1. Record Nr. UNINA9911019278803321

Autore	Huizinga Dorota
Titolo	Automated defect prevention : best practices in software management / / Dorota Huizinga, Adam Kolawa
Pubbl/distr/stampa	Hoboken, N.J., : Wiley-Interscience, : IEEE Computer Society, c2007
ISBN	9786611032296 9781281032294 1281032298 9780470165171 0470165170 9780470165164 0470165162
Descrizione fisica	1 online resource (454 p.)
Altri autori (Persone)	KolawaAdam
Disciplina	005
Soggetti	Software failures - Prevention - Data processing Software maintenance - Data processing Debugging in computer science - Automatic control Computer programs - Testing - Data processing Computer programs - Correctness
Lingua di pubblicazione	Inglese
Formato	Materiale a stampa
Livello bibliografico	Monografia
Note generali	Description based upon print version of record.
Nota di bibliografia	Includes bibliographical references and index.
Nota di contenuto	Preface -- Features and Organization -- Practice Descriptions -- Intended audience -- Acknowledgements -- Permissions -- Disclaimer -- 1. The Case for Automated Defect Prevention -- 1.1 What is ADP? -- 1.2 What are the goals of ADP? -- 1.2.1 People: Stimulated and Satisfied -- 1.2.2 Product: High Quality -- 1.2.3 Organization: Increased Productivity and Operational Efficiency -- 1.2.4 Process: Controlled, Improved, and Sustainable -- 1.2.5 Project: Managed through Informed Decision Making -- 1.3 How is ASDP implemented? -- 1.3.1 Principles -- 1.3.2 Practices -- 1.3.3 Policies -- 1.3.4 Defect Prevention Mindset -- 1.3.5 Automation -- 1.4 From the waterfall to modern software development process models -- 1.5 Acronyms -- 1.6 Glossary -- 1.7 References -- 1.8 Exercises -- 2. Principles of

Automated Defect Prevention -- 2.1 Introduction -- 2.2 Defect Prevention: Definition and Benefits -- 2.3 Historical Perspective: Defect Analysis and Prevention in Auto Industry - What Happened to Deming? -- 2.4 Principles of Automated Defect Prevention -- 2.4.1 Principle 1: Establishment of Infrastructure: "Build a strong foundation through integration of people and technology" -- 2.4.2 Principle 2: Application of General Best Practices: "Learn from others' mistakes" -- 2.4.3 Principle 3: Customization of Best Practices: "Learn from your own mistakes" -- 2.4.4 Principle 4: Measurement and Tracking of Project Status: "Understand the past and present to make decisions about the future" -- 2.4.5 Principle 5: Automation: "Let the computer do it" -- 2.4.6 Principle 6: Incremental Implementation of ADP's Practices and Policies -- 2.5 Automated Defect Prevention based Software Development Process Model -- 2.6 Examples -- 2.6.1 Focus on Root Cause Analysis of a Defect -- 2.6.2 Focus on Infrastructure -- 2.6.3 Focus on Customized Best Practice -- 2.6.4 Focus on Measurements of Project Status -- 2.7 Acronyms -- 2.8 Glossary -- 2.9 References -- 2.10 Exercises.
