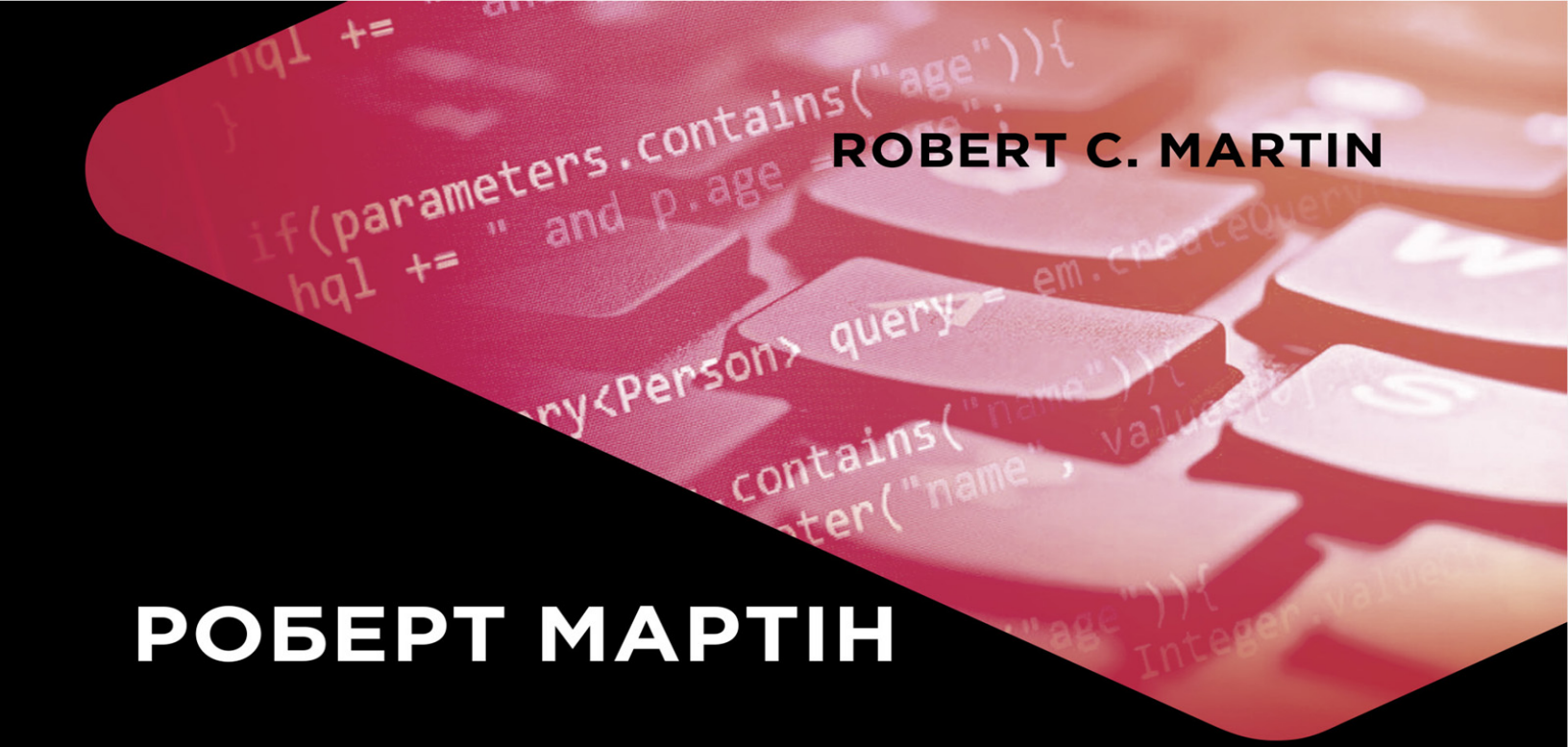
ЗМІСТ

Чистий кодер. Чистий кодер

Про книгу

Роберт Мартін, також відомий як Дядечко Боб,— знакова постать у світі розробки ПЗ, блискучий професіонал, міжнародний консультант, один із тих, хто створював 2001 року всесвітньо відомий Agile-маніфест. Кожна його наступна книжка — джерело безцінного досвіду.

