Твори, що мають відкритий доступ, будучи одного дня перетворені в цифрову форму і завантажені в Інтернет, стають легкою здобиччю для порушників авторських прав.

Інформація - відомості, представлені у вигляді сигналів, знаків, звуків, рухливих або нерухомих зображень або іншим способом.

На початковому етапі розвитку Інтернету більшість його користувачів дотримувалися позиції про неможливість поширення діючих норм на правовідносини, що виникають в цифрових мережах. Як наслідок цього, почалася підготовка законопроектів, присвячених спеціальному регулюванню правовідносин, що виникають в мережевому просторі, зокрема проект Закону про регулювання російської частини Інтернету. Проте більшість юристів на сьогодні вважають, що чинне

законодавство цілком в змозі адекватно регулювати хоч би деякі сфери взаємин, що виникають в мережі.

Найчастіше через Інтернет передаються, а у тому числі шляхом такої передачі порушуються і авторські права, наступні об'єкти прав : літературні, музичні і аудіовізуальні твори, комп'ютерні програми, а також витвори образотворчого мистецтва, фотографії і так далі. Серед причин такого масового незаконного відтворення екземплярів творів, що охороняються авторським правом можна виділити технічну простоту здійснення операції. З розвитком техніки, користувачеві Інтернет все менше часу треба для отримання на своєму комп'ютері тотожної копії твору, або практично що не поступається оригіналом за якістю. Досі через інтернет складно передавати лише відео зображення, у зв'язку з великим розміром відеоінформації. Передача ж літературних і музичних творів здійснюється з швидкістю, яка набагато перевищує швидкість прочитання або прослуховування таких творів. Інша причина незаконного обороту творів полягає в елементарній відсутності привабливих варіантів легального отримання необхідних творів за наявності нелегальних.

Очевидно, що окремі правовласники не в змозі відстежувати поширення об'єктів, що охороняються, в цифрових мережах і їх використання при створенні продуктів мультимедіа. Правовласники фактично позбавлені можливості захищати свої права в цифровому середовищі тими ж способами, що і при звичайному використанні об'єктів, що охороняються авторським правом.

Проте існує явна зацікавленість не лише практично усіх правовласників, але і більшості користувачів в знаходженні легальних способів рішення виникаючих проблем.

Існує ряд помилок відносно законності використання творів в мережі. Одна з них - розміщення твору на сайті за усним погодженням з автором. Відповідно до Закону про авторське право форма авторського договору, на підставі якого мають бути передані майнові права, має бути письмовою. Звичайно, недотримання письмової форми договору не тягне його недійсність, а лише позбавляє сторони права посилатися на показання свідків. Проте, у зв'язку з особливостями авторського договору (зокрема, необхідністю погоджувати в нім не лише майнові права, які передаються, але і термін, територію, можливість переуступання прав третім особам та ін.), усна домовленість майже ніколи не тягне передачу авторських прав. Як наслідок використання твору стає незаконним і дає можливість авторові або його правонаступникові подавати до суду.

Ще однією глибокою помилкою є твердження, що створення електронних бібліотек не порушує нічиїх прав. Відомо, що чинний Закон про авторське право робить певні пільги відносно використання творів бібліотеками, зокрема бібліотека має право без дозволу автора і без виплати йому гонорару здійснювати репрографічне відтворення, тобто репродукція, під якою розуміється відтворення шляхом фотокопіювання або за допомогою інших технічних засобів, інших, ніж ви-

дання. Проте ця норма зовсім не дозволяє ні оцифровувати (відтворювати) твір, ні розміщувати твір на сайті.

Інформаційно-комунікаційні технології перетворять життя суспільства всюди у світі. Інновації створюють нові ринки товарів і послуг. Такі технології вносять революційні зміни в процеси праці, підвищують продуктивність в традиційних галузях і збільшують швидкість руху капіталу і об'єми його потоків. Проте зміни в економіці - лише одна сторона питання. Суспільства переживають глибокі зміни у сфері культури, формуючи засоби масової інформації, які, у свою чергу, формують суспільства, а також адаптуючись до лавиноподібного зростання Інтернету. Швидкий розвиток нових информационно- комунікаційних технологій по всьому світу має і свою негативну сторону: створюються можливості для появи нових форм експлуатації, нових різновидів злочинної діяльності і навіть нових форм злочинності.

Визначення поняття "Злочин, пов'язаний з використанням комп'ютерів" або аналогічних йому, таких як "кіберзлочин", обговорювалося упродовж останніх 30 років.

Уперше подібний термін був використаний в одній з доповідей Стэнфордского дослідницького інституту, а потім, в злегка зміненому виді, він знову з'явився в документах 1979 і 1989 років. Ця класифікація широко вживалася в опублікованих пізніше статтях по кіберзлочинності: комп'ютер як суб'єкт злочину; комп'ютер як об'єкт злочину; чи комп'ютер як інструмент (четвертий варіант, запропонований в 1973 році, - комп'ютер як символ - очевидно, вийшов із вживання в 1980-х роках). Можливо, корисно інакше сформулювати цю концептуальну модель, розглядаючи злочини, пов'язані з використанням комп'ютерів, як що забороняється законом і/або судовою практикою поведінка, яка

a) спрямоване власне на комп'ютерну сферу і комунікаційні технології;

b) включає використання цифрових технологій в процесі здійснення правопорушення; чи

c) включає використання комп'ютера як інструменту в процесі скоювання інших злочинів, і, відповідно, комп'ютер виступає при цьому як джерело електронних процесуальних доказів.

У законах і договорах, у тому числі в прийнятої Радою Європи Конвенції по кібер злочинах, дані визначення різних видів злочинів, пов'язаних з використанням комп'ютерів (таких, як злочини проти конфіденційності, цілісності або доступності комп'ютерних систем, правопорушення відносно контенту і правопорушення відносно інтелектуальної власності).

Але з проектом в Інтернеті пов'язані ще і домен, тобто об'єкт, пов'язаний з назвою сайту. А це вже реєстрація торговельної марки, а ніяк не авторські права. Важлива також і для проекту в мережі Реєстрація домена.

Графічні зображення, як об'єкти авторського права, які найчастіше можуть бути розміщені в Інтернеті, : малюнок, ескіз, картина, план, креслення, кінокадр, телекадр, відеокадр, фотографія і так далі.

Згідно ст. 421 діючого Цивільного кодексу України суб'єктами права інтелектуальної власності на вище вказані твори є творець об'єкту права інтелектуальної власності (автор) і інші особи, яким належать особисті немайнові права, а також (чи) власник майнових авторських прав інтелектуальної власності Також і згідно ст.7 закону України "Про авторське право і суміжні права" суб'єктами авторського права є автори творів, їх спадкоємці і особи, яким автори або їх спадкоємці передали свої авторські майнові права.

Дуже важливим є питання правової оцінки розміщення будь-якого зображення на сторінці веб-сайту, тобто встановлення того, чи являється це використанням твору. Посилаючись, по-перше, на ст. 441 Цивільного кодексу України, яка вказує, що використанням твору є "публікація (випуск у світ); відтворення будь-яким способом і в будь-якій формі" можна зробити висновок, що розміщення будь-якого зображення на сторінці веб-сайту і є використанням твору в розумінні ст. 441 ГК.

Така ж норма відносно відтворення є і в ст.1 чинного закону України "Про авторське право і суміжні права", яка вказує, що відтворенням є "виготовлення одного або більше екземплярів твору, а також їх запис для тимчасового або постійного зберігання в електронній (у тому числі цифровий), оптичній або іншій формі, яку може прочитувати комп'ютер".

Таким чином, законодавство України чітко визначає, що розміщення зображення на веб-сайті - це вже є його відтворення. А тому суб'єкт авторського права має можливість згідно ч.1 ст.440 Цивільного кодексу України застосовувати свої майнові права інтелектуальної власності на твір, тобто "виняткове право дозволяти використання твору і право перешкоджати неправомірному використанню твору, у тому числі - забороняти таке використання".

Використання фотографії на веб-сайті. У разі, якщо на фотографії зафіксований об'єкт, який не відноситься до об'єктів авторського, користувач належний укласти договір лише з особою, яке володіє винятковими правами на фотографію. Але в той же час, на фотографіях нерідко фіксуються об'єкти, які охороняються також нормами авторського права, наприклад: творі живопису, скульптури, графіки, дизайну, витвору декоративно-прикладного мистецтва, твору архітектури, містобудування або садово-паркового мистецтва. Використовуючи такі твори, слід пам'ятати про те, що використання вказаних об'єктів авторських прав, як і використання фотографій, може здійснюватися тільки на підставі письмового дозволу автора (правовласника) і згідно ст.15 чинного закону України "Про авторське право і суміжні права". Інакше кажучи, користувачеві слід укладати договір не лише з автором фотографій (правовласником), але і з автором (правовласником) твору, що охороняється авторським правом, який відтворюється на фотографії.

Потрібно вказати, що в Інтернет з'являються і нові об'єкти авторського права, охорона яких за існуючим законодавством досить скрутна. Найважливіше питання - визначення правової природи основного компонента всесвітньої мережі - файлу HTML. Файл HTML утілює в собі відразу декілька об'єктів інтелектуальної власності. З точки зору внутрішньої структури - це комп'ютерна програма. Але з точки зору зовнішнього оформлення сторінка HTML може бути літературним твором, твором художника, дизайнера і тому подібне. Жоден з існуючих способів охорони не зважає повною мірою на специфіку HTML. Безперечно, виходом була б розробка особливого механізму охорони HTML сторінок і внесення відповідних змін в законодавство про інтелектуальну власність.

Незважаючи на відсутність спеціального регулювання, права автора сторінки HTML можуть бути захищені нормами авторського права. За бажання, автор HTML файлу може зареєструвати свій HTML - файл в Державному підприємстві "Українське агентство з авторських і суміжних прав" (надалі - ДП УААСП) як комп'ютерну програму, тим самим, підтвердити своє авторство і пріоритет.

Складнішим представляється проблема використання так званих "Java applets" - програм, написаних мовою Java, призначених для роботи в Інтернет. Якщо мова HTML досить примітивна, то мову "Java" повною мірою можна назвати повноцінною мовою програмування. Особливості функціонування програм, написаних мовою "Java" полягають в тому, що програма існує увесь час як некомпильованный текст. При запуску через інтернет програм, написаних мовою "Java", користувач копіює на свій комп'ютер не лише ту програму, яку він запускає, але і компонент, необхідний для компіляції і запуску виконуваної програми безпосередньо на комп'ютері користувача. Таким чином використовується відразу декілька програм, що охороняються авторським правом.

Придбаваючи звичайну програму, користувач не цікавиться, наскільки правомірно програміст використовував ту або іншу мову програмування, і чи дотримані права автора компілятора, оскільки програма поступає в користувача у вигляді об'єктного коду. У випадку ж з програмами, написаними мовою "Java", і програма і компілятор поступають користувачеві окремо друг від друга, і компіляція програми для її запуску відбувається вже на комп'ютері користувача, причому компілятор - самостійна програма - залишається на комп'ютері користувача і може бути використана необмежена кількість разів з іншими програмами мовою Ява. Поки проблема використання компіляторів не стоїть гостро тільки з однієї причини - основний постачальник програмного забезпечення до Інтернет компанія "Майкрософт" надає своїм користувачам усе необхідні для роботи в Інтернет програми абсолютно безкоштовно. Проте вже ця ситуація може докорінно змінитися.

Реєстрація авторського права відбувається шляхом подачі Заявки на реєстрацію об'єкту авторського права.

Документи для реєстрації авторського права потрібні наступні:

- Доручення встановленого зразка на Кондратюка І. В., який представлятиме ваші інтереси по реєстрації об'єкту авторського права;
- Заява українською мовою, яка складається у формі, затвердженій Державним департаментом інтелектуальної власності;
- Екземпляр твору (опублікованого або неопублікованого) в друкарському виді мовою оригіналу на паперовому або електронному носієві;
- Документи, який свідчать про факт і дату публікації твору;
- Документ про плату збору за підготовку до реєстрації авторського права на твір;
- Документ про плату державного мита за видачу посвідчення (готується нашим Агентством).
- Вказаний документ подається до Державного департаменту інтелектуальної власності після отримання нами Рішення про реєстрацію авторського права;
- Документ, який засвідчує перехід у спадок майнового права автора (якщо заявка подається спадкоємцем автора).
- Для реєстрації авторського права ми здійснюємо усі необхідні дії:
- Консультації автора про реєстрацію об'єкту авторського права;
- Правильне заповнення Заявки на реєстрацію об'єкту авторського права;
- Плату зборів і мит за реєстрації об'єкту авторського права;
- Правильне оформлення усіх документів, що додаються до Заявки;
- Отримання Рішення про реєстрацію авторського права;
- Отримання Свідоцтва про реєстрацію авторського права;
- Відправку авторові Рішення об реєстрації і Свідчення про реєстрацію авторського права;

Реєстрація авторського права - найбільш ефективний спосіб фіксації ваших прав на об'єкт інтелектуальної власності.

1.6 Суспільна зацікавленість в якості програмного продукту.

РОЗДІЛ 2 ПРАКТИКА ЗАСТОСУВАННЯ ТЕХНОЛОГІЙ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Принципи гнучких технологій

Agile Manifesto -маніфест розробки програмного забезпечення

Основні ідеї:

- Особистості та їхні взаємодії важливіші, ніж процеси та інструменти;
- Робоче програмне забезпечення важливіше, ніж повна документація;
- Співпраця із замовником важливіша, ніж контрактні зобов'язання;
- Реакція на зміни важливіша, ніж дотримання плану.

Принципи, які роз'яснють Agile Manifesto

1. Задоволення клієнта за рахунок **ранньої та безперебійної** поставки цінного програмного забезпечення;
2. Вітання **зміни вимог** навіть наприкінці розробки (це може підвищити конкурентоспроможність отриманого продукту);
3. **Часта поставка** робочого програмного забезпечення (кожен місяць або тиждень або ще частіше);
4. **Тісне, щоденне спілкування замовника з розробниками** впродовж всього проекту;
5. Проектом займаються **умотивовані особистості**, які забезпечені потрібними умовами роботи, підтримкою і довірою;
6. Рекомендований **метод передачі інформації — особиста розмова** (віч-на-віч);
7. **Робоче програмне забезпечення — найкращий вимірювач прогресу;**
8. Спонсори, розробники та користувачі повинні мати можливість підтримувати **постійний темп на невизначений термін**;
9. Постійну увагу **поліпшенню технічної майстерності** та зручному дизайну;
10. **Простота** — мистецтво не робити зайвої роботи;
11. Найкращі технічні вимоги, дизайн та архітектура виходять у **самоорганізованої команди**;
12. Постійна **адаптація до мінливих обставин**.

Збираєтеся почати практикувати Scrum у себе в компанії? Чи ви вже працюєте по Scrum 'у пару місяців? У вас вже є базові поняття, ви прочитали декілька книг, а, можливо, навіть отримали сертифікат Scrum Master 'а? Поздоровляю!

Проте, погодитеся, якась неясність все одно залишається.

За словами Кена Швабера, Scrum — це не методологія, це фреймворк. А це означає, що Scrum не дає готових рецептів, що робити в тих або інших випадках.

Але у мене для вас є хороша новина: я розповім, як саме я практикую Scrum. дуже детально і з усіма деталями. Проте, і без поганої новини не обійдеться: я розповім усього лише про те, як практикую Scrum я. Це означає, що вам не обов'язково робити усе точно так же. Насправді, залежно від ситуації, я і сам можу робити щось по-іншому.

Переваги Scrum'а і одночасно найбільший його недолік в тому, що його необхідно адаптувати до вашої конкретної ситуації.

Моє бачення Scrum 'а формувалося упродовж цілого року і стало результатом експериментів в команді чисельністю біля 40-ка чоловік. Одна компанія потрапила в украй складну ситуацію: постійні переробки, аврали, серйозні проблеми з якістю, провалені терміни і інші неприємності. Ця компанія вирішила перейти на Scrum, але впровадити його толком у неї не виходило. У результаті це завдання було доручене мені. На той час для більшості співробітників слово «Scrum» було терміном, що просто набив оскому, відлуння якого звучало час від часу в коридорах без якого-небудь відношення до їх повсякденної роботи.

Через рік роботи ми впровадили Scrum на усіх рівнях компанії: поекспериментували зі всілякими розмірами команд (від 3 до 12 чоловік), спробували спринти різної довжини (від 2 до 6 тижнів) і різні способи визначення критерію готовності, різноманітні формати product і sprint backlog 'а, різні стратегії тестування, способи демонстрації результатів, способи синхронізації Scrum -команд і так далі. Також ми випробували безліч XP практик : з різними способами безперервної інтеграції, з парним програмуванням, TDD (розробкою через тестування), і т. д. А найголовніше — розібралися, як уся ця справа поєднується з Scrum 'ом.

Scrum має на увазі постійний процес розвитку, так що історія на цьому не закінчується. Я упевнений, згадана мною компанія продовжуватиме рухатися вперед (якщо в ній і далі проводитимуть ретроспективи спринтів) і постійно знаходити нові і нові способи ефективного застосування Scrum 'а, враховуючи особливості кожної з ситуацій, що склалися.

Ця книга не претендує на звання «єдино правильного» навчального посібника по Scrum! Вона усього лише пропонує вам приклад вдалого досвіду, отриманого протягом року в результаті постійної оптимізації процесу. До речі, у вас запросто може виникнути відчуття, що ми усі зрозуміли не так.

Під час вивчення Scrum 'а я читав книги, присвячені Scrum 'у і Agile 'у, рився по різних сайтах і форумах, проходив сертифікацію у Кена Швебера, засипав його питаннями і проводив купу часу, обговорюючи Scrum з моїми колегами. Проте одним з найбільш цінних джерел знань сталі реальні історії впровадження Scrum 'а. Вони перетворюють Принципи і Практики ст. ну, в те, як ви це робите. Вони допомогли мені передбачати типові помилки Scrum -новичков, а іноді і уникнути їх.

Отже, ось і випав мій шанс поділитися чимось корисним. Це моя реальна історія

2.2 Як працювати з з Product backlog 'ом

Product backlog — це основа Scrum'a. З нього усе починається. По суті, **product backlog є списком вимог, історій, функціональності, які впорядковані по мірі важливості.** При цьому усі вимоги описані на зрозумілій для замовника мові.

Елементи цього списку називатимемо «користувацькими історіями», user story, а іноді елементами backlog'у.

Опис кожної нашої історії включає наступні поля:

• **ID** — унікальний ідентифікатор — просто порядковий номер. Застосовується для ідентифікації історій у разі їх перейменування.

• **Назва** — короткий опис історії. Наприклад, «Перегляд журналу своїх транзакцій». Воно має бути однозначним, щоб розробники і product owner (власник продукту) могли приблизно зрозуміти, про що йде мова, і відрізнити одну історію від іншої. Звичайні від 2 до 10 слів.

• **Важливість** (Importance) — міра важливості цього завдання, на думку product owner'a. Наприклад, 10. Чи 150. Чим більше значення, тим вище важливість.

Краще не використовувати термін «пріоритет», оскільки зазвичай в цьому випадку 1 означає найвищий пріоритет. Це дуже незручно: якщо згодом ви вирішите, що якась історія ще важливіша, то який «пріоритет» ви тоді їй поставите? Пріоритет 0? Пріоритет-1?

• **Попередня оцінка** (initial estimate) — початкова оцінка об'єму робіт, необхідного для реалізації історії в порівнянні з іншими історіями. Вимірюється в story point'ах. Приблизно відповідає числу «ідеального людино-дня».

Запитаєте вашу команду: «Якщо зібрати команду з оптимальної кількості людей, тобто не занадто велику і не занадто маленьку (частіше всього з двох чоловік), закритися в кімнаті з достатнім запасом їжі і працювати ні на що не відволікаючись, то, скільки днів тоді знадобиться на розробку завершеного, протестованого продукту, готового до демонстрації і релизу»?. Якщо відповідь буде «Для трьох чоловік, закритих в кімнаті, на це знадобиться 4 дні», це означає, що первинна оцінка складає 12 story point'ов.

В цьому випадку важливо отримати не максимально точні оцінки (наприклад, для історії в 2 story point'и знадобиться 2 дні), а зробити так, щоб оцінки вірно відбивали відносну трудомісткість історій (наприклад, на історію, оцінену в 2 story point'a зажадає приблизно в два рази менше роботи в порівнянні з історією в 4 story point'a).

• **Як продемонструвати** (how to demo) — коротке пояснення того, як завершене завдання буде продемонстровано у кінці спринту. По суті, це простий тестовий сценарій типу «Зробіть це, зробіть те — повинне вийти те-то».

Якщо у вас практикується Test Driven Development (розробка через тестування або коротко TDD), то цей опис може послужити псевдокодом [3] для приймального тесту.

• **Примітки** — будь-яка інша інформація: пояснення, посилання на додаткові джерела інформації, і т. д. Зазвичай вона представлена у формі коротких тез

Саме ці 6 полів виявилися самими застосовними.

Зазвичай product backlog зберігається в Excel таблиці або в хмарному сервісі з можливістю спільного доступу (декілька користувачів можуть редагувати файл одночасно). Хоча офіційно документ належить product owner 'у, не забороняємо іншим користувачам редагувати його. Адже розробникам досить часто доводиться заглядати в product backlog, щоб щось уточнити або змінити оцінку передбачуваних трудовитрат.

З цієї ж причини, не поміщаємо product backlog в систему контролю версій, а зберігаємо його на мережевому диску або у хмарному сервісі. Цей простий спосіб дозволяє уникнути взаємних блокувань доступу і конфліктів синхронізації змін.

Проте майже усі інші артефакти зберігаються в системі контролю версій.

2.3 Додаткові поля для user story

Іноді використовуються додаткові поля в product backlog в основному для того, щоб допомогти product owner 'у визначитися з його пріоритетами.

Категорія (track) — Наприклад, «панель управління» або «оптимізація». За допомогою цього поля product owner може легко вибрати усі пункти категорії «оптимізація» і встановити їм низький пріоритет.

Компоненти (components) — вказує, які компоненти (наприклад, база даних, сервер, клієнт) торкнуться при реалізації історії. Це поле складається з групи checkbox 'ов, які відзначаються, якщо відповідні компоненти вимагають змін. Поле «компоненти» виявиться особливо корисним, якщо у вас декілька Scrum команд, наприклад, одна, яка працює над панеллю управління і інша, яка відповідає за клієнтську частину. В даному випадку це поле істотно спростить для кожної з команд процедуру вибору історії, за яку вона могла б взятися.

Ініціатор запиту (requestor). product owner може захотіти зберігати інформацію про усіх замовників, зацікавлених в цьому завданні. Це треба для того, щоб тримати їх в курсі справи про хід виконання робіт.

ID в системі обліку дефектів (bug tracking id) — якщо ви використовуєте окрему систему для обліку дефектів (наприклад. Jira), тоді в описі історії корисно зберігати посилання на усі дефекти, які до неї відносяться.

2.4 Як орієнтуєнтувати product backlog на бізнес?

Якщо product owner — технар, то він цілком може додати історію «Додати індекси в таблицю Events». Навіщо йому це треба? Та тому, що реальною метою цієї історії, швидше за все, буде «прискорення пошуку подій в панелі управління».

І взагалі, може виявитися, що не індекси були вузьким місцем, що призводить до повільної роботи пошукової форми. Причиною могло бути щось абсолютне інше. Зазвичай команді видніше, яким чином краще розв'язати подібну проблему, тому product owner повинен ставити цілі з точки зору бізнесу (що потрібно).

Робиться це до тих пір, поки не проявиться істинна причина появи історії (у наведеному прикладі — підвищити швидкість пошуку подій в панелі управління). Первинний технічний опис проблеми можна помістити в колонку з примітками: «Індексування таблиці Events може розв'язати проблему».

2.5 Як планувати спринт

Переконайтеся, що product backlog знаходиться в потрібній кондиції, перш ніж починати планування.

Що означає в потрібній кондиції? Що усі user story відмінно описані? Що усі оцінки трудовитрат коректні? Що усі пріоритети розставлені?

Це означає:

- Product backlog повинен існувати! (Хто б міг подумати?)
- У кожного продукту має бути один product backlog і один product owner.
- Усі найбільш важливі завдання мають бути класифіковані по рівню важливості, а їх числові значення не повинні співпадати.
- Не хвилюйтеся, якщо завдання з низьким рівнем важливості мають однакові значення, швидше за все, вони не потраплять в поточний спринт, а, отже, не обговорюватимуться.
- Усі user story, які, на думку Product owner 'а мають гіпотетичну можливість потрапити в наступний спринт, повинні мати унікальне значення важливості.
- Рівень важливості використовується виключно для впорядковування історій. Т. е. якщо історія А має рівень важливості 20, а історія Б важливість 100, це означає що Б важливіше А. Це не означає, що Б в п'ять разів важливіше А. Якщо Б присвоїти важливість 21 — сенс не поміняється!
- Корисно залишати проміжки з цілих чисел між значеннями на той випадок, якщо з'явиться історія В, важливіша, ніж А, але менш важлива, чим Б. Звичайно, можна викрутитися, присвоївши їй рівень важливості 20.5, але виглядає це кострубато, тому для проміжків ми вирішили використовувати тільки цілі числа!

- Product owner повинен розуміти кожну історію (найчастіше він їх автор, хоча іноді інші члени команди теж можуть вносити пропозиції, і тоді product owner зобов'язаний призначити їм пріоритетність). Він не зобов'язаний знати в усіх подробицях, що конкретно слід зробити, але він повинен розуміти, чому ця user story була включена в product backlog.
- Хоча зацікавлені особи можуть додавати user story в product backlog, вони не мають права привласнювати їм рівень важливості. Це прерогатива Product owner 'а. Вони також не можуть додавати оцінки трудовитрат, оскільки це прерогатива команди.

Ми також спробували:

- Використовувати Jira (нашу систему обліку дефектів) для зберігання product backlog 'а. Але для більшості Product owner 'а навігація по Jira була занадто обтяжлива. У Excel маніпулювати історіями набагато простіше і приємніше. Ви можете розфарбовувати текст, переставляти пункти, додавати, у разі потреби, нові колонки, коментувати, імпортувати і експортувати дані і т. д.
- Використовувати програми, заточені під Agile процес, такі як VersionOne, ScrumWorks, Xplanner і т. д.

Планування спринту — найважливіша частина Scrum'у. Погано проведене планування може зіпсувати увесь спринт.

Мета планування полягає в тому, щоб, з одного боку, дати команді досить інформації для спокійної роботи впродовж декількох тижнів, а з іншої — переконати Product owner 'а в тому, що команда зможе зробити свою роботу.

Добре-добре, це було дуже розпливчате визначення. Давайте краще поговоримо про те, що має бути підсумком планування :

• Мета спринту.

• Список учасників команди (і міра їх зайнятості, якщо вона не стовідсоткова).

• Sprint backlog (список історій, які увійшли до спринту).

• Дата демонстрації.

• Місце і час проведення щоденного Scrum 'а.

Чому без product owner 'а не обійтися

Іноді product owner з великим небажанням витрачає свій час на планування разом з командою. Команді і product owner 'у просто необхідно планувати разом, тому що кожна user story має три параметри, які дуже тісно пов'язані між собою.

Об'єм робіт і пріоритети завдань визначаються product owner 'ом. Зате **оцінка трудовитрат** — це прерогатива команди. Завдяки взаємодії команди і

product owner 'a в ході планування спринту виробляється оптимальне співвідношення усіх трьох змінних.

Найчастіше product owner починає планування наступного спринту з опису основних цілей і найбільш значущих історій. Після цього команда робить оцінку трудовитрат усіх user story, починаючи з найважливішої. У процесі у команди виникають дуже важливі питання з приводу об'єму майбутніх робіт. Наприклад, чи «Має на увазі історія »видалити користувача« видалення усіх його незавершених транзакцій»?. Іноді відповідь на це питання буде великим сюрпризом для команди і зажадає перегляду усіх оцінок для даної user story.

В деяких випадках час, який знадобиться на виконання user story, не співпадатиме з очікуваннями product owner 'a. Отже, він захоче переглянути пріоритет для story або змінити об'єм роботи. Це, у свою чергу, змусить команду виконати переоцінку і так далі, і так далі.

Така взаємна залежність є основою Scrum 'у, такі усього Agile 'у.

Але що якщо product owner все-таки наполегливо відмовляється виділити пару годин на планування спринту? У такому разі я зазвичай намагаюся послідовно застосувати наступні стратегії:

• Спробуйте донести до product owner 'a, чому його участь украй важлива.

• Спробуйте знайти у своїх рядах добровольця, який зміг би стать представником product owner 'a. Скажіть своєму product owner 'у: «У вас немає часу на планування, Петро виконуватиме вашу роль. У нього будуть усі повноваження на зміну пріоритетів і об'ємів робіт. Раджу вам обговорити з ним якомога більше нюансів до початку планування. Якщо ви проти Джефа, тоді виберіть когось іншого, але тільки при умові, що він буде присутнім на плануванні від початку до кінця».

• Відкладіть початок спринту до того моменту, поки у product owner 'a не з'явиться час для спільного планування. А доки не беріть на себе ніяких нових зобов'язань. Нехай в цей час ваша команда займеться будь-якою іншою корисною роботою.

2.6 Чому якість не обговорюється

Спробуємо пояснити різницю внутрішньою якістю і зовнішньою якістю.

Зовнішня якість — це те, як користувачі сприймають систему. Повільний і не інтуїтивний призначений для користувача інтерфейс — це приклад поганої зовнішньої якості.

Внутрішня якість торкається речей, які як правило не видно користувачеві, але при цьому роблять величезне значення на зручність супроводу системи. Це продуманість дизайну системи, покриття тестами, читаність коду, рефакторинг і т. п.

По правді кажучи, у системи з високою внутрішньою якістю іноді може бути досить низька зовнішня. Але навпаки буває украй рідко. Складно побудувати щось хороше на прогнилому фундаменті.

Зовнішнюя якість – це частина загального об'єму робіт. Адже з точки зору бізнесу буває дуже доцільно якнайшвидше випустити версію системи з трохи кострубатим і повільним призначеним для користувача інтерфейсом, і лише потім підготувати версію з доопрацюваннями і виправленнями. Тут право вибору повинне залишатися за product owner 'ом, оскільки саме він відповідає за визначення об'єму робіт. І навпаки — внутрішня якість не може бути предметом дискусії. Команда постійно повинна стежити за якістю системи, тому вона просто не обговорюється.

Як розрізняти завдання, пов'язані з внутрішньою і зовнішньою якістю?

Уявіть, що product owner говорить: «Добре хлопці, я розумію, чому ви оцінили це завдання в 6 story point 'а, але я упевнений, що, якщо ви трішки помізкуєте, то зможете по-швидкому «залатати» проблему.

Він намагається використовувати внутрішню якість як змінну! Як я здогадався? Так, тому що він хоче, щоб ми зменшили оцінку завдань, не зменшивши при цьому об'єм робіт. Слово «латочка» повинне викликати у вас тривогу.

Чому ж ми так жорстко стоїмо на своєму?

По моєму особистому досвіду, жертвувати внутрішньою якістю — це практично завжди дуже і дуже погана ідея. Заощаджений час нікчемно малий в порівнянні з тією ціною, яку вам доведеться заплатити як в найближчому майбутньому, так і в перспективі. Як тільки якість вашого коду погіршає, відновити його буде дуже важко.

В цьому випадку намагаємося перейти до обговорення об'єму завдань. «Раз вам так важливо виконати цю історію якомога раніше, тоді може бути варто скоротити об'єм завдань, щоб ми могли зробити її швидше? Можливо, варто спростити обробку помилок і зробити «Поліпшену обробку помилок« окремою історією залишивши її на майбутнє? Чи може знизити пріоритет інших історій, щоб ми могли зосередити усі свої зусилля на цій»?.

Найбільша складність при плануванні спринту полягає в наступному:

1. Люди не розраховують, що це займе так багато часу.
2. Але так воно і відбувається!