3. Initial Planning and Infrastructure -- 3.1 Introduction -- 3.2 Initial Software Development Plan -- 3.2.1 Product -- 3.2.2 People -- 3.2.3 Technology -- 3.2.4 Process -- 3.3 Best Practices for Creating People Infrastructure -- 3.3.1 Defining Groups -- 3.3.2 Determining a Location for Each Group's Infrastructure -- 3.3.3 Defining People Roles -- 3.3.4 Establishing Training Program -- 3.3.5 Cultivating a Positive Group Culture -- 3.4 Best Practices for Creating Technology Infrastructure -- 3.4.1 Automated Reporting System -- 3.4.2 Policy for Use of Automated Reporting System -- 3.4.3 Minimum Technology Infrastructure -- 3.4.4 Intermediate Technology Infrastructure -- 3.4.5 Expanded Technology Infrastructure -- 3.5 Integrating People and Technology -- 3.6 Human Factors and Concerns -- 3.7 Examples -- 3.7.1 Focus on Developer Ideas -- 3.7.2 Focus on Reports Generated by the Minimum Infrastructure -- 3.8 Acronyms -- 3.9 Glossary -- 3.10 References -- 3.11 Exercises -- 4. Requirements Specification and Management -- 4.1 Introduction -- 4.2 Best Practices for Gathering and Organizing Requirements -- 4.2.1 Creating the Product Vision and Scope Document -- 4.2.2 Gathering and Organizing Requirements -- 4.2.3 Prioritizing Requirements -- 4.2.4 Developing Use Cases -- 4.2.5 Creating a Prototype to Elicit Requirements -- 4.2.6 Creating Conceptual Test Cases -- 4.2.7 Requirements Documents Inspection -- 4.2.8 Managing Changing Requirements -- 4.3 Best Practices in Different Environments -- 4.3.1 Existing Versus New Software Project -- 4.3.2 In-House Versus Outsourced Development Teams -- 4.4 Policy for Use of the Requirements Management System -- 4.4.1 The project manager should approve the final version of the vision and scope document, which should be entered into, and tracked in, the requirements management system -- 4.4.2 The architect should approve the final version of the requirements specification (SRS) document. The requirements from SRS should be entered into, and their changes tracked in, the requirements management system. 4.4.3 The architect or lead developer should define the scope and test requirements for each feature to be implemented, and then enter those details in the requirements management system -- 4.4.4 The developer should create test cases for each feature she is assigned to implement, and add those test cases to the requirements management system -- 4.4.5 After the developer implements a feature, she should modify the test cases to verify the new feature, then, once the tests pass, she should mark the feature as "implemented" -- 4.4.6 Measurements Related to Requirement Management System -- 4.4.7 Tracking of Data

Related to the Requirements Management System -- 4.5 Examples -- 4.5.1 Focus on Customized Best Practice -- 4.5.2 Focus on Monitoring and Managing Requirement Priorities -- 4.5.3 Focus on Change Requests -- 4.6 Acronyms -- 4.7 Glossary -- 4.8 References -- 4.9 Exercises -- 5. Extended Planning and Infrastructure -- 5.1 Introduction -- 5.2 Software Development Plan -- 5.3 Defining Project Objectives -- 5.4 Defining Project Artifacts and Deliverables -- 5.4.1 The Vision and Scope document -- 5.4.2 SRS, describing the product key features -- 5.4.3 Architectural and detailed design documents and models -- 5.4.4 List of COTS (Commercial-Off-the-Shelf-Components) used -- 5.4.5 Source and executable code -- 5.4.6 Test plan -- 5.4.7 Acceptance plan -- 5.4.8 Periodic reports generated by the reporting system -- 5.4.9 Deployment plan -- 5.4.10 User and operational manuals -- 5.4.11 Customer training plan -- 5.5 Selecting a Software Development Process Model -- 5.6 Defining Defect Prevention Process -- 5.7 Managing Risk -- 5.8 Managing Change -- 5.9 Defining Work Breakdown Structure (WBS) - An Iterative Approach -- 5.10 Best Practices for Estimating Project Effort -- 5.10.1 Estimation by Using Elements of Wideband Delphi -- 5.10.2 Estimation by Using Effort Analogy -- 5.10.3 Estimation by Using Parametric Models -- 5.10.4 Estimations of Using COTS and Code Reuse.