У «Чистому кодері» автор викладає свої очікування від професійного розробника у всіх можливих аспектах: із погляду управлінських взаємодій, тайм-менеджменту, зовнішнього тиску, співпраці в команді та вибору відповідних інструментів. Не оминає він і питань трудової етики. Ви також ді-знаєтеся, що навіть гуру програмування далеко не завжди є професіоналами. Натомість Роберт Мартін пропонує читачеві шлях до справжнього розробницького професіоналізму — і робить це надзвичайно цікаво й дотепно.



ROBERT C. MARTIN

РОБЕРТ МАРТІН

Чистий кодер

КОДЕКС ПОВЕДІНКИ ДЛЯ ПРОФЕСІЙНИХ РОЗРОБНИКІВ

ВИДАВНИЧИЙ ДІМ
ФАБУЛА
#PRO

Роберт Мартін

ЧИСТИЙ КОДЕР:
Кодекс поведінки для професійних розробників

Видавничий дім «Фабула»
2023

ISBN 978-617-522-110-5 (epub)

Електронна версія створена за виданням:

Мартін Роберт С.

М29 Чистий кодер: Кодекс поведінки для професійних розробників / пер. з англ. Г. Якубовська. — Харків : ВД Фабула, 2023. — 256 с.

ISBN 978-617-522-082-5

Роберт Мартін, також відомий як Дядечко Боб,— знакова постать у світі розробки ПЗ, блискучий професіонал, міжнародний консультант, один із тих, хто створював 2001 року всесвітньо відомий Agile-маніфест. Кожна його наступна книжка — джерело безцінного досвіду.

У «Чистому кодері» автор викладає свої очікування від професійного розробника у всіх можливих аспектах: із погляду управлінських взаємодій, тайм-менеджменту, зовнішнього тиску, співпраці в команді та вибору відповідних інструментів. Не оминає він і питань трудової етики. Ви також дізнаєтеся, що навіть гуру програмування далеко не завжди є професіоналами. Натомість Роберт Мартін пропонує читачеві

шлях до справжнього розробницького професіоналізму — і робить це надзвичайно цікаво й дотепно.

УДК 004.41

Шановний читачу!

Спасибі, що придбали цю книгу.

A
Налаштування

1978 року я працював у *Teradyne* над телефонною тестовою системою, про яку вже згадував раніше. Система складалася приблизно з 80 тисяч рядків коду асемблера M365. Початковий код зберігався на магнітних стрічках.

Ті стрічки нагадували восьмидоріжечні стерео-касети, що були шалено популярні в 1970-х роках. Кінець і початок стрічки були склеєні, а стрічковий накопичувач міг перемотувати її лише в одному напрямку. Стрічки в касетах мали довжину 10, 25, 50 та 100 футів. Що довша була стрічка, то більше часу забирало перемотування, бо накопичувачу

доводилося просто рухатися вперед до «точки завантаження». На перемотування стофутової стрічки до точки завантаження витрачалося близько п’яти хвилин, тому ми дуже обачно підходили до вибору довжини стрічок{51}.

На логічному рівні стрічки поділялися на файли. На одну стрічку можна було записати стільки файлів, скільки вона могла вмістити. Для того щоби знайти потрібний файл, належало завантажити стрічку, а потім послідовно читати файли, поки потрібний не буде знайдений. На дошці в лабораторії завжди висіла роздруківка каталогу з вихідним кодом — за нею ми визначали скільки файлів потрібно пропустити, щоби знайти потрібний.

На полиці в лабораторії також лежала стофутова еталонна копія стрічки з вихідним кодом і позначкою «Майстер». Аби відредагувати файл, ми встановлювали в один накопичувач еталонний екземпляр, а в інший — десятифутову порожню (робочу) стрічку. Еталонна стрічка промотувалася до потрібного файлу, після чого файл копіювався на робочу стрічку. Тоді ми перемотували обидві стрічки на початок, й еталонна стрічка поверталася на полицю.

Окрім цього, на дошці в лабораторії була присутня спеціальна роздруківка еталонної стрічки «Майстер». Коли ми створювали копії файлів, котрі належало відредагувати, то встромляли в роздруківку кольорову кнопку поруч з ім’ям цього файлу. Ось так і здійснювався контроль за внесенням змін!