У Scrum 'і усе обмежене часом. Так що ж ми робимо, коли обмежене за часом планування спринту наближається до кінця, а мета спринту або sprint backlog все ще не визначені? Просто обриваємо планування? Подовжуємо на годину? Чи, можливо, ми завершуємо збори і продовжуємо його наступного дня?

Це трапляється знову і знову, особливо в нових командах. Як ви зазвичай вирішувати цю проблему? Я не знаю. А як вирішуємо її ми? Ну, зазвичай, я безцеремонно обриваю зустріч. Закінчую її. Нехай спринт постраждає. Точніше, я гово-

рю команді і product owner 'у: «Отже, зустріч закінчується через 10 хвилин. Ми, досі, повністю не спланували спринт. Чи можемо ми почати працювати з тим, що у нас є, або призначимо ще одно 4-х годинне планування спринту на завтра в 8 ранку»?. Можете здогадатися, що вони відповідають.: о)

Я кілька разів давав можливість нараді затягнутися, але звичайно це, ні до чого не призводило. Головна причина цьому — втома учасників зустрічі. Якщо команда не виробила відповідний план спринту за 2-8 годин (залежить від конкретно ваших обмежень за часом), вони, швидше за все, не впораються з ним і в додатковий час. Другий варіант, по правді, досить хороший: призначити нову зустріч наступного дня. За винятком того, що люди зазвичай нетерплячі і хочуть швидше почати спринт, а не витрачати ще пару годин на планування.

Отже, я урізаю тривалість зустрічі. Так, спринт від цього страждає. Але з іншого боку, команда отримала дуже цінний урок, і наступне планування спринту пройде ефективніше. Крім того, коли ви наступного разу запропонуєте збільшити тривалість зустрічі, люди обурюватимуться значно менше.

Вчитеся залишатися у рамках встановленого часу, вчитеся давати реалістичні оцінки. Це стосується як тривалості зустрічей, так і тривалості спринту.

Розпорядок зустрічі по плануванню спринту

Наявність хоч би зразкового розкладу значно збільшить ваші шанси закінчити планування спринту у відведений для цього час.

Наприклад, наш розклад виглядає так:

Планування спринту : з 13:00 до 17:00 (після кожної години перерва на 10 хвилин)

13:00–13:30. product owner роз'яснює мету спринту і розповідає про бізнес-процеси з product backlog'a. Обговорюється час і місце проведення демо.

13:30–15:00. Команда проводить оцінку часу, який знадобиться на розробку бізнес-процесів і, при необхідності дробить їх на дрібніші. В деяких випадках product owner може змінити пріоритет їх виконання. З'ясовуємо усі питання, що цікавлять нас. Для найбільш важливих заповнюємо поле «Як продемонструвати».

15:00–16:00. Команда визначає набір user story, які увійдуть до наступного спринту. Щоб перевірити наскільки це реально, обчислюємо продуктивність команди.

16:00–17:00. Домовляємося про час і місце проведення щоденного Scrum'a (якщо вони змінилися в порівнянні з минулим спринтом). Після чого приступаємо до розбиття user story на завдання.

Наявність розкладу ні в якому разі не має на увазі наявності жорстких обмежень. Залежно від того, як проходить зустріч, ScrumMaster може збільшити, або зменшити тривалість кожного етапу.

2.6 Визначаємо довжину спринту

Одне з основних завдань планування спринту — це визначення дати демо. А це означає, що вам доведеться визначитися з довжиною спринту.

Яка ж довжина оптимальна?

Короткі спринти — зручні. Вони дозволяють компанії бути максимально «гнучкими», а значить готовою часто коригувати свої плани.

Короткий спринт = короткий цикл зворотного зв'язку = часті релізи = швидкі відгуки від клієнтів = менше часу витрачається на роботу в неправильному напрямі = швидке навчання, вдосконалення і т. д.

Але з іншого боку довгі спринти теж хороші. У команди залишається більше часу, щоб набрати темп, більше простору для маневрів, щоб вирішити виниклі проблеми, а також більше часу для досягнення мети спринту, а у вас менше накладних витрат, таких як планування спринту, демо і т. д.

В основному короткі спринти більше подобаються product owner'у, а довгі — розробникам. Тому довжина спринту — це завжди компроміс. Більшість команд (але не усі) використовують тритижневий спринт. Він досить короткий, щоб надати адекватну корпоративну «гнучкість», але в той же час досить довгий, для того, щоб команда змогла досягти максимальної продуктивності і встигнути вирішити усі питання, які виникнуть у ході спринту.

Ми дійшли важливого висновку:

Експериментувати з довжиною спринту треба на початковому етапі. Не витрачайте надто багато часу на аналіз, просто виберіть відповідну довжину і використовуйте її упродовж одного або двох спринтів, після чого можете вибрати нову.

Проте, як тільки ви знайдете відповідною довжину, надовго зафіксуйте її. Після декількох місяців експериментів нам підійшла довжина в 3 тижні, тому тепер ми дотримуємося цього правила і використовуємо тритижневі спринти. Іноді спринт здаватиметься занадто довгим, іноді — занадто коротким. Але збереження фіксованої довжини спринту дозволяє виробити корпоративний ритм, в якому усі почувають себе достатньо комфортно. До того ж зникнуть спори, щодо дати релізу, оскільки усі знають, що випуск нової версії продукту кожні 3 тижні.

2.7 Визначення мети спринту

Це трапляється практично завжди, коли в ході нашого планування я ставлю питання: «Отже, яка ж мета спринту»?. Усі починають дивитися на мене здивованими очима, а product owner — морщити лоб, почухувавши своє підборіддя.

Чомусь сформулювати мету спринту буває досить непросто

. Але я досі переконаний, що зусилля, витрачені на спроби сформулювати мету, виправдовують себе. Краще паршива мета, чим її відсутність. Наприклад,

цілі можуть бути наступні: «заробити більше грошей», «завершити три історії з найвищими пріоритетами», «здивувати виконавчого директора», «підготувати систему до бета-тестированию», «додати можливість адміністрування» або щонебудь в цьому дусі. Найголовніше, щоб мета була позначена в термінах бізнесу, а не в технічних термінах. Тобто мовою, зрозумілим навіть людям поза командою.

Мета спринту повинна відповідати на головне питання «Навіщо ми працюємо над цим спринтом? Чому ми усі просто не підемо у відпустку»?. Насправді, найпростіший спосіб витягнути мету спринту з product owner'a — безпосередньо поставити йому це питання.

Метою має бути щось, що не було ще досягнуто. «Здивувати виконавчого директора» може бути непоганою метою. Але тільки не у тому випадку, коли він і так в захваті від поточного стану системи. В цьому випадку, усі можуть просто зібратися і піти додому, а мета спринту все одно буде досягнута.

Мета спринту може здатися злегка безглуздою і надуманою упродовж усього планування. Але найчастіше, основна її цінність починає проявлятися до середини спринту, коли люди починають забувати чого вони хочуть досягти в цьому спринті. Якщо у вас працюють декілька Scrum -команд (як у нас) над різними продуктами, дуже корисно мати можливість проглянути список цілей спринтів для усіх команд на одній wiki -странице (чи ще де-небудь), а також вивісити їх на видному місці, щоб усі (а не тільки топ-менеджери) знали, чим займається компанія і навіщо!

Вибір історій, які увійдуть до спринту

Основне в плануванні спринту — процедура вибору історій, які увійдуть до спринту. А точніше, вибір історій, які треба скопіювати з product backlog'a в sprint backlog.

Погляньте на картинку. Кожен прямокутник є історією, розташування якої відповідає рівню її важливості. Найбільш важлива історія знаходиться нагорі списку. Розмір історії (т. е. кількість story point 'a) визначає розмір кожного прямокутника. Висота блакитної дужки означає

прогнозовану продуктивність команди

, т. е. кількість історій, яку команда збирається завершити в наступному спринті.

Чесно кажучи, sprint backlog — це вибірка історій з product backlog'a. Він є списком історій, які команда зобов'язалася виконати впродовж спринту.

Саме

команда

вирішує, скільки історій увійде до спринту. Ні product owner, ні хто-небудь ще.

У зв'язку з цим, виникають два питання:

1. Яким чином команда вирішує, які історії потраплять в спринт?
2. Як product owner може вплинути на їх рішення?
Почну з другого питання.

Як product owner може впливати на те, які історії потраплять в спринт?

Допустимо, на плануванні спринту виникла наступна ситуація:
Product owner 'а розчарував той факт, що історія «Г» не потрапляє в спринт. Що він може зробити в ході наради?
Другий варіант — зміна об'єму робіт : product owner починає зменшувати об'єм історії «А» до тих пір, поки команда не вирішить, що історію «Г» можна втиснути в спринт.
Перший варіант — зміна пріоритетів. Якщо product owner призначить історії «Г» вищий пріоритет, то команда буде зобов'язана включити її в спринт першої (виключивши при цьому історію «В»).
Третій варіант — розбиття історії. Product owner може вирішити, що деякі частини історії «А» не так вже і важливі. Таким чином, він розбиває історію «А» на дві історії «А1» і «А2», а потім призначає ним різний пріоритет.
І так, не дивлячись на те, що у більшості випадків product owner не може контролювати прогнозовану продуктивність, у нього існує безліч способів вплинути на те, які історії потраплять в спринт.

Як команда приймає рішення про те, які історії включати в спринт?

Ми використовуємо два підходи:
1. На основі інтуїції.
2. На основі підрахунку продуктивності.

Планування, засноване на інтуїції. ScrumMaster: «Хлопці, ми закінчимо історію »А« в цьому спринті»? (Показує на найважливішу історію в product backlog 'а)
Ліза: «Звичайно, закінчимо. У нас є три тижні, а це досить тривіальна функціональність». ScrumMaster: «Добре. А як на рахунок історії »б«»? (Показує на другу по важливості історію) Том і Ліза одночасно: «Легко»!
ScrumMaster: «Добре. Як на рахунок історій »А«, »Б« і »В«»?
Сем (звертаючись до product owner) : чи «Треба реалізовувати розширену обробку помилок для історії »В«»?
Product owner: «Немає. Поки вистачить базової». Сем: «У такому разі історію »В« ми теж закінчимо». ScrumMaster: «Добре, як на рахунок історії »Г«»? Ліза: «Хмм».

Том: «Гадаю, що зробимо». ScrumMaster: «Вірогідність 90 % або 50 %»? Ліза і Том: «швидше 90 %».

ScrumMaster: «Добре, значить, включаємо історію Г в цей спринт. Що скажете на рахунок історії »Д«»? Сем: «Можливо».

ScrumMaster: «90 %? 50 %»? Сем: «Ближче до 50 %». Ліза: «Сумніваюся».

ScrumMaster: «У такому разі, не включаємо історію »Д«. Зобов'язуємося реалізувати історії »А«, »Б«, »В« і »Г«. Звичайно, якщо встигнемо, то реалізуємо і історію »Д«, проте не стоїть на це расчитывать. Тому історію »Д« виключаємо з плану спринту. Згодні»? Усе: «Згодні».

Інтуїтивне планування добре працює для маленьких команд і коротких спринтів.

Планування, засноване на методі оцінки продуктивності. Цей підхід включає два етапи:

1. Визначити

прогнозовану продуктивність.

2. Порахувати, скільки історій ви можете додати без перевищення прогнозованої продуктивності.

Продуктивність є мірою «кількості виконаної роботи». Вона розраховується як сума первинних оцінок усіх історій, які були реалізовані впродовж спринту.

На наступному малюнку показаний приклад прогнозованій продуктивності на початку спринту і реальній продуктивності у кінці спринту. Кожен прямокутник означає історію, число усередині прямокутника — це його початкова оцінка.

Пам'ятайте, що реальна продуктивність розраховується на підставі початкової оцінки кожної історії. Будь-які зміни оцінки впродовж спринту ігноруються.

Я вже чую ваші заперечення: «Яка від цього користь? Високий або низький рівень продуктивності залежить від мільйона чинників! Недалекі програмісти, неправильна початкова оцінка, зміна об'єму робіт, незаплановані потрясіння в ході спринту і т. д».

Згоден, продуктивність — це приблизна величина. Але, проте, дуже корисна. Вона краща, ніж нічого. Продуктивність дає нам наступне: «Незалежно від причин, ми маємо різницю між запланованим і виконаним об'ємом робіт».

А що, якщо історія була майже закінчена? Чому ми не використовуємо дробові значення для таких історій при підрахунку реальної продуктивності? Тому, що Scrum (як і гнучка розробка (agile), та і бережливе виробництво (lean)) орієнтований на те, щоб створювати закінчений, готовий до постачання продукт! Цінність реалізованої наполовину історії нульова (а то і негативна). Див. книгу «Managing the Design Factory» автора Donald Reinertsen або одну з книг авторів Mary Poppendieck і Tom Poppendieck.

Так яким же магічним способом ми оцінюємо продуктивність?

Найпростіше оцінити продуктивність, проаналізувавши попередні результати команди. Яка продуктивність була впродовж декількох останніх спринтів? Приблизно такою ж вона буде і в наступному спринті.

Цей підхід відомий під назвою вчорашня погода. Він виправданий для тих команд, які вже провели декілька спринтів (є статистичні дані) і планують наступний спринт без істотних змін, т. е. той же склад команди, робочі умови і т. д. Звичайно, це не завжди можливо.

Більше просунутий варіант оцінки продуктивності полягає у визначенні доступних ресурсів. Допустимо, ми плануємо тритижневий спринт (15 робочих днів). Команда складається з 4-ех чоловік. Ліза бере два відгули. Петро зможе приділити проекту тільки 50 % часу плюс бере один відгул. Складемо всі разом .

Чи отримали ми прогнозовану продуктивність? Ні! Тому що наша одиниця виміру — це story point, який, в нашому випадку приблизно дорівнює «ідеальному людино-дню». Ідеальний людино-день — це максимально продуктивний день, коли ніхто і ніщо не відволікає від основного заняття. Такі дні — рідкість. Крім того, треба брати до уваги, що в ході спринту може бути додана незапланована робота, людина може захворіти і т. д.

Без жодного сумніву, наша прогнозована продуктивність буде менше п'ятдесяти. Питання в тому, наскільки? Для відповіді на нього введемо визначення фокус- фактору.

Фокус-чинник — це коефіцієнт того, наскільки команда сфокусована на своїх основних завданнях. Низький фокус-чинник може означати, що команда чекає неодноразового втручання у свою роботу або припускає, що оцінки занадто оптимістичні.

Вибрати розумний фокус-чинник краще всього, узявши його з останнього спринту (а ще краще — середнє значення за декілька останніх спринтів).

За реальну продуктивність приймається сума початкових оцінок для тих історій, які були завершені в ході останнього спринту.

Допустимо, в ході останнього спринту командою з трьох чоловік у складі Тома, Лізи і Сема реалізоване 18 story point 'ов. Тривалість спринту була 3 тижні, що складає 45 людино-дня. Необхідно спрогнозувати продуктивність команди на майбутній спринт. Злегка ускладнимо завдання появою Степана — нового учасника команди. Зважаючи на відгули членів команди і інші вищезгадані обставини, отримаємо 50 людино-днів.

Таким чином, наша прогнозована продуктивність майбутнього спринту — це 20 story point 'ов. Це означає, що команді слід включати історії в план спринту до тих пір, поки їх сума приблизно не дорівнюватиме 20.

У нашому випадку команда може вибрати 4 найбільш важливі історії (що складає 19 story point 'ов) або 5 найбільш важливих історій (24 story point 'а). Зупинимося на чотирьох історіях, т. до. їх сума близька до 20. Якщо виникають сумніви, вибирайте менше історій.

З огляду на те, що вибрані 4 історії складають 19 story point 'а, остаточна прогнозована продуктивність майбутнього спринту складає 19.

Техніка «вчорашньої погоди» дуже зручна, проте використовувати її треба, покладаючись на здоровий глузд. Якщо останній спринт був надзвичайно поганим внаслідок того, що усі члени команди хворіли протягом тижня, це зовсім не означає, що подібна ситуація повториться в ході наступного спринту. Таким чином, фокус-чинник може бути збільшений. Якщо команда нещодавно впровадила надшвидку систему безперервної інтеграції, фокус-чинник також може бути збільшений. У разі, якщо до команди приєднався новий учасник, фокус-чинник треба зменшити, зважаючи на час, необхідне йому на те, щоб влитися в проект, і на навчання. І т. д.

Для того, щоб отримувати достовірніші оцінки, по можливості використовуйте усереднені дані за останні декілька спринтів.

Що якщо команда нова і не має ніякої статистики? В цьому випадку можна використати фокус-чинник інших команд, які працюють в схожих умовах.

Немає можливості узяти дані інших команд? Виберіть фокус-чинник навмання. Хороша новина полягає в тому, що фокус-чинник доведеться вгадувати лише для першого спринту. Після першого спринту ви матимете в розпорядженні статистичні дані і зможете безперервно вимірювати і удосконалювати ваш фокус-чинник і прогнозовану продуктивність.

В якості «значення за умовчанням» фокус-фактору для нових команд ми зазвичай використовуємо 70 %, т. до. це саме та межа, якої нашим командам вдавалося досягти за весь час їх роботи.

Яку техніку ми використовуємо для планування?

Я згадував декілька підходів підрахунку продуктивності : на основі інтуїції, на основі «вчорашньої погоди» і на основі визначення доступних ресурсів з урахуванням фокус-фактора.

Який же підхід використовуваний ми?

Зазвичай ми поєднуємо усі перераховані підходи. На це не вимагається багато часу.

Ми аналізуємо фокус-чинник і реальну продуктивність останнього спринту і проаналізувавши доступні ресурси, отримуємо фокус-чинник майбутнього спринту. Обговорюємо будь-які відмінності цих двох фокус-факторів і вносимо необхідні корективи.

Як тільки стане відомий приблизний список історій на майбутній спринт, ми проводимо перевірку з використанням підходу, заснованого на інтуїції. Просимо команду на деякий час забути про цифри і просто прикинути, наскільки реально реалізувати ці історії в одному спринті. Якщо здається, що їх багато, то виключаємо одну-дві історії.

Взагалі мета — прийняти рішення про те, які історії включати в спринт. А фокус-чинник, доступні ресурси і прогнозована продуктивність є лише інструментами її досягнення.

2.8 Як сформувати задачі для спринту?

Більшість зустрічей по плануванню спринту присвячена історіям з product backlog 'a: їх оцінці, розставлянню пріоритетів, уточненню вимог, розбиттю на частини і т. д.

Зазвичай команда включає проектор, який показує backlog. Хтось (зазвичай product owner або ScrumMaster) сідає за клавіатуру, швидко зачитує історію і починає обговорення. У міру того, як команда і product owner обговорюють пріоритети і деталі, хлопець за клавіатурою оновлює історію прямо в backlog.

Є спосіб трохи кращий — зробити облікові картки прикріпити їх на стіну.

Такий «призначений для користувача інтерфейс» виграє в порівнянні з комп'ютером і проектором, з наступних причин:

• Люди вимушені вставати і ходити, тому вони більше сконцентровані і не засинають.

• Кожен почуває себе причетним до процесу (а не тільки хлопець за клавіатурою).

• Можна редагувати декілька історій одночасно.

• Змінити пріоритети дуже просто — досить поміняти місцями облікові картки.

• Після закінчення зустрічі облікові картки можна забрати в кімнату, де працює команда, і використовувати їх на дошці завдань.

Ви можете заповнити облікові картки власноручно або (як ми зазвичай робимо) згенерувати версію для друку прямо з product backlog 'a, використовуючи простий скрипт.

Важливо: Після планування спринту наш scrum master вручну оновлює product backlog, щоб врахувати усі зміни, які були зроблені на картках. Так, це певна морока, але ми на це згодні з урахуванням того, наскільки ефективніше проходять зустрічі по плануванню спринту з використанням паперових облікових карток.

Декілька слів про поле «Важливість» (Importance). Це значення «важливості» з product backlog'a на момент роздруківки картки. Її позначення на картці допомагає нам відсортувати картки на стіні. Зазвичай ми розташовуємо важливіші елементи лівіше, а менш важливі — правіше. Проте, коли картки вже на стіні, можна забути про значення поля «важливість» і, замість цього, використовувати порядок, щоб показати відносну важливість історії. Якщо product owner міняє місцями картки, не витрачайте час на оновлення значень на папірцях. Просто після зу-

стрічі переконайся, що відновили значення міри важливості історій в product backlog'i.

Історію зазвичай можна оцінити легше і точніше, якщо вона розбита на завдання (task).

Використання карток також спрощує процедуру розбиття історій на завдання. Можна розбити команду на пари, тоді вони зможуть одночасно розбивати історії на завдання — кожна свою.

На нашій дошці ми відображаємо завдання у вигляді маленьких стікерів під кожною історією. Кожен стікер відповідає одному завданню у рамках цієї історії.

Ми не додаємо завдання в product backlog з двох причин:

1. Список завдань зазвичай часто міняється: приміром, завдання можуть змінюватися і переглядатися упродовж sprint'у. Щоб синхронізувати з ними product backlog потрібне надто багато зусиль.

2. Швидше за все, на цьому рівні деталізації product owner не братиме участь в процесі.

Так само, як і облікові картки, стікери із завданнями можна використовувати в sprint backlog'i.

Критерій готовності

Важливо, щоб і product owner, і команда спільними зусиллями визначили критерій готовності. Чи можна вважати історію готовою, якщо увесь код був доданий в репозиторій? Або ж вона вважається готовою, лише після того, як була розгорнута на тестовому сервері і пройшла усі інтеграційні тести? Де тільки це можливо, ми використовуємо наступне визначення критерію готовності : «історія готова до розгортання на живому сервері», проте, іноді ми вимушені мати справу з визначенням типу «розгорнуто на тестовому сервері і готове до приймального тестування».

Спочатку ми використовували деталізовані контрольні листи для визначення того, що історія готова. А зараз ми часто просто говоримо: «історія готова тоді, коли так вважає тестувальник з нашої Scrum - команди». В цьому випадку перевірка того, що побажання product owner 'а були правильно сприйняті командою, залишається на совісті тестувальника. У його завдання також входить контроль того, що історія досить «готова» для того, щоб задовольнити прийнятому критерію готовності.

Врешті-решт, ми усвідомили, що не можна усі історії рівняти під одну гребінку. Історія «форма пошуку користувачів» дуже сильно відрізнятиметься від історії під назвою «керівництво по експлуатації». У останньому випадку «готово» може просто означати «прийнято відділом підтримки клієнтів». Тому здоровий глузд часто краще, ніж формальний контрольний лист.

Якщо ви часто плутаєтеся з визначенням критерію готовності (як це було спочатку у нас), то вам, напевно, варто ввести поле «критерій готовності» для кожної історії.

2.9 Оцінка трудовитрат за допомогою гри в planning poker

Оцінка — це командна робота, і, частенько, усі члени команди беруть участь в оцінці кожної історії. Чому?

• Під час планування ми зазвичай не знаємо, хто виконуватиме ту або іншу частину.

• Реалізація історій зазвичай вимагає участі різних фахівців (дизайн призначеного для користувача інтерфейсу, кодування, тестування, і т. д.).

• Для того, щоб кожен учасник команди міг видати якусь оцінку, він повинен більш менш розуміти, в чому суть цієї історії. Отримуючи оцінку від кожного члена команди, ми переконуємося, що усі розуміють, про що йде мова. Це збільшує вірогідність взаємодопомоги по ходу спринту. А також це збільшує вірогідність того, що найбільш важливі питання по цій історії спливуть якомога раніше.

• При оцінці історії спільними зусиллями різнобічне бачення проблеми призводить до сильного розкиду оцінок. Такі розбіжності краще виявляти і обговорювати якомога раніше.

Якщо попросити усіх оцінити історію, то зазвичай людина, яка розуміє її краще за інших, видасть оцінку першим. Це сильно впливає на оцінки інших людей.

Але існує прекрасна практика, яка дозволяє цього уникнути. Вона називається planning poker (придумана Майком Коном).

Кожен член команди отримує колоду з 13-ти карт, таких же, як на рис. вище. Всякий раз, коли треба оцінити історію, кожен член команди вибирає карту з оцінкою (у story point 'ах), яка, на його думку, підходить, і кладе її на стіл сорочкою вгору. Коли усі члени команди визначилися з оцінкою, карти одночасно розкриваються. Таким чином, члени команди вимушені оцінювати самостійно, а не «списувати» чужу оцінку.

Якщо виходить велика різниця в оцінках, то цю різницю обговорюють і намагаються виробити загальне розуміння того, що повинно бути зроблено для реалізації цієї історії. Можливо, вони розіб'ють завдання на дрібніші. Після цього команда оцінить історію наново. Цей цикл повинен повторюватися до тих пір, поки оцінки не зійдуться, т. е. не стануть приблизно однаковими.

Дуже важливо нагадувати усім членам команди, що вони повинні оцінювати загальний об'єм робіт по історії, а не тільки «свою частину». Тестувальник повинен оцінювати не лише роботи по тестуванню.

Зверніть увагу, послідовність значень на картах — нелінійна. Ось, наприклад, між 40 і 100 нічого немає. Чому так?

Це треба, щоб уникнути появи помилкового почуття точності для великих оцінок. Якщо історія оцінюється приблизно в 20 story point 'а, те немає сенсу обговорювати чи повинна вона бути 20, або 18, або 21. Усе, що нам треба знати, це те, що її складно оцінити. Тому ми приблизно призначаємо їй оцінку в 20.

Якщо у вас виникло бажання детальніше переоцінити цю історію, то краще розбийте її на дрібніші частини і оціните вже їх!

І, до речі, шахраювати, викладаючи карти 5 і 2, щоб отримати 7, не можна. Ви повинні вибрати або 5 або 8 — сімки немає.

Є ще декілька спеціальних карт:

• 0 = чи «історія вже готова» або ж її оцінка «пара хвилин роботи».

•? = «Я поняття не маю. Абсолютно«.

• Чашка кави = «Я занадто втомився, щоб думати. Давайте зробимо перерву».

2.10 Уточнення описів історій

Немає нічого жахливішого, ніж ситуація, коли команда з пафосом демонструє нову функціональність продукту, а product owner тяжко зітхає і говорить: «ну так — усе це красиво, ось тільки не те, що я просив»!

Як переконатися, що product owner і команда розуміють історію однаково? Чи що усі члени команди розуміють усі історії однаково? Є прості способи виявити різницю в розумінні. Найбільш проста практика — завжди заповнювати усі поля для кожної історії (точніше, для усіх історій, які можуть потрапити в поточний спринт).

Приклад № 1:

Команда і product owner цілком задоволені планом на спринт і вже готові закінчити планування, але тут ScrumMaster говорить: «Хвилиночку! У нас немає оцінки для історії «додати користувача». Давайте оцінимо! Після пари здач в planning poker, команда сходиться на оцінці в 20 story point 'а, на що product owner схоплюється з криком: «Що?!»?. Пара хвилин запеклих суперечок і раптом з'ясовується, що команда мала на увазі «зручний web -интерфейс для функцій додати, редагувати, видалити і шукати користувачів», а product owner мав на увазі тільки «додавати користувачів безпосередньо у базу даних за допомогою SQL - клієнта». Команда оцінює історію наново і зупиняється на 5-ти story point 'ах.

Приклад № 2:

Команда і product owner цілком задоволені планом на спринт і вже готові закінчити планування, але тут Scrum master говорить: «Хвилиночку! Ось тут у нас є історія «додати користувача». Як вона може бути продемонстрована»?. Народ нашепчеться і через хвилину хтось встане і почне: «ну, спершу потрібно залогінитися на сайт, потім»., а product owner тут же переб'є: «залогінитися на сайт?! Ні, ця

функція взагалі до сайту не повинна мати ніякого відношення — це буде просто маленький SQL - скрипт, тільки для адміністраторів».

Поле «як продемонструвати» може (і повинно) бути дуже коротким!

Інакше ви не встигнете вчасно закінчити планування спринту. По суті, цей лаконічний опис на звичайній російській мові, як вручну виконати найбільш загальний тестовий приклад: «Зробити це, потім це, потім перевірити, що вийшло так-то».

Простий опис часто дозволяють виявити різне розуміння об'єму робіт для історій. Добре адже дізнатися про це заздалегідь, чи не так?

2.11 Розбиття історій на дрібніші історії

Історії мають бути не занадто маленькими, але і не занадто великими (у сенсі оцінок). Якщо ви отримали купу історій в половину story point'a, то ви напевно ляжете жертвою мікроменеджменту. З іншого боку, історія в 40 story point'oB несе в собі ризик того, що до кінця спринту її встигнуть закінчити лише частково, а незавершена історія не представляє цінності для вашої компанії, вона тільки збільшує накладні витрати. Далі — більше: якщо ваша прогнозована продуктивність 70 story point'oB, а дві найбільш важливі історії оцінені в 40, то планування декілька ускладниться. Команда стане перед вибором: або розслабитися (т. е. включити в спринт тільки одну історію), або узяти на себе нездійсненні зобов'язання (т. е. включити обоє).

Практично завжди є можливість розбити історію на дрібніші. Проте, в цьому випадку треба стежити за тим, щоб менші історії все ще представляли цінність з точки зору бізнесу.

Зазвичай ми прагнемо отримати історії об'ємом від двох до восьми людино-дня. Продуктивність нашої середньостатистичної команди зазвичай знаходиться в межах 40-ка — 60-ти людино-дня, що дозволяє нам включати в спринт приблизно по 10 історій. Іноді всього 5, а іноді цілих 15. До речі, таким числом облікових карток досить зручно оперувати.

2.12 Розбиття історій на задачі

У чому різниця між «завданнями» і «історіями»? Дуже правильне питання.

А відмінність дуже проста: історії це щось, що можна продемонструвати, що представляє цінність для product owner'a, а завдання або не можна продемонструвати, або вони не представляють цінності для product owner'a.

Приклад розбиття історії на дрібніші:

Приклад розбиття історії на завдання:

Декілька цікавих спостережень:

• Молоді Scrum -команды не люблять витрачати час на попереднє розбиття історій на завдання. Деякі вважають це «водоспадним» підходом.

• Абсолютно зрозумілі історії розбивати на завдання заздалегідь так само легко, як і у міру їх виконання.

• Таке розбиття часто дозволяє виявити додаткову роботу, яка збільшує оцінку, чим забезпечується реалістичніший план на спринт.

• Таке попереднє розбиття помітно збільшує ефективність щоденного Scrum 'а (див. стор. 46 «Як ми проводимо щоденний Scrum»).

• Навіть неточне розбиття, яке змінюватиметься по ходу робіт, все одно дає нам усі перелічені вище вигоди.

Отже, щоб встигнути розбити історії на завдання, ми намагаємося виділити досить часу на планування спринту. Проте, якщо час підтискає, то розбиття на завдання ми можемо і пропустити (див. наступну главу «Коли пора зупинитися»).

2.13 Вибір часу і місця для щоденного Scum 'у

Усі часто забувають, що на плануванні спринту, окрім усього іншого, необхідно вибрати час і місце проведення щоденного Scrum 'а. Без цього ваш спринт приречений на невдалий старт. Адже перший щоденний Scrum — це, здебільшого, час, коли кожен вирішує з чого почати роботу.

Краще проводити щоденний Scrum уранці. Недоліки обіднього Scrum'а : приходячи на роботу уранці, вам потрібно спробувати згадати, чим ви обіцяли команді займатися сьогодні.

Недоліки уранішнього Scrum'а : приходячи на роботу уранці, вам потрібно спробувати згадати, чим ви займалися учора, щоб можна було відзвітувати про це.

Перший недолік гірший, оскільки найбільш важливе те, що ви збираєтеся робити, а не, що ви вже зробили.

Зазвичай вибираємо самий ранній час, який не викликає нарікань ні у кого в команді. Зазвичай це 9:00, 9:30 або 10:00. Дуже важливо, щоб усі в команді щиро погодилися на цей час.

Коли пора зупинитися? Ну ось, час закінчується. Чим ми можемо пожертвувати з усього того, що ми збиралися зробити на плануванні спринту, якщо час підтискає?