5.10.5 Estimation Quality Factor and the Iterative Adjustments of Estimates -- 5.11 Best Practices for Preparing the Schedule -- 5.12 Measurement and Tracking for Estimation -- 5.13 Identifying Additional Resource Requirements -- 5.13.1 Extending the Technology Infrastructure -- 5.13.2 Extending the People Infrastructure -- 5.14 Examples -- 5.14.1 Focus on the Root Cause of a Project Scheduling Problem -- 5.14.2 Focus on Organizing and Tracking Artifacts -- 5.14.3 Focus on Scheduling and Tracking Milestones -- 5.15 Acronyms -- 5.16 Glossary -- 5.17 References -- 5.18 Exercises -- 6. Architectural and Detailed Design -- 6.1 Introduction -- 6.2 Best Practices for Design of System Functionality and its Quality Attributes -- 6.2.1 Identifying Critical Attributes of Architectural Design -- 6.2.2 Defining the Policies for Design of Functional and Non-functional Requirements -- 6.2.3 Applying Design Patterns -- 6.2.4 Service Oriented Architecture -- 6.2.5 Mapping Requirements to Modules -- 6.2.6 Designing Module Interfaces -- 6.2.7 Modeling Modules and their Interfaces with UML -- 6.2.8 Defining Application Logic -- 6.2.9 Refining Test Cases -- 6.2.10 Design Document Storage and Inspection -- 6.2.11 Managing Changes in Design -- 6.3 Best Practices for Design of Graphical User Interface -- 6.3.1 Identifying Critical Attributes of User Interface Design -- 6.3.2 Defining the User Interface Design Policy -- 6.3.3 Identifying Architectural Patterns Applicable to the User Interface Design -- 6.3.4 Creating Categories of Actions -- 6.3.5 Dividing Actions into Screens -- 6.3.6 Prototyping the Interface -- 6.3.7 Testing the Interface -- 6.4 Examples -- 6.4.1 Focus on Module Assignments and Design Progress -- 6.4.2 Focus on the Number of Use Cases per Module -- 6.4.3 Focus on Module Implementation Overview -- 6.4.4 Focus on Customized Best Practice for GUI Design -- 6.5 Acronyms -- 6.6 Glossary -- 6.7 References -- 6.8 Exercises -- 7. Construction.
7.1 Introduction -- 7.2 Best Practices for Code Construction -- 7.2.1 Applying coding standards throughout development -- 7.2.2 Applying the test-first approach at the service and module implementation level -- 7.2.3 Implementing service contracts and/or module interfaces before their internal functionality -- 7.2.4 Applying Test Driven Development for algorithmically complex and critical code units -- 7.2.5 Conducting white box unit testing after implementing each unit

and before checking the code to the source control system -- 7.2.6 Verifying code consistency with the requirements and design -- 7.3 Policy for Use of the Code Source Control System -- 7.3.1 Each developer should have a local copy (sandbox) of files related to her current work -- 7.3.2 Each team should have a sandbox with copies of all files needed to build each application -- 7.3.3 Parallel development practices should be well defined and understood by participating developers -- 7.3.4 Each developer should check out only code that she is actively modifying -- 7.3.5 Each developer should check in to source control only code that complies with the required coding standards and passes the designated quality checks -- 7.3.6 Each developer should clean the sandbox and re-shadow relevant files after major changes -- 7.3.7 The entire team should store code for different software versions in physically independent locations of the source control systems -- 7.3.8 The entire team should use versioning features only for small variations within one software version -- 7.3.9 Measurements Related to Source Control -- 7.3.10 Tracking of Source Control Data -- 7.4 Policy for Use of Automated Build -- 7.4.1 Creating a special build account -- 7.4.2 Cleaning the build area before each build -- 7.4.3 Shadowing or cloning the source code to the build directory -- 7.4.4 Building the application at regularly scheduled intervals after cleaning the build directory and shadowing or cloning the source code.
7.4.5 Completely automating the build process -- 7.4.6 Creating hierarchies for Makefiles and/or other build files -- 7.4.7 Parameterizing scripts and build files -- 7.4.8 For n-tier applications, establishing and creating a build on a staging area as well as a production area -- 7.4.9 Fully integrating automated builds with the source control system -- 7.4.10 Integrating testing into the automated build process -- 7.4.11 Measurements Related to Automated Builds -- 7.4.12 Tracking of Data Related to Automated Builds -- 7.5 Examples -- 7.5.1 Focus on a Customized Coding Standard Policy -- 7.5.2 Focus on Features/Tests Reports -- 7.6 Acronyms -- 7.7 Glossary -- 7.8 References -- 7.9 Exercises -- 8. Testing and Defect Prevention -- 8.1 Introduction -- 8.2 Best Practices for Testing and Code Review -- 8.2.1 Conducting White Box Unit Testing: Bottom-Up Approach -- 8.2.2 Conducting Black Box Testing and Verifying the Convergence of Top Down and Bottom Up Tests -- 8.2.3 Conducting Code Reviews as a Testing Activity -- 8.2.4 Conducting Integration Testing -- 8.2.5 Conducting System Testing -- 8.2.6 Conducting Regression Testing -- 8.2.7 Conducting Acceptance Testing -- 8.3 Defect Analysis and Prevention -- 8.4 Policy for Use of Problem Tracking System -- 8.4.1 During development, problem tracking systems should be used to store only severe defects, feature requests and developer ideas -- 8.4.2 After a code freeze, the problem tracking system should be used to record all defect reports and all feature requests -- 8.4.3 During release planning, recorded feature requests should be prioritized and their implementation scheduled -- 8.4.4 Measurements of Data Related to the Problem Tracking System -- 8.4.5 Tracking of Data Related to Problem Tracking System -- 8.5 Policy for Use of Regression Testing System -- 8.5.1 Configuring the regression system so that it provides detailed result information -- 8.5.2 Executing regression tests automatically after each build.