Редагування проводилося в екранному режимі. Ми користувалися дуже вдалим текстовим редактором ED-402 — близьким аналогом vi. Сторінка читалася зі стрічки, ми редагували її вміст, записували назад і бралися за наступну. Сторінка зазвичай складалася із приблизно 50 рядків коду. Ми не мали можливості зазирнути вперед, аби побачити наступні сторінки, і не могли повернутися назад — до вже відредагованих сторінок. Тому ми використовували лістинги.

У лістингах позначалися всі зміни, після чого файли редагувалися відповідно до позначок. *Ніхто* не писав і не змінював код спонтанно! Це було би чимось на кшталт самогубства. Після внесення змін до всіх

файлів, що потребували редагування, ми поєднували їх із вмістом еталонного екземпляра. Отриману стрічку використовували для здійснення компіляції і тестування.

Завершивши тестування і переконавшись, що зміни працюють, ми дивилися на роздруківку. Якщо на ній не було нових кнопок, робоча стрічка просто перейменовувалась на еталонну, а всі «відпрацьовані» кнопки знімалися з дошки. Якщо ж на дошці з’являлися нові кнопки, ми прибирали всі попередні і передавали робочу стрічку тому, хто їх виставив, для злиття результатів.

У команді було лише троє розробників, і кожен із них мав кнопки відповідного кольору. Тому ми завжди могли визначити, хто взяв на редагування ті чи інші файли. А оскільки всі ми працювали в одній лабораторії і постійно спілкувалися один з одним, поточний стан кодової бази постійно зберігався в наших головах. Найчастіше навіть роздруківка виявлялася зайвою, і ми обходилися без неї.

Інструменти

Сучасним розробникам доступні найрізноманітніші інструменти програмування. Від деяких, на мою думку, варто триматися якнайдалі, але є серед них і такі, якими повинен упевнено володіти будь-який розробник. У цьому розділі описаний мій особистий поточний інструментарій. Я не наводжу повного опису всіх представлених інструментів, тому огляд не є вичерпним, і розповідаю лише про те, що використовую сам.

Управління вихідним кодом

Що стосується управління вихідним кодом, то, як правило, краще використовувати програми з відкритим кодом. Чому? Бо вони пишуться розробниками для розробників. Інакше кажучи, розробники пишуть програми з відкритим кодом для себе, коли їм необхідне працездатне рішення.

Існує багато дорогих комерційних («корпоративних») систем контролю за версіями. На мою думку, ними спокушаються не стільки розробники, скільки керівники, менеджери та «інструментальні групи». Список функцій таких програм завжди виглядає вражаюче. Але, на жаль, у ньому часто не виявляється того, що дійсно потрібне розробникам. До того ж, головною проблемою для користувачів зазвичай стає їх *швидкість*.

«Корпоративні» системи керування вихідним кодом

Цілком можливо, що ваша компанія вже вклала цілий статок у «корпоративну» систему керування вихідним кодом. У такому разі — прийміть мої співчуття. Звісно, із політичних міркувань ви не зможете просто бовкнути: «Дядечко Боб каже, що цим лайном не треба користуватися». Однак існує простий вихід.

Ви можете реєструвати вихідний код у «корпоративній» системі наприкінці кожної ітерації (кожні два тижні або близько того), а в середині ітерацій використовувати систему з відкритим кодом. Усі будуть задоволені, ви не порушите корпоративних настанов, а ваша продуктивність залишиться високою.

Песимістичне та оптимістичне блокування

У 1980 роках песимістичне блокування здавалося гарною ідеєю. Зрештою, найпростіший шлях до розв’язання проблем паралельного оновлення — «розпаралелювання». Якщо я редагую файл, то вам краще

його не чіпати. Система кольорових кнопок, яку ми використовували наприкінці 1970-х, була своєрідним механізмом песимістичного блокування. Якщо файл був позначений певною кнопкою, інші не повинні були його редагувати.

Звісно, песимістичне блокування має свої недоліки. Якщо заблокувати файл і піти у відпустку, то решта користувачів, які забажають працювати із цим файлом, опиняться в безвиході. Навіть якщо файл залишиться заблокованим на день-два, це затримає роботу інших людей.

