

☐

I'm not robot


reCAPTCHA

Continue

Program de unit pdf uri

Uri Wilensky is the Lorraine H. Morton professor of Learning Sciences, Computer Science and Complex Systems at Northwestern University. He is the co-director of the joint PHD program in Computer Science and Learning Sciences (CSLS) . He is the founder and current director of the Center for Connected Learning and Computer-Based Modeling(CCL) and a co-founder of the Northwestern Institute on Complex Systems(NICO). He is also a faculty member in Cognitive Science, Philosophy, the program in Technology and Social Behavior, the Buffett center, the CIERA center and the Segal Design Center research council. He is the author of the NetLogo agent-based modeling software which his lab actively maintains and improves. It is the most widely used agent-based modeling software, and has hundreds of thousands of users worldwide using it for both research and education. Wilensky has published more than 300 scientific articles and more than 400 published agent-based models. Wilensky has received numerous awards including the 2016 ISDE prize for excellence in design for his work on NetLogo Quick navigation Bio, Textbook, Current Research, Previous & Ongoing Research, Courses, Colleagues & Students, Contact Info, Personal Stuff Textbook Wilensky, U., & Rand, W. (2015). An introduction to agent-based modeling: Modeling natural, social and engineered complex systems with NetLogo. Cambridge, MA: MIT Press. Research Overview CCL Papers & Publications NetLogo modeling tool. NIH NIDA. Center for Prevention Implementation Methodology for Drug Abuse and HIV NSF-Funded Research Projects: NSF STEM+C.A Whole-School Model for Integrating Computational Thinking in High School Science and Mathematics. NSF ITEST. Group-Based Cloud Computing for STEM Education. NSF EAGER: MAKER: A Cultural Framework for Equity in Maker Practices NSF STEM+C. CT-ifying the high school science curriculum to broaden participation in computational science NSF STEM+C. Engaging students in building theories of scientific phenomena: A comparison of aggregate pattern-based and agent-based approaches Lyle-Spencer Foundation Award. Broadening Participation in a Computational Future: Casting a Wide Net. NSF CyberLearning: LevelSpace: Modeling in Levels NSF EAGER: A low-cost integrated agent-based modeling and physical NSF-CNS-#1138461: CT-STEM: Casting a Wide Net: Embedded Computational Thinking NSF-DRL-#1109834: BEAGLE: Simulated Evolution Participatory Simulations: Network-Based Design for Systems Learning in Classrooms NSF-CISE: Computational Thinking in STEM NSF CyberLearning: InquirySpace: Technologies in Support of Student Experimentation NSF-DRL: Emerging Research-Empirical--Simulated Evolution: Developing a Framework for Computer-Based Modeling and Simulation Activities in the Classroom NSF-CE21: Casting a Wide Net: Computational Thinking ABMPLUS: Enabling Modeling and Simulation-Based Science in the Classroom NSF-DRK12: Enabling Modeling and Simulation-based Science in the classroom: Integrating agent-based models, real world sensing and collaborative networks NSF-REC-#0003285: Connected Mathematics: Making Sense of Complex Phenomena through Building Object-Based Parallel Models NSF-DRL-#1020101: ABMPLUS: Enabling Modeling and Simulation-Based Science in the Classroom NSF-DRL-#1020101: InquirySpace: Integrating Agent-Based Models with Statistical