8.5.3 Reviewing regression test results at the beginning of each work day and updating the test suite as needed -- 8.5.4 Using regression results to assess the deployment readiness of the system -- 8.5.5 Measurements Related to the Regression Testing System -- 8.5.6 Tracking of Data Related to the Regression Testing System -- 8.6

Examples -- 8.6.1 Focus on Defect Tracking Reports -- 8.6.2 Focus on Test Type Reports -- 8.6.3 Example of a Root Cause Analysis of a Design and Testing Defect -- 8.7 Acronyms -- 8.8 Glossary -- 8.9 References -- 8.10 Exercises -- 9. Trend Analysis and Deployment -- 9.1 Introduction -- 9.2 Trends in Process Control -- 9.2.1 Process Variations -- 9.2.2 Process Stabilization -- 9.2.3 Process Capability -- 9.3 Trends in Project Progress -- 9.3.1 Analyzing Features/Requirements Implementation Status -- 9.3.2 Analyzing Source Code Growth -- 9.3.3 Analyzing Test Results -- 9.3.4 Analyzing Defects -- 9.3.5 Analyzing Cost and Schedule -- 9.4 Best Practices for Deployment and Transition -- 9.4.1 Deployment to a Staging System -- 9.4.2 Automation of the Deployment Process -- 9.4.3 Assessing Release Readiness -- 9.4.4 Release: Deployment to the Production System -- 9.4.5 Non-intrusive Monitoring -- 9.5 Acronyms -- 9.6 Glossary -- 9.7 References -- 9.8 Exercises -- 10. Managing External Factors -- 10.1 Introduction -- 10.2 Best Practices for Managing Outsourced Projects -- 10.2.1 Establishing A Software Development Outsource Process -- 10.2.2 Phase 0: Decision to Outsource -- 10.2.3 Phase 1: Planning -- 10.2.4 Phase 2: Implementation -- 10.2.5 Phase 3: Termination -- 10.3 Best Practices for Facilitating IT Regulatory Compliance -- 10.3.1 Section 508 of the US Rehabilitation Act -- 10.3.2 Sarbanes-Oxley Act of 2002 -- 10.4 Best Practices for Implementation of CMMI -- 10.4.1 Capability and Maturity Model Integration (CMMI) -- 10.4.2 Staged Representation -- 10.4.3 Putting Staged Representation Based Improvement into Practice Using ASDP. 10.4.4 Putting Continuous Representation Based Improvement into Practice Using ASDP -- 10.5 Acronyms -- 10.6 Glossary -- 10.7 References -- 10.8 Exercises -- 11. Case Studies: Automation as an Agent of Change -- 11.1 Case Study I: Implementing Java Coding Standards in a Financial Application -- 11.1.1 Company Profile -- 11.1.2 Problems -- 11.1.3 Solution -- 11.1.4 Data Collected -- 11.1.5 The Bottom Line Results - Facilitating Change -- 11.1.6 Acronyms -- 11.1.7 Glossary -- 11.1.8 References for Case Study I -- 11.2 Case Study II: Implementing C/C++ Coding Standards in an Embedded Application -- 11.2.1 Introduction -- 11.2.2 C/C++ Coding Standards -- 11.2.3 Considerations on Choosing a Coding Standards Checker -- 11.2.4 Experiences Using the Checker -- 11.2.5 Lessons Learned -- 11.2.6 Conclusion -- 11.2.7 References for Case Study II -- Appendix A: A Brief Survey of Modern Software Development Process Models -- A.1 Introduction -- A.2 Rapid Application Development (RAD) and Rapid Prototyping -- A.3 Incremental Development -- A.4 Spiral Model -- A.5 Object Oriented Unified Process -- A.6 Extreme and Agile Programming -- A.7 References -- Appendix B: Mars Polar Lander (MPL), Loss and Lessons -- B.1 No Definite Root Cause -- B.2 No Mission Data -- B.3 Root Cause Revisited -- Appendix C: Service-Oriented Architecture: Example of an Implementation with ADP Best Practices -- C.1 Introduction -- C.2 Web Service Creation: Initial Planning and Requirements -- C.2.1 Functional Requirements -- C.2.2 Non-Functional Requirements -- C.3 Web Service Creation: Extended Planning and Design -- C.3.1 Initial Architecture -- C.3.2 Extended Infrastructure -- C.3.3 Design -- C.4 Web Service Creation: Construction and Testing Stage 1 : Module Implementation -- C.4.1 Applying Coding Standards -- C.4.2 Implementing Interfaces and Applying a Test-first Approach for Modules and Sub-modules -- C.4.3 Generating White Box Junit Tests -- C.4.4 Gradually Implementing the Sub-module until all Junit Tests Pass and Converge with the Original Black Box Tests.