Пріоритети для зустрічі з плануванню спринту такі:

Пріоритет № 1: Мета спринту і дата демонстрації. Це той мінімум, з яким можна починати спринт. У команди є мета і крайній термін, а працювати вони можуть прямо по product backlog 'у. Та це недобре, і треба запланувати ще одну зустріч по плануванню sprint backlog 'а на завтра, але якщо вам украй необхідно

стартувати спринт, то це, швидше за все, спрацює. Хоча, якщо бути чесним, я так ніколи і не стартував спринт, маючи усього лише мету і дату.

Пріоритет № 2: Список історій, які команда включила в sprint backlog.

Пріоритет № 3: Оцінки для кожної історії з sprint backlog 'у.

Пріоритет № 4: Поле «Як продемонструвати» для кожної історії з sprint backlog 'а.

Пріоритет № 5: Розрахунки продуктивності і ресурсів в якості «випробування реальністю» для плану на спринт. Також потрібний список членів команди з вказівкою їх міри участі в проекті (без цього не можна розрахувати продуктивність).

Пріоритет № 6: Певний час і місце проведення щоденного Scrum 'а. Взагалі, вибрати час і місце можна за одну хвилину, але якщо час зборів вже закінчився, то ScrumMaster просто вибирає їх сам після зборів і оповіщає усіх по електронній пошті.

Пріоритет № 7: Історії, розбиті на завдання. Розбивати історії на завдання можна також і у міру їх вступу в роботу, поєднуючи це з щоденним Scrum 'ом, але такий підхід злегка порушує течію спринту.

2.14 Технічні історії

А тепер ще одна складна проблема: технічні історії. Це нефункціональні вимоги.

Я маю на увазі усе, що повинно бути зроблено, але невидимо для замовника, не відноситься ні до однієї user story, і не дає прямої вигоди product owner 'у

Ми називаємо це «технічними історіями».

Наприклад:

1. Встановити сервер безперервної інтеграції. Чому це потрібно зробити? Тому, що це заощадить багато часу розробникам і понизить ризик проблеми «великого вибуху» при інтеграції у кінці ітерації.
2. Описати архітектуру системи. Чому це потрібно зробити? Тому, що розробники часто забувають загальну архітектуру і тому пишуть неузгоджений код. Потрібний документ, що надає усім однакову загальну картину.
3. Рефакторинг шару доступу до даних. Чому це потрібно зробити? Тому, що шар доступу до даних став дуже заплутаним, і розробники втрачають багато часу на те, щоб розібратися і виправити виникаючі дефекти. Рефакторинг збереже час усієї команди і підвищить стійкість системи.
4. Відновити Jira (систему обліку дефектів). Чому це потрібно зробити? Поточна версія дуже нестабільна і повільна: оновлення збереже час усій команді.

Чи мають сенс ці історії? Чи це завдання не пов'язані ні з однією історією? Хто задає пріоритети? Чи треба залучати сюди Product owner 'а?

Ми спробували різні варіанти роботи з технічними історіями. Ми пробували вважати їх звичайнісінькими user story. Це була невдала ідея: для Product owner 'а приоритезировать їх в product backlog 'а було все одно, що порівняти тепле з м'яким. З очевидних причин технічні історії отримували найнижчий пріоритет з поясненням: «Так, хлопці, поза сумнівом, ваш сервер безперервної інтеграції — дуже важлива штука, але давайте спершу реалізуємо деякі прибуткові функції? Після цього ви можете прикрутити вашу технічну цукерочку?

В деяких випадках product owner дійсно правий, але частіше все-таки ні. Ми дійшли висновку, що product owner не завжди компетентний, щоб йти на компроміс. І ось що ми робимо:

1. Намагаємося уникати технічних історій. Шукаємо спосіб перетворити технічну історію на нормальну історію з вимірюваною цінністю для бізнесу. У такому разі у Product owner 'а більше шансів знайти розумний компроміс.

2. Якщо ми не можемо перетворити технічну історію на звичайну, дивимося чи не можна включити цю роботу у вже існуючу історію. Наприклад, «рефакторинг доступу до даних» міг би стати частиною історії «редагувати користувача», оскільки вона має на увазі роботу з даними.

3. Якщо обидва підходи не пройшли, відмічаємо це як технічну історію і зберігаємо список таких історій окремо. Нехай product owner бачить список, але не редагує. У переговорах з product owner 'ом використовуваний параметри «фокус-фактора» і «прогнозованій продуктивності» і виділяємо час в спринті для реалізації технічних історій.

Приклад (діалог дуже схожий на те, що сталося під час одного з планувань спринту).

Команда: «У нас є деякі внутрішні технічні роботи, які мають бути зроблені. Ми б хотіли закласти на це 10 % усього часу, т. е. понизити фокус-чинник з 75 % до 65 %. Це можливо»?

Product owner: «Природно, немає! У нас немає часу»!

Команда: «Добре, давайте подивимося на наш останній спринт (усі кидають погляд на графік продуктивності на білій дошці). Наша прогнозована продуктивність була 80, але реальна продуктивність виявилася 30, вірно»?

Product owner: «У нас немає часу на внутрішні технічні роботи! Нам потрібний новий функціонал»!

Команда: «Добре. Але причина погіршення нашій продуктивність в тому, що ми витрачаємо надто багато часу на створення повної збірки для тестування«.

Product owner: «Ну і що»?

Команда: «А те, що продуктивність і далі буде такою низькою, якщо ми нічого не зробимо».

Product owner: «Так і що ви пропонуєте»?

Команда: «Ми пропонуємо виділити приблизно 10 % наступного спринту на установку билд сервера і інші речі, щоб зробити інтеграцію менш хворобливою. Це, швидше за все, збільшить продуктивність усіх наступних спринтів як мінімуна 20 %»!

Product owner: «Серйозно? Чому ж ми це не зробили на попередньому спринті»?!

Команда: «Тому що ви не захотіли, щоб ми це зробили».

Product owner: «Гаразд, тоді логічно, якщо ви це зробите зараз»!

Звичайно, є і інший варіант: не вести переговори з product owner 'ом з приводу технічних історій, а просто поставити його перед фактом, що у нас фокус-чинник такий-то. Але це не правильно навіть не спробувати досягти компромісу.

Якщо product owner виявився кмітливим і компетентним, рекомендується інформувати його якнайкраще і дати йому можливість визначати загальні пріоритети. Адже прозорість — це один із засадничих принципів Scrum 'у!

2.15 Як використати систему обліку дефектів для ведення product backlog 'у

Є ще одне непросте завдання. З одного боку, Excel дуже хороший формат для product backlog 'а. З іншого боку, вам все одно потрібна система обліку дефектів, і Excel тут явно не тягне. Ми використовуємо Jira.

Отже, як ми переносимо завдання з Jira в планування спринту? Не можемо ж ми просто їх проігнорувати і зосередитися тільки на історіях.

Ми пробували наступні підходи:

1. Product owner роздруковує самі високопріоритетні завдання з Jira, виносить їх на планування спринту і вішає їх на стінку з іншими історіями (неявно вказуючи їх відносний пріоритет).

2. Product owner створює історії, що відповідають завданням з Jira. Наприклад, «Виправити найкритичніші помилки звітності в админке, Jira - 124, Jira - 126, і Jira - 180».

3. Роботи по виправленню помилок не включаються в спринт, тобто команда визначає досить низький фокус-чинник (наприклад, 50 %), щоб вистачало часу на виправлення. Потім, вводиться припущення, що команда в кожну ітерацію витрачатиме певну частину часу на помилки в Jira.

4. Заноситься product backlog в Jira (просто переносимо з Excele). Вважаємо помилки звичайними історіями.

Який підхід для нас найкращий? Насправді він може відрізнятися в різних командах і мінятися від спринту до спринту. Ми більше схиляємось до першого підходу: він простий і зрозумілий.

Планування спринту закінчене!

Планування спринту — найважливіша річ в Scrum 'і. Вкладете більше зусиль в планування — і усе інше піде як по маслу.

Планування спринту пройшло успішно, якщо усі (і команда, і product owner) з посмішкою завершують зустріч, з посмішкою прокидаються наступним ранком і з посмішкою проводять перший щоденний Scrum.

2.16 Як донести інформацію про спринт до усіх в компанії?

Важливо інформувати усю компанію про те, що відбувається у вашій команді. Якщо цього не робити, то інші почнуть скаржитися, або — що ще гірше — придумувати всякі жахи про вас.

Ми для цієї мети використовуємо «сторінку з інформацією про спринт».

Іноді ми також додаємо до назви історії поле «як продемонструвати»

Відразу ж після зустрічі по плануванню спринту цю сторінку створює ScrumMaster. Він поміщає її в wiki, і тут же спамит на усю компанію:

Окрім цього у нас є «панель» в wiki, в якій міститися посилання на поточні спринти усіх команд.

На додаток до усього цього наш ScrumMaster роздруковує сторінку інформації про спринт і вивішує її на стіну в коридорі. Таким чином хто завгодно, проходячи мимо, може поглянути на цю сторінку і дізнатися, чим же займається наша команда.

Коли спринт добігає кінця, ScrumMaster нагадує усім про демонстрацію, що наближається.

Якщо усе це робити, то ні у кого не вийде сказати, що він не міг дізнатися, чим займається команда.

3. Як створити sprint backlog?

Отже, ми тільки що закінчили планування і повідомили на весь світ про початок нашого нового спринту. Тепер настала черга ScrumMaster 'а створити sprint backlog. Це необхідно зробити потім планування спринту, але до початку першого щоденного Scrum 'а.

3.1 Формат sprint backlog 'у

Ми поекспериментували з різними форматами sprint backlog 'а, включаючи Jira, Excel, не забули спробувати і звичайну настінну дошку. Спершу в основному використовували Excel, в інтернеті можна знайти досить багато шаблонів для sprint backlog 'а, з автогенерацією burndown діаграм і всякими іншими примочками.

Опишемо в усіх подробицях формат sprint backlog 'a, який ми визнали найбільш ефективним — дошку задач на стіні.

Знайдіть велику стіну, на якій або нічого немає, або висить всяка нісенітниця на зразок логотипу компанії, старих діаграм або жахливих картинок. Очистите її (якщо необхідно, запитаєте дозволи). Склейте скотчем величезне полотно паперу (як мінімум 2x2 або 3x2 метри для великої команди). А потім зробіть щось подібне до цього:

Як варіант, можна використовувати білу дошку. Але таке її використання неефективне. Якщо це можливо, збережете білу дошку для нарисів майбутнього дизайну, а в якості дошки для завдань використовуйте «паперові стіни».

Примітка: якщо ви користуєтеся стікерами для завдань, не забудьте прикріпити їх скотчем, бо одного прекрасного дня ви знайдете їх акуратною купкою на підлозі.

3.2 Як працює дошка завдань?

Ви, звичайно, можете додати будь-які додаткові поля. Наприклад, «В очікуванні інтеграційного тестування» або «Скасованого». Але перш ніж усе ускладнювати, гарненько подумайте, чи дійсно ці доповнення так вже потрібні?

Простота украй цінна в такого роду речах, тому робимо ускладнення тільки у разі, якщо ціна занадто велика.

Приклад 1: після першого щоденного Scrum 'у.

Після першого щоденного Scrum'у дошка завдань може виглядати приблизно так:

Як видно, три завдання знаходяться «в процесі», тобто команда займатиметься ними сьогодні.

Іноді, у великих командах, завдання зависає в змозі «в процесі» тому, що ніхто не пам'ятає, хто над нею працював. Якщо таке трапляється часто, команда зазвичай вживає заходи. Наприклад, відмічає на картці завдання ім'я людини, яка взялася над нею працювати.

Приклад 2: ще через пару днів.

Через пару днів дошка завдань може виглядати приблизно так:

Як видно, ми закінчили історію «Депозит» (т. е. вона була зафіксована в системі контролю версій, протестована, отрефакторена і т. д.) «Автоматичне оновлення» зроблене частково, «Адмінка: вхід» розпочатий, а «Адмінка: управління користувачами» ще немає.

У нас виникло 3 незаплановані завдання, як видно справа внизу. Про це корисно буде згадати на ретроспективі.

Ось приклад сьогодення sprint backlog 'а ближче до кінця спринту. Він, насправді, стає досить безладним по ходу спринту, але нічого страшного, оскільки це ненадовго. На кожен новий спринт ми починаємо sprint backlog з чистого аркуша.

3.3 Як працює burndown –диаграмма?

Давайте придивимося до burndown - діаграми:

Ось про що вона говорить:

• У перший день спринту, 1-го серпня, команда визначила, що роботи залишилося приблизно на 70 story point 'ів. Це якраз і є прогнозована продуктивність на цей спринт.

• 16-го серпня роботи залишилося приблизно на 15 story point 'а. Пунктирна прямна показує, що вони на вірному шляху, тобто в такому темпі вони встигнуть закінчити усі завдання до кінця спринту.

Ми пропускаємо вихідні по осі X, оскільки в цей час рідко щось робиться. Раніше ми їх включали, але burndown при цьому виглядав дивно, оскільки вона «вирівнювався» на вихідних, і це схоже на привід для занепокоєння.

Тривожні сигнали на дошці задач. Побіжний погляд на дошку завдань повинен дати можливість будь-якій людині зрозуміти, наскільки успішно просувається ітерація. ScrumMaster несе відповідальність за те, щоб команда вживала відповідні заходи при виявленні перших тривожних симптомів :

Кращий варіант відстежування змін, який можна запропонувати при цьому підході — це робити фотографію дошки завдань щодня. Робіть так, якщо це необхідно.

Якщо відстежування змін для вас дуже важливе, тоді можливо підхід з дошкою завдань вам взагалі не підходить.

І все-таки, я б запропонував вам спочатку оцінити значення детального відстежування змін в спринті. Після того, як спринт закінчений і робочий код разом з документацією був відісланий замовникові, чи буде хто-небудь розбиратися, скільки історій було закінчено на 5-й день спринту? Чи буде хто-небудь хвилюватися про те, яка була реальна оцінка для завдання «Написати приймальний тест для історії Депозит» після того, як історія була закінчена?

3.4 Як оцінювати час виконання задач?

У більшості книг і статей, присвячених Scrum 'у, для оцінки часу виконання завдань використовуються години, а не дні. І ми так раніше робили. Загальна формула була такою: один ефективний людино-день дорівнює шести ефективним людино-годинам.

Проте ми відмовилися від цієї практики з наступних причин:

• Оцінки в людино-годинах занадто дрібні, що призводить до появи великої кількості крихітних завдань по годині або два, і, як результат, до мікроменеджменту (micromanagement).

• Виявилось, що все одно усі оцінюють в людино-днях, а коли треба отримати оцінку в людино-годинах, просто множать дні на шість. «Це завдання займе приблизно день. Ага, я повинен дати оцінку в годинах. Що ж, це означає шість годин».

• Дві різні одиниці виміру вводять в оману. «Це оцінка в людино-днях або людино-годинах»?

Тому ми використовуємо людино-день як основну одиницю при оцінці часу (ми називаємо їх story point 'и). Наше найменше значення — 0.5. Т. е. будь-які завдання, оцінені менш ніж в 0.5, або віддаляються, або комбінуються з іншими, або оцінка залишається 0.5 (нічого страшного в злегка завищеній оцінці немає). Витончено і просто.

3.5 Як улаштувати кімнату команди

Більшість цікавих і корисних обговорень виникають спонтанно, прямо перед дошкою завдань. Тому ми намагаємося оформити це місце як окремий «куточок обговорень». Це і справді корисно. Немає кращого способу подивитися на систему в цілому, чим стати в куточок обговорень, подивитися на обидві стіни, а потім сісти за комп'ютер і поклацувати останню збірку системи.

«Стіна проектування» — це просто велика дошка, на якій розвішані найважливіші для розробки блок-схеми, ескізи інтерфейсу, моделі предметної області і т. п.

На фотографії: щоденний Scrum у вищезгаданому куточку.

Ця burndown діаграма виглядає підозріло красивою і гладкою, правда? Але команда наполягає на тому, що це правда.

Усадіть команду разом. Коли приходить час розставити столи і розсадити команду, є одно правило, яке складно переоцінити. Усадіть команду разом!

Людям не подобається переїжджати. Вони не хочуть збирати усі свої дрібнички, висмикувати шнури з комп'ютера, переносити усе своє добро на інший

стіл, і знову встромляти усі шнури. Чим менша відстань, тим більше невдоволення. «Гаразд, шеф, навіщо мене пересаджувати всього на 5 метрів управо»?

Але коли ми будуємо ефективну команду по Scrum 'у іншого виходу немає. Просто зберіть команду разом. Навіть якщо доведеться їм погрожувати, самому переносити усе їх майно, і самому витирати застарілі сліди від чашок на столах. Якщо для команди немає місця — знайдіть місце. Де завгодно. Навіть якщо доведеться посадити команду в підвалі. Пересуньте столи, підкупіть офіс-менеджера, робіть усе, що треба. Але як би то не було, зберіть команду разом.

Як тільки ви зберете усю команду разом, результат з'явиться негайно. Вже після першого спринту команда погодиться, що це була хороша ідея — зібратися усім в одному місці, хоча немає ніяких гарантій, що команда не виявиться занадто упертою, щоб це визнати.

Так, до речі, а що означає «разом»? Як повинні стояти столи? Ну, особисто у мене немає однозначної думки про найкраще розташування столів. Але навіть якщо б і було, я думаю, у більшості команд немає такої розкоші — вирішувати, як будуть розставлені столи в їх кімнаті. Завжди існують фізичні обмеження — сусідня команда, двері в туалет, здоровий автомат з напоями посеред кімнати — та що завгодно.

Разом означає:

- В межах чутності: кожен в команді може поговорити з будь-яким іншим членом команди без крику і не встаючи з-за свого столу.
- В межах видимості: кожен член команди може побачити будь-якого іншого. Кожен може бачити дошку завдань. Не обов'язково бути досить близько, щоб читати, але, принаймні бачити.
- Автономно: якщо раптом уся ваша команда підніметься і почне раптову і дуже жваву дискусію про архітектуру системи, нікого з не членів команди не виявиться досить близько, щоб йому це перешкодило. Автономно не означає, що команда має бути повністю ізольована. У просторі, розділеному на секції, цілком може вистачити окрема секція для команди, з досить високими стінами, щоб не пропускати велику частину шуму ззовні.
- А як бути з розподіленою командою? Ну, тоді вам не повезло. Щоб зменшити негативні наслідки, використовуйте якомога більше технічних засобів: відеоконференції, вебкамеры, засоби для спільного використання робочого столу і т. п.

Не підпускайте product ownera занадто близько. Product owner повинен знаходитися настільки близько до команди, щоб у разі виникнення питань, команда могла б запитати його особисто, і щоб він мав можливість на своїх двох підійти до дошки завдань. Але він не повинен сидіти в одній кімнаті з командою. Чому? Тому що є вірогідність, що він не зможе не вдаватися до подробиць, а команда не

зможе правильно працювати, тобто не досягне стану повної автономності, самомотивации і надпродуктивності.

Це усього лише припущення: насправді, я сам ніколи не бачив, щоб product owner сидів поряд з командою, а це означає, що немає підстав говорити, що це погана ідея. Мені це просто підказує внутрішнє чуття ScrumMaster 'а.

Я дотримуюся наступної точки зору : якщо ви Scrum - тренер (і можливо поєднуєте цю роль з роллю менеджера), не бійтеся дуже щільно працювати з командою. Але тільки **впродовж певного проміжку часу**, **а потім залиште команду** в спокої і дайте їй можливість спрацьовувати і самоорганізовуватися. Час від часу контролюйте її (проте не дуже часто). Це можна робити, відвідуючи демо, вивчаючи дошку завдань або беручи участь в щоденному Scrum 'і. Якщо ви побачите як можна поліпшити процес, відведіть ScrumMaster 'а в сторону і дайте йому ділову пораду. Не варто повчати його на очах у усієї команди. Відвідування ретроспективи теж буде не зайвим. Якщо міра довіри до вас з боку команди невелика, то зробіть добру справу, не ходіть на ретроспективи, а то ваша присутність змусить команду мовчати про свої проблеми.

Якщо Scrum-команда працює добре, забезпечте її усім необхідним, а потім залиште в спокої.

3.6 Як проводити щоденний Scrum?

Щоденний Scrum починається в один і той же час, в одному і тому ж місці. Спочатку, ми вважали за краще проводити його в окремій кімнаті (це були дні, коли ми використовували sprint backlog'n в електронному форматі), проте зараз ми проводимо щоденний Scrum біля дошки завдань в кімнаті команди. Немає нічого кращого!

Зазвичай ми проводимо зустріч стоячи, оскільки це зменшує вірогідність того, що тривалість нашої «планерки» перевищить 15 хвилин.

Зазвичай ми оновлюємо дошку завдань під час щоденного Scrum'у. У міру того, як кожен член команди розповідає про те, що він зробив за вчорашній день і чим займатиметься сьогодні, він переміщає стікери на дошці завдань. Як тільки розповідь торкається якогось незапланованого завдання, то для нього клеїться новий стікер. При оновленні тимчасових оцінок, на стікері пишеться нова оцінка, а стара закреслюється. Іноді стікерами займається Scrum Master, поки учасники говорять.

У деяких командах прийнято, що усі члени команди оновлюють дошку завдань перед кожною зустріччю. Це теж добре працює. Просто вирішіть, що вам ближче, і дотримуйтеся цього.

Незалежно від того, який формат sprint backlog'у ви використовуєте, намагайтеся залучити усю команду у підтримку sprint backlog'у в актуальному стані.

Ми пробували проводити спринти, в яких тільки ScrumMaster займався підтримкою sprint backlog'a. Він повинен був щодня обходити усіх членів команди і запитувати про оцінки часу, що залишився до закінчення. Недоліки цього підходу в тому, що :

• ScrumMaster витрачає надто багато часу на «паперову роботу», замість того, щоб займатися підтримкою команди і усуненням перешкод.

• Члени команди не в курсі стану спринту, оскільки їм не треба піклуватися про sprint backlog'e. Ця нестача зворотного зв'язку зменшує загальну гнучкість і сконцентрированность команди.

Якщо sprint backlog має належний вигляд те будь-який член команди може легко його відновити.

Відразу ж після щоденного Scrum'у, хтось підсумовує оцінки усіх завдань на дошці (звичайно, окрім тих, які знаходяться в колонці «Готово») і ставить нову точку на burndown -діаграму.

Як бути з тими, що запізнилися. Деякі команди заводять спеціальну скарбничку. Якщо ви запізнилися, навіть на хвилину, ви кидаєте в скарбничку певну суму. Без варіантів. Навіть якщо ви подзвонили перед початком щоденного Scrum'a і попередили, заплатити все одно доведеться.

Відкрутитися можна лише у виняткових випадках. Наприклад, візит до лікаря, власне весілля або щось не менш важливе.

Гроші із скарбнички використовуються на громадські потреби. Наприклад, на них можна замовити піцу. Цей підхід працює непогано. Але користуватися їм треба лише у тому випадку, коли люди часто спізнюються. Деяким командам це просто не треба.

3.7 Що робити з тими, хто не знає, чим себе зайняти?

Іноді хтось говорить: «Учора я робив те-то і те-то, а сьогодні немає чіткого представлення у мене, чим зайняти себе». Наші дії?

Допустимо, Ваня і Ліза не знають, чим сьогодні зайнятися.

Якщо я виступаю в ролі ScrumMaster 'а, я просто передаю слово наступному. При цьому беру на олівець тих, хто не знає, чим йому зайнятися. Після того, як усі висловилися, я пробігаю разом з командою по дошці завдань і перевіряю, що дані на дошці завдань актуальні і що усі розуміють сенс кожної історії. Далі я пропоную кожному учасникові команди приклеїти нові стікери. Після цього повертаюся до тих, хто не знав, чим себе зайняти, з питанням «Після того, як ми пройшлися по дошці, чи не з'явилося у вас уявлення про те, чим зайнятися»? Сподіваюся, на той час воно вже з'явиться.

Якщо ж ні, то я з'ясовую, чи є можливість для парного програмування. Допустимо, сьогодні Коля планує реалізувати інтерфейс для адмінки. Ви можете за-

пропонувати Вані або Лізі попрацювати в парі з Колею над цією функціональністю. Зазвичай це працює.

Якщо ні, то ось вам наступний прийом.

ScrumMaster: «Так, і хто хоче продемонструвати нам готову бета-версию»? (припускаючи, що випуск бета-версии — мета спринту) Команда: здивована тиша ScrumMaster : «Ми що — не готові»? Команда: «ммм. ні».

ScrumMaster: «Чому? Що ще залишилося незавершеним»?

Команда: «Так у нас навіть немає тестового сервера. Крім того треба полагодити білд-скрипт». ScrumMaster: «Ага» (наклеює два нових стикера). «Ваня і Ліза, вам все ще нічим зайнятися сьогодні»?

Ваня: «Думаю, що я спробую роздобути тестовий сервер». Ліза: «А я постараюся полагодити білд-скрипт».

Якщо повезе, хтось дійсно зможе продемонструвати готовий до випуску бета-версії реліз. Відмінно! Мета спринту досягнута. Але що робити, якщо пройшла тільки половина часу, відведеного на спринт. Усе просто. Привітайте команду за відмінну роботу, візьміть одну-дві історії із стопки «Наступні» і помістите їх в колонку «В планах». Після цього повторно проведіть щоденний Scrum. Повідомите Product owner 'а про те, що ви додали декілька нових історій в спринт.

Що ж робити, якщо команда не досягла мети спринту, а Ваня з Лізою все ще не можуть визначитися з тим, яку користь вони можуть принести? Я зазвичай користуюся однією з методик (усі вони не дуже хороші, але все таки це останній засіб) :

- Присоромити: «Гаразд, якщо не знаєш, як принести користь команді, йди додому, читай книги і т. д. Чи просто сиди тут, поки комусь не знадобиться твоя допомога».
- По-старому: Просто призначити їм завдання.
- Моральний тиск: Скажіть їм: «Ваня і Ліза! Не смію вас більше затримувати. А ми усі просто постоїмо тут, поки у вас не з'являться ідеї, як допомогти нам в досягненні мети».
- Закабалити: Скажіть їм: «Ви зможете допомогти команді, виконуючи роль прислуги сьогодні. Готуйте каву, робіть масаж, винесіть сміття, приготуйте обід: робіть усе, про що вас може попросити команда».

Ви будете здивовані, наскільки швидко Ваня і Ліза знайдуть для себе корисні технічні завдання.

Якщо у вас є людина, яка часто примушує вас заходити так далеко, можливо, ви повинні відвести цього товариша в сторону і культурно пояснити йому, що він не правий. Якщо і після цього проблема не вирішиться, треба зрозуміти, чи важлива ця людина для команди або ні?

Якщо вона не дуже важлива, постарайтеся **виключити її зі своєї команди**.

Якщо ж вона важлива для команди, спробуйте знайти їй напарника-наставника. Ваня може бути класним програмістом і відмінним архітектором, просто він вважає за краще, щоб йому інші люди говорили, що він повинен робити. Призначте Колю наставником Вані. Якщо Ваня дійсно важливий для команди, такою буде ціна досягнення мети.

3.8 Як проводити демо?

Демонстрація спринту — дуже важлива частина Scrum 'у, яку багато хто все ж недооцінює:

- Ой, а нам що обов'язково робити демо?
- Ми все одно нічого цікавого не покажемо!
- У нас немає часу на підготовку різних демо!
- У мене купа роботи, не вистачає ще дивитися чужі демо!

Чому ми наполягаємо на тому, щоб кожен спринт закінчувався демонстрацією?

- Добре виконане демо чинить великий вплив на розробку, навіть якщо воно не здалося захоплюючим.
- Позитивна оцінка роботи надихає команду.
- Усі інші дізнаються, чим займається ваша команда.
- На демо зацікавлені сторони обмінюються життєво важливими відгуками.
- Демо проходить в дружній атмосфері, тому різні команди можуть вільно спілкуватися між собою і обговорювати насущні питання. Це цінний досвід.
- Проведення демо примушує команду дійсно дороблять завдання і випускати їх (навіть якщо це усього лише на тестовий сервер).
- Без демо постійно виявлялась купа на 99 % зробленої роботи. Проводячи демо, ми можемо отримати менше зроблених завдань, але вони будуть дійсно закінчені, що набагато краще, ніж коли функціонал буде перенесений у наступний спринт.
- Якщо команду примушувати проводити демо, коли у них нічого толком не працює, їм буде ніяково. Команда запинатиметься і спотикатиметься, показуючи функціональність, і добре, якщо у кінці ви почуєте ріденькі оплески.

Людям буде шкода цю команду, а деяких може навіть розсердити те, що вони тільки втратили час на цьому демо. Це дуже неприємно. Але це діє, як гірка пігулка. У наступному спринті команда дійсно постарається усе доробити!

Вона думатиме «гаразд, може, в наступному спринті варто показати всього дві задачі замість п'яти, але, цього разу вони ПРАЦЮВАТИМУТЬ»!. Команда знає, що демо доведеться проводити не дивлячись ні на що, і завдяки цьому шанси побачити там щось пристойне значно зростають.

При підготовці і проведенню демо:

- Постарайтеся як можна чіткіше озвучити мету цього спринту. Якщо на демо є присутніми люди, які нічого не знають про ваш продукт, то не полінуєтеся приділити пару хвилин, щоб ввести їх в курс справи.
- Не витрачайте багато часу на підготовку демо, особливо на створення ефектної презентації. Викиньте усе непотрібне і сконцентруйтеся на демонстрації тільки реально працюючого коду.
- Стежте, щоб демо проходило в швидкому темпі. Сконцентруйтеся на створенні не стільки красивого, скільки динамічного демо.
- Нехай ваше демо буде бізнес-орієнтованим, забудьте про технічні деталі. Сфокусуйтеся на тому «що ми зробили», а не на тому «як ми це робили».
- Якщо це можливо, дайте аудиторії самій спробувати запустити продукт.
- Не треба показувати купу виправлень дрібних багів і елементарних задач. Ви можете згадати про них, але демонструвати їх не варто, тому що це забере у вас багато часу і понизить увагу до важливіших історій.

Що робити з «недемонстрованими» речами:

Член команди : «Я не збираюся демонструвати це завдання, тому що її неможливо продемонструвати. Я говорю про історію 'Поліпшити масштабованість системи так, щоб вона могла обслуговувати одночасно 10 000 користувачів'. Я, побудь-якому, не зможу запросити на демо 10 000 користувачів».

ScrumMaster: «Так, ти закінчив з цим завданням»?

Член команди : «Ну, звичайно».

ScrumMaster: «А як ти дізнався, що воно потягне»?

Член команди : «Я конфігурував нашу систему в середовищі, призначеному для тестування продуктивності, і навантажив систему одночасними запитами з восьми серверів відразу».

ScrumMaster: «Так у тебе є дані, які підтверджують, що система може обслужити 10 000 користувачів»?

Член команди : «Так. Хоч тестові сервери і слабенькі, проте, в ході тестирования вони все одно впоралися з 50 000 одночасних запитів».

ScrumMaster: «Так, а звідки у тебе ця цифра»?

Член команди (засмучений) : «Ну, добре, у мене є звіт! Ти можеш сам глянути на нього, там описано як уся ця справа була конфігурована і скільки запитів було відіслано»!

ScrumMaster: «О, відмінно. Це і є твоє демо. Просто покажи цей звіт аудиторії і коротко пробіжися по ньому. Все ж краще, ніж нічого, правда»?

Член команди : «А що, цього вистачає? Правда він виглядає якось корявенько, потрібно б його трішки шлифонути».

ScrumMaster: «Добре, тільки не витрачай на це надто багато часу. Він не зобов'язаний бути красивим, головне — інформативним».

3.9 Як проводити ретроспективи?

Чому ми наполягаємо на тому, щоб усі команди проводили ретроспективи?

Найбільш важлива річ відносно ретроспектив - це їх проведення.