Сучасні інструменти значно краще справляються зі злиттям паралельно редагованих вихідних файлів. Адже це нетривіальне завдання: програма аналізує два різні файли разом із попередниками цих двох файлів, а потім застосовує відповідні стратегії для визначення способу інтеграції паралельних змін. І мушу сказати, вона робить це добре.

Отже, час песимістичного блокування минув. Тепер нам не потрібно робити блокування файлів під час редагування. Ба більше, нам узагалі не потрібно турбуватися про блокування окремих файлів — ми запитуємо відразу всю систему й редагуємо потрібні файли.

Коли все буде готове до реєстрації змін, виконується операція оновлення. Вона повідомляє нам, чи не було більш ранньої реєстрації змін, здійсненої іншими користувачами, виконує автоматичне злиття більшості змін, знаходить конфлікти і допомагає вирішити їх. Після цього об’єднаний код додається в кодову базу.

Далі я більш докладно висвітлю роль автоматизованих тестів і безперервної інтеграції у цьому процесі. А тут варто підкреслити ось що: код, який не пройшов всіх тестів, за жодних умов не повинен додаватися в кодову базу. *Ніколи*.

CVS/SVN

Одна із традиційних систем управління вихідним кодом — CVS — була гарною для свого часу, але в сучасних проєктах уже починає відставати. Хоча CVS відмінно працює з окремими файлами і теками, вона не дуже

добре справляється з перейменуванням файлів та видаленням тек. Що вже казати про підтеки… І на цьому — завіса. Що менше говорити про це, то краще.

Із іншого боку, система Subversion працює дуже добре. Вона дозволяє отримати доступ до всієї системи за одну операцію. У ній легко виконуються операції оновлення, злиття та закріплення змін. За відсутності розгалужених змін, системи SVN доволі прості в управлінні.

Розгалуження

До 2008 року я уникав будь-яких форм розгалуження (branching), окрім найпростіших. Якщо розробник створює гілку, то ця гілка має бути повернена до основної лінії ще до кінця ітерації. При цьому я так суворо ставився до розгалуження, що воно дуже рідко застосовувалося у тих проєктах, у яких я брав участь.

Якщо ви використовуєте SVN, то я, як і раніше, вважаю, що це вдала політика. Однак останнім часом з’явилися нові інструменти, що цілковито змінили ситуацію. Я маю на увазі *розподілені* системи управління вихідним кодом. У цій категорії моїм улюбленим інструментом є git. Зараз я розповім про цю систему докладніше.

git

Я почав використовувати git наприкінці 2008 року. Ця система цілковито змінила мій підхід до управління вихідним кодом. Пояснення того, чому ця програма так відчутно змінила правила гри, виходять за межі цієї книжки. Проте порівняння рис. Д.1 із рис. Д.2 набагато красномовніше будь-яких слів.

На рис. Д.1 показаний перебіг розробки проєкту FitNesse за кілька тижнів під керуванням SVN. Ви бачите наслідки моєї жорсткої політики відмови від розгалуження. Ми просто його не використовували. Натомість у головній гілці часто виконувалися операції оновлення, злиття та закріплення.