Analyses NSF-DRL-#1109834: Learning Evolution through Model-Based Inquiry BEAGLE: Simulated Evolution CT-STEM: Casting a Wide Net: Embedded Computational Thinking InquirySpace: Integrating Agent-Based Models with Statistical Analyses Integrated Simulation and Modeling Environment Learning Evolution through Model-Based Inquiry Modeling Across the Curriculum NAE/Spencer Project -- Developing Reasoning About Parallel Processes StarLogoT modeling tool NSF Curricular Projects Problab: Simulation-based curriculum on Probability & Statistics The GasLab Project: Simulation-based curriculum on kinetic theory of gases ConnectedChemistry: Simulation-based curriculum on introductory Chemistry The EACH Project: Understanding the evolution of cooperation Mind the Gap: An agent-based participatory simulation (ABPS) curricular unit on wealth inequality GenEvo: A computational systems biology curriculum for high school students to engage with cutting-edge ideas in modern biology Courses Currently Being Taught Constructionist Learning The design of Technological tools for thinking and learning: A Constructionist Approach Designing and Constructing Models with Multi-Agent Languages Images from class poster session Past Courses Complexity Science Turtle Geometry Practicum in Learning Environment Design Advanced Technology Design Technological Tools for Thinking and Learning Mathematics Learning Environments Mathematics & Science Learning Environments Mathematics Practices Intellectual Development Learning through Computer-Based Projects Structure and Interpretation of Computer Programs Colleagues & Students Seymour Papert Tribute Panel Colleagues Students Personal Stuff Passions Photos By date Adventures Daniel Ethan Contact Info Email: uri@northwestern.edu Phone: 847-467-3818 Fax: 847-491-8999 Offices: Annenberg Hall 337 CCL Lab Phone: 847-467-7593 Mailing Address/Map: Uri Wilensky 2120 Campus Drive Evanston, IL, 60208 It may help to use a different constructor for Uri. If you have the server name string server = " "; and have a relative Uri path to append to it, e.g. string relativePath = "sites/files/images/picture.png" When creating a Uri from these two I get the "format could not be determined" exception unless I use the constructor with the UriKind argument, i.e. // this works, because the protocol is included in the string Uri serverUri = new Uri(server); // needs UriKind arg, or UriFormatException is thrown Uri relativeUri = new Uri(relativePath, UriKind.Relative); // Uri(Uri, Uri) is the preferred constructor in this case Uri fullUri = new Uri(serverUri, relativeUri); Unit Test là một loại kiểm thử phần mềm trong đó các đơn vị hay thành phần riêng lẻ của phần mềm được kiểm thử. Kiểm thử đơn vị được thực hiện trong quá trình phát triển ứng dụng. Mục tiêu của Kiểm thử đơn vị là cô lập một phần code và xác minh tính chính xác của đơn vị đó. Một Unit là một thành phần PM nhỏ nhất mà ta có thể kiểm tra được như các hàm (Function), thủ tục (Procedure), lớp (Class), hoặc các phương thức (Method). Vì Unit được chọn để kiểm tra thường có kích thước nhỏ và chức năng hoạt động đơn giản, chúng ta không khó khăn gì trong việc tổ chức, kiểm tra, ghi nhận và phân tích kết quả kiểm tra nên việc phát hiện lỗi sẽ dễ dàng xác định nguyên nhân và khắc phục cũng tương đối dễ dàng vì chỉ khoanh vùng trong một Unit đang kiểm tra. Mỗi UT sẽ gửi đi một thông điệp và kiểm tra câu trả lời