З деяких причин команди не виявляють належної цікавості до проведення ретроспектив. Без невеликого тиску з боку багато команд часто пропускають ретроспективу і відразу переходять до наступного спринту. Хоча при цьому, усі ніби погоджуються, що ретроспективи украй корисні. Можна сказати ретроспектива є другим за значимістю заходом в Scrum'і (перший — це планування спринту), тому що це самий відповідний момент для початку поліпшень!

Звичайно, щоб виникла хороша ідея, вам не потрібна ретроспектива, вона може прийти до вас в голову! Але чи підтримає команда вашу ідею? Можливо, але вірогідність значно вища, якщо ідея народжується усередині команди, тобто, під час ретроспективи, коли кожен може зробити свій вклад в обговорення ідеї.

Без ретроспектив ви виявите, що команда наступає на одні і ті ж граблі знову і знову.

Хоча основний формат ретроспективи трохи варіюється, але в основному ми робимо так:

- Виділяємо 1-3 години, залежно від того наскільки довга очікується дискусія.
- Беруть участь : product owner, уся команда і ScrumMaster власною персоною.
- Розташовуємося або в окремій кімнаті із затишним м'яким куточком, або на терасі, або в якомусь другом схожому місці, оскільки нам подобається вести дискусію в спокійній і невимушеній атмосфері.
- Частенько ми намагаємося не проводити ретроспективи в робочій кімнаті, оскільки це розсіює увагу учасників.
- Вибираємо когось в якості секретаря.
- ScrumMaster показує sprint backlog і за участю команди підводить підсумки спринту. Важливі події, виводи і т. д.

- Починаємо «серію» обговорень. У цей момент кожен має шанс висловитися про те, що, на його думку, було хорошого, що можна було б поліпшити і що б він зробив по-іншому в наступному спринті. При цьому його ніхто не перебиває.
- Ми порівнюємо прогнозовану і реальну продуктивність. Якщо є істотні розбіжності, то намагаємося проаналізувати і зрозуміти, чому так вийшло.
- Коли час добігає кінця, ScrumMaster намагається узагальнити усі конкретні пропозиції з приводу того, що ми можемо поліпшити в наступному спринті.

Ось приклад дошки з нашої ретроспективи:

Головна тема — завжди одна і та ж : «Що ми можемо поліпшити в наступному спринті». У нас є три колонки:

Добре: Якщо треба було б повторити цей спринт ще раз, то ми б зробили це точно так же.

Могло б бути і краще: Якщо треба було б повторити цей спринт ще раз, то ми б зробили це по-іншому.

Поліпшення: Конкретні ідеї про те, як в майбутньому можна щось поліпшити.

Таким чином, перша і друга колонки відноситься до минулого, тоді як третя — спрямована в майбутнє.

Після того, як команда закінчив свій мозковий штурм з приводу усіх цих стікерів, вони проводять «точкове голосування» для визначення поліпшень, яким слід приділити особливу увагу в ході наступного спринту. У кожного члена команди є три магнітики, якими він може скористатися для голосування. Кожен член команди може ліпити магнітики як йому надумається, хоч усі три відразу на одну задачу.

Ґрунтуючись на цьому голосуванні, ми вибираємо 5 поліпшень, які ми спробуємо впровадити в наступному спринті, а на наступній ретроспективі ми перевіримо, що у нас вийшло. Дуже важливо не переоцінити свої можливості. Виберіть всього декілька поліпшень для наступного спринту.

Інформація, яка спливає в ході ретроспектив, зазвичай украй важлива. Для команди настали нелегкі часи, тому що менеджери по продажах почали забирати програмістів з роботи на свої зустрічі, щоб ті грали роль «технічних експертів»? Це дуже важлива інформація. Можливо, що і у інших команд такі самі проблеми. Можливо, нам варто провести спеціальні тренінги по нашому продукту для відділу маркетингу, щоб вони самостійно змогли відповідати на усі питання клієнтів?

Можливі способи рішення проблем, знайдених командою на ретроспективі, можуть виявитися корисними не лише для неї самої, але і для інших.

Як же зібрати усі ці результати? Ми вибрали досить простий спосіб. Одна людина бере участь в усіх ретроспективах в ролі «сполучної ланки». Без всяких формальностей.

Найбільш важливі вимоги для людини, яка буде «сполучною ланкою» :

- Він має бути хорошим слухачем.
- Якщо ретроспектива проходить дуже в'яло, він має бути готовий поставити просте, але влучне питання, яке підштовхне людей на дискусію. Наприклад: «Якби можна було повернути час назад і переробити цей спринт з найпершого дня, щоб ви зробили по-іншому»?.
- Він має бути згоден витрачати свій час на відвідування усіх ретроспектив усіх команд.
- Він повинен мати необхідні повноваження, які допомогли б йому взятися за виконання запропонованих командою поліпшень, що виходять за межі можливостей самої команди.

Типові проблеми, які обговорюють на ретроспективах

1. «Нам потрібно було більше часу витратити на розбиття історій на підзадачі.

Щодня на Scrum 'і можна почути, як люди вимовляють побиту до болю фразу: «Я не знаю, що мені сьогодні робити». І вам доводиться день у день витрачати купу часу для того, щоб після Scrum 'у знайти завдання для цих хлопців. Моя порада — робіть це заздалегідь.

Стандартні дії: ніяких. Можливо, команда сама розв'яже цю проблему на наступному плануванні. Якщо ж це повторюється з разу в раз, збільште час на планування спринту.

2. «Дуже часто турбують ззовні».

Стандартні дії:

- Попросіть команду зменшити фокус-чинник на наступний спринт, щоб у них був реалістичніший план.
- Попрость команду детальніше записувати випадки втручання (хто і як довго). Потім буде легше розв'язати проблему.
- Попросите команду перекладати усі зовнішні запити на ScrumMaster 'а або Product owner 'а.
- Попросіть команду вибрати одну людину в якості «голкіпера» і перенаправляти на нього усі питання, які можуть відвернути команду від роботи. Це дозволить іншій частині команди сконцентруватися на своїх завданнях. У цій ролі може виступати, як ScrumMaster, так і будь-який член команди, якого треба буде періодично міняти.

3. «Ми узяли величезний шматок роботи, а закінчили тільки половину».

Стандартні дії: ніяких. Швидше за все, наступного разу команда не стане братися за нереальний об'єм робіт. Чи, принаймні, поскромніше оцінить свої можливості.

4. «У нас в офісі безлад і дуже шумно».

Стандартні дії:

- Спробуйте створити сприятливішу атмосферу або перевезіть команду на інше місце. Куди завгодно. Можете зняти кімнату в готелі (див. стор. 43 «Як ми улаштували кімнату команди»).
- Якщо це можливо, попросите команду зменшити фокус-чинник на наступний спринт з чітким описом причини : шум і безлад в офісі. Можливо, це змусить Product owner 'а почати штовхати менеджмент щодо вашої проблеми.

3.10 Відпочинок між спринтами

У реальному житті неможливо постійно бігти як спринтер. Між забігами вам у будь-якому випадку потрібний відпочинок. Якщо ви біжите з постійною максимальною швидкістю, то, по суті, ви просто біжите підтюпцем.

Те ж саме спостерігається в Scrum 'і, так і в розробці програмного забезпечення в цілому. Спринти дуже вимотують. Як у розробника, у вас ніколи немає часу, щоб розслабитися, щодня ви повинні стояти на цьому проклятому Scrum - мітингу і розповідати усім, чого ви добилися учора. Мало хто може похвалитися: «Я витратив велику частину дня, поклавши ноги на стіл, переглядаючи блоги і попиваючи каву».

Окрім власне відпочинку, є ще одна причина для паузи між спринтами. Потім демо і ретроспективи, як команда, так і product owner будуть переповнені інформацією і всілякими ідеями, які їм слід було б обмізкувати. Якщо ж вони негайно займуться плануванням наступного спринту, тобто ризик, що вони не зможуть упорядкувати усю отриману інформацію або зробити належні висновки. До того ж у Product owner 'а не вистачить часу для коригування його пріоритетів після проведеного демо і т. д.

Погано:

Перед початком нового спринту (якщо бути точним, після ретроспективи спринту і перед плануванням наступного спринту) ми намагаємося додавати невеликий проміжок вільного часу. На жаль, у нас це не завжди виходить.

Як мінімум, ми намагаємося добитися того, щоб ретроспектива спринту і наступне планування спринту не проходили в один і той же день. Перед початком нового спринту кожен повинен гарненько виспатися, не думаючи при цьому про спринти.

Краще:

Один з варіантів для цього — «інженерні дні» (чи як би ви їх не називали). Це дні, коли розробникам дозволяється робити по суті усе, що вони хочуть. (ОК, я визнаю, в цьому винен Google). Наприклад, читати про останні засоби розробки і API, готуватися до сертифікації, обговорювати комп'ютерні новини з колегами, займатися своїм особистим проектом.

Краще нікуди?

Наша мета — інженерний день між кожним спринтом. Так між спринтами з'являється реальна можливість відпочити, а команда розробки отримує хороший шанс підтримувати актуальність своїх знань. До того ж, ця досить вагома перевага роботи в компанії.

Зараз у нас один інженерний день між спринтами. Якщо конкретно, то це перша п'ятниця кожного місяця. Чому ж не між спринтами? Ну, тому що я вважав важливим, щоб уся компанія брала інженерний день в один і той же час. Інакше люди не сприймають його серйозно. І оскільки ми (поки що) не погоджували спринти між усіма продуктами, я був вимушений вибрати інженерний день, незалежний від спринтів.

Коли-небудь ми можемо спробувати погоджувати спринти між продуктами (тобто одна і та ж дата для початку спринту і одночасне закінчення спринтів для усіх продуктів і команд). В цьому випадку, ми точно помістимо інженерний день між спринтами.

3.11 Як планувати релізи?

Іноді треба планувати далі, ніж на один спринт вперед. Це типова ситуація для контрактів з фіксованою вартістю, коли нам доводиться планувати наперед, або ж є ризик підписатися під нереальною датою постачання.

Як правило, планування релізу для нас — це спроба відповісти на питання :

«коли, в самому гіршому випадку, ми зможемо поставити версію 1.0».

Спосіб планування простий, але може послужити вам хорошою відправною точкою.

Визначаємо свою приймальну шкалу. У доповненні до звичайного product backlog, product owner визначає приймальну шкалу, яка є ні що інше, як просте розбиття усіх історій product backlog ‘а на групи залежно від їх рівня важливості в контексті контрактних зобов’язань.

Ось приклад діапазонів з нашої приймальної шкали:

• Усі елементи з важливістю >= 100 зобов’язані бути включені у версію 1.0, інакше нас оштрафують за повною програмою.

• Усі елементи з важливість 50-99 повинні бути включені у версію 1.0, але у разі чого ми можемо викотити цю функціональність в наступному додатковому релізі.

• Елементи з важливістю 25-49 потрібні, але можуть бути зроблені в наступному релізі версії 1.1.

• Важливість елементів < 25 дуже спірна, оскільки можливо, що вони взагалі ніколи не згодяться.

Ось приклад product backlog ‘а, розфарбованого відповідно до вищеописаних правил:

Червоні

=

обов’язково

мають бути додані у версію 1.0 (банан — груша)

Жовті

=

бажано

включити у версію 1.0 (родзинки — лук)

Зелені

= можуть бути додані пізніше (грейпфрут — персик)

Отже, якщо до крайнього терміну ми закінчимо усе: від банана до лука, то нам боятися нічого. Якщо час нас підтискатиме, то ми ще

встигнемо викрутитися

, прибравши родзинки, арахіс, пампушку і лук. Усе, що нижче за лук — бонус.

Оцінюємо найважливіщі історії. Щоб спланувати реліз, product owner 'а потрібні оцінки, як мінімум оцінки усіх включених в контракт історій. Як і у разі планування спринту, це — колективна праця команди і Product owner 'а. Команда планує, а product owner пояснює і відповідає на питання.

Оцінка вважається цінною, якщо згодом вона виявилася близькою до реальності. Менш корисною, якщо відхилення склало, скажімо, 30 %. І абсолютно даремною, якщо вона не має нічого спільного з реально згаяним часом.

А ось що я думаю про залежність цінності оцінки від того, хто і як довго її робить.

Резюмуючи вищесказане:

1. Нехай команда проведе оцінку.

2. Не давайте їм витрачати на це багато часу.

3. Переконаєтеся, що команда розуміє, що треба отримати приблизні оцінки, а не контракт, під яким потрібно ставити підпис.

Зазвичай product owner збирає усю команду, робить ввідний огляд і повідомляє, що метою цієї наради є оцінка двадцяти (наприклад) найбільш значущих історій з product backlog 'а. Він проходиться по кожній історії і дозволяє команді приступити до процесу оцінки. Product owner залишається з командою, щоб, у разі потреби, відповідати на питання і прояснити об'єм робіт історій. Так само, як і при

плануванні спринту, колонка «як продемонструвати» — відмінний засіб, щоб уникнути нерозуміння.

Нарада має бути строга обмеженим за часом — команди схильні витрачати дуже багато часу на оцінку всього декількох історій.

Якщо product owner захоче витратити більше часу на оцінку, він просто призначить іншу нараду пізніше. Команда повинна переконатися в тому, що product owner усвідомлює, що проведення подібних нарад відіб'ється на їх поточному спринті. Тобто, що він розуміє, що за усе (і за оцінку у тому числі) треба платити.

Нижче наведений приклад результатів оцінки (у story point 'a) :

Прогнозуємо продуктивність. Добре, тепер у нас є приблизні оцінки для найбільш важливих історій. Наступний крок — прогноз середньої продуктивності команди.

Це означає, що спершу ми повинні визначити наш фокус-фактор.

За великим рахунком, фокус-фактор показує: «наскільки ефективно команда використовує свій час для роботи над вибраними історіями». Це значення ніколи не досягне 100 %, оскільки команда завжди витрачає час на незаплановані завдання, допомогу іншим командам, перемикання між завданнями, перевірку електронної пошти, ремонт своїх поламаних комп'ютерів, політичні дебати на кухні і т. д.

Припустимо, що фокус-фактор нашої команди дорівнює 50 % (це досить низьке значення, у моєї команди значення коливається в районі 70 %). Допустимо також, що довжина нашого спринту буде 3 тижні (15 днів), а розмір команди — 6 чоловік.

Таким чином, кожен спринт — це 90 людино-днів, проте, у кращому разі ми можемо сподіватися тільки на 45 людино-днів (оскільки наш фокус-фактор складає всього 50 %).

Отже, прогнозована продуктивність складе 45 story point 'ів.

Якщо у кожної історії оцінка дорівнюватиме 5 дням (хоча такого і не буває), тоді ця команда зможе видавати приблизно по 9 історій за спринт.

Зводимо усе в план релізу

Зараз, коли у нас є оцінки і прогнозована продуктивність, ми можемо легко розбити product backlog на спринти:

Кожен спринт складається з набору історій, кількість яких не перевищує спрогнозовану продуктивність 45.

Тепер видно, що, швидше за все, нам знадобиться 3 спринти для завершення усієї обов'язкової і бажаної функціональності.

3 спринту = 9 календарних тижнів = 2 календарні місяці. Чи стане це крайнім терміном, який ми озвучимо клієнтові? Це повністю залежить від виду контракту, від того, наскільки фіксований об'єм робіт, і т. д. Зазвичай ми беремо час зі

значним запасом, тим самим захищаючи себе від помилкових оцінок, можливих проблем, необумовленого функціонала і т. д. Значить, в цьому випадку ми встановимо термін постачання в 3 місяці, щоб мати місяць в резерві.

Дуже добре, що ми можемо демонструвати клієнтові що-небудь придатне до використання кожні 3 тижні і дозволяти йому змінювати вимоги протягом всього часу співпраці (звичайно в залежності тому, як виглядає контракт).

Коригуємо план релізу. Реальність не підлаштується під план, тому доводиться його коригувати.

Після закінчення спринту ми дивимося на реальну продуктивність команди. Якщо ця продуктивність істотно відрізняється від прогнозованої, ми змінюємо прогнозовану продуктивність для майбутніх спринтів і оновлюємо план релиза. Якщо це загрожує нам зривом терміну постачання, product owner може почати переговори з клієнтом або почати шукати шлях зменшення об'єму робіт без порушення контракту. Чи, можливо, він і команда зможуть збільшити продуктивність або фокус-фактор шляхом усунення серйозних перешкод, які були виявлені під час спринту.

Product owner може подзвонити клієнтові і сказати: «Привіт, ми злегка не вписуємося в графік, але я вважаю, що ми зможемо укластися в строк, якщо приберемо вбудований функціонал, розробка якого займає багато часу. Ми можемо додати його в наступному релізі, який буде через три тижні після першого релізу».

Нехай це і не краща новина, але, хоч би, ми були чесні і дали можливість клієнтові заздалегідь зробити вибір: або ми поставляємо тільки найважливішу функціональність в строк, або ж усе повністю, але із затримкою. Зазвичай, це не дуже складний вибір.

3.12 Як поєднати Scrum з XP

Те, що Scrum і XP (eXtreme Programming) можуть бути ефективно об'єднані, не викликає сумнівів. Більшість міркувань в інтернеті підтримують це припущення, і я не хочу витрачати час на додаткові обґрунтування.

Проте, одну річ я все-таки повинен згадати. Scrum вирішує питання управління і організації, тоді як XP спеціалізується на інженерних практиках. Ось чому ці дві технології добре працюють разом, доповнюючи один одного.

Тим самим я приєднуюся до прибічників думки, що Scrum і XP можуть бути ефективно об'єднані!

Парне програмування. Ось декілька висновків після застосування парного програмування :

• Парне програмування дійсно покращує якість коду.

• Парне програмування дійсно збільшує зосередженість команди (наприклад, коли напарник говорить: «А ця штуковина точно потрібна для цього спринту»?)

• Дивно, але багато розробників, які виступають проти парного програмування, насправді не практикували його, проте раз спробувавши — швидко розуміють усі переваги.

• Парне програмування вимотує, так що не варто займатися ним цілий день.

• Часта зміна пар дає добрий результат.

• Парне програмування дійсно сприяє поширенню знань усередині команди, помітно прискорюючи цей процес.

• Деякі люди почувають себе некомфортно, працюючи в парах. Не варто позбавлятися від хорошого програміста, тільки тому, що йому не подобається парне програмування.

• Ревю коду — хороша альтернатива парному програмуванню.

• У «штурмана» (людини, яка не пише код) має також бути свій комп'ютер, але не для розробки, а для виконання дрібних завдань, коли це необхідно — перегляду документації, якщо «водій» (людина, яка пише код) затнувся і так далі.

• Не нав'язуйте парне програмування людям. Надихнете їх, дайте необхідні інструменти і дозвольте самим дійти до цього.

3.13 Розробка через тестування (Test Driven Development)

Розробка через тестування важливіше, ніж Scrum і XP разом узяті.

Розробка через тестування означає, що ви спочатку повинні написати автоматизований тест (який не проходить — прим. перекладача). Після цього потрібно написати рівно стільки коду, щоб тест пройшов. Потім необхідно провести рефакторинг, в основному, щоб поліпшити читабельність коду і усунути дублювання. При необхідності повторити.

Декілька фактів про TDD:

- Розробка через тестування — це непросто. На ділі виявляється, що демонструвати TDD програмістові практично марно — часто єдиний дієвий спосіб заразити його TDD полягає в наступному. Програміста потрібно зобов'язати працювати в парі з кимось, хто в TDD хороший. Але як тільки програміст вникнув у TDD, то він вже заражений серйозно і про розробку іншим способом навіть чути не хоче.
- TDD чинить глибокий позитивний вплив на дизайн системи.
- Щоб TDD стало приносити користь в новому проекті, необхідно докласти немало зусиль. Особливо багато витрачається на інтеграційні тести методом «чорного ящика». Але ці зусилля окупаються дуже швидко.
- Потратіть досить часу, але зроби так, щоб писати тести було просто. Тобто потрібно отримати необхідний інструментарій, навчити персо-

нал, забезпечити створення правильних допоміжних і базових класів і т. д.

Можна використати наступні інструменти для розробки через тестування:

- jUnit / httpUnit / jWebUnit. Ми придивляємося до TestNG і Selenium.
- HSQLDB в якості вбудованої БД в пам'яті (in - memory) для тестових цілей.
- Jetty в якості вбудованого web -контейнера в пам'яті (in - memory) для тестових цілей.
- Cobertura для визначення міри покриття коду тестами.
- Spring framework для написання різних типів тестових фикстур (у т. ч. з використанням моков (mock - object) і без, із зовнішньою БД і БД в пам'яті (in - memory) і т. д.)

У наших складних продуктах (з точки зору TDD) у нас реалізовані автоматизовані приймальні тести методом «чорного ящика». Ці тести завантажують усю систему в пам'ять, включаючи бази даних і web -сервера, і взаємодіють з системою тільки через зовнішні інтерфейси (наприклад, через HTTP).

Такий підхід дозволяє отримати швидкі цикли «розробка-зборка-тест». Він так само виступає страховкою, надаючи розробникам упевненість в успішності частого рефакторингу коду. У свою чергу, це забезпечує простоту і елегантність дизайну навіть у разі розростання системи.

TDD і новий код. Ми використовуємо TDD для усіх нових проектів, навіть якщо це означає, що фаза розгортання робочого оточення проекту зажадає більше часу (тому що потрібно більше зусиль на налаштування і підтримку тестових утиліт). Неважко зрозуміти, що вигода переважить будь-який привід не впроваджувати TDD.

TDD і існуючий код. Ми вже говорили, що TDD — це непросто, але що дійсно складно, так це намагатися застосовувати TDD для коду, який спочатку не був спроектований для застосування цього підходу! Цю тему можна довго обговорювати, але я, мабуть, зупинюся.

Якщо від ручного регресійного тестування відмовитися не можна, але дуже хочеться його автоматизувати — краще робити це не потрібно. Замість цього зробіть усе для полегшення процесу ручного тестування.

А вже після цього можна подумати і про автоматизацію.

Еволюційний дизайн. Це означає почати з простого дизайну і постійно покращувати його, а не намагатися зробити усе ідеально з першого разу і більше ні за що і ніколи не чіпати.

З цим ми справляємося досить добре. Ми приділяємо досить часу рефакторингу і поліпшенню існуючого дизайну, і дуже рідко займаємося детальним проє-

ктуванням на роки вперед. Іноді ми, звичайно, можемо напортачить, внаслідок чого поганий дизайн настільки вростає в систему, що рефакторинг перетворюється на дійсно велике завдання. Але, в цілому, ми повністю задоволені цим підходом.

Якщо практикувати TDD, то, за великим рахунком, постійне поліпшення дизайну виходить само собою.

Безперервна інтеграція (Continuous integration). Щоб впровадити безперервну інтеграцію нам довелося для більшості наших продуктів створити досить складне рішення, побудоване на Maven і QuickBuild 'е. Це архиполезно і економить масу часу. До того ж це дозволило нам раз і назавжди позбавиться від класичної фрази: «але у мене ж це працює»!. Наш сервер безперервної інтеграції є «суддею» або еталоном, по якому визначається працездатність усього початкового коду. Кожного разу, коли хтось зберігає свої зміни в системі контролю версій, сервер безперервної інтеграції починає збирати наново усі доступні йому проекти і проганяє усі тести на сервері. Якщо хоч щось піде не так, то сервер обов'язково розішле усім учасникам команди повідомлення. Такі електронні листи містять в собі інформацію про те, які саме зміни поламали зборку, посилання на звіти по тестах і т. д.

Щоночі сервер безперервної інтеграції збирає заново кожен проект наново і публікує на наш внутрішній портал останні версії бинарников (EAR, WAR і т. д. [5]), документацію, звітів по тестах, по покриттю тестами, по залежностях між модулями і бібліотеками і ще багато чого корисного. Деякі проекти також автоматично встановлюються на тестових серверах.

Щоб це усе запрацювало, довелося витратити силу-силенну часу, але, повірте мені, це того коштувало.

Спільне володіння кодом (Collective code ownership). Ми виступаємо за спільне володіння кодом, хоча ще не усі наші команди впровадили у себе цю практику. На власному досвіді ми переконалися, що парне програмування і постійна зміна пар автоматично збільшують рівень спільного володіння кодом. А команди, у яких спільне володіння кодом на високому рівні, довели свою найвищу надійність. Приміром, вони ніколи не провалюють спринти через те, що у них хтось захворів.

Інформативний робочий простір. У усіх команд є доступ до дощок і незайнятих стін, якими вони дійсно користуються. У багатьох кімнатах стіни завішані всякого роду інформацією про продукт і проект. Найбільша проблема у нас — старизна, що постійно накопичується, на стінах. Вже подумуємо завести роль хатньої «робітниці» в кожній команді.

Хоча ми і заохочуємо використання дошки завдань, ще не усі команди впровадили їх (див. стор. 43 «Як ми улаштували кімнату команди»)

Стандарти кодування. Нещодавно ми почали визначати стандарти кодування. Дуже корисно — шкода не зробили цього раніше. Це не займе багато часу, почни з простого і поступово доповнюй. Записуй тільки те, що може бути зрозуміле не усім, при цьому, по можливості, не забути послатися на існуючі матеріали.

У більшості програмістів є свій індивідуальний стиль кодування. Дрібні деталі: як вони обробляють виключення, як коментують код, в яких випадках повертають null і так далі. У одних випадках ці відмінності не грають особливої ролі, в інших можуть привести до серйозної невідповідності дизайну системи і важко читаного коду. Стандарти кодування — ідеальне рішення цієї проблеми, якщо вони, звичайно, регламентують важливі моменти.

Ось невеликий витяг з наших стандартів кодування :

- Ви можете порушити будь-яке з цих правил, але на те має бути вагома причина і це повинно бути задокументовано.
- За умовчанням використовуйте стандарти кодування Sun : http://iava.sun.com/docs/codeconv/html/CodeConvTOC.doc.html
- Ні за яких обставин не перехоплюйте виключення без збереження повного стека викликів програми (stack trace), або повторної генерації виключення (rethrow). Допустиме використання log.debug(), тільки не втратьте стек викликів.
- Для усунення тісного зв'язування між класами застосовуйте впровадження залежностей на основі сетерів(зрозуміло, за винятком випадків, коли таке зв'язування просто потрібне).
- Уникайте абревіатур. Загальновідомі абревіатури, такі як DAO, допустимі.
- Методи, які повертають колекції або масиви, не повинні повертати null. Повертайте порожні колекції і масиви замість null.

Стійкий темп / енергійна робота. Безліч книг з Agile - разробки програмного забезпечення стверджують, що тривала переробка веде до падіння продуктивності.

Близько року тому одна з наших команд (найбільша) дуже-дуже багато працювала наднормово. Якість коду була жахлива і велику частину часу команда «Гасила пожежі». У тестувальників (які теж працювали наднормово) не було нінайменшої можливості нормально протестувати систему. Через декілька місяців ми змогли зменшити кількість переробки до прийнятного рівня. Люди перестали працювати наднормово (окрім рідкісних криз в проекті), і — ось сюрприз — і продуктивність, і якість коду помітно покращали.

Звичайно, зменшення кількості робочих годин не було єдиною причиною підвищення продуктивності і поліпшення коду. Але ми упевнені, що це зіграло істотну роль.

3.14 Як ми тестуємо?

Це найскладніша частина чи то Scrum 'у, чи то розробки програмного забезпечення в цілому.

Організація тестування може досить сильно відрізнятися в різних компаніях. Усе залежить від кількості тестувальників, рівня автоматизації тестування, типу системи (просто сервер + інтернет додаток або, можливо, ви випускаєте «коробкові» версії програм?), частоти релизов, критичності ПО (блог-сервер або система управління польотами) і т. д.

Ми досить багато експериментували, щоб зрозуміти, як організувати процес тестування в Scrum 'е. Зараз я спробую розповісти про те, що ми робили і чому ми встигли навчитися за цей час.

Швидше за все, вам не уникнути фази приймального тестування

У ідеальному світі Scrum 'а результатом кожного спринту має бути система, потенційно готова до використання. Бери і встановлюй, так?

А ось і ні!

По нашому досвіду, такий підхід зазвичай не працює. Там буде купа осоружних багов. Якщо «якість» для вас хоч що-небудь означає, тоді доведеться потурбуватися про ручне приймальне тестування. Це коли спеціально виділені тестувальники, які

не являються

частиною команди, бомблять систему тими видами тестів, про яких Scrum -команда не могла навіть і подумати, або на які у неї не було часу, або відповідного устаткування. Тестувальники працюють з системою точно так, як і користувачі, а значить, що це треба робити вручну (якщо, звичайно ж, ваша система розроблена для людей).

Команда тестувальників знайде баги, Scrum -команде доведеться підготувати версію з виправленими багами, і рано чи пізно (сподіватимемося що рано) ви зможете випустити для своїх користувачів версію 1.0.1 з необхідними виправленнями. Вона буде куди стабільнішою, ніж глючная 1.0.0.

Коли я говорю «фаза приймального тестування», я маю на увазі увесь період тестування, відладки і перевыпуска, поки версія не буде досить хороша для виходу у світ готового продукту.

Мінімізуйте фазу приймального тестування

Приймальне тестування, м'яко кажучи, приносить деякі незручності. Воно ясно не має нічого спільного з гнучкими методологіями. І, не дивлячись на те, що

ми не можемо позбавитися від цієї фази, ми можемо спробувати звести її до мінімуму. А якщо точніше, зменшити

кількість часу

, яке для нього потрібне. Цього можна досягти наступними способами:

• Максимально поліпшити якість початкового коду, створюваного Scrum -командой.

• Максимально збільшити ефективність ручного тестування (т. е. знайти кращих тестувальників, забезпечити їх кращими інструментарієм і переконатися, що вони повідомляють про ті завдання, які віднімають багато часу і можуть бути автоматизовані).

Оскільки ж ми можемо підняти якість коду команди? Ну, взагалі-то способів існує дуже багато. Я зупинюся на тих, які, як нам здалося, діють краще всього :

• Включите тестувальників в Scrum -команду.

• Робіть менше за спринт.

Підвищуйте якість, включивши тестувальників в Scrum -команду

О, я вже чую ці заперечення:

«Але це ж очевидно! Scrum -команды мають бути

кросс-функциональными«!

«У Scrum -команде не має бути виділених ролей! У нас не може бути людини, що займається

тільки

тестуванням«!

Я б хотів дещо роз'яснити. В даному випадку, під «тестувальником» я маю на увазі людину, головна спеціалізація якої тестування, а не людину, чия роль — це винятково тестування.

Розробники досить часто бувають огидними тестувальниками.

Особливо

, коли вони тестують свій власний код.

Тестувальник — це «остання інстанція»

Крім того, що тестувальник — звичайний член команди, він ще і виконує дуже важливу функцію. Він — людина, за якою завжди «останнє слово». Ніщо не може вважатися готовим в спринті, поки

він

не підтвердить, що це дійсно так. Я виявив, що розробники часто говорять, що щось готове, хоча насправді це не так. Навіть, якщо у вас є дуже чітке визначення критерію готовності (яке у вас має бути, див. стор. 29 «Критерій готовності»), у більшості випадків, розробники будуть часто його забувати. Ми, програміс-

ти, люди дуже нетерплячі і прагнемо перейти до наступного пункту списку якнайшвидше.

Оскільки ж містер Т (наш тестувальник) дізнається, що щось дійсно готове? Ну, передусім, він може це (сюрприз!)

протестувати!

У більшості випадків виявляється, що те, що розробник вважає готовим, насправді навіть

неможливо протестувати!

Або унаслідок того, що воно не було зафіксоване в репозиторії, або не встановлено на сервер, або не може бути запущено, або ще що-небудь. Як тільки містер Т завершив тестування, насамперед він повинен пройтися за критеріями готовності (якщо такі у вас є) і, бажано, разом з розробником. Приміром, якщо критерій готовності вимагає наявності заміток до релизу, то містер Т перевіряє їх наявність. Якщо ж потрібна формальніша специфікація функціонала (хоча це буває рідко), містер Т перевіряє і це теж. І так далі.