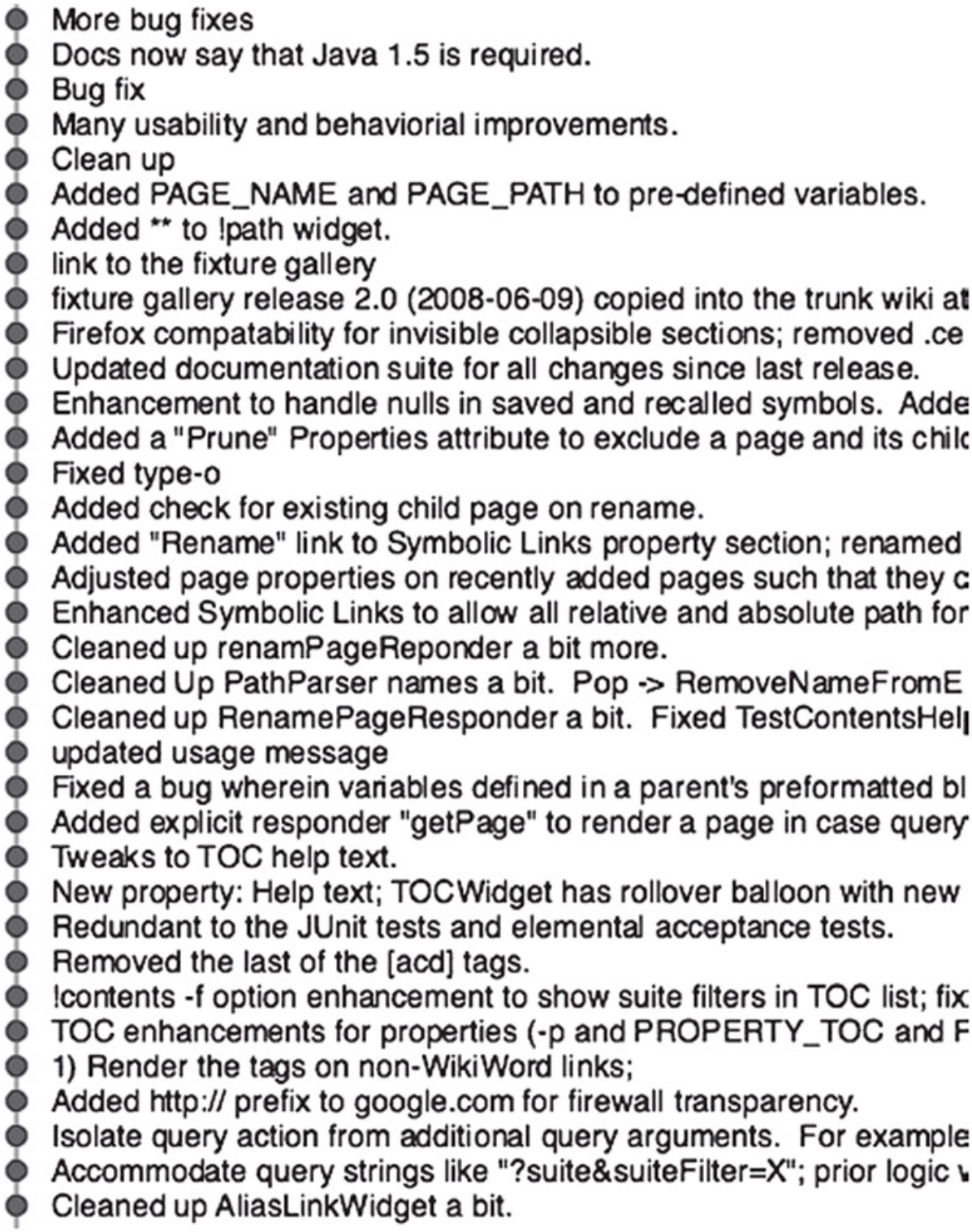


Рис. Д.1. Проєкт FitNesse під керівництвом subversion

На рис. Д.2 показана структура кількох тижнів розробки того самого проєкту з використанням git. Як бачите, операції розгалуження та злиття відбуваються постійно. І річ не в тім, що я чомусь змінив своє ставлення до розгалуження. Просто такий робочий процес став очевидним та ефективним. Окремі розробники створюють гілки з дуже коротким терміном життя, а потім поєднують їх із результатами колег тоді, коли вважають за потрібне.