C.4.5 Checking Verified Tests into the Source Control System and

Running Nightly Regression Tests -- C.5 Web Service Creation: Construction and Testing Stage 2: The WSDL Document Implementation -- C.5.1 Creating and Deploying the WSDL Document on the Staging Server as Part of the Nightly Build Process -- C.5.2 Avoiding Inline Schemas when XML Validation Is Required -- C.5.3 Avoiding Cyclical Referencing, when Using Inline Schemas -- C.5.4 Verifying WSDL document for XML Validity -- C.5.5 Avoiding "Encoded" Coding Style by Checking Interoperability -- C.5.6 Creating Regression Tests for the WSDL documents and Schemas to Detect Undesired Changes -- C.6 Web Service Creation: Server Deployment -- C.6.1 Deploying the Web Service to a Staging Server as Part of the Nightly Build Process -- C.6.2 Executing Web Service Tests that Verify the Functionality of the Web Service -- C.6.3 Creating "Scenario-Based" Tests and Incorporating them Into the Nightly Test Process -- C.6.4 Database Testing -- C.7 Web Service Creation: Client Deployment -- C.7.1 Implementing the Client According to the WSDL Document Specification -- C.7.2 Using Server Stubs to Test Client functionality - Deploying the Server Stub as Part of the Nightly Deployment Process -- C.7.3 Adding Client Tests into the Nightly Test Process -- C.8 Web Service Creation: Verifying Security -- C.8.1 Determining the Desired Level of Security -- C.8.2 Deploying Security Enabled Web Service on Additional Port of the Staging Server -- C.8.3 Leveraging Existing Tests: Modifying Them to Test for Security and Incorporating Them into the Nightly Test Process -- C.9 Web Service Creation: Verifying Performance through Continuous Performance/Load Testing -- C.9.1 Starting Load Testing As Early As Possible and Incorporating it into the Nightly Test Process -- C.9.2 Using Results of Load Tests to Determine Final Deployment. Configuration -- Appendix D: AJAX Best Practice: Continuous Testing -- D.1 Why AJAX? -- D.2 AJAX Development and Testing Challenges. D.3 Continuous Testing -- Appendix E: Software Engineering Tools -- Glossary -- Index.

Sommario/riassunto

Improve Productivity by Integrating Automation and Defect Prevention into Your Software Development ProcessThis book presents an approach to software management based on a new methodology called Automated Defect Prevention (ADP). The authors describe how to establish an infrastructure that functions as a software "production line" that automates repetitive tasks, organizes project activities, tracks project status, seamlessly collects project data, and sustains and facilitates the improvement of human-defined processes. Well-grounded in software engineering research and in industry best practices, this book helps organizations gain dramatic improvement in both product quality and operational effectiveness.Ideal for industry professionals and project managers, as well as upper-level undergraduates and graduate-level students in software engineering, Automated Defect Prevention is complete with figures that illustrate how to structure projects and contains real-world examples, developers' testimonies, and tips on how to implement defect prevention strategies across a project group.