Приємний додатковий ефект цього підходу полягає в тому, що у команди з'являється людина, яка чудово підходить на роль організатора демонстрації результатів спринту.

Чим займається тестувальник, коли нічого тестувати?

Це питання ніколи не втрачає своєї актуальності. Містер Т : «Scrum-мастер, зараз нічого тестувати. Чим

я

можу зайнятися«?. Може пройти тиждень, поки команда закінчить першу історію, так чим же займатиметься тестувальник

усе це

час?

Спершу, йому слід зайнятися

підготовкою до тестування

. А саме: написанням специфікацій тестів, підготовкою тестового оточення і так далі. Таким чином, коли у розробника з'явиться що-небудь готове до тестування, містер Т має бути готовий почати тестування.

Якщо команда практикує TDD, люди з першого дня зайняті написанням тестуючого коду. В цьому випадку, тестувальник може зайнятися парним програмуванням з розробниками, що пишуть тестуючий код. Якщо ж тестувальник взагалі не уміє програмувати, йому слід працювати в парі з розробником в ролі «штурмана», давши напарникові можливість друкувати. У хорошого тестувальника зазвичай виходить вигадати більше різних тестів, чим у хорошого розробника, тому вони один одного доповнюють. Якщо ж команда не займається TDD або ж кількості що підлягають написанню тестів недостатньо, щоб повністю завантажити тестува-

льника, він просто може робити усе що завгодно, щоб допомогти команді досягти мети спринту. Як і будь-який інший член команди. Якщо тестувальник уміє програмувати — відмінно. Якщо ні, команді доведеться виявити усі завдання, що не вимагають навичок програмування, але які необхідно виконати за спринт.

Коли на плануванні спринту історії розбиваються на завдання, то в першу чергу команда намагається сфокусуватися на завданнях, що вимагають

програмування

. Проте, як правило, існує безліч завдань, які

не вимагають програмування

, але, проте, підлягають виконанню по ходу спринту. Якщо ви впродовж планування спринту витратите трохи часу на

визначення завдань, які не вимагають програмування

, те містер Т отримає можливість зробити для досягнення цілі спринту дуже багато. Навіть якщо він не уміє програмувати або у цей момент нічого тестувати.

Ось деякі приклади завдань, які не вимагають програмування, але які часто мають бути закінчені до кінця спринту:

• Встановити і настроїти тестове оточення.

• Уточнити вимоги.

• Детально обговорити процес установки.

• Написати документи по установці (замітки до релизу, readme.txt або що там потрібно у вашій компанії).

• Поспілкуватися з підрядниками (наприклад, з дизайнерами призначеного для користувача інтерфейсу).

• Поліпшити скрипти автоматизованої збірки.

• Наступне розбиття історій на завдання.

• Зібрати ключові питання від розробників і простежити, щоб вони не залишилися без відповідей.

З іншого боку, що ми робимо у разі, коли містер Т виявляється «вузьким місцем»? Скажімо, зараз останній день спринту і велика частина функціонала вже реалізована, а містер Т не має можливості протестувати усе необхідне. Що ми робимо в цьому випадку? Ну, ми можемо зробити усіх членів команди помічниками містера Т. Він вирішує, що він може зробити самостійно, а прості завдання доручає іншим членам команди. Ось, що таке кросс-функціональна команда!

Отже, містер Т дійсно грає особливу роль в команді, але він, крім того, може виконувати роботу інших членів команди, так само, як і інші члени команди можуть робити його роботу.

Підвищуйте якість — робіть менше за спринт!

Це вирішується ще на плануванні спринту. Простіше кажучи, не намагайтеся зробити якомога більше історій за спринт. Якщо у вас існують проблеми з якіс-

тю або вам доводиться витрачати надто багато часу на приймальне тестування, просто робіть менше за спринт! Це автоматично приведе до підвищення якості, зменшить тривалість приймального тестування і кількість багов, які вилізуть у кінцевого користувача. Врешті-решт, це повинно підняти продуктивність усієї команди, адже вона зможе сконцентруватися на нових завданнях, замість того, щоб постійно витрачати час на виправлення старого коду, який постійно ламається.

Майже завжди виходить дешевше зробити менше, але якісніше, чим більше, але потім в паніці латати діри.

Чи варто робити приймальне тестування частиною спринту?

Тут у нас повний «розбрід і хитання». Деякі наші команди включають приймальне тестування в спринт, проте, більшість — ні. І ось чому:

Спринт обмежений в часі. Приймальне тестування (яке, якщо брати до уваги моє визначення, включає відладку і повторний випуск продукту), досить складно втиснути в чіткі часові рамки. Що якщо час вже вийшов, а в системі залишився критичний баг? Що тоді? Збираєтеся чекати завершення ще одного спринту? Чи випустите готовий продукт з цим багом? У більшості випадків обоє варіант неприйнятні. Саме тому ми виносимо ручне приймальне тестування за межі спринту.

Якщо дві Scrum -команды працюють над одним продуктом, тоді ручне приймальне тестування необхідно проводити, зібравши результати роботи обох команд. Якщо обидві команди включають в спринт ручне приймальне тестування, тоді все одно потрібна команда, якій доведеться протестувати фінальний реліз, який виходить після інтеграції результатів роботи обох команд.

Це далеко не ідеальне рішення, але у більшості випадків воно нас влаштовує.

Співвідношення спринтів і фаз приймального тестування

У ідеальному Scrum -світі фаза приймального тестування не потрібна, оскільки кожна Scrum -команда після кожного спринту видає нову, готову до реального використання версію системи

Насправді, усе виглядає трохи по-іншому:

Після першого спринту випускається глючная версія 1.0.0. Під час другого спринту починають поступати повідомлення про помилки, і команда велику частину часу займається відладкою, а потім випускає версію з виправленнями 1.0.1 в середині спринту. Потім, у кінці другого спринту виходить версія 1.1.0 з новим функціоналом, яка, природно, виявляється ще більше глючною, оскільки у команди просто не вистачило часу довести її до розуму через те, що доводилося підчищати хвости, що залишилися з минулого спринту. І так по колу.

Червоне штрихування похилої другого спринту символізує хаос.

Неприваблива картина, так? А найсумніше в тому, що ця проблема залишається навіть за наявності команди приймального тестування. Єдина різниця полягає в тому, що основна маса повідомлень про помилки поступає від команди тестування, а не від обурених користувачів. Але ця різниця просто величезна з точки зору бізнесу, хоча для розробників нічого і не міняється. Ну, окрім тільки того, що тестувальники зазвичай менш агресивні, чим кінцеві користувачі.

Простого рішення цієї проблеми ми так і не знайшли. Але наекспериментировались з різними підходами вдосталь.

Насамперед, знову ж таки, необхідно забезпечити найвищу якість коду, який створює Scrum -команда. Вартість раннього виявлення і виправлення помилки (в межах спринту) незрівняно нижче за вартість виявлення і виправлення помилки після закінчення спринту.

Але факт залишається фактом: як би ми не зменшували кількість помилок, вони обов'язково знайдуться і після завершення спринту. Так що ж з цим робити?

Підхід № 1: «Не починати нові історії, поки старі не будуть готові до реального використання»

Звучить класно, чи не так? Вас теж гріє ця думка?:)

Кілька разів ми вже було наважувалися на цей підхід, і навіть малювали теоретичні моделі того, як би могли це зробити. Але кожного разу ми зупинялися, коли розглядали зворотний бік медалі. Нам би довелося ввести необмежену за часом ітерацію між спринтами, в яку б ми тільки тестували і виправляли помилки до тих пір, поки у нас на руках не було б готового до використання релиза.

Наявність необмеженої в часі ітерації між спринтами нам не подобалася в основному через те, що вона б порушила регулярність спринтів. Ми не змогли б більше заявляти: «Кожні три тижні ми починаємо новий спринт». А крім того вона не вирішує проблему. Навіть якщо ми введемо таку ітерацію, все одно час від часу з'являтимуться помилки, що терміново вимагають уваги, і нам потрібно бути до цього готовими.

Підхід № 2: «Починати реалізовувати нові історії, але найвищим пріоритетом ставити доведення старих до розуму»

Ми віддаємо перевагу цьому підходу. Принаймні, досі так і було.

По суті, він полягає в наступному: коли ми закінчуємо спринт, ми переходимо до наступного, але враховуємо, що в наступному спринті нам знадобиться час на виправлення помилок минулого спринту. Якщо наступний спринт виявляється переобтяжений роботою над виправленням дефектів минулого, то ми намагаємося зрозуміти причину такої кількості дефектів і виробити спосіб підняти якість. І ми вибираємо довжину спринту достатньою, щоб встигнути впоратися з пристойним об'ємом роботи по виправленню помилок минулого спринту.

Поступово, за декілька місяців, кількість роботи по усуненню дефектів минулих спринтів зменшилася. Крім того, ми змогли добитися, щоб на усунення дефекту вимагалося відволікати менше людей, тобто, немає нужди турбувати усю команду з приводу кожної помилки. Тепер хаос в наших спринтах знизився до прийнятного рівня.

При плануванні спринту, щоб врахувати той час, який ми плануємо витратити на усунення дефектів, ми встановлюємо зменшене значення фокус-фактору. З часом команди починають дуже добре визначати потрібне значення фокус-фактору. У цьому також дуже допомагає статистика реальної продуктивності.

Не забувайте про обмеження системи

Припустимо приймальне тестування — це ваше найвужче місце. Наприклад, у вас занадто мало тестувальники або фаза приймального тестування забирають багато часу через жахливу якість коду.

Допустимо, команда, яка виконує приймальне тестування, устигає перевірити 3 задачі в тиждень. Також допустимо, що ваші розробники можуть зробити 6 нових задач в тиждень.

Для менеджера або Product owner 'а (навіть для самої команди) буде спокусою запланувати розробку 6-ти нових задач в тиждень.

Краще при плануванні розраховувати на 3 задачі в тиждень, а час, що залишився, направити на усунення вузького місця.

Наприклад:

• Змусити розробників потестировать (приготуйтеся до «вдячності» за це.).

• Впровадити нові інструменти і скрипти, які спростять тестування.

• Додати більше автоматизації.

• Зробити довше спринт і включити приймальне тестування в спринт.

• Виділити декілька «тестових спринтів», де уся команда працюватиме над приймальним тестуванням.

• Найняти більше тестувальників (навіть якщо це означає звільнити деяких розробників).

Ми пробували усі ці рішення (окрім останнього). З точки зору довгострокової перспективи кращими є пункти 2 і 3, а саме: ефективніші інструменти, скрипти і автоматизація тестування.

А для виявлення вузьких місць краще всього підходять ретроспективи.

Повертаючись до реальності

У вас, напевно, склалося враження, що у нас є тестери в усіх Scrum -командах, і що ми обзавелися ще однією величезною командою тестерів, які після кожного спринту проводять приймальне тестування усіх готових продуктів.

Ну . це не так зовсім.

У нас

всього кілька разів

вийшло виділити час на усі ці процедури, але тоді ми на власному досвіді переконалися наскільки це корисно. Можу сказати, що на даний момент ми все ще далекі від бажаного процесу забезпечення якості, і нам як і раніше є чому вчитися.

Як ми управляємо декількома Scrum -командами

Коли у вас над одним продуктом працюють відразу декілька Scrum -команд, усе набагато складніше. Це загальна проблема і вона характерна не лише для Scrum 'а. Чим більше розробників, тим більше проблем.

Ми і з цим експериментували (як завжди). Максимальний розмір команди, що працювала у нас над одним проектом, був порядку 40-ка людина.

Як виявилось, ключовими є наступні питання:

• Скільки сформувати команд?

• Як розподілити людей по командах?

Скільки сформувати команд

Якщо настільки складно працювати по Scrum 'у з декількома командами, то навіщо взагалі морочитися? Чому просто не зібрати усіх в одну команду?

Найбільша Scrum -команда, яка у нас коли-небудь була, складалася з 11-ти чоловік. І це працювало, правда, не дуже добре. Щоденний Scrum завжди тривав більше 15-ти хвилин. Членам команди було складно тримати в голові інформацію про те, чим кожен з них займається, через це могли виникати непорозуміння. ScrumMaster 'у теж було складно направити роботу в потрібне русло, і було складно знайти час, щоб розібратися з усіма виниклими труднощами.

Як варіант, можна виділити дві команди. Але чи буде це краще? Не факт.

Якщо команда досвідчена, їй комфортно працювати по Scrum 'у, і ви знаєте, як правильно розділити роботу на два окремі напрями, що дозволить уникнути одночасної роботи над одними і тими ж первинниками, тоді я скажу, що це хороша ідея: розділити на окремі команди. У осоружному ж випадку, не дивлячись на усі мінуси роботи у великій команді, я б подумав про те, як залишити одну велику команду.

Мій досвід показує, що набагато краще мати декілька великих команд, чим багато маленьких, які постійно заважатимуть один одному. Створюйте маленькі команди тільки тоді, коли вони не потребують взаємодії один з одним.

Віртуальні команди

Як же зрозуміти, наскільки правильно були оцінені переваги і недоліки при виборі між «великою командою» або «декількома командами»?

Будьте гранично уважні, і ви помітите формування «віртуальних команд».

Приклад 1: Ви зупинилися на одній великій команді. Але як тільки почнете спостерігати за тим, хто з ким спілкується упродовж спринту, то помітите, що команда фактично розбивається на дві підкоманди.

Приклад 2: Ви вирішили зробити три невеликі команди. Але як тільки почнете прислухатися, хто і з ким говорить в ході спринту, то помітите, що перша і друга команди спілкуються між собою, тоді як третя працює сама по собі.

Що б це означало? Що ваша стратегія розподілу була неправильною? І так і ні (якщо віртуальні команди тимчасові).

Погляньте ще раз на перший приклад. Якщо склад обох віртуальних підкоманд міняється (т. е. люди переходять з однієї віртуальної підкоманди в іншу), тоді можливо ви прийняли правильне рішення, створивши одну велику Scrum -команду. Якщо ж дві віртуальні підкоманди в ході спринту залишаються незмінними, то, можливо, для наступного спринту їх слід розбити на дві справжні незалежні Scrum -команды.

Тепер повернемося до другого прикладу. Якщо перша і друга команди спілкуються один з одним (але не з третьою) упродовж усього спринту, тоді можливо для наступного спринту слід об'єднати першу і другу команду в одну Scrum -команду. Якщо перша і друга команда дуже щільно спілкуються в першій половині спринту, а в другій половині спринту вже перша і третя команди ведуть жваві бесіди один з одним, тоді думаю, вам варто було б розглянути можливість об'єднання усіх трьох команд в одну, або все-таки залишити ці команди як є. Підніміть це питання на ретроспективі і дайте можливість командам самим вирішити його.

Розбиття на команди — це дійсно одне з найскладніших завдань в Scrum 'е. Не намагайтеся копати дуже глибоко або займатися дуже складною оптимізацією. Експериментуйте, спостерігайте за віртуальними командами, і не забувайте приділяти обговоренню цього питання досить часу на ретроспективах. Рано чи пізно ви знайдете для себе правильне рішення. Проте запам'ятаєте, що вам слід зробити усе, щоб команди почували себе комфортно і не заважали один одному занадто часто.

Оптимальний розмір команди

Практично в усіх книгах, які я читав, стверджується, що «оптимальний розмір» команди складає приблизно 5-9 чоловік.

Виходячи з того, що я бачив, залишається тільки погодитися з цією думкою. Хоча, як на мене, все-таки краще мати команду від 3-ех до 8-ми людина. Поднапрягитесь, але створіть команди такого розміру.

Допустимо, у вас є одна Scrum -команда з 10-ти чоловік. Подумайте, може бути варто викинути двох найбільш слабких учасників команди. Ой-йо-йой, я що — правда, сказав це?

Припустимо, ви розробляєте два різні продукти, у вас по три людини в кожній команді, проте обидві команди працюють дуже повільно.

Можливо

, їх слід об'єднати в одну команду з 6-ти чоловік. І нехай ця команда одночасно створює обидва продукти. В цьому випадку одному з двох product owner 'ов доведеться піти (чи почати грати роль консультанта, ну або ще щось на кшталт цього).

Допустимо, у вас одна Scrum -команда з 12 чоловік, яку нереально розбити на дві незалежні команди тільки тому, що початковий код страшно запущений. Вам доведеться прикласти максимум зусиль, щоб отрефакторить (замість того щоб клепати новий функціонал) початковий код до такої міри, щоб над ним могли працювати незалежні команди. Повірте мені, що, швидше за все, ваші «інвестиції» окупляться з лишком.

Синхронізувати спринти або ні?

Припустимо, є три Scrum команди, які працюють над одним проектом. Чи повинні їх спринти бути синхронізованими, т. е. починатися і закінчуватися одночасно? Або ж вони повинні присікатися один з одним?

Спочатку ми вирішили, що потрібні пересічні спринти (за часом).

Це звучало круто. У будь-який момент часу у нас був би спринт, який ось-ось закінчиться, і спринт, який ось-ось почнеться. Навантаження на Product owner 'а була б розподілена рівномірно за часом. Постійні релизы продукту. Демонстрації кожного тижня. Алілуя.

Так-так, утопія. але тоді це дійсно звучало переконливо!

Ми тільки-тільки почали так працювати, але тут мені підвернулася можливість поспілкуватися з Кеном Швабером(у рамках моєї Scrum -сертификации). Він вказав на те, що це неправильно, і що було б набагато краще синхронізувати спринти. Я точно не пам'ятаю його аргументів, але в ході нашої дискусії він мене переконав в цьому.

Відтоді ми використовували це рішення і жодного разу про це не пошкодували. Я ніколи не дізнаюся, провалилася б стратегія пересічних спринтів або ні, але думаю, що так. Переваги синхронізованих спринтів в наступному:

• З'являється природна можливість перетасовувати команди між спринтами! При пересічних спринтах немає можливості реорганізовувати команди так, щоб не потурбувати жодної команди у розпалі спринту.

• Усі команди можуть працювати на одну мету впродовж спринту і проводити планування спринту разом, що призводить до кращої співпраці між командами.

• Менше адміністративної мороки, наприклад менша кількість зустрічей для планування спринту, демонстрацій і релизів.

Чому ми ввели роль «тимліда»

Припустимо, що у нас над одним продуктом працюють три команди.

Червоним помічений product owner. Чорним — Scrum Master. А інші це піхота. ну тобто. поважні члени команди.

Хто при такому розподілі ролей повинен вирішувати, які люди будуть віднесені до якої команди. Може, product owner? Чи може усе три ScrumMaster 'а разом? Чи взагалі кожна людина сама вирішує, в якій команді працювати? Але якщо так, то що робити у разі, коли усі захочуть в одну і ту ж команду (тому що там красива ScrumMaster 'ша)?

А що коли потім виявиться, що над цим кодом більше двох команд працювати не зможуть, і нам доведеться трансформувати три команди по 6 чоловік в дві по 9? Це означає, що буде всього 2 ScrumMaster 'а. Так кого ж з трьох поточних ScrumMaster 'ов потрібно позбавити титулу?

У багатьох компаніях ці питання потрібно вирішувати дуже делікатно.

Є спокуса віддати розподіл людей по командах і їх наступний перерозподіл на відкуп product owner 'у. Але ж це не зовсім те, чим повинен займатися product owner, правда ж? Product owner це фахівець з предметної області, який вказує команді напрям руху. У загальному випадку його не повинне хвилювати усе інше. Особливо, враховуючи його роль «курчати» (це якщо ви чули про метафору «свиней і курчат», а якщо не чули, то погуглите).

Ми розв'язали проблему введенням ролі «тимлид». Людину в цій ролі можна описати як «Scrum - of - Scrums master», «бос» або «головний ScrumMaster». Йому не треба очолювати яку-небудь команду, але він відповідає за питання, що не входять в компетенцію команд, такі як, наприклад, «кого призначити ScrumMaster 'ом», «як розподілити людей по командах» і т. д.

Придумати дійсно відповідні назва для цієї ролі у нас так толком і не вийшло. А «тимлид» виявилося найменш непідходящим, з тих, що ми перепробували.

Такий підхід спрацював в нашому випадку, і я його рекомендую (не залежно від того, як ви вирішите назвати цю роль у себе).

Як ми розподіляємо людей по командах

На випадок, коли у вас декілька команд працюють над одним і тим же продуктом, існує дві стратегії розподілу людей по командах.

• Дозволити спеціально призначеній людині провести розподіл, наприклад «тимлиду», про якого я писав вище, product owner 'а або будь-якому іншому менеджерові (якщо у нього досить інформації, щоб з цим впоратися).

• Дозволити командам якимсь способом самоорганізовуватися.

Ми спробували усі три стратегії. Три?!? Ну так — стратегію № 1, стратегію № 2 і їх комбінацію. І виявилось, що комбінація двох стратегій працює краще всього.

Перед плануванням спринту тимлид запрошує Product owner 'а і усіх ScrumMaster 'ов на нараду з приводу формування команд. Ми обговорюємо минулий спринт і вирішуємо, чи є необхідність в переформатуванні команд. Можливо, нам треба об'єднати дві команди або перевести декілька чоловік з однієї команди в іншу. Ми вирішуємо, що саме нам треба, записуємо це на листочок і несемо на планування спринту як попередній розподіл по командах.

Насамперед на плануванні спринту ми проходимося по найбільш пріоритетних історіях з product backlog 'а. А потім тимлид говорить щось подібне до:

«Усім привіт. От як ми пропонуємо сформувати команди на наступний спринт«.

«Як видно, ми збираємося зменшити кількість команд з чотирьох до трьох. Склади команд вказані. Будь ласка, групуйтеся згідно із списками і виберіть собі відповідну стіну для планування«.

(тимлид чекає доки народ побродить по кімнаті, і через деякий час з'являються три групи людей, кожна збереться біля своєї частини стіни).

«Поточний розподіл — тільки прикидка!

Просто щоб з чого почне. У міру планування спринту будь-якої з вас вільний переходити у будь-яку іншу команду, можна розділяти команди, можна, навпаки, об'єднувати їх. Загалом, робіть усе, що підкаже здоровий глузд, враховуючи пріоритети, які озвучує product owner«.

Описаний вище підхід виявився для нас найбільш ефективним. Трохи директивного управління, яке оптимізується розумною долею самоврядування.

Чи потрібні вузькоспеціалізовані команди?

Припустимо, ваша система складається з трьох основних компонентів:

Клієнт

Сервер

БД

Допустимо, що над вашим продуктом працюють 15 чоловік, і вам не дуже хочеться збирати їх в одну Scrum -команду. Як же розділити людей на команди?

Підхід № 1: команди, що спеціалізуються на компонентах

Можна створити команди, що спеціалізуються на конкретних компонентах. Тоді ми отримаємо «команду для клієнтської частини», «команду для серверної частини» і «команду для бази даних».

Саме з цього підходу ми колись починали. Працює не дуже добре, по крайній у тому випадку, коли більшість історій зачіпають відразу декілька компонентів.

Приміром, візьмемо, історію, яка називається «Дошка оголошень, де користувачі можуть залишати один одному повідомлення». Для створення такої дошки оголошень нам доведеться відновити призначений для користувача інтерфейс в клієнтській частині, додати бізнес-логіку на стороні сервера, і додати парочку таблиць у базу даних.

Це означає, що усім трьом командам доведеться досить щільно співпрацювати, щоб закінчити цю історію. Не дуже зручно.

Підхід № 2: універсальні команди

А можна створити універсальні команди, тобто команди, які не заточені на роботу усього лише з одним специфічним компонентом.

Якщо велика частина усіх історій припускає роботу над декількома компонентами, тоді такий розподіл на команди працює набагато краще. Кожна команда зможе реалізувати історію цілком: клієнтську частину, серверну і базу даних. Завдяки чому команди зможуть працювати набагато більше незалежно один від одного, що насправді ДУЖЕ ДОБРЕ.

Коли ми почали впроваджувати Scrum, насамперед ми зробили з усіх наших спеціалізованих команд (підхід № 1) універсальні команди (підхід № 2). Це зменшило кількість ситуацій «ми не можемо закінчити завдання, оскільки чекаємо, поки ці хлопці закінчать серверну частину».

Проте іноді нам все-таки доводиться збирати тимчасові команди, що спеціалізуються на розробці окремих компонентів.

Чи варто змінювати склад команди між спринтами?

Зазвичай кожен спринт має свої власні особливості залежно від того, якого роду завдання ми намагаємося вирішити. Як наслідок, оптимальний склад команди для кожного спринту може відрізнятися.

Фактично, майже кожен спринт нам доводилося говорити собі щось подібне до: «цей спринт не зовсім звичайний спринт, тому що (ля-ля-ля)». Через деякий час ми припинили використовувати поняття »звичайний« спринт. Звичайних спринтів просто немає. Так само як немає звичайних сімей або звичайних людей.

Для одного спринту може здатися хорошою ідеєю, створити команду, яка займається клієнтською частиною додатка і включає усіх, хто добре знає код клієнта. Для іншого спринту хорошою ідеєю може бути створення двох універсальних команд і розподіл фахівців з клієнтської частини між ними.

Одним з ключових аспектів Scrum 'а являється «спрацьованість команди», т. е. якщо члени команди працюють разом впродовж багатьох спринтів, вони зазвичай стають дуже згуртованими. Вони навчаться входити в груповий потік, і досягнуть неймовірного рівня продуктивності. Але щоб досягти цього потрібно декілька спринтів. Якщо ви часто змінюватимете склад команди, то ви ніколи не досягнете справжньої командної спрацьованості.

Тому, якщо ви вирішили змінити склад команди, враховуйте усі наслідки. Чи будуть це довготривалі або короткочасні зміни? Якщо короткочасні, коштує їх пропустити. На довготривалі зміни можна піти.

Є одно виключення: велика команда, яка тільки-тільки почала працювати по Scrum 'у В цьому випадку можливі деякі експерименти з розподілом команди на підкоманди, поки не буде знайдений варіант, який повністю влаштовував би усіх. Упевніться, що усі розуміють, що негативний результат — теж результат, що перші декілька ітерацій можуть бути грудкою — і це нормально, за умови, що ви працюєте над поліпшеннями.

Учасники команди з частковою зайнятістю

Можу тільки підтвердити те, що говорять книги, присвячені Scrum 'у наявність в Scrum -команде учасників з частковою зайнятістю — не дуже хороша ідея.

Припустимо, ви розглядаєте можливість узяти Джо у свою команду як учасника з частковою зайнятістю. Спочатку добре усе обдумайте. Чи дійсно Джо потрібний вашій команді? Упевнені, що не можете дістати його на повний день? Які у нього ще обов'язки? Чи можна передати обов'язки Джо комусь іншому і перекласти його на роль консультанта? Чи можна дістати Джо на повний день, починаючи з

наступного

спринту, а доки передати його обов'язки комусь іншому [6]?

Але іноді просто немає вибору. Вам до зарізу потрібний Джо тому, що він єдиний адміністратор баз даних (DBA) в усій будівлі. Іншим командам він потрібний так само сильно, як і вам, тому він ніяк не може працювати з повною зайнятістю у вашій команді. Крім того, компанія не може собі дозволити найняти ще одного DBA. Ну і гаразд. Це аргументований випадок, щоб узяти його на неповну зайнятість (що, до речі, було у нас). Але перш, ніж зробити це, завжди проводите подібний аналіз.

У звичайній ситуації я б віддав перевагу команді, що складається з трьох учасників з повною зайнятістю, чим з восьми, але з частковою.

Якщо у вас є людина, що є учасником декількох команд, як, наприклад вищезгаданий DBA, все одно непогано б його закріпити за однією командою. Визначите, яка команда потребує його найбільше, і призначте її в якості «домашньої команди». Коли його ніхто не смикатиме, він буде присутнім на щоденних Scrum 'ах, плануваннях спринтів, ретроспективах і т. д. цієї команди.

Як ми проводимо Scrum - of - Scrums

Scrum — of — scrums — це регулярні зустрічі, мета яких — обговорення різних питань між Scrum -мастерами.

Якось ми працювали над чотирма продуктами. Над трьома з них працювало по одній Scrum -команде, а над четвертим — 25 чоловік, які були розділені на декілька Scrum -команд. Це виглядало таким чином:

У нас були два рівні Scrum - of - Scrums: «рівня продукту», який проводився за участю команд продукту Д, і «рівня компанії» для учасників усіх команд.

Scrum - of - Scrums рівня продукту

Ця зустріч була дуже важливою. Ми проводили її не рідше за один раз в тиждень. Ми обговорювали проблеми інтеграції, балансування команд, підготовку до наступного планування спринту і т. д. Ми виділяли на це 30 хвилин, але часто нам їх бракувало. В якості альтернативи можна було б проводити щоденний Scrum - of - Scrums, проте, ми так і не зібралися випробувати його.

Наш порядок денний мав наступний вигляд:

1. Кожен по черзі розповідав, що його команда зробила минулого тижня, що планує закінчити на цього тижня, і з якими труднощами вони зіткнулися.

2. Будь-які інші проблеми, що відносяться до компетенції декількох команд одночасно, які треба обговорити. Наприклад, питання інтеграції.

Насправді порядок денний Scrum — of — Scrums не так вже і важлива — важливіше, щоб ця зустріч проводилася регулярно.

Scrum - of - Scrums рівня компанії

Ми назвали цю зустріч «Пульсом». Ми пробували проводити її в різних форматах з різними учасниками. Врешті-решт, ми відмовилися від усіх інших ідей на користь щотижневих зборів тривалістю 15 хвилин, в якому бере участь увесь колектив (взагалі-то все ті, хто беруть участь в процесі розробки).

Чегооо? 15 хвилин? Увесь колектив? Усі учасники усіх продуктових команд? І це працює?

Так — працює, якщо ви (чи відповідальний за проведення цих зборів) дуже строгі відносно того, щоб збори були стислими.

Формат зборів :

1. Новини і уточнення з боку керівника розробки. Наприклад, інформація про майбутні заходи, події.

2. «Карусель». Одна людина з кожної продуктової групи [2] звітує в тому, що було зроблено за минулий тиждень, що планується зробити цього тижня і про проблеми. Деякі інші люди так само звітують (наприклад, начальник відділу по роботі з клієнтами, начальник відділу контролю якості).

3. Усі, хто хоче, можуть вільно висловитися і поставити будь-які питання.

Ці збори для подачі стислої інформації, а не для дискусій або рефлексії. Якщо цим і обмежитися, то 15-ти хвилин цілком вистачає. Іноді воно займає більше часу, але дуже рідко більше за 30ти хвилини. Якщо зав'язується цікава дискусія, я її припиняю і пропоную усім зацікавленим залишитися і продовжити її послові зборів.

Чому ми проводимо «Пульс» усім колективом? Тому, що ми помітили, що Scrum - of - Scrums рівня компанії присвячений переважно звітності. На нім дуже рідко виникали дискусії. Крім того, інформація, що озвучується на Scrum, - of - Scrum 'е, дуже цікава і багато чим з тих, хто на нього не потрапляє. Зазвичай командам цікаво, чим займаються інші команди. І ми порахували, якщо все одно треба витрачати час на інформування один одного, то чом би не бути присутнім усім.

Чергування щоденних Scrum 'ів

Якщо у вас є багато Scrum команд, працюючих над одним продуктом, і у усіх щоденний Scrum відбувається в один і той же час, може виникнути проблема. Product Owner 'ы (і понад міру докучливі люди на зразок мене) мають фізичну можливість відвідувати тільки один щоденний Scrum водночас часу. Тому ми просимо команди рознести час проведення щоденних Scrum 'ов.

Цей приклад розкладу відноситься до того періоду, коли у нас був щоденний scrum в різних кімнатах, а не в кімнаті нарад команди. Зазвичай зустрічі плануються на 15 хвилин, але кожна команда отримувала 30 хвилинний часовий слот на випадок, якщо їй потрібно було трохи затриматися.

Це
дуже зручно
з двох причин:

1. Люди, подібні до мене і product owner 'а можуть відвідати
усе
щоденні scrum 'ы за одно ранок. Немає кращого способу отримати уявлення про те, як проходить спринт, і в чому основні проблеми.

