The DP&T Methodology: The Defect Prevention and Traceability–Driven Methodology for Software Engineering

Jay Xiong
International Software Automation, Inc., & ISA Shanghai, Ltd., Shanghai, China, www.ISApanorama.com, jay@ISApanorama.com

Jonathan Xiong
Electrical Engineering & Computer Science, University of California, Berkeley, 2650 Durant Ave., Freeborn #801, Berkeley, CA 94720, USA, jxiong@berkeley.edu

Abstract: This paper describes a new methodology for software engineering: the DP&T Methodology. It is the core component of the new DP&T System. Different from traditional methodologies with linear process models, the DP&T Methodology uses the nonlinear DP&T Model with the capacity for forward and backward traceability. Driven by six types of automated traceability and two types of defect prevention, the DP&T Methodology supports component-based development, iterative and incremental development, and much more.

Keywords: traceability, defect prevention, methodology, software engineering

1 The DP&T System and DP&T Methodology

The Defect Prevention and Traceability (DP&T) System is a new type of system for software engineering. It consists of six parts: the DP&T Model, the DP&T Methodology, the DP&T Support Graphics, the DP&T Support Technologies, the DP&T Support Tools, and the DP&T Support Platforms.

The DP&T Methodology is the core section of the DP&T System. Differing from traditional methodologies that use linear process models, the DP&T Methodology uses a nonlinear process model called the DP&T Model with the capability for forward and backward traceability for validating requirements, preventing defects, finding and solving inconsistencies among all artifacts generated in each phase of the software life cycle, and so on.

2 Driving forces

DP&T Methodology is driven by six types of automated traceability and two kinds of defect prevention. The six types of automated traceability are as follows:

- Automated and bidirectional traceability in multiple levels among documents and source code obtained from project planning, requirement analysis, design, coding, testing, and maintenance. This type of traceability is essential to software validation, verification, debugging, and the identification of unimplemented requirements and useless code modules, requirements that are related to a module to be changed (for consistent modification), test cases that can be used for regression testing (whereby the efficiency of regression testing can be improved tenfold!), etc.

- Automated and bidirectional traceability within the source code, among source files, classes, functions, and detailed statements. It is better to diagram the entire program, then create the traceability automatically between header file and "#include" statement, program tree and function body, function definition and function call statement, class instance and class definition, goto statement and label, etc. This type of traceability is essential for efficient code inspection and walkthrough, testing, bug checking, consistent code modification, etc. An example of this kind of
traceability is shown in Fig. 1, provided by Panorama®++ for C/C++, which was developed by International Software Automation, Inc.

Fig. 1 – Automated traceability within the entire source code of a diagrammed program.

- Automated capability to trace a runtime error to the execution path and any related functions. This type of traceability is useful for