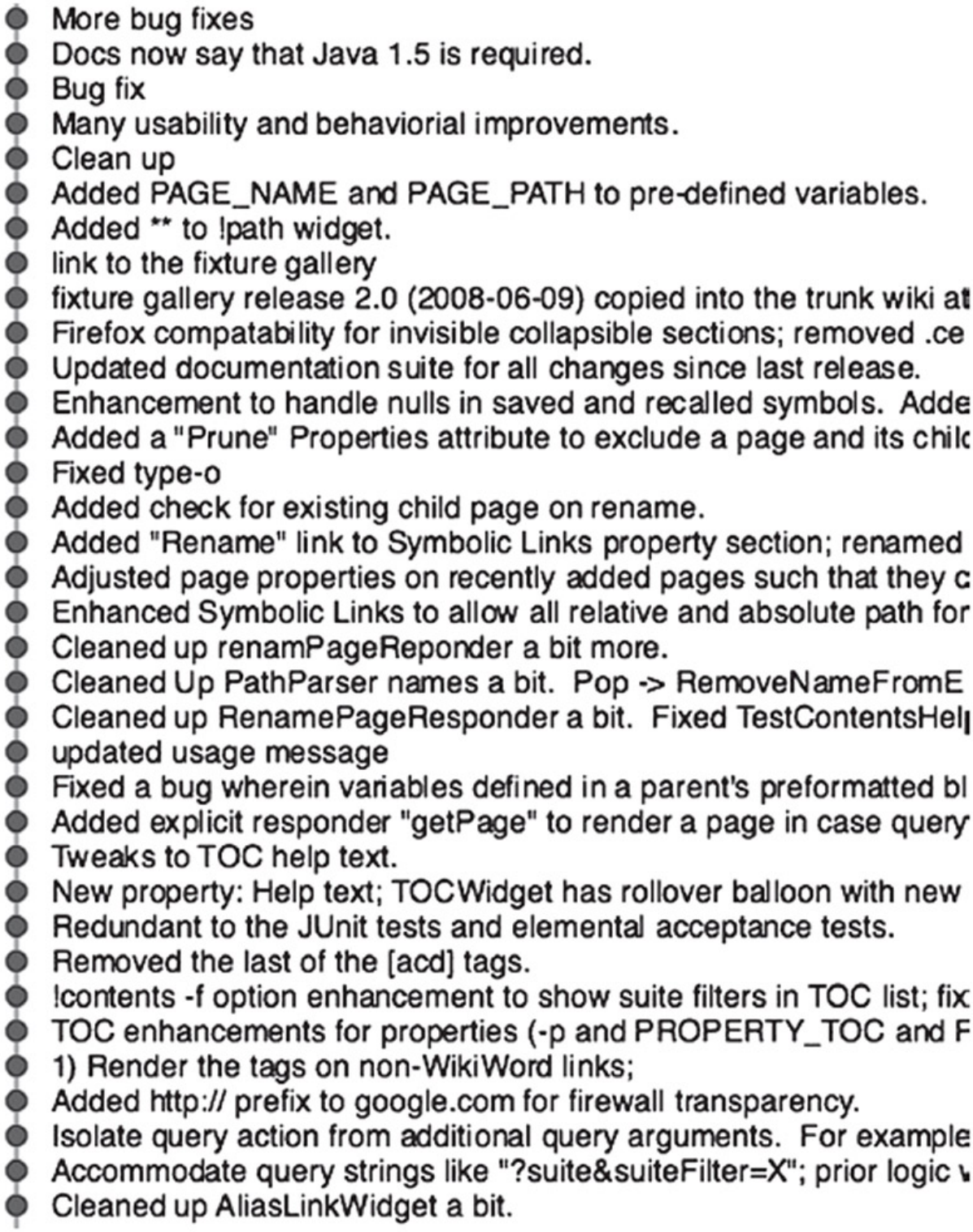


Рис. Д.2. Проєкт FitNesse під керівництвом git

Також зауважте, що на рисунку не видно «справжньої» головної гілки. Річ у тім, що її *просто немає*. Розробники зберігають на своїх локальних комп’ютерах копії *всієї* історії проєкту. Вони вносять зміни і реєструють їх у своїй локальній копії, а потім синхронізують із копіями колег у разі потреби.

Щоправда, я підтримую спеціальний «золотий репозиторій», у якому зберігаються всі опубліковані версії та внутрішні збірки. Але називати його «головною гілкою» означало би не розуміти суті справи. Насправді це лише зручна «світлина» всієї історії, що зберігається локально кожним розробником.

Якщо ви не зрозуміли щось зі сказаного, це нормально. Спочатку git викликає чимало труднощів. До того, як працює ця система, потрібно звикнути. Але запевняю вас: git та інші подібні системи — це майбутнє керування вихідним кодом.