2. Команди можуть відвідувати щоденні Scrum'bi один одного. Не дуже часто, але іноді трапляється, що дві групи працюють в суміжних областях, і учасники

груп заглядають один до одного на щоденні Scrum 'ы, щоб бути в курсі того, що відбувається.

Зворотний бік медалі полягає в обмеженнях для команди — втрачається можливість змінити час проведення щоденного Scrum 'а. Хоча, насправді, для нас це не складало проблеми.

«Пожежні» команди

Ми зіткнулися з ситуацією, коли було неможливо впровадити Scrum у великому проекті через те, що його команда постійно гасила пожежі, т. е. у паніці усувала дефекти передчасно випущеної системи. Це був порочний круг: через те, що увесь час йшло на постійну боротьбу з пожежами, не було часу на їх запобігання (т. е. на поліпшення архітектури, впровадження автоматичного тестування, створення систем моніторингу і сповіщення, і т. п.)

Ми розрубали цей гордіїв вузол тим, що виділили спеціальну «пожежну» команду і окрему Scrum -команду.

Завданням Scrum -команды (з благословення Product owner 'а) була стабілізація системи і запобігання потенційним пожежам. А «пожежна» команда (її ми назвали «командою підтримки») виконувала два завдання:

1. Гасити пожежі.

2. Прикривати Scrum -команду від всяких подразників, на зразок несподіваних запитів на зміну функціонала, які незрозуміло звідки беруться.

«Пожежну» команду ми розмістили ближче до дверей, а Scrum -команду — чимдалі в кімнаті. Таким чином, «пожежна» команда могла навіть фізично захищати Scrum -команду від подразників типу прагнучих спілкування «продажників» або розсерджених клієнтів.

У кожну команду були включені старші розробники, щоб команди не занадто залежали один від одного.

В основному цей підхід був вирішенням проблеми впровадження Scrum 'а. Як взагалі можна почати практикувати Scrum, якщо команда не може спланувати свою роботу більше, ніж на один день вперед? Нашою відповіддю на це, як сказано вище, був розподіл команд.

Спрацювало це вистачає добре. Оскільки Scrum -команде дали можливість планувати свою роботу, вона, врешті-решт, змогла стабілізувати систему. А тим часом пожежна команда повністю відмовилася від спроб що-небудь планувати і працювала повністю по запитах, тобто тільки усувала черговий жахливий дефект, що проявився.

Природно повністю залишити Scrum -команду в спокої не вийшло. Частенько пожежна команда вимушена була залучати до вирішення питань окремих людей з Scrum -команды, або, у гіршому разі, усю команду.

У будь-якому випадку через пару місяців система стала настільки стабільною, щоб ми могли відмовитися від пожежної команди на користь додаткових Scrum -команд. «Пожежники» були просто щасливі здати свої шоломи і приєднатися до звичайним Scrum -командам.

Розбивати product backlog чи ні?

Припустимо, у вас є один продукт і дві Scrum -команды. Скільки вам потрібне product backlog 'ов? Скільки Product owner 'а? Вибір досить сильно вплине на те, як проходитимуть зустрічі по плануванню спринту. Ми оцінили три можливі підходи.

Підхід перший : Один product owner — один backlog

«Повинен залишитися тільки один«. Наш улюблений підхід.

Перевага цього підходу в тому, що можна дозволити команді самій планувати роботу на основі пріоритетів, розставлених product owner 'ом. Product owner може зосередитися на тому

що йому потрібне

, і надати командам самим, розбивати історії на завдання.

Ближче до справи. Давайте подивимося, як проходить зустріч по плануванню спринту для цієї команди. Зустріч по плануванню спринту проходить на нейтральній території.

Прямо перед зустріччю product owner називає одну із стін «стіною product backlog» і розвішує на ній історії (облікові картки), відсортовані по пріоритету. Він продовжує вішати їх, поки не займе усю стіну. Як правило, такої кількості історій більш ніж достатньо для спринту.

Кожна Scrum -команда вибирає порожню ділянку стіни і вішає там свою назву. Це буде їх «командна стіна». Після цього кожна команда відклеює історії з «стіни product backlog», починаючи з найважливіших, і переклеює їх на свою «командну стіну».

На малюнку нижче показана описана ситуація. Жовті стрілки зображують рух облікових карток з «стіни product backlog» на стіни команд.

По ходу зустрічі product owner і команди торгуються за облікові картки, рухають їх між командами, пересувають картки вгору-вниз, міняючи пріоритети, розбивають їх на частини і т. п. Десь за годину кожна з команд отримує попередню версію sprint backlog 'а на своїй «командній стіні». Далі за команду працюють незалежно, оцінюючи історії і розбиваючи їх на підзадачі.

Так, хоч цей дурдом і забирає масу сил, та зате це ефективно, прикольний і сприяє спілкуванню. Коли час закінчується, у усіх команд, як правило, досить інформації, щоб почати спринт.

Підхід другий : Один product owner — декілька backlog 'ов

У цьому підході product owner веде
декілька
product backlog 'а, по одному на команду. Ми насправді не застосовували цей підхід, але ми робили щось схоже. Це був наш запасний план на випадок, якщо перший підхід не спрацює.

Недоліком цього підходу є те, що тут історії командам роздає product owner, хоча команди, ймовірно, впоралися б з цим краще самі.

Підхід третій : Декілька product owner 'ов — декілька backlog 'ов

Схоже на другий варіант, по окремому product backlog на команду, тільки ще і з окремим
product owner 'ом
на кожну команду.

Ми не пробували так робити, і, швидше за все, пробувати не будемо.

Якщо два product backlog 'а торкаються одного і того ж початкового коду, це швидше за все приведе до серйозного зіткнення інтересів між Product owner 'ами.

Якщо ж два product backlog 'а мають справу з різними первинниками, то це, за великим рахунком, все одно, що розбити продукт на окремі підпродукти, і розробляти їх незалежно. І це означає, що ми повернулися до ситуації «одна команда розробляє один продукт», яка для нас приємна і зрозуміла.

Паралельна робота з кодом

За наявності декількох команд, одночасно працюючих над одним початковим кодом, нам неминуче доведеться мати справу з паралельними гілками коду в системі SCM (software configuration management). Є багато книг і статей, що розповідають, як забезпечити одночасну роботу з кодом для декількох людей, тому я не вдаватимуся до деталей. Навряд чи мені вдасться додати щось нове. Проте я хотів би коротко поділитися напрацюваннями нашої команди :

• Завжди підтримуйте основну гілку проекту в робочому стані. Це, як мінімум, означає, що усе повинно компілюватися, і усі юнит-тесты повинні проходити. Таким чином, ми дістаємо можливість у будь-який момент випустити робочий релиз. Бажано, щоб сервер безперервної інтеграції будував і автоматично встановлював готовий продукт в тестовому оточенні щоночі.

• Позначайте кожного релиз тегом. Всякий раз, коли для приймального тестування або реального використання випускається черговий релиз, відповідна вер-

сія коду має бути помічена тегом. Це дозволить вам при необхідності у будь-який момент створити в цій точці окрему гілку для підтримки випущеного продукту.

• Створюйте нові гілки коду тільки тоді, коли це дійсно необхідно. Хорошим практичним правилом буде створення нової гілки тільки за умови, якщо ви вимушені порушувати стратегію використання поточний гілки. Якщо у вас є сумніви, то не робіть галужень. Чому? Тому що кожне галуження вимагає додаткової роботи і наступної підтримки.

• Використовуйте галуження для розподілу коду різних стадій розробки. Ви можете створити для кожної Scrum команди окрему гілку розробки. Але якщо ви змішаєте короткострокові зміни з довгостроковими змінами в одній і тій же гілці, вам дуже складно буде випускати невеликі латочки!

• Частіше синхронізуйтеся. Якщо ви працюєте в окремій гілці, синхронізуйтеся кожного разу, коли ви хочете що-небудь побудувати. Щодня, коли ви починаєте роботу, синхронізуйтеся з головною гілкою розробки, так щоб ваша гілка була в адекватному стані і враховувала усі зміни, зроблені іншими групами. Навіть якщо це приведе до кошмару злиття (merge - hell), врахуйте, що усе могло б бути ще гірше, якби ви затягнули злиття.

Ретроспектива для декількох команд

Як ми проводимо ретроспективи у разі, якщо над одним продуктом працює відразу декілька команд?

Відразу ж після того, як закінчилася демонстрація спринту і відшуміли бурхливі овації, команди розходяться по своїх кімнатах або вирушають в якесь зручне містечко. І кожна команда проводить свою ретроспективу як це описано на сторінці 50 «Як ми проводимо ретроспективи».

А на плануванні (де є присутніми усі команди, оскільки ми намагаємося синхронізувати спринти усіх команд, які працюють над одним продуктом), ми насамперед вислуховуємо представників кожної команди, на тему найбільш важливих моментів останньої ретроспективи. Виходить приблизно по 5 хвилин на команду. Після цього проводимо вільне обговорення хвилин на 10-20. І, власне, тільки потім починається планування спринту.

Ми не пробували ніяких інших способів, тому що і такий підхід працює досить добре. Проте, найбільший недолік цього підходу, полягає у відсутності можливості трохи передихнути між ретроспективою і початком планування (див. стор. 55 «Відпочинок між спринтами»).

В ході планування з поодинокою командою ми не підводимо підсумки ретроспективи. У цьому немає необхідності, адже кожен член команди був присутнім на ретроспективі.

Як ми управляємо географічно розподіленими командами

Що ж робити, якщо учасники команди знаходяться в географічно рознесених місцях? Велика частина «магії» Scrum 'а і XP заснована на спільній тісній взаємодії учасників команд, які програмують парно і зустрічаються лицем до лиця щодня.

У нас є географічно розподілені команди. Так само деякі учасники працюють удома час від часу.

Наша політика відносно розподілених команд дуже проста. Ми використовуємо усі можливі способи, щоб максимізувати пропускну спроможність засобів зв'язку між фізично розділеними учасниками команди. Під «пропускною спроможністю засобів зв'язку» я розумію не лише Mbit/sec (хоча це теж важливий показник). Я маю на увазі пропускну спроможність засобів зв'язку в ширшому сенсі:

1. Можливість практикувати парне програмування.
2. Можливість зустрічатися лицем до лиця в ході щоденного Scrum 'а.
3. Можливість побачити один одного у будь-який момент.
4. Можливість особистих зустрічей і живого спілкування.
5. Можливість проводити незаплановані наради усією командою.
6. Можливість бачити одні і ті ж версії sprint backlog, sprint burndown, product backlog 'а і інших джерел інформації за проектом.

Ось деякі заходи, зроблені нами (чи що робляться зараз) для досягнення мети :

• Web -камера і навушники з мікрофоном на кожній робочій станції.

• Кімната для проведення телеконференцій, обладнана web -камерами, мікрофонами, комп'ютерами з усім необхідним ПО-для загального доступу до робочого столу, проведення телеконференцій і т. д.

• «Видалені вікна«. Великі монітори в кожному офісі, на яких постійно можна бачити, що відбувається в інших офісах. Щось типу віртуального вікна між двома відділами. Можна стояти перед ним і спостерігати. Наприклад, хто на своєму робочому місці або хто з ким розмовляє. Усе для того, щоб створити відчуття, що усі знаходяться разом.

Використовуючи ці і інша техніка, ми повільно, але упевнено починаємо розуміти, як проводити планування спринтів, демонстрації, щоденні scrum 'ы і ін. в географічно розподілених командах.

Як завжди, головне — не припиняти експериментувати. Обговорення => адаптація => обговорення => адаптація => обговорення => адаптація => обговорення => адаптація => обговорення => адаптація.

Офшорна розробка

У нас є декілька офшорних команд. Ми експериментували з тим, як ефективно ними управляти, використовуючи Scrum.

Існують дві основні стратегії: видалені команди і видалені учасники команд.

Перша стратегія — видалені команди — очевидний вибір. І все-таки, ми почали з використання другої стратегії — видалені учасники команд. На те існує декілька причин.

1. Ми хочемо, щоб учасники команд краще упізнали один одного.

2. Ми хочемо, щоб була налагоджена ефективна комунікація, і щоб команди зрозуміли її важливість.

3. На початковій стадії офшорна команда занадто маленька, щоб самоорганізовуватися в ефективну scrum -команду.

4. Ми хочемо, щоб був період інтенсивного обміну знаннями, перш ніж задіювати незалежні офшорні команди.

У довгостроковій перспективі ми цілком можемо перейти до стратегії «видалені команди».

Члени команди, працюючі будинки

Робота будинку іноді може бути дійсно ефективною. Іноді, за один день будинку можна зробити більше, ніж за увесь тиждень на роботі. Принаймні, якщо у вас немає дітей: о)

Проте, усадити команду разом — одна із засадничих ідей Scrum 'а. І що ж робити в цьому випадку?

Найчастіше ми дозволяємо команді вирішувати, як часто і коли саме її члени можуть працювати удома. Деякі люди регулярно працюють удома, оскільки живуть далеко. І все-таки, ми намагаємося робити так, щоб велику частину часу команда проводила разом.

Коли члени команди працюють удома, вони беруть участь в щоденному Scrum 'е, використовуючи Skype -конференции (іноді відео конференції). Вони доступні в чаті впродовж усього дня. Не так добре, як знаходитися в тій же кімнаті, але цього вистачає.

Якось ми випробували ідею виділення середовища, як спеціального дня для роботи вдома. По суті це означало: «Хочеться попрацювати удома? Немає проблем, але тільки по середовищах. І погоджуйте це з командою». Для команди, на якій ставився експеримент, це спрацювало відмінно. По середовищах велика частина команди зазвичай залишалася удома і виконувала значний об'єм робіт, будучи при цьому на зв'язку один з одним. Один такий день не сильно порушував синхронізацію людей в команді. Але з якихось причин з іншими командами цей підхід не спрацював.

В цілому, люди, працюючі будинки, не стали для нас проблемою.

Пам'ятка ScrumMaster 'а

Наприкінці я познайомлю вас з нашою пам'яткою ScrumMaster 'a. Вона містить найбільш важливі адміністративні завдання, за які відповідає ScrumMaster. Є речі, про які дуже легко забути. Але є і очевидні, такі як «усувати перешкоди на шляхи команди», які ми не включаємо в наш список.

На початку спринту

• Після планування створити «сторінку з інформацією про спринт».

a) На стартовій сторінці wiki -портала помістити посилання на створену сторінку.

b) Роздрукувати цю сторінку і повісити її на стіні, яка у усіх на очах.

• Розіслати e - mail 'ы з повідомленням про початок нового спринту. Не забути вказати мету спринту і дати посилання на «сторінку з інформацією про спринт».

• Відновити статистику спринтів. Додати оцінку попередньої продуктивності, розміру команди, довжини спринту і т. д.

Щодня

• Стежити за тим, щоб щоденний Scrum починався і закінчувався вчасно.

• Стежити за тим, щоб у разі додавання або видалення історії з sprint backlog 'a усе було зроблено, як годиться, щоб ці зміни не зірвали графік робіт.

a) Стежити за тим, щоб product owner знав про ці зміни.

• Стежити за тим, щоб команда постійно оновлювала burndown -диаграмму.

• Стежити за тим, щоб усі проблеми вирішувалися. Як варіант можна проінформувати про них Product owner 'a і начальника відділу розробки.

У кінці спринту

1. Провести відкриту демонстрацію результатів спринту.

2. За декілька днів до демонстрації нагадати усім про її проведення.

3. Провести ретроспективу за участю усієї команди і Product owner 'a. Запросити начальника відділу розробки, щоб він допоміг знайти оптимальне рішення проблем.

4. Відновити статистику спринтів. Внести значення реальної продуктивності і основні тези минулої ретроспективи.

Microsoft Solutions Framework

Модель проектної групи MSF

Microsoft Solutions Framework

Бізнес-рішення повинні створюватися з урахуванням поставлених термінів і у рамках відведеного бюджету. Цього легше добитися, застосовуючи підхід, що затвердився, перевірений. MSF пропонує випробувані методики для планування, створення і впровадження успішних рішень в IT-індустрії. Він не містить жорстких інструкцій; замість цього пропонується гнучкий і масштабований підхід, здатний задовольнити потреби організації або проектної групи будь-якого розміру. Рекомендації MSF складаються із застосовних для більшості проектів принципів, моделей і дисциплін по управлінню людьми, процесами і технологічними елементами.

Для отримання подальшої інформації по MSF, см http://www.microsoft.com/msf

Microsoft Operations Framework

MOF пропонує рекомендації, що допомагають забезпечити надійність (reliability), доступність (availability), зручність в супроводі (supportability) і керованість (manageability) IT-рішень, побудованих на базі продуктів і технологій Майкрософту. Принципи, моделі і дисципліни MOF порушують кадрові, технологічні і технічні питання, що відносяться до управління складними (complex), розподіленими (distributed), різнорідними (heterogeneous) IT -системами.

Для отримання подальшої інформації по MOF, см http://www.microsoft.com/mof

Інформація про IT Infrastructure Library (ITIL) - збірку найбільш методик, що добре зарекомендували себе, є базою MOF, може бути отримана на сайті http://www.itil.co.uk/index.html

Основи моделі проектної групи MSF

Модель проектної групи MSF розроблялася протягом декількох років і виникла в результаті осмислення недоліків пірамідальної, ієрархічної структури традиційних проектних груп.

Відповідно до моделі MSF проектні групи будуються як невеликі багатопрофільні команди, члени яких розподіляють між собою відповідальність і доповнюють області компетенцій один одного. Це дає можливість чітко сфокусувати увагу на потребах проекту. Проектну групу об'єднують єдине бачення проекту, прагнення до втілення його в життя, високі вимоги до якості роботи і бажання самоудосконалюватися. Цей документ описує різні ролеві кластери (role clusters) усередині проектної групи, визначаючи їх цілі і області компетенції. Також даються рекомендації по використанню підходу Майкрософту до формування проектної групи при масштабуванні моделі як для малих, так і для великих і складних проектів.

Нижче описуються основні принципи, ключові ідеї і випробувані методики MSF в застосуванні до моделі проектної групи.

Основні принципи

MSF включає ряд основних принципів. У цьому розділі говориться про тих з них, які мають відношення до успішної роботи команди.

Розподіл відповідальності при фіксації звітності

MSF поєднує розподіл відповідальності за виконувану роботу із строго певною звітністю про її виконання.

Модель проектної групи MSF грунтується на твердженні про равноправности ролей в команді. Кожен ролевий кластер представляє унікальну точку зору на проект, і в той же час ніхто не не в змозі успішно представляти усі можливі погляди, що відбивають якісно різні цілі. Для дозволу цієї дилеми команда соратників (команда рівних, team of peers), що працює над проектом, повинна мати чітку форму звітності перед зацікавленими сторонами (stakeholders) при розподіленій відповідальності за досягнення загального успіху.

У рамках проектної групи кожен ролевий кластер звітує перед усією командою (як і перед усією організацією) про досягнення своєї мети. Говорячи інакше, кожен ролевий кластер звітує за свій вклад в остаточний результат. Відповідальність розподіляється серед членів команди, які взаимозависимы з двох причин, : по-перше, діяльність кожного з ролевих кластерів не може бути ізольована від роботи інших; по-друге, команда ефективніша, якщо кожному її членові зрозуміла повна картина того, що відбувається. Цей взаємозв'язок диктує необхідність для усіх членів команди висловлювати свою думку і вносити певний вклад у вирішення питань, що лежать поза областю їх звітності, забезпечуючи повноцінне використання усього спектру знань, компетенції і досвіду команди. Усі члени команди відповідають за успіх проекту; вони розділяють честь і славу у разі позитивного результату і повинні удосконалювати свій професійний рівень, працюючи над уроками менш вдалих проектів.

Наділяйте членів команди повноваженнями

У ефективно працюючій команді кожен її член має необхідні повноваження для виконання своїх обов'язків і упевненість в отриманні від колег усього необхідного. З іншого боку, замовник може бути упевнений в результатах роботи команди і будувати свої плани виходячи з цієї упевненості. У гіршому разі, замовник має бути в найкоротший строк повідомлений про затримку, що відбувається, або зміну.

Проектна група MSF наділяє своїх членів необхідним для роботи рівнем повноважень. Натомість вона чекає від своїх співробітників наступне:

- готовність переймати на себе зобов'язання перед іншими;
- чітке визначення тих зобов'язань, які вони на себе беруть;
- прагнення докладати належні зусилля до виконання своїх зобов'язань;

- готовність чесно і негайно інформувати про погрози виконанню своїх зобов'язань.

У випадках, коли для здійснення якого-небудь кроку потрібно спільну роботу декількох членів команди, вклад кожного з них залежатиме від дій інших. Проте люди не можуть витрачати час на контроль усіх залежностей, що впливають на їх діяльність. У ефективних проектних групах співробітники упевнені в тому, що вони можуть покластися на роботу своїх колег, які мають усі необхідні повноваження і прихильні загальній меті.

Можна розглянути аналогію з естафетною командою бігунів. Коли другий учасник команди починає бігти, він не сповільнюється для того, щоб озирнутися і побачити, як близько знаходиться його попередник. Замість цього він прагне якнайшвидше набрати швидкість і потім просто витягає назад руку, щоб узяти естафетну паличку. При цьому він упевнений, що паличка дійсно буде передана. Ця упевненість грунтується на тренованості, досвіді і довірі.

При роботі над складними проектами члени проектних груп повинні виробляти подібний рівень упевненості один в одному, що зміцнюється кожного разу при виконанні зобов'язання (нехай навіть невеликого), узятого на себе членом команди. Ось деякі прості рекомендації по зміцненню довіри в команді:

- Наділяйте членів проектної групи повноваженнями, необхідними для виконання їх обов'язків. Це означає, що у членів проектної групи є ресурси, необхідні для їх роботи, відповідні повноваження і розуміння їх меж. При цьому члени проектної групи повинні знати про наявні шляхи ескалації (escalation) ухвалення рішень з питань, що виходять за рамки їх компетенції.
- Будьте готові брати зобов'язання перед іншими. Така готовність включає усвідомлення і розуміння наслідків того, що бере зобов'язань і їх впливу на поточну завантаженість і ресурси. Як результат, того, що бере великих зобов'язань не повинне відбуватися перш, ніж усі їх наслідки добре зрозумілі. Замість цього члени проектної групи повинні брати менші зобов'язання, суть яких їм добре ясна. Наприклад, перш ніж взятися за велике завдання співробітник може попросити час на її обдумування. Успішне виконання малих зобов'язань надає членам команди упевненість один в одному.
- Чітко визначайте узяті зобов'язання. Це дозволяє уникнути нерозуміння, яке може пошкодити упевненості членів команди один в одному.
- Додайте максимум виправданих зусиль для виконання узятих на себе зобов'язань. Якщо проектна група включає людей з різних організацій, їх розуміння виправданості може відрізнятися. Наприклад, деякі члени проектної групи можуть вважати прийнятною роботу у вихідні дні. Інші ж можуть допускати це тільки в крайніх випадках, або ж можуть зазнавати утруднення з доступом до робочих місць в неурочний час.

- Коли виконання зобов'язання знаходиться під загрозою, прямо говорите про це. Неминуче виникатимуть випадки, коли ситуація міняється. Це може статися із-за зміни пріоритетів або непередбаченої події; або ж просто виконання поставленого завдання зажадає більше часу, чим передбачалося. Раннє попередження дозволяє іншим членам команди, залежним від вашої роботи, скоректувати свої плани. Можливо, вони навіть зможуть запропонувати підхід до вирішення виниклої проблеми.

У багатьох організаціях такі принципи поведінки є невід'ємною частиною корпоративної культури, і їх наслідують так чітко, що їх майже ніколи не доводиться обговорювати. Проте практикуючим MSF проектним групам доводиться час від часу працювати з організаціями, в яких ці цінності не до кінця розуміються і признаються. Усередині таких організацій часто існує дуже недоброзичлива атмосфера, що обмежує вільний потік інформації. У подібних випадках керівники проектних груп повинні чітко роз'яснювати прийняті в команді принципи колективної роботи і допомагати новим членам в засвоєнні MSF -подхода до виробничого процесу.

Концентруйтеся на бізнес-пріоритетах

Модель проектної групи MSF відстоює необхідність ухвалення рішень проектною групою на основі повного розуміння бізнесу замовника і при активній його участі в реалізації проекту. Ролевий кластер "Управління продуктом" (product management) діє по відношенню до проектної групи як представник замовника і частенько формується із співробітників організації-замовника. Кластер "Управління продуктом" представляє бізнес-сторону проекту і забезпечує його узгодженість із стратегічними цілями замовника. У обов'язку кластера "Управління продуктом" входить контроль за повним розумінням інтересів бізнесу при ухваленні ключових проектних рішень.

Ролевий кластер "Управління випуском" (release management) безпосередньо відповідальний за безперешкодне впровадження проекту і його функціонування. Цей ролевий кластер бере на себе зв'язок між розробкою рішення, його впровадженням і наступним супроводом, забезпечуючи інформованість членів проектної групи про наслідки їх рішень.

Єдине бачення проекту

MSF відстоює необхідність вироблення єдиного бачення (shared vision) проекту, що формує цілісний підхід проектної групи до розробки IT -решения.

Необхідно чітко розуміти цілі і завдання проекту або процесу, оскільки це основа усіх допущень про функціонування рішення у рамках організації замовника. Це відноситься до сприйняття рішення, як проектною групою, так і самим замовником. Загальне бачення чітко обкреслює зроблені припущення і забезпечує єдність мети усіх зацікавлених сторін. Це одна з основ моделі проектної групи MSF.

Коли усі учасники проекту розуміють це і виробляють єдине бачення, вони придбавають можливість погоджувати свої дії із загальною командною метою.

Не маючи такого бачення, члени проектної групи можуть керуватися такими, що суперечать один одному поглядами на меті проекту, що значно утруднить роботу команди як єдиного цілого. І навіть у разі успіху члени команди навряд чи зможуть оцінити свій вклад, оскільки ця оцінка залежить від точки зору, що приймається.

Проявляйте гнучкість - будьте готові до змін

MSF виходить з того, що усе навкруги безперервно міняється. Неможливо ізолювати IT-проекти від цих змін. Від початку до завершення проекту модель проектної групи MSF гарантує присутність усіх командних ролей і їх залученість в процес ухвалення рішень, обумовлених змінами, що відбуваються. Ця модель заохочує гнучкість (agility) при роботі в умовах постійного виникнення нових завдань і потреб. Участь усіх ролей в процесі ухвалення рішень забезпечує розгляд питань з урахуванням повного спектру точок зору.

Заохочуйте вільне спілкування

Історично багато організацій будували свою діяльність на основі зведення інформованості співробітників до мінімуму, необхідного для виконання роботи (need-to-know). Частенько такий підхід призводить до непорозумінь і знижує шанси команди на досягнення успіху.

MSF проповідує відкритий і чесний обмін інформацією як усередині команди, так і з ключовими зацікавленими особами поза нею. Вільний обмін інформацією не лише скорочує ризик виникнення непорозумінь і невиправданих витрат, але і забезпечує максимальний вклад усіх учасників проектної групи в зниження існуючої в проекті невизначеності.

Формування команди соратників (команди рівних) припускає залучення усіх існуючих ролей до ухвалення ключових рішень. Тому єдине бачення - неодмінна умова досягнення успіху, на якому, до речі, базується і прийнятий в MSF підхід до управління рисками. Він припускає залучення усіх членів проектної групи до виявлення і аналізу ризиків, а також створення позитивної, доброзичливої атмосфери для заохочення цієї діяльності. Відкрита, чесна дискусія про позитивний досвід, що мається, і про можливі напрями роботи над недоліками дає основу тієї культури самоудосконалення, яку проповідує MSF.

Існують чинники, здатні обмежити відкритість обміну інформацією в команді, наприклад, конфіденційність відомостей приватного або комерційного характеру. Проте при ухваленні рішення про приховання інформації треба оцінити, чи дійсно причини цієї секретності такі важливі? Якщо в команді вироблена атмосфера довіри, то в тих окремих випадках, коли обмеження поширення інформації дійсно потрібне, буде нескладно пояснити своїм колегам його причини і отримати замість їх довіра і упевненість у виправданості обмежень.

Ключові концепції

Успішне використання моделі проектної групи MSF грунтується на ряду ключових концепцій (key concepts), представлених в цьому розділі.

Команда соратників

Концепція "команди соратників" (teem of peers) означає рівноправне положення кожної з ролей в команді. Це сприяє вільному спілкуванню, збільшує командну відповідальність і зумовлює рівну важливість кожної з шести якісних цілей. Щоб досягти успіху у рамках команди соратників (команди рівних), кожен з її членів, незалежно від ролі, повинен нести відповідальність за якість продукту, розуміти інтереси замовника і суть вирішуваного бізнес-завдання.

Хоча соратники рівні, ухвалення рішення методом консенсусу між ролями не тотожно ухваленню рішення методом консенсусу між співробітниками. Кожен ролевий кластер вимагає певної внутрішньої організаційної ієрархії для розподілу роботи і управління його ресурсами. Керівники ролевих кластерів відповідальні за організацію роботи, управління і координацію дій команди, тоді як її члени мають можливість зосередитися на своїх індивідуальних завданнях.

Сфокусована на потребах замовника

Задоволення потреб замовника - головний пріоритет будь-якої добре працюючої проектної групи. Концентрація на потребах замовника (customer - focused mindset) означає обов'язкове розуміння його бізнес-завдань і прагнення до їх рішення з боку команди. Одним із способів визначення успіху такої уваги до замовника є здатність відстежити зв'язок кожного елементу в дизайні системи з відповідною йому початковою вимогою замовника або користувача (trace each feature in the design back to a customer or user requirement). Іншим ключовим моментом в задоволенні потреб замовника є його активна участь в проектуванні рішення і отримання його відгуків в ході процесу розробки. Це дозволяє проектній групі і замовникові успішно погоджувати свої очікування і потреби.

Націленість на кінцевий результат

Неважливо, чи займаєтеся ви, подібно до співробітників Майкрософту, виробництвом "коробкового" ПО або розробляєте програми для внутрішніх цілей вашого підприємства. Важливо, як ви відноситеся до результатів своєї індивідуальної праці, чи сприймаєте ви їх як продукт.

Перший крок в досягненні гідного рівня якості - це розглядати власну працю самостійним проектом або ж вкладом в який-небудь більший проект. MSF виступає за наділ проектів індивідуальними рисами. В результаті члени проектної групи починають почувати себе членами команди, а не відособленими працівниками. Наприклад, в Майкрософті для досягнення цього проектам дають імена. Такий підхід допомагає виділити проект і його команду, посилити в ній почуття відповідальності і створити механізм, що формує в команді моральний дух. Друк назви проекту на футболках, кавових чашках і інших сувенірах допомагає створити і

зміцнити дух команди. Це особливо корисно в проектах, над якими працюють "віртуальні колективи", складені з працівників різних підрозділів організації.

Усвідомивши свій вклад в проект, вам буде легко зрозуміти, що всякий очікуваний від вас результат може розглядатися як кінцевий продукт. Принципи і методики, застосовані до створення продуктів "взагалі", наприклад, пропоновані MSF, можуть допомогти вам досягти успіху в цьому.

Установка на кінцевий продукт (product mindset) також означає, що отриманню кінцевого результату проекту приділяється більше уваги, чим процесу його досягнення. З цього не виходить, що сам процес може бути поганий або непродуманий - просто він існує для отримання кінцевої мети, а не заради себе самого. Коли увага до кінцевого продукту стає частиною виробничої культури, кожен член колективу починає відчувати відповідальність за результат проекту.

У своїй презентації 1991-го року колишній менеджер програми Майкрософт Chris Peters так описував цю концепцію стосовно розробки програмного забезпечення :

"У кожного .. абсолютно однакова робота. Вона має одну і ту ж назву. І це - постачання продукту. Ваша робота - це не написання програмного коду. Ваша робота - не це тестування. Ваша робота - це не складання специфікацій. Ваша робота - це постачання продукту. Це якраз те, чим займається команда розробників.

Ваша роль як розробника або тестувальника вторинна. Я не говорю, що це не важливо, але це вторинно по відношенню до вашої справжньої роботи, якою є постачання продукту.

Коли ви прокидаєтеся уранці і приходите на роботу, ви говорите: "Що є пріоритетом - ми створюємо продукт або пишемо програмний код"? Відповіддю є: ми створюємо продукт. Не намагайтеся створювати код - створюйте продукт".

Установка на відсутність дефектів

У успішній команді кожен співробітник відчуває відповідальність за якість продукту. Вона не може бути делегована одним членом команди іншому або ж одним ролевим кластером іншому. Відповідно, кожен член команди повинен представляти інтереси замовника, враховуючи в ході розробки продукту його споживчі якості.

Установка на відсутність дефектів (zero - defect mindset) - це прагнення до найвищого рівня якості. Вона означає, що мета команди - виконання своєї роботи з максимально можливою якістю, причому таким чином, що якщо від команди зажадають поставити результат завтра, вона буде здатна поставити щось працює. Потрібне розуміння того, що щодня продукт має бути практично готовий до постачання. Це не означає постачання програмного коду, повністю вільного від дефектів. Але це означає, що продукт відповідає вимогам якості (quality bar), встановленим спонсором (sponsor) проектуі прийнятим проектною групою на етапі вироблення концепції.

Як аналогія, що якнайкраще описує цю ідею, приведемо лінію по виробництву автомобілів. Традиційно робітники виконували зборку з комплектуючих і відповідали тільки за якість своєї власної роботи. Коли автомобіль сходив з лінії, інспектор перевіряв, чи досить його якість для виставляння на продаж. Проте, у разі виявлення проблем виникали великі витрати на їх усунення, оскільки всяка переробка вже зібраної машини обходиться дуже дорого. На додаток, оскільки якість не була легко передбачуваною, кількість часу на тестування автомобіля також була важко прогнозованою.

Пізніше в автомобільній промисловості якість стала "завданням номер один". Це означає, що як тільки певна операція (наприклад, приєднання дверей або установка радіо) завершена, інспектор перевіряє поточний стан зразка і упевняється, що стандарти якості дотримані. Якщо в ході усього процесу зборки зберігається належний рівень якості, вимагається значно менше часу і ресурсів на остаточну перевірку придатності автомобіля. Крім того, це робить процес перевірки набагато більше передбачуваним, оскільки інспекторові треба лише перевірити якість зборки продукту в цілому, а не стан окремих його частин.

Прагнення до самоудосконалення

Прагнення до самоудосконалення (willingness to learn) - це прихильність ідеї невпинного саморозвитку за допомогою накопичення досвіду і обміну знаннями. Воно дозволяє членам проектної групи отримувати користь з негативного досвіду зроблених помилок, так само як і відтворювати успіхи, використовуючи перевірені методи роботи інших людей. Проведення по закінченню основних фаз проекту відкритих обговорень його стану і доброзичливий, але об'єктивний аналіз проекту після його закінчення - це ключові компоненти моделі процесу MSF. Проектні групи, що виділяють час на аналіз результатів своєї роботи і витягання з них уроків, створюють базу для постійного самоудосконалення і довготривалого успіху. Крім того, Майкрософт успішно розвиває культуру самоудосконалення, включаючи аналіз витягнутих уроків і обмін знаннями в плани індивідуальної роботи співробітників.

Зацікавлені команди працюють ефективно

Команди з низькою мотивацією програють з двох причин. На індивідуальному рівні їх члени не працюють з повною віддачею, що веде до зниження якості роботи і продуктивності праці. Окрім цього, увага працівників зосереджена на вузьких цілях, і вони не піклуються про вплив своєї діяльності на роботу колег. Обоє ці чинника чинять істотний вплив на IT -проекты, розробка яких завжди має на увазі високий рівень завдань, що стоять перед колективом, і інтелектуальну складність підходів до їх рішення.

MSF прагне до створення зацікавленості і високого морального духу команди. Люди, що працюють в Майкрософті, сприймають це як одну з основних характеристик компанії. Приведемо деякі методи стимулювання такої зацікавленості :

- Створюйте у членів команди єдине бачення вирішуваної задачі.

- Формуйте індивідуальність команди, використовуючи назви проектів і відзнаки команди - брелки, футболки, чашки і т. д.
- Ближче знайомтеся з колегами, використовуючи для цього неформальні зустрічі і заходи.
- Для створення командного духу організовуйте заходи, на яких люди можуть ближче познайомитися один з одним. Як правило, такі зустрічі проводяться поза офісом.
- Переконаєтеся, що особисті цілі і прагнення членів команди не залишаються без уваги. Наприклад, це може бути створення можливостей для особистого і професійного зростання або турбота про те, щоб посадові обов'язки не чинили негативного впливу на особисте життя.
- Наділяйте членів команди реальними повноваженнями; прислухайтеся до їх думок.
- Святкуйте успіхи команди.

Випробувані методики

Наступні випробувані методики (proven practices) є загальними засобами досягнення успіху команд, практикуючих MSF.

Малі багатопрофільні проектні групи

Малі багатопрофільні команди (small, multidisciplinary teams) мають ряд безперечних переваг. Одне з них - велика оперативність дій порівняно з великими колективами. Тому при роботі над великими проектами краще створювати команди команд - набір малих груп, що працюють паралельно. При цьому працівники, що є експертами в певних областях або специфічні функції, що мають, отримують повноваження діяти у рамках своїх областей компетенції.

Усередині проектної групи - і навіть усередині окремого ролевого кластера - існують завдання, що вимагають різних професійних навичок. Усі члени команди або ролевого кластера, різні, що мають, освіту, досвід і спеціалізацію, вносять свій унікальний вклад в роботу колективу і, у результаті, в створення кінцевого продукту.

Колективна робота

Одна з цілей моделі проектної групи - зменшення витрат взаємодії. Як наслідок, команди мають менше перешкод для ефективного обміну інформацією. Окрім структури організації, важливу роль в ефективності її внутрішніх і зовнішніх інформаційних потоків грає просторова близькість співробітників.

У своїй книзі Microsoft Secrets, втаємниченій Майкрософтом, Michael A. Cusumano і Richard W. Selby пишуть:

".. Єдине місце роботи дозволяє співробітникам, зайнятим над проектом, регулярно збиратися разом, обговорюючи різні ідеї. Часте і безперешкодне спілкування здатне запобігти великим проблемам".

Також сусідство робочих місць допомагає команді зміцнити почуття згуртованості і єдності.

Зближення співробітників - наприклад, робота в одній секції будівлі, об'єднання їх офісів або виділення спеціального місця для збору команди - найбільш ефективний спосіб створення умов для вільного спілкування, що являється істотної складової формули командного успіху MSF.

Проте, хоча ідеальним вибором є зближення робочих місць, вимоги бізнесу і сучасні технологічні нововведення в комунікаціях роблять можливим створення успішних "віртуальних" команд.

Віртуальні команди - це команди, що взаємодіють і спілкуються головним чином за допомогою електронних комунікацій. Обмін інформацією в них йде через межі країн і організацій, простір і час. Зв'язок з колегами через Internet в режимі реального часу докорінно міняє принципи роботи і обміну інформацією. Internet стає для членів команди новим комунікаційним стандартом, і програмне забезпечення, що робить можливою видалену колективну роботу, відкриває шлях до подальшого підвищення продуктивності в нових умовах.

Для віртуальних команд модель проектної групи MSF є ключовою. Без належного рівня організації, що об'єднує усі робочі ролі в єдине ціле, віддаленість працівників зажадала б ще більшої кількості дискусій, договорів і взаємозв'язків, жорсткішого планування і додаткових інструментів моніторингу проекту для забезпечення злагодженості роботи.

Життєво важлива складова віртуальної команди - це упевненість кожного її члена в можливості покластися на інших, що досягається виробленням загальної корпоративної культури, невпинною управлінською роботою і - як тільки це стає можливим - перенесенням роботи в одне місце (чи хоч би спільним відпочинком).

Дослідження показують, що при формуванні віртуальних команд комунікаційним здібностям і соціальним якостям працівників зазвичай приділяється недостатня увага. Аналітики говорять, що це упущення зазвичай є головною причиною невдач багатьох проектів. Тому при створенні віртуальної команди підбирайте людей, які, :

- можуть працювати самостійно;
- мають лідерські якості;
- мають необхідні для створення продукту професійні навички;
- здатні передавати свої знання колегам;
- можуть допомогти в розвитку ефективних методів роботи.

Загальна участь в проектуванні

Кожен ролевий кластер бере участь в створенні специфікації рішення, оскільки має унікальний погляд на продукт з точки зору його відповідності своїм завданням, також як і завданням усієї команди. Це створює клімат, стимулюючий виникнення і втілення кращих ідей, що охоплюють усі точки зору.

Огляд моделі команди MSF

MSF заснований на постулаті про шість якісних цілей, досягнення яких визначає успішність проекту. Ці цілі обумовлюють модель проектної групи. Тоді як

за успіх проекту відповідальна уся команда, кожен з її ролевих кластерів, визначуваних моделлю, асоційований з однією із згаданих шести цілей і працює над її досягненням.

Шість ролевих кластерів моделі проектної групи - це "Управління продуктом" (product management), "Управління програмою" (program management), "Розробка" (development), "Тестування" (test), "Задоволення споживача" (user experience) і "Управління випуском" (release management). Вони відповідальні за різні області компетенції (functional areas) і пов'язані з ними цілі і завдання. Іноді ролеві кластери називаються просто ролями. Але у будь-якому випадку суть концепції залишається тією ж - побудувати основу виробничих стосунків і пов'язану з нею модель команди такими, щоб вони були пристосовуваними (масштабованими) для задоволення потреб будь-якого проекту. Одна роль (чи один кластер) може бути представлена одним або декількома співробітниками, залежно від розміру проекту, його складності і професійних навичок, потрібних для реалізації усіх областей компетенції кластера.

Модель проектної групи MSF підкреслює важливість побудови ролевих кластерів відповідно до потреб бізнесу. Угрупування пов'язаних областей компетенції, кожна з яких має свою специфіку, забезпечує хорошу збалансованість команди. Чітке визначення цілей підвищує рівень відповідальності і сприяє кращому їх сприйняттю проектною командою, що негайно позначається якнайкраще на якості продукту, що випускається. Оскільки кожна з цілей однаково потрібна для успішності проекту, усі ролі знаходяться в рівноправних партнерських взаєминах з рівною значущістю при ухваленні рішень.

Помітимо, що використання ролевих кластерів не має на увазі і не нав'язує ніякої спеціальної структури організації або обов'язкових посад. Адміністративний склад ролей може широко варіюватися в різних організаціях і проектних групах. Найчастіше ролі розподіляються серед різних підрозділів однієї організації, але іноді частина їх відводиться співтовариству споживачів або зовнішнім по відношенню до організації консультантам і партнерам. Ключовим моментом є чітке визначення працівників, відповідальних за кожен ролевий кластер, їх функцій, відповідальності і очікуваного вкладу в кінцевий результат.

Ролевий кластер	**Мета**	**Область компетенції**	**Функції**
Управління продуктом	Задоволені замовники	Маркетинг Бізнес-віддача (бізнес-пріоритети) Представлення інтересів замовника Планування продукту	Виступає в ролі представника замовника Формує загальне бачення/рамки проекту Організовує роботу з вимогами замовника Розвиває сфери застосування в бізнесі Формує очіку-

			вання замовника Визначає компроміси між параметрами "можливості продукту / час / ресурси" Організовує маркетинг, PR і євангелізацію Розробляє, підтримує і виконує план комунікацій
Управління програмою	Досягнення результату у рамках проектних обмежень	Управління проектом Вироблення архітектури рішення Контроль виробничого процесу Адміністративні служби	Управляє процесом розробки з метою отримання готового продукту у відведені терміни Формулює специфікацію продукту і розробляє його архітектуру Регулює взаємини і комунікацію усередині проектної групи Стежить за тимчасовим графіком проекту і готує звітність про його стан Проводить в життя важливі компромісні рішення Розробляє, підтримує і виконує звідний план і календарний графік проекту Організовує управління рисками
Розробка	Створення продукту відповідно до специфікації	Технологічне консультування Проектування і здійснення реалізації Розробка додатків Розробка інфраструктури	Визначає деталі фізичного дизайну Оцінює необхідні час і ресурси на реалізацію кожного елементу дизайну Розробляє або контролює розробку елементів Готує продукт до впровадження Консультує ко-

			манду з технологічних питань
Тестування	Схвалення випуску продукту тільки після того, як усі дефекти виявлені і улагоджені	Планування тестів Розробка тестів Звітність по тестах	Забезпечує виявлення усіх дефектів Розробляє стратегію і плани тестування Здійснює тестування
Задоволення споживача	Підвищення ефективності користувача, збільшення споживчої цінності продукту	Забезпечення технічної підтримки Навчання Ергономіка Графічний дизайн Інтернаціоналізація Загальнодоступність (забезпечення можливості роботи для користувачів з обмеженими фізичними можливостями)	Представляє інтереси споживача в команді Організовує роботу з вимогами користувача Проектує і розробляє системи підтримки продуктивності Визначає компроміси, що відносяться до зручності використання і споживчих якостей продукту Визначає вимоги до системи допомоги і її зміст Розробляє учбові матеріали і здійснює навчання користувачів
Управління випуском	Безпроблемне впровадження і супровід продукту	Інфраструктура Супровід Бізнес-процеси Управління випуском готового продукту	Представляє інтереси відділів постачання і обслуговування продукту Організовує постачання проектної групи Організовує впровадження продукту Виробляє компроміси в керованості і зручності супроводу продукту Організовує супровід і інфраструктуру постачання Організовує логістичне забезпечення проектної групи

Задоволені замовники

Успішний проект повинен породжувати рішення, що задовольняє замовників і споживачів. Можна повністю укластися у відведений час і бюджет, але при цьому потерпіти повну невдачу, якщо потреби замовників не будуть задоволені.

Досягнення результату у рамках проектних обмежень

Ключова мета будь-якої проектної групи - створення рішення у рамках встановлених обмежень. Основні обмеження всякого проекту - це виділений бюджет і терміни. Успішність більшості проектів визначається саме витраченим часом і засобами.

Створення продукту відповідно до специфікації

Специфікація продукту детально описує, що саме проектна група повинна поставити замовникові. Для проектної групи важливо максимально точно наслідувати специфікацію продукту, що поставляється, оскільки вона складає суть угоди між проектною групою і замовником.

Схвалення випуску продукту лише після того, як усі дефекти виявлені і улагоджені

Будь-яке програмне забезпечення містить дефекти. Проте усі дефекти мають бути виявлені і улагоджені (address) до того, як продукт випущений. Залагоджування дефекту може мати на увазі різні рішення, починаючи від усунення і закінчуючи документуванням способів його обходу (work - around). Постачання продукту з відомим дефектом, але з описом способів його обходу прийнятніше постачання продукту з невиявленим дефектом, який надалі стане сюрпризом, як для проектної команди, так і для замовника.

Підвищення ефективності користувача, збільшення споживчої цінності продукту

Для того, щоб продукт був успішним, він повинен покращувати роботу і продуктивність користувачів. Постачання продукту, що має широкі можливості і відмінні якості, але надмірно складного з точки зору споживача, недопустима.

Безпроблемне впровадження і супровід продукту

Іноді забувають про необхідність забезпечення гладкого впровадження продукту. Впровадження створює вірне або невірне, але у будь-якому випадку незгладиме враження про продукт. Наприклад, збої в інсталяційній програмі можуть змусити споживачів думати, що сама програма має такий же рівень якості (що може і не відповідати дійсності). Як наслідок, проектна група повинна не просто впроваджувати продукт. Треба боротися за гладке, безпроблемне його впровадження і заздалегідь підготувати супровід і підтримку продукту. Це може включати, наприклад, завчасне навчання споживачів і створення інфраструктури впровадження.

Ролеві кластери моделі проектної групи

Ролеві кластери моделі проектної групи MSF

Ролевий кластер "Управління продуктом"

Ключова мета ролевого кластера "Управління продуктом" (product management) - задоволені замовники. Проект не може вважатися успішним, якщо він не привів до задоволення потреб замовника. Проте, передусім, замовник має бути ідентифікований і зрозумілий! В деяких випадках замовником може бути не та сторона, яка спонсорує проект (тобто підтримує витрачені на нього зусилля і покриває витрати). Отже, має бути зроблена чітка відмінність між цими сторонами і вироблений аналіз вимог кожної з них в цілях успішної взаємодії з ними обома. Лише після цього може бути сформульований набір вимог і очікувань, яким повинні слідувати відповідні проектні ролі. Можлива ситуація, коли проектна група уклалася до бюджету і терміни, але успіх все ж не досягнутий, оскільки не задоволені бізнес-потреби замовника.

Модель проектної групи MSF розділяє області компетенції кожного ролевого кластера, щоб чіткіше визначити набір зон відповідальності, що входять в них, твірних в сукупності загальний потенціал команди.

Для досягнення своєї мети ролевий кластер "Управління продуктом" повинен містити в собі наступні області компетенції : планування продукту (product planning), бізнес-віддача (business value), представлення інтересів замовника (customer advocacy) і маркетинг (marketing).

Області компетенції

Маркетинг

- Проведення маркетингових і PR акцій, націлених на (потенційних) клієнтів.
- Демонстрація своїх конкурентних переваг.

- Забезпечення попадання продукту у відповідні канали дистрибуції, що гарантують його максимальну доступність для споживача.
- Консультування споживача для створення у нього позитивного враження від купівлі і використання продукту.

Бізнес-віддача

- Визначення і контроль обумовленості роботи над проектом з точки зору бізнесу.
- Контроль (в т.ч. кількісний) отримання замовником бізнес-віддачі від проекту.

Представлення інтересів замовника

- Вироблення загального бачення проекту і рішення.
- Обмін інформацією із замовником і формування його очікувань.

Планування продукту

- Збір, аналіз і приоритезация вимог замовника і бізнесу.
- Проведення дослідження ринку і наявного попиту; аналіз і вивчення конкурентів.
- Визначення критеріїв бізнес-успішності проекту.
- Розробка плану випуску серії версій продукту.

Маркетинг

Маркетинг - це процес або методика просування, поширення і продажу на ринку продукту, рішення або послуги. Він має декілька різновидів - стартовий маркетинг, довготривалий маркетинг і PR. На різних етапах життєвого циклу рішення маркетинг фокусується на різних завданнях. Те, на якому саме етапі життєвого циклу знаходиться ваше рішення, є визначальний для вибору адекватних маркетингових заходів.

Бізнес-віддача

Для замовників і осіб, що приймають бізнес-рішення, ця область компетенції забезпечує максимально достовірну оцінку віддачі від інвестування в проект.

Щоб отримати добрий результат, менеджери продукту повинні накопичувати знання про бізнес замовника, чинники успіху і ключові використовувані метрики. Отримання цієї інформації може розглядатися як бізнес-аналіз, що виявляє ключові складові успіху. Тільки знаючи, що принесе замовникові віддачу, а що - ні, ви зможете спроектувати і побудувати відповідне рішення (solution). Все частіше за інвестицію в область інформаційних технологій піддаються скрупульозним перевіркам, і багато IT -проекты вимагають ретельного вивчення фінансової обгрунтованості. Проведення аналізу співвідношення витрат і прибутку істотно збільшує шанси задоволення замовника. Визначення фінансових результатів - невід'ємна частина процесу інвестування в IT-проект.

Представлення інтересів замовника

Ця область компетенції відповідає за контакти із замовником і формування його очікувань. Такі контакти включають PR, брифінги для вищого керівництва і

замовників, споживчий маркетинг, презентації, анонси і прем'єри продуктів. Формування очікувань - це ключове завдання кластера "Управління продуктом" починаючи з моменту, коли єдине бачення результатів проекту вже сформоване. Це дуже важливо, адже саме задоволення очікувань є визначальним чинником успіху або невдачі проекту.

Наведемо приклад. Допустимо, замовник в певний день чекає постачання проектною групою продукту, що реалізовує десять заздалегідь обумовлених функцій. Якщо у результаті поставляється продукт тільки з двома функціями з цих десяти, то як замовником, так і проектною групою, проект вважатиметься невдалим.

Проте, якщо менеджери продукту постійно підтримують двосторонній зв'язок із замовником упродовж усього процесу розробки, внесення змін в план може відбуватися відповідно до очікувань замовника, і це сприятиме успіху проекту. Менеджери продукту можуть залучити замовника в процес ухвалення компромісних рішень і проінформувати його про наростаючі риски і інші труднощі. На відміну від попереднього випадку, замовник може оцінити ситуацію і погодитися з проектною групою в тому, що реалізація усіх десяти функціональних можливостей у відведений час не реалістична, і що наявність лише двох з них є прийнятним компромісом. При такому розвитку подій реалізація двох функцій відповідає очікуванням замовника і обидві сторони вважатимуть проект таким, що вдався.

Планування продукту

Планування продукту визначає вимоги і необхідні характеристики для послідовності версій рішення. У завдання планування продукту входить забезпечення швидкого розуміння вимог до рішення з боку менеджера програми або розробника. Це включає, по-перше, повне розуміння поточних вимог до рішення: які бізнес-потреби воно задовольняє, як замовники його використовуватимуть, які можуть виникнути проблеми супроводу і які альтернативні рішення можливі. По-друге, це дослідження додаткових чинників, які збільшили б цінність рішення для замовника, : можливість входження в нові сектори бізнесу, інтеграція з іншими системами, підвищення продуктивності, спадкоємність, скорочення витрат на супровід і так далі. Знаючи це, розробник плану продукту може рекомендувати для його наступних версій різні функціональні можливості і пріоритети їх реалізації.

Основні складові планування продукту - дослідницька робота і аналіз. Незалежно від того, чи йде мова про розуміння замовника і потреб його бізнесу або про розуміння умов конкуренції, важливо приділяти досить уваги відповідним дослідженням і аналізу. Такий підхід також запобігає непродуктивній витраті ресурсів на реалізацію непотрібних елементів.

Ролевий кластер "Управління програмою"

Основне завдання цього ролевого кластера - забезпечити реалізацію рішення у рамках обмежень проекту, що може розглядатися як задоволення вимог спонсора проекту до його результату. Для цього "Управління програмою" контролює календарний графік проекту, об'єм роботи і відведений на проект бюджет. Даний

кластер забезпечує своєчасне досягнення необхідних результатів і задоволення очікувань спонсора упродовж проекту. Нижче описуються області компетенції ролевого кластера "Управління програмою".

Управління проектом

- Моніторинг і управління бюджетом.
- Складання звідного плану і звідного календарного графіку проекту.
- Організація управління рисками.
- Сприяння обміну інформацією і досягненню домовленостей усередині проектної групи.
- Моніторинг прогресу і звітність про стан проекту.
- Управління виділенням ресурсів.

Вироблення архітектури рішення

- Організація високорівневого проектування рішення.
- Управління функціональною специфікацією.
- Визначення рамок проекту і ключових компромісних рішень.

Контроль виробничого процесу

- Організація контролю виробничого процесу.
- Вироблення рекомендацій по його вдосконаленню.

Адміністративні служби

- Організація процесів управління проектом і допомога лідерам команд (team leads) в їх використанні.
- Організація ряду адміністративних служб, необхідних для підтримки ефективної роботи команди.

Управління проектом

Будучи відповідальним за календарний план, менеджмент проекту (project management) стежить за усіма тимчасовими графіками усередині команди, контролює їх правомірність і інтегрує їх в звідний календарний графік проекту. Він, у свою чергу, піддається моніторингу, і звітність про хід його виконання прямує проектній групі і спонсорові проекту.

Будучи відповідальним за бюджет, менеджмент проекту організовує його створення, збираючи вимоги до ресурсів з боку кожного з ролевих кластерів проектної групи. Менеджмент проекту має бути залучений в усі рішення, що стосуються ресурсів (як людських, так і матеріальних). Також менеджмент проекту відстежує співвідношення реальних і планових витрат і надає звіти про них проектній групі і спонсорові.

На додаток до цього у рамках цієї області компетенції знаходиться координування ресурсів, сприяння ефективній взаємодії членів проектної групи і допомога в ухваленні ключових рішень.

Детальнішу інформацію про область компетенції "Управління проектом" і підході MSF до управління проектами, см http://www.microsoft.com/msf/

Вироблення архітектури рішення

"Вироблення архітектури рішення" (solution architecture) - це область компетенції ролевого кластера "Управління програмою", відповідальна за логічний дизайн рішення і опис його функцій. Її основне завдання - забезпечення результативного і ефективного використання продукту споживачем для досягнення своїх цілей.

Зони відповідальності "Вироблення архітектури рішення" включають:

- Організацію високорівневого проектування рішення.
- Управління функціональною специфікацією.
- Визначення рамок проекту і ключових компромісних рішень.

Будучи відповідальною за логічний дизайн рішення, "Вироблення архітектури рішення" здійснює життєво важливий зв'язок між бізнес-стороною проекту (що представляється кластером "Управління продуктом" за допомогою концептуального дизайну) і технологічною його стороною (що представляється кластером "Розробка" за допомогою фізичного дизайну). Ця область компетенції "опікає" функціональну специфікацію. Вона прагне до досягнення внутрішньокомандного консенсусу про дизайн і обгрунтовує вибраний підхід перед зацікавленими в проекті сторонами. Ця область компетенції також відповідає за відповідність усіх елементів дизайну первинним вимогам (і, зрештою, забезпечення бізнес-віддачі). Кожен елемент рішення повинен реалізовувати деяку вимогу: таким чином проектна група може оцінити наслідки будь-яких змін дизайну для бізнес-корисності кінцевого продукту.

Заходи, архітектуру рішення", що проводиться "Виробленням, включають:

- Створення концепції рішення і його узгодження з архітектурою підприємства замовника; розробка стратегії версионирования продукту; вироблення рекомендацій до планів збору вимог.
- Збір вимог до интероперабельности (interoperability) рішення, його відповідності корпоративній архітектурі і діючим нормативам; здійснення логічного проектування; складання "карти" відповідності (traceability map) елементів дизайну первинним вимогам замовника; створення функціональної специфікації; специфікація проміжних версій продукту.
- Управління змінами функціональної специфікації; підтримка "карти" відповідності; роз'яснення специфікації іншим ролевим кластерам проектної групи і зовнішнім зацікавленим особам; зв'язок з іншими проектними групами для забезпечення интероперабельности.
- Участь в процесі приоритезации; формування очікувань зацікавлених сторін.
- Зв'язок з архітектурною групою підприємства; коригування вимог до майбутніх версій рішення.

Співробітники, що працюють в цій області компетенції, мають бути утвореними в технологічному плані і мати великий досвід у поєднанні із здатністю зіставляти технологічні проблеми з потребами бізнесу, що стоять за ними. Хоча архітектори рішення можуть покластися на досвід команди розробників в питаннях специфічних технологій, вони повинні дуже чітко усвідомлювати наслідки тих або інших технічних кроків і розуміти їх взаємозв'язок і їх вплив на середовище впровадження. Також архітектори рішення мають бути в змозі обговорити вищезгадані наслідки з архітекторами замовника і негайно вирішити будь-які конфлікти між пропонованим рішенням і архітектурою підприємства замовника.

Контроль виробничого процесу

Область компетенції "Контроль виробничого процесу" (process assurance) ролевого кластера "Управління програмою" забезпечує контроль за дотриманням проектною групою виробничим процесам, націленим на досягнення високоякісного результату. При цьому основна увага приділяється виявленню і усуненню першопричин дефектів. "Контроль виробничого процесу" має дві основні зони відповідальності :

- Визначення ключових виробничих процесів, використовуваних командою при роботі над проектом, і консультування команди з питань їх впровадження.
- Проведення аналізу ефективності процесів, вироблення рекомендацій по їх вдосконаленню, моніторинг їх реалізації і адекватності.

Ця область компетенції здійснює наступну діяльність:

- Розробка внутрішньопроектних виробничих протоколів і процесів.
- Консультування з питань ефективного використання виробничих процесів; контроль реалізації процесів; визначення їх адекватності; вироблення рекомендацій по вдосконаленню процесів.

Співробітники, що працюють в цій області компетенції, повинні мати певну незалежність від проектної групи, можливість поглянути на проект "з боку". Часто вони навіть підзвітні особі, що не входить безпосередньо до проектної групи.

Адміністративні служби

Ця область компетенції ролевого кластера "Управління програмою" відповідальна за реалізацію процесів по управлінню проектом і за адміністративну підтримку проектної групи.

Вона забезпечує реалізацію проектною групою процесів, що відповідають специфікації дизайну. Адміністративні служби (administrative services) створюють умови, при яких проектна команда може працювати ефективно, випробовуючи мінімум бюрократичних перешкод.

Відповідальність адміністративних служб включає:

- Реалізацію процесів управління проектами і допомога лідерам груп в їх використанні. Наприклад, збір необхідної для звітності інформації,

допомога лідерам груп в складанні звітів і управлінні планом і календарним графіком проекту.

- Забезпечення ряду адміністративних послуг, необхідних для підтримки ефективної роботи команди, таких як складання розкладу зборів, постачання і складання контрактів.

Адміністрація проекту здійснює:

- Забезпечення стартових процесів проекту, таких як залучення співробітників з інших організацій, підготовка контрактів, забезпечення інфраструктури проектної групи (робочі місця, телефони, контроль безпеки і так далі)
- Узгодження правил планування; допомога лідерам груп у виробленні планів і календарних графіків; збір інформації, необхідної для створення звідного плану і календарного графіку проекту; організацію процесів звітності про хід здійснення проекту і про фінансовий стан.
- Допомога лідерам груп в складанні звітності про хід проекту, формування звідних проектних звітів.
- Забезпечення згортання усіх адміністративних систем по закінченню проекту.
- Організацію загальної діяльності по адмініструванню проекту (планування зборів, управління рисками і проблемами, підтримка таблиці головних ризиків, списку планових заходів і так далі); складання звітів про хід проекту і фінансових звітів; організація суміжних робочих місць членів команди з метою об'єднання колективу.

Компетенція адміністратора проекту вимагає поєднання хороших адміністративних здібностей з уважним відношенням до деталей за наявності істотного досвіду в методиках планування проектів і чіткого розуміння корпоративних правил і інструкцій організацій-постачальників. У великих проектах для новачків є прекрасна можливість роботи у рамках адміністративних служб в цілях набуття досвіду, необхідного для наступних проектів.

Ролевий кластер "Розробка"

Первинним завданням ролевого кластера "Розробка" є побудова рішення відповідно до специфікації. Її виконання означає створення рішення, відповідного очікуванням замовника і умовам, сформульованим у функціональній специфікації. Також цей ролевий кластер строго наслідує вироблену архітектуру і дизайн рішення, які спільно з функціональною специфікацією складають звідний опис кінцевого продукту.

На додаток до функції безпосередньої розробки рішення на цей ролевий кластер покладаються обов'язки по технологічному консультуванню проектної групи. У цій якості розробники займаються підготовкою початкових даних для проектування і вибору технологічного інструментарію рішення. Також ролевий

кластер "Розробка" створює функціональні прототипи для перевірки правильності прийнятих рішень і зменшення ризиків.

Як творців рішення розробники здійснюють низькорівневе проектування рішення і його елементів, оцінюють трудовитрати на реалізацію і потім здійснюють побудову самого рішення. Розробники самі оцінюють власні витрати і відстежують розклад, оскільки їх повсякденна діяльність зв'язана з різними нештатними ситуаціями. Ця концепція називається оцінюванням від низу до верху (bottom - up estimation) і є фундаментальною частиною філософії MSF. Мета цієї концепції полягає в досягненні більшої обгрунтованості календарного плану і збільшенні почуття відповідальності тих, хто визначає терміни цього плану, і від чиєї продуктивності залежить його виконання.

Технологічне консультування

- Виконання функцій технологічних консультантів.
- Оцінювання і верифікація технологій.
- Активна участь в створенні і обговоренні функціональної специфікації.
- Вклад в створення корпоративних стандартів розробки програмного забезпечення.