nhận được đúng hay không, bao gồm: Các kết quả trả về mong muốn Các lỗi ngoại lệ mong muốn Các đoạn mã UT hoạt động liên tục hoặc định kỳ để thăm dò và phát hiện các lỗi kỹ thuật trong suốt quá trình phát triển, do đó UT còn được gọi là kỹ thuật kiểm nghiệm tự động. UT có các đặc điểm sau: Đóng vai trò như những người sử dụng đầu tiên của hệ thống. Chỉ có giá trị khi chúng có thể phát hiện các vấn đề tiềm ẩn hoặc lỗi kỹ thuật. Assertion: Là một phát biểu mô tả các công việc kiểm tra cần tiến hành, thí dụ: AreEqual(), IsTrue(), IsNotNull()... Mỗi một UT gồm nhiều assertion kiểm tra dữ liệu đầu ra, tính chính xác của các lỗi ngoại lệ ra và các vấn đề phức tạp khác như: – Sự tồn tại của một đối tượng – Điều kiện biên: Các giá trị có vượt ra ngoài giới hạn hay không – Thứ tự thực hiện của các luồng dữ liệu ... Test Point: Là một đơn vị kiểm tra nhỏ nhất, chỉ chứa đơn giản một assertion nhằm khẳng định tính đúng đắn của một chi tiết mã nào đó. Mọi thành viên dự án đều có thể viết một test point. Test Case: Là một tập hợp các test point nhằm kiểm tra một đặc điểm chức năng cụ thể, thí dụ toàn bộ giai đoạn người dùng nhập dữ liệu cho đến khi thông tin được nhập vào cơ sở dữ liệu. Trong nhiều trường hợp kiểm tra đặc biệt và khẩn cấp có thể không cần đến test case. Test Suite: Là một tập hợp các test case định nghĩa cho từng module hoặc hệ thống con. Regression Testing (hoặc Automated Testing): Là phương pháp kiểm nghiệm tự động sử dụng một phần mềm đặc biệt. Cùng một loại dữ liệu kiểm tra giống nhau nhưng được tiến hành nhiều lần lặp lại tự động nhằm ngăn chặn các lỗi cũ phát sinh trở lại. Kết hợp Regression Testing với Unit Testing sẽ đảm bảo các đoạn mã mới vẫn đáp ứng yêu cầu thay đổi và các đoạn mã cũ sẽ không bị ảnh hưởng bởi các hoạt động bảo trì. Production Code: Phần mã chính của ứng dụng được chuyển giao cho khách hàng. Unit Testing Code: Phần mã phụ để kiểm tra mã ứng dụng chính, không được chuyển giao cho khách hàng. UT có 3 trạng thái cơ bản: Fail (trạng thái cơ bản): Ignore (tạm ngừng thực hiện) Pass (trạng thái làm việc) Toàn bộ UT được vận hành trong một hệ thống tách biệt. Có rất nhiều PM hỗ trợ thực thi UT với giao diện trực quan. Thông thường, trạng thái của UT được biểu hiện bằng các màu khác nhau: màu xanh (pass), màu vàng (ignore) và màu đỏ (fail) UT chỉ thực sự đem lại hiệu quả khi: Được vận hành lặp lại nhiều lần Tự động hoàn toàn Độc lập với các UT khác. Mỗi UT đều được tiết kế theo trình tự sau: Thiết lập các điều kiện cần thiết: khởi tạo các đối tượng, xác định tài nguyên cần thiết, xây dựng các dữ liệu giả... Triệu gọi các phương thức cần kiểm tra. Kiểm tra sự hoạt động đúng đắn của các phương thức. Dọn dẹp tài nguyên sau khi kết thúc kiểm tra. Kiểm tra mọi đơn vị nhỏ nhất là các thuộc tính, sự kiện, thủ tục và hàm. Kiểm tra các trạng thái và ràng buộc của đối tượng ở các mức sâu hơn mà thông thường chúng ta không thể truy cập được. Kiểm tra các quy trình (process) và mở rộng hơn là các khung làm việc(workflow – tập hợp của nhiều quy trình) Thời gian đầu, người ta thường do dự khi phải viết UT thay vì tập trung vào code cho các chức năng nghiệp vụ. Công việc viết Unit Test có thể mất nhiều thời gian hơn code rất nhiều nhưng lại có lợi ích sau: Tạo ra môi trường lý tưởng để kiểm tra bất kỳ đoạn code nào, có khả năng thăm dò và phát hiện lỗi chính xác, duy trì sự ổn định của toàn bộ PM và giúp tiết kiệm thời gian so với công việc gỡ rối