IDE/Редактор

Ми, розробники, проводимо більшу частину часу за читанням та редагуванням коду. Інструменти, що ми використовуємо для цього, значно змінилися за минулі роки. Деякі з них мають неймовірну міць, а деякі майже не змінилися з 1970-х років.

VI

Здається, що доба використання vi як основного редактора для розробки давно минула. У наші дні з’явилися інструменти, що значно перевершують vi за своїми можливостями, та інші нескладні текстові редактори того самого типу. Однак останнім часом спостерігається помітний сплеск популярності vi. Це пояснюється його простотою, зручністю використання, швидкістю та гнучкістю. Хоча vi й поступається Emacs або Eclipse за широтою можливостей, він залишається швидким та потужним редактором.

Водночас, я вже не досвідчений користувач vi. Колись мене називали «богом» vi, але цей час давно минув. Я використовую vi час від часу, коли потрібно швидко відредагувати текстовий файл. Нещодавно я застосовував його для швидкої зміни вихідного файлу Java у віддаленому режимі. Але обсяг повноцінного коду, написаного мною у vi за останні 10 років, істотно зменшився.

Emacs

Emacs досі залишається одним із найпотужніших редакторів і, ймовірно, залишиться ним ще кілька найближчих десятиліть. Його потужність забезпечується внутрішньою моделлю з урахуванням lisp. У категорії інструментів редагування загального призначення жоден інший редактор навіть віддалено не може з ним конкурувати. З іншого боку, на мою думку, Emacs не може повноцінно конкурувати зі спеціалізованими інтегрованими середовищами розробки (IDE), що

зараз займають провідні позиції. Редагування коду не є редагуванням загального призначення.

У 1990 роках я був фанатом Emacs. Я просто не розглядав інших варіантів. Тодішні редактори з керуванням мишею були просто кумедними іграшками, до яких жоден розробник не ставився серйозно. Але на початку 2000-х я познайомився з IntelliJ — моїм фаворитом серед IDE, і вже не озирався назад.

Eclipse/IntelliJ

Я — відданий користувач IntelliJ. Я люблю цю систему й використовую її для написання коду на Java, Ruby, Clojure, Scala, Javascript та багатьох інших мовах. Вона була створена розробниками, які розуміють, що саме потрібно колегам під час написання коду. За минулі роки IntelliJ майже ніколи не підводила мене, а враження від неї завжди залишалися позитивними.

Середовище Eclipse за своєю потужністю і масштабом можна порівняти з IntelliJ. Ці два середовища якісно перевершують Emacs за можливостями редагування Java-коду. У цій категорії існують інші IDE, але я не згадую їх тут, бо в мене немає безпосереднього досвіду їх використання.

Ці інтегровані середовища відрізняються від таких інструментів, як Emacs, насамперед надзвичайно багатими можливостями маніпулювання кодом. Наприклад, IntelliJ дозволяє отримати суперклас із класу всього однією командою. Ви можете перейменовувати змінні, отримувати методи, перетворювати наслідування на композицію — і це далеко не весь перелік.

Із цими інструментами редагування коду переходить від рівня рядків і символів до більш складних маніпуляцій. Розробник при цьому думає не про кілька наступних символів і рядків, що він збирається ввести, а про наступні перетворення. Коротше кажучи, модель програмування у цих нових системах значно змінюється і стає більш продуктивною.

Звісно, за широту можливостей доводиться платити. Освоєння інтегрованих середовищ вимагає багато часу і зусиль, а фаза початкової підготовки проєкту стає досить помітною. Крім того, ці інструменти дуже вимогливі — для їхньої роботи необхідні значні обчислювальні потужності.

TextMate

Редактор TextMate потужний і невимогливий до ресурсів. Щоправда, він не вміє виконувати такі приголомшливі маніпуляції, на які здатні IntelliJ й Eclipse. У нього немає потужного lisp-ядра та бібліотеки Emacs. Він не має швидкості та гнучкості vi. З іншого боку, його можна швидко освоїти, а всі операції є інтуїтивно зрозумілими.

Я використовую TextMate лише іноді, особливо для спонтанного програмування на C++. У великому проєкті я би скористався Emacs, але для невеликих завдань на C++ мені просто ліньки з ним возитися.