Проектування і здійснення реалізації

- Співвідношення архітектури рішення з архітектурою підприємства.
- Створення і реалізація логічного і фізичного дизайну рішення.

Розробка додатків

- Програмування тих, що становлять рішення відповідно до проектної документації.
- Аналіз і обговорення програмного коду (code reviews) з метою обміну знаннями і досвідом.
- Здійснення тестування модулів (unit testing) відповідно до плану і в координації з ролевим кластером "Тестування".

Розробка інфраструктури

- Створення тих, що становлять рішення відповідно до проектної документації.
- Аналіз і обговорення програмного коду з метою обміну знаннями і досвідом.
- Здійснення тестування модулів відповідно до плану і в координації з ролевим кластером "Тестування".
- Розробка скриптів автоматизації.
- Створення внедренческой документації.

Технологічне консультування

Область компетенції "Технологічне консультування" (technology consulting) служить ресурсом дозволу технічних проблем. Виконуючи роль технологічних консультантів, розробники повинні надавати необхідну інформацію для проектування, оцінки і верифікації технологій; проводити попередні дослідження з метою мінімізації ризиків.

На фазі вироблення концепції ця область компетенції аналізує вимоги замовників і споживачів з точки зору тієї, що їх реалізовується. "Технологічне консультування" вносить свій вклад в підготовку документу загального опису проекту, оцінюючи технічні проблеми, що впливають на можливість реалізації проекту при заданих параметрах. "Технологічне консультування" зважує усе "за і проти" кожного підходу до реалізації і визначає адекватність спочатку вибраних технологічних засобів. У рамках цієї діяльності розробники можуть вести дослідницьку роботу, проводити консультації з іншими сторонами як усередині організації, так і поза нею, вести обговорення з постачальниками технологій. В цілях додаткової верифікації "Технологічне консультування" може розробляти прототипи ("іграшкові" версії рішення з неповною функціональністю), щоб підтвердити з їх допомогою життєздатність аналізованих концепцій. Це особливо важливо в проектах, що вимагають використання нових технологій, або ж в тих предметних областях, де у проектної групи відсутній достатній досвід.

Проектування і здійснення реалізації

Область компетенції "Проектування і здійснення реалізації" (implementation architecture and design) пов'язана з набором завдань, що відносяться до визначення архітектури рішення і його проектування.

Ролевий кластер "Управління програмою" відповідальний за загальну архітектуру рішення і її позиціонування у рамках архітектури підприємства. Проте ролевий кластер "Розробка" відповідальний за відповідність архітектури реалізації рішення архітектурі підприємства. Це стосується специфіки додатків, даних і технологічного інструментарію рішення.

MSF пропонує трирівневий процес проектування : концептуальний дизайн (conceptual design), логічний дизайн (logical design) і фізичний дизайн (physical design). "Управління програмою" і "Управління продуктом" спільно здійснюють концептуальний дизайн. Він включає сценарії використання (user scenarios), високорівневий аналіз вимог до зручності експлуатації (usability), концептуальне моделювання даних і початковий вибір використовуваних технологій. Розробники ж займаються логічними і фізичними аспектами дизайну рішення. Ця діяльність вимагає адекватних технологічних знань і уміння визначити вплив того або іншого технологічного вибору на створюване рішення.

Розробка додатків

Область компетенції "Розробка додатків" (application development) пов'язана із завданнями розробки програмних застосувань у рамках проекту. Головні цілі

цієї області компетенції - створення тих, що становлять рішення відповідно до проектної документації, проведення тестування модулів, виправлення дефектів, виявлених в процесі тестування і здійснення інтеграції усіх компонент в остаточний продукт.

Розробники вносять свій вклад у вироблення стандартів і досконально наслідують їх в процесі роботи над рішенням. Вони також здійснюють аналіз і обговорення програмного коду (code reviews), щоб оцінити якість виконаної роботи. Проведення такого аналізу дозволяє членам проектної групи ділитися накопиченими знаннями і досвідом, утілюючи в життя фундаментальний принцип MSF - витягання уроків на рівні команди. Від розробників потрібно проведення належного тестування модулів (unit testing) і адекватне документування цього процесу. Така робота здійснюється в тісному зв'язку з ролевим кластером "Тестування", який планує і виконує незалежну оцінку якості рішення.

Розробка інфраструктури

Область компетенції "Розробка інфраструктури" (infrastructure development) пов'язана із завданнями розробки системної інфраструктури і інфраструктури програмного забезпечення, що входить до складу рішення. Системна інфраструктура включає мережеву інфраструктуру, клієнтські і серверні комп'ютери і усі супутні компоненти. Інфраструктура програмного забезпечення включає операційні системи клієнтів і серверів, а також програмні продукти, що забезпечують необхідні сервіси (наприклад, служби каталогів, системи обміну повідомленнями, бази даних, інтеграція додатків підприємства, адміністрування системи, адміністрування мережі і так далі).

Команда розробників створює інфраструктуру відповідно до проектної документації. Це включає налаштування технологічних засобів рішення, як наприклад, налаштування мережі і систем "клієнт-сервер". Складові інфраструктури схильні до впливу вимог до додатків і навпаки. Наприклад, якщо критичними чинниками є надійність і продуктивність, може бути необхідним використання кластеризації (clustering) і балансування завантаження (load-balancing) серверів. Операційні системи і системні продукти, в середовищі яких використовуватиметься рішення, мають бути відповідним чином встановлені, конфігуровані і оптимізовані. Після закінчення проведення необхідного тестування і стабілізації компоненти інфраструктури впроваджуються на широкій основі. Це впровадження здійснюється ролевим кластером "Управління випуском", який забезпечує задоволення вимог до інфраструктури рішення.

Ролевий кластер "Тестування"

Завдання ролевого кластера "Тестування" (test) - схвалення випуску продукту тільки після того, як усі дефекти виявлені і улагоджені. Будь-яке програмне забезпечення містить дефекти. Але треба виявити і улагодити (address) усе з них до того, як продукт випущений. Залагоджування дефекту може мати на увазі різні рішення, починаючи від усунення і закінчуючи документуванням способів обходу

дефекту (work - around). Постачання продукту з відомим дефектом, але з описом способів його обходу є прийнятнішою, ніж постачання продукту з невиявленим дефектом, який надалі стане сюрпризом, - як для проектної команди, так і для замовника.

Щоб досягти успіху, команда тестувальників повинна фокусуватися на певних ключових завданнях. Вони структуруються у вигляді трьох областей компетенції.

Планування тестів

- Розробка методології і плану тестування.
- Участь у встановленні стандарту якості (quality bar).
- Розробка специфікацій тестів.

Розробка тестів

- Розробка і підтримка автоматизованих тестів (automated test cases), інструментів і скриптів.
- Проведення тестів з метою визначення стану проекту.
- Управління билдами (manage the build process).

Звітність про тести

- Доведення до зведення проектної групи інформації про якість продукту.
- Моніторинг знайдених помилок з метою забезпечення їх залагоджування до випуску продукту.

Планування тестів

Ця область компетенції (планування тестів - test planning) ролевого кластера "Тестування" формулює методологію знаходження і врегулювання проблем якості продукту.

Команда тестувальників розробляє плани і методики тестування і таким чином формує стратегію, використовувану в проекті для тестування рішення. Плани тестування включають опис типів тестів, тестованих складових і інформацію про необхідні ресурси (як людських, так і технічних).

Істотна частина роботи цієї області компетенції полягає в участі у виробленні необхідного рівня якості (quality bar) продукту. Ця діяльність включає надання проектній групі метрик контролю якості і критеріїв успішності вирішення.

Ще один рід діяльності, здійснюваний цією областю компетенції, полягає в розробці специфікацій тестів. Його суть - в детальному описі інструментарію і програмного коду, необхідних для виконання плану тестування.

Розробка тестів

Ця область компетенції (розробка тестів - test engineering) відповідальна за передбачені планом тестування заходи, спрямовані на знаходження і врегулювання усіх проблем якості створюваного продукту. У їх числі - робота із створення і підтримки тестових сценаріїв (test cases), розробка засобів, скриптів і документації

процесу тестування, управління щоденними билдами (daily builds), проведення на них тестів з метою чіткого визначення рівня завершеності продукту.

Звітність про тести

Ця область компетенції (звітність про тести - test reporting) забезпечує проектну групу інформацією про поточні вади в рішенні, також як і про досягнуті успіхи. Завдяки цьому проектна група має чітку картину поточного стану розробки.

Щоб усі знайдені проблеми були вирішені до остаточного випуску продукту, проводиться їх моніторинг (tracking). Регулярно здійснюється документування стану проблем (включаючи завдання по їх дозволу, пріоритети, методи врегулювання і можливі шляхи їх обходу), що дозволяє проектній групі постійно мати поточні дані про якість продукту і детальний аналіз тенденцій його зміни.

Ролевий кластер "Задоволення споживача"

Мета цього ролевого кластера (задоволення споживача - user experience) - підвищення ефективності використання продукту. Кластер складається з шести областей компетенції : загальнодоступність (accessibility), інтернаціоналізація (internationalization), забезпечення технічної підтримки (technical communications), навчання користувачів (training), зручність експлуатації (usability) і графічний дизайн (graphic design). У рамках кожної зі своїх областей компетенції ролевий кластер "Задоволення споживача" має декілька зон відповідальності, необхідних для споживчого успіху рішення. Нижче слідує їх перерахування.

Загальнодоступність

Облік вимог загальнодоступності (доступності для людей з недоліками зору, слуху і тому подібне) в дизайні рішення.

Інтернаціоналізація

Поліпшення якості і зручності експлуатації рішення в іншомовних середовищах.

Забезпечення технічної підтримки

- Проектування і розробка документації для служб підтримки (настільне керівництво співробітників служб підтримки, бази знань і так далі)
- Наповнення системи допомоги.

Навчання користувачів

Вироблення і реалізація стратегії навчання користувачів.

Зручність експлуатації (ергономіка)

- Збір, аналіз і приоритезация вимог користувачів (user requirements).
- Аналіз і обговорення дизайну продукту.
- Розробка сценаріїв і прикладів використання (usage scenarios and use cases).
- Представлення інтересів споживача в проектній групі.

Графічний дизайн

Дизайн призначеного для користувача інтерфейсу.

Загальнодоступність

Ця область компетенції забезпечує доступність продукту для людей з фізичними недоліками, вводячи концепцію і вимоги загальнодоступності в дизайн рішення. Загальнодоступність важлива з багатьох причин. Передусім, усі люди - незалежно від фізичного стану - повинні мати можливість використовувати продукти, що розробляються, і рішення. Якщо продукт не задовольняє вимозі загальнодоступності, його ринок звужується. Крім того, загальнодоступність рішення часто потрібна для виконання законодавчих актів.

Вимоги загальнодоступності повинні враховуватися на всьому протязі розробки рішення і зазвичай включають:

- Додавання пункту про загальнодоступність в специфікацію кожного елементу рішення.
- Включення розділів про загальнодоступність в систему допомоги.
- Забезпечення повноти документації по загальнодоступності.
- Забезпечення наявності документації по загальнодоступності (accessibility) в загальнодоступних (accessible) форматах.

Інтернаціоналізація

Відповідальність цієї області компетенції - поліпшення якостей і зручності експлуатації рішення в іншомовних середовищах. Вона включає забезпечення як глобалізації продукту, так і його локалізації.

Глобалізація

Глобалізація (globalization) - процес проектування і розробки рішення, при якому передбачається можливість його локалізації без зміни самого рішення і зайвої роботи. Іншими словами, випущене рішення належним чином глобализировано, якщо можлива його локалізація з мінімальними витратами.

Локалізація

Локалізація (localization) рішення включає зміни в його призначеному для користувача інтерфейсі, файлах допомоги, друкарській і електронній документації, маркетингових матеріалах і веб-сторінках. Іноді в якій-небудь іншомовній версії продукту можуть мінятися певні графічні елементи або навіть частина змісту.

Забезпечення технічної підтримки

"Забезпечення технічної підтримки " займається розробкою системи документування і підтримки рішення.

Головна функція цієї області компетенції - створення компонент підтримки, таких як система допомоги. Цей інструмент служить для користувача наставником, забезпечуючи його визначеннями ключових слів, поясненнями допущених помилок і відповідями на питання, що часто ставляться. Від системи допомоги виграє як користувач, так і організація. Користувачі отримують негайні і чіткі відповіді на насущні для них питання. Організація скорочує витрати на супровід.

Додаткове завдання "Технічної підтримки" - проектування і розробка документації до рішення, що може включати розробку керівництва по його установці, модернізації, використанню і залагоджуванню виникаючих труднощів.

Навчання

Ця область компетенції забезпечує підвищення продуктивності користувача шляхом навчання його навичкам, необхідним для ефективного використання рішення. Це досягається застосуванням дієвої стратегії навчання. Завдання розробки такої стратегії покладається на ролевий кластер "Задоволення споживача".

Стратегія навчання може бути розроблена усередині організації або доручена іншій фірмі, що спеціалізується на наданні послуг з навчання і розвитку. Незалежно від того, хто розробляє цю стратегію, відповідний процес зазвичай включає:

- Проведення аналізу профілів споживачів (користувачів) і цілей і завдань организаци-замовника.
- Визначення набору необхідних навичок.
- Розробку і реалізацію плану навчання.
- Після реалізації плану - визначення його ефективності і, якщо потрібно, внесення коригувань.

Стратегія навчання може включати один або декілька з наступних підходів: навчання під керівництвом інструктора, навчання з використанням електронних засобів, самонавчання або використання засобів допомоги в роботі. Багато організацій обирають змішаний шлях, що дозволяє людям використовувати найбільш зручний для них спосіб навчання.

Зручність експлуатації (ергономіка)

Ця область компетенції (зручність експлуатації - usability) забезпечує високу ефективністю і рівень зручності споживачів продукту, що розробляється.

Основна її діяльність - дослідження зручності експлуатації, що включає збір, аналіз і приоритезацию споживчих вимог. При належних витратах часу на розуміння потреб споживача починаючи з ранніх стадій розробки рішення проект має значно більше шансів досягти успіху в задоволенні споживчих вимог.

Інше основне завдання цієї області компетенції - розробка сценаріїв і прикладів використання. Ключова ідея тут - побачити, як створюване рішення може бути використане в цілому. Це допомагає команді розробників зрозуміти, як рішення виглядає з точки зору його споживача - в прямому і переносному сенсі. Частенько це веде до удосконалення дизайну рішення, що призводить до більшої його ефективності.

І, нарешті, ця область компетенції забезпечує аналіз і обговорення споживчих якостей рішення з користувачами. Завдяки витратам на такий зворотний зв'язок, рішення покращується і досягається вищий рівень задоволення споживача.

Графічний дизайн

Ця область компетенції забезпечує адекватний дизайн графічних елементів рішення. Її основна діяльність - управління розробкою призначеного для користувача інтерфейсу, що включає проектування графічних об'єктів, з якими повинен буде взаємодіяти користувач, їх функціональності і розробку екранів інтерфейсу.

Ролевий кластер "Управління випуском"

Мета цього ролевого кластера - безперешкодне впровадження і супровід продукту. Ця роль служить сполучною ланкою між проектною групою і групами процесів супроводу. Цей ролевий кластер включає наступні області компетенції :

- Інфраструктура (infrastructure)
- Супровід (support)
- Бізнес-процеси (operations)
- Управління випуском готового продукту (commercial release management).

"Управління випуском":

- виконує посередницькі функції між групами розробки і супроводу;
- вибирає інструментарій впровадження і сприяє його автоматизації і оптимізації;
- встановлює операційні критерії готовності продукту до випуску;
- бере участь в розробці дизайну, відповідаючи за керованість (manageability), зручність супроводу (supportability) і зручність впровадження (deployability) кінцевого продукту;
- організовує навчання персоналу супроводу;
- готує і створює систему супроводу досвідченого впровадження (pilot deployment);
- планує і забезпечує потоковий випуск продукту (deployment into production);
- контролює відповідність результатів стабілізації продукту критеріям прийнятності.

Інфраструктура

- Планування інфраструктури підприємства.
- Координація використання приміщень і кросс-географическое планування (центри даних, лабораторії, філії і периферійні офіси).
- Розробка правил і інструкцій для узгодженого управління інфраструктурою.
- Забезпечення інфраструктурного обслуговування проектної групи (сервери, стандартні образи дисків для робочих станцій, установка програмного забезпечення).

- Організація постачання проектної групи апаратним і програмним забезпеченням.
- Створення тестових і випробувальних середовищ (test and staging environments), відтворюючих робоче середовище (production environment).

Супровід

- Забезпечення зв'язку і обслуговування замовників і споживачів.
- Управління угодами про рівень послуг (SLA - service level agreement) і забезпечення їх належного виконання.
- Організація оперативного дозволу призначених для користувача проблем і нештатних ситуацій.
- Надання команді розробників зворотного зв'язку.
- Розробка процедур дії в нештатних ситуаціях.

Бізнес-процеси

- Управління обліковими записами.
- Підтримка засобів обміну повідомленнями, баз даних, телекомунікацій і мереж.
- Системне адміністрування.
- Управління брандмауером (firewall); адміністрування системи безпеки.
- Обслуговування додатків.
- Інтеграція серверів.
- Підтримка служб каталогів.

Управління випуском готового продукту

- Реєстраційні коди продукту; процес верифікації реєстрації.
- Управління ліцензіями.
- Підготовка дистрибутивних комплектів.
- Управління каналами дистрибуції продукту.
- Друкарські і електронні публікації.

Інфраструктура

Ця область компетенції включає ряд завдань, пов'язаних з організацією інфраструктури підтримки. Її характеристики схожі з характеристиками ролевого кластера "Інфраструктура" (infrastructure) MOF (Microsoft Operation Framework).

Супровід

Ця область компетенції стежить за тим, щоб створюване і впроваджуване рішення було зручним в супроводі. Її характеристики схожі з характеристиками ролевого кластера "Супровід" (support) MOF.

Бізнес-процеси

Ця область компетенції пов'язана з рядом операційних завдань, які повинні виконуватися при здійсненні проекту. Вона відповідальна за те, щоб створюване і впроваджуване рішення було узгодженим з тими, що мають до нього відношення

бізнес-процесами. Її характеристики схожі з характеристиками ролевого кластера "Бізнес-процеси" (operations) MOF.

Управління випуском кінцевого продукту

Ця область компетенції пов'язана з рядом завдань, що відносяться до випуску комерційних програмних продуктів. Основне її завдання - виведення продукту на діючі канали постачання.

Масштабування моделі проектної групи

У своїй книзі "Rapid Development" колишній розробник програмного забезпечення Майкрософтом Steve McConnell пише:

"Великі проекти вимагають організаційних методик, які формалізують і направляють інформаційний обмін. .. Усі способи організації обміну інформацією полягають в створенні деякої ієрархії, тобто створенні малих груп, що працюють як команди, і призначення представників цих груп, що йде за цим, для взаємодії один з одним і керівною ланкою".

Модель проектної групи MSF пропонує розбиття великих команд (більше 10 чоловік) на малі багатопрофільні групи напрямів (feature teams). Ці малі колективи працюють паралельно, регулярно синхронізуючи свої зусилля.

Крім того, коли ролевому кластеру потрібно багато ресурсів, формуються т.з. функціональні групи (functional teams), які потім об'єднуються в ролеві кластери.

Групи напрямів

Кожен ролевий кластер проектної групи має одну або декілька складових, що утворюють ієрархічну структуру (бажано, максимально просту). Наприклад, тестувальники підзвітні менеджерові тестування.

У цю структуру вплетені так звані групи напрямів (feature teams). Це компактні міні-команди, що утворюють матричну організаційну структуру. У них входять по одному або декілька членів з різних ролевих кластерів. Такі команди мають чітко певне завдання і відповідальні за питання, що все відносяться до неї, починаючи від проектування і складання календарного графіку. Наприклад, може бути сформована спеціальна група проектування і розробки сервісів друку.

У "Rapid Development" Steve McConnel писав:

> "Перевагою групи напряму є баланс відповідальності і повноважень. Таку команду легко наділити повноваженнями, оскільки вона містить представників кожної із залучених сторін. Через це група враховуватиме при ухваленні рішень усі істотні точки зору, і навряд чи виникнуть ситуації, що вимагають перегляду її рішень.
>
> З тієї ж причини команда стає відповідальнішою. У її розпорядженні є усі люди, необхідні для здійснення вірних кроків. Якщо члени групи напряму приймають погане рішення, то в цьому вони не можуть винити нікого, окрім самих себе. І ця команда добре збалансована. Ви не хотіли б, щоб розробники, маркетологи або тестувальники отримали право вирішального го-

лосу. Але від групи напряму ви зможете отримати зважене судження, оскільки вона включає представників усіх вищеназваних категорій".

Рисунок 1. Групи напрямів

Функціональні групи

Функціональні групи - це групи, існуючі усередині ролевих кластерів. Вони створюються у великих проектах, коли необхідно згрупувати працівників усередині ролевих кластерів по їх областях компетенції. Наприклад, в Майкрософті група управління продуктом зазвичай включає фахівців з планування продукту і фахівців з маркетингу. Як перша, так і друга сфери діяльності відносяться до управління продуктом: одна з них сосредотачивается на виявленні якостей продукту, що дійсно цікавлять замовника, а друга - на інформуванні потенційних споживачів про переваги продукту.

Аналогічно, в команді розробників можливе угрупування співробітників відповідно до призначення модулів, що розробляються ними, : інтерфейс користувача, бізнес-логіка або об'єкти даних. Часто програмістів розділяють на розробників бібліотек і розробників рішення. Програмісти бібліотек зазвичай використовують низькорівневу мову C і створюють повторно використовувані компоненти, які можуть згодитися усьому підприємству. Творці ж рішення зазвичай сполучають ці компоненти і працюють з мовами більше високого рівня, такими як, наприклад Microsoft® Visual Basic®.

Часто функціональні групи мають внутрішню ієрархічну структуру. Наприклад, менеджери програми можуть бути підзвітні провідним менеджерам програми, які у свою чергу звітують перед головним менеджером програми. Подібні

структури можуть також з'являтися усередині областей компетенцій. Але важливо пам'ятати, що ці ієрархії не повинні затінювати модель команди MSF на рівні проекту в цілому. Цілі усіх ролей залишаються тими ж самими, також як і відповідальність перед проектною командою за досягнення цих цілей.

Об'єднання ролей

Хоча модель проектної групи складається з шести ролей, це не означає, що команда обов'язково повинна налічувати не менше шести чоловік. Модель не вимагає призначення окремого співробітника на кожен ролевий кластер. Сенс полягає в тому, що в команді має бути представлені усі шість якісних цілей. Зазвичай, виділення як мінімум однієї людини на кожен ролевий кластер забезпечує повноцінна увага до інтересів кожної з ролей, але це економічно виправдано не для усіх проектів. Частенько члени проектної групи можуть об'єднувати ролі.

У малих проектних групах об'єднання ролей є необхідним. При цьому повинні дотримуватися два принципи. По-перше, роль команди розробників не може бути об'єднана ні з якою іншою роллю. Розробники - це творці проекту, і вони не повинні відволікатися від свого головного завдання. Наділ розробників додатковими обов'язками лише робить вірогіднішим вихід з календарного графіку проекту.

Другий принцип - це уникнення поєднання ролей, що мають зумовлені конфлікти інтересів. Наприклад, управління продуктом і управління програмою мають ті, що суперечать один одному інтереси і, отже, не повинні об'єднуватися. Менеджмент продукту має мету задовольнити замовника, тоді як менеджмент програми забезпечує готовність продукту у відведений час і у рамках наявного бюджету. У разі поєднання цих ролей виникає ризик, що зміна, що зажадалася замовником, або не буде розглянута з належною увагою, або буде прийнята без належного аналізу його впливу на проект. Представлення цих ролей різними людьми в проектній команді забезпечує рівновага двох точок зору, що суперечать. Те ж саме відноситься до спроби об'єднання ролей розробки і тестування.

	Управление продуктом	Управление программой	Разработка	Тестирование	Удовлетворение потребителя	Управление выпуском
Управление продуктом		-	-	+	+	±
Управление программой	-		-	±	±	+
Разработка	-	-		-	-	-
Тестирование	+	±	-		+	+
Удовлетворение потребителя	+	±	-	+		±
Управление выпуском	±	+	-	+	±	

+ Допустимо ± Нежелательно - Нельзя

Рисунок 2. Об'єднання ролей в малих проектах

Приведена таблиця показує невдалі (не "можна") і можливі ("допустимо") поєднання ролей. Але, як і в будь-якій іншій командній діяльності, відповідна комбінація ролей залежить від самих членів команди, їх досвіду і професійних навичок.

Знак "-" на перетині стовпця з рядком означає, що поєднання відповідних ролей неприпустимо, за винятком випадків, коли це абсолютно необхідно, і коли пов'язані з цим риски можуть бути зменшені ефективними планами їх запобігання і пом'якшення наслідків. Ясно, що є різні рівні конфліктів між ролями, що робить модель проектної групи динамічною і утрудняє об'єднання ролей. Проте на практиці поєднання ролей зустрічається нерідко. І якщо проектна група робить це обдумано і управляє пов'язаними з таким об'єднанням рисками, виникаючі проблеми будуть мінімальними.

Зовнішня координація - на кому лежить відповідальність?

Для досягнення успіху проектній групі необхідно взаємодіяти, обмінюватися інформацією і координувати зусилля з іншими (зовнішніми) групами. Їх діапазон починається від замовників і споживачів і закінчується іншими командами розробників. В більшості випадків, відповідно до укладеного контракту, проектна група повинна звітувати безпосередньо перед замовником про рішення, що розробляється. І хоча концепція команди соратників (команди рівних) має на увазі розподілену відповідальність за успішне створення рішення, важливо чітко відрізняти її від схеми звітності, визначуваної комунікаційним планом. Як замовник, так і команда повинні знати, хто конкретно відповідальний за надання необхідної інформації.

Мал. 4 ілюструє типовий розподіл відповідальності за координацію як з питань бізнесу, так і з питань технологій. "Управління програмою", "Управління

продуктом", "Задоволення споживача" і "Управління випуском" є основними зовнішніми координаторами. Ці ролі вирішують як внутрішні, так і зовнішні завдання, тоді як розробники і тестувальники сфокусовані на внутрішніх завданнях і не беруть участь в зовнішньому обміні інформацією.

Це не означає, що розробники і тестувальники мають бути ізольовані від зовнішнього світу. Контакти з організацією замовника і споживачами продукту можуть істотно допомогти співробітникам реально сконцентруватися на потребах замовника, що є однією з цілей MSF, особливо на ранніх стадіях проекту. Проте ці контакти не повинні виходити за рамки неформального спілкування.

Ця діаграма показує узагальнену картину. Зазвичай проектні команди повинні координувати зусилля з множиною зовнішніх груп, контролюючих якість, фінанси і юридичні питання. Важливо, щоб способи контакту із зовнішніми групами були чіткими і зрозумілими, і щоб розробники і тестувальники продовжували працювати відособлено, не відволікаючись постійно на зайві подразники.

Рисунок 3.

Рисунок 4. Комунікації

Варто підкреслити, що хоча зовнішня комунікація може надати корисну інформацію і рекомендації, ні окремі члени команди, ні уся команда в цілому не може змінити пріоритети проекту і прийняті рішення про бюджет, терміни і об'єм

робіт проекту. Такі зміни є прерогативою замовника або спонсора проекту, а проектна команда лише реалізує їх. Це приклад того, як команда рівноправних партнерів (соратників) співіснує з організаційними ієрархічними структурами.

Висновок

Модель проектної групи MSF не забезпечує успіх сама по собі. Є багато інших чинників, що визначають успіх або невдачу проекту, але структура проектної групи, безумовно, вносить істотний вклад.

Пояснюючи це, Steve McConnell в "Rapid Development" пише:

"Навіть за наявності компетентних зацікавлених і працелюбних людей невірна структура команди здатна звести нанівець їх зусилля, замість того щоб привести їх до успіху. Погана структура команди може послужити причиною збільшення часу розробки, зниження якості, пониження морального духу, підвищення плинності кадрів і, зрештою, привести до відміни проекту".

ЛІТЕРАТУРА

Рекомендована

1. Швабер К. Скрам. Гибкое управление продуктом и бизнесом. Альпина Паблишер, 2019. 236с.
2. Книберг Г. Scrum и XP: заметки с передовой. Как мы делаем Scrum. URL: http://scrum.org.ua/wp-content/uploads/2008/12/scrum_xp-from-the-trenches-rus-final.pdf (дата звернення 20.02.2020)
3. Амблер С. Гибкие технологии: экстремальное программирование и унифицированый процесс разаботки. Библиотека программиста. С-Петербург : Питер, 2005. 412 с.
4. Мартин Р. Быстрая разработка программ: принципы, примеры, практика. Пер. с англ. Москва : Вильямс, 2009.752 с.
5. Бек К. Экстремальное программирование: разработка через тестирование. Библиотека программиста. С-Петербург : Питер, 2003. 224 с.
6. Липаев В.В. Программная инженерия. Методологические основы, Гос. ун-т Высшая школа экономики. Москва : ТЕИС, 2006, 608 с.
7. Соммервиль И. Инженерия программного обеспечения, Москва : "Вильямс", 2004. 448с.
8. Орлов С.А. Технологии разработки программного обеспечения. С-Петербург : Питер, 2002. 464 с.
9. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. С-Петербург : Питер, 2001. 368 с.
10. Маккарти М., Маккарти Д. Правила разработки программного обеспечения. С-Петербург : Питер, 2007. 240 с.

Використана

11. Влиссидес Д. Применение шаблонов проектирования. Дополнительные штрихи.: Пер. с англ. Москва : Вильямс, 2003. 144 с.
12. Вендров А.М. Проектирование программного обеспечения экономических информационных систем. Москва : Финансы и статистика, 2006. 544 с.
13. Гецци К.Джазайери М., Мандриоли Д. Основы инженерии программного обеспечения. 2-е изд. Перев. с англ. С Петербург, 2005. 832 с.
14. Ларман К. Применение UML и шаблонов проектирования. 2-е изд. С-Петербург : Вильямс, 2002. 624 с.
15. Буч Г., Максимчук Р., Энгл М., Янг Б., Коналлен Д., Хьюстон К. Объектно-ориентированный анализ и проектирование с примерами приложений (UML2). Третье издание. Москва : Вильямс, 2008. 720 с.

16. Р.Т.Фатрепп, Д.Ф.Шафер, Л.И.Шафер, Управление программными проектами: достижение оптимального качества при минимуме затрат. Москва : Вильямс. 2003. 1125 с.
17. Сидоров М. О., Мендзебровский І. Б., Орехов А. А. «Професійна практика програмної інженерії» - досвід викладання. Інженерія програмного забезпечення. 2010. № 2. С. 54-60.
18. Кримінальний кодекс України (ККУ). 2018. URL: https://urist-ua.net/кодекси/кримінальний_кодекс_україни / (дата звернення 20.02.2020).
19. Стандарт вищої освіти України : перший (бакалаврський) рівень, галузь знань –12 Інформаційні технології, спеціальність – 121 Інженерія програмного забезпечення. Затверджено і введено в дію наказом Міністерства освіти і науки України від 29.10.2018 № 1166.

Інформаційні ресурси

20. The Computing Curricula 2020 (CC2020) Project. URL: https://www.cc2020.net/ (дата звернення 20.02.2020).
21. Software Engineering Curricula 2015. URL: https://www.acm.org/binaries/content/assets/education/ce2016-final-report.pdf (дата звернення 20.02.2020).

Навчально-методичне видання
(українською мовою)

Безверхий Анатолій Ігорович
Вербицький Володимир Григорович

ПРОФЕСІЙНА ПРАКТИКА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Навчально-методичний посібник
для здобувачів ступеня вищої освіти бакалавра
спеціальності «Інженерія програмного забезпечення»
освітньо-професійної програми «Програмне забезпечення систем»

Рецензент *М.Ю. Пазюк*
Відповідальний за випуск *В.Г. Вербицький*
Коректор *А.І. Безверхий*