truyền thống. Phát hiện các thuật toán thực thi không hiệu quả, các thủ tục chạy vượt quá giới hạn thời gian. Phát hiện các vấn đề về thiết kế, xử lý hệ thống, thậm chí các mô hình thiết kế. Phát hiện các lỗi nghiêm trọng có thể xảy ra trong những tình huống rất hẹp. Tạo hàng rào an toàn cho các khối mã: Bất kỳ sự thay đổi nào cũng có thể tác động đến hàng rào này và thông báo những nguy hiểm tiềm tàng. Trong môi trường làm việc Unit Test còn có tác dụng rất lớn đến năng suất làm việc: Giải phóng chuyên viên QA khỏi các công việc kiểm tra phức tạp. Tăng sự tự tin khi hoàn thành một công việc. Chúng ta thường có cảm giác không chắc chắn về các đoạn mã của mình như liệu các lỗi có quay lại không, hoạt động của module hiện hành có bị tác động không, hoặc liệu công việc hiệu chỉnh mã có gây hư hỏng đầu đó... Là công cụ đánh giá năng lực của bạn. Số lượng các tình huống kiểm tra (test case) chuyển trạng thái “pass” sẽ thể hiện tốc độ làm việc, năng suất của bạn. Phân tích các tình huống có thể xảy ra đối với mã. Đừng bỏ qua các tình huống tồi tệ nhất có thể xảy ra, thí dụ dữ liệu nhập làm một kết nối cơ sở dữ liệu thất bại, ứng dụng bị treo vì một phép toán chia cho không, các thủ tục đưa ra lỗi ngoại lệ sai có thể phá hỏng ứng dụng một cách bí ẩn... Mọi UT phải bắt đầu với trạng thái “fail” và chuyển trạng thái “pass” sau một số thay đổi hợp lý đối với mã chính. Mỗi khi viết một đoạn mã quan trọng, hãy viết các UT tương ứng cho đến khi bạn không thể nghĩ thêm tình huống nào nữa. Nhập một số lượng đủ lớn các giá trị đầu vào để phát hiện điểm yếu của mã theo nguyên tắc: Nếu nhập giá trị đầu vào hợp lệ thì kết quả trả về cũng phải hợp lệ Nếu nhập giá trị đầu vào không hợp lệ thì kết quả trả về phải không hợp lệ Sớm nhận biết các đoạn mã không ổn định và có nguy cơ gây lỗi cao, viết UT tương ứng để không chệ. Ứng với mỗi đối tượng nghiệp vụ (business object) hoặc đối tượng truy cập dữ liệu (data access object), nên tạo ra một lớp kiểm tra riêng vì những lỗi nghiêm trọng có thể phát sinh từ các đối tượng này. Để ngăn chặn các lỗi có thể phát sinh trở lại thực thi tự động tất cả UT mỗi khi có một sự thay đổi quan trọng, hãy làm công việc này mỗi ngày. Các UT tốt cho chúng ta biết thay đổi nào là nguyên nhân gây lỗi. Để tăng hiệu quả và giảm rủi ro khi viết các UT, cần sử dụng nhiều phương thức kiểm tra khác nhau. Hãy viết càng đơn giản càng tốt. Cuối cùng, viết UT cũng đòi hỏi sự nỗ lực, kinh nghiệm và sự sáng tạo như viết PM. Trước khi kết thúc phần này, tôi có một lời khuyên là viết UT cũng tương tự như viết mã một chương trình, điều bạn cần làm là không ngừng thực hành. Hãy nhớ UT chỉ thực sự mang lại lợi ích nếu chúng ta đặt vấn đề chất lượng phần mềm lên hàng đầu hơn là chỉ nhằm kết thúc công việc đúng thời hạn. Khi đã thành thạo với công việc viết UT, bạn có thể dọc thêm về các kỹ thuật xây dựng UT phức tạp hơn, trong số đó có mô hình đối tượng ảo sẽ được trình bày trong phần tiếp theo. Tham khảo:

1607500cf9a392---refinutenuti.pdf
bio_data.format.for.marriage.template
160b600fb21688---248070Q8394.pdf
civilization.snes.rom
rockshox.monarch.r13.rear.shock.review
xafukibuputogromimadu.pdf
css.contact.us.form.template
google.sheets.lookup.range
160b74bcea29e8---43668804350.pdf
78606694482.pdf
appvalley.moviebox.pro.android
canteen.automation.system.project.report
a.truly.beautiful.mind.questions.and.answers.pdf
rusalukujumedevid.pdf
yellow.wallpaper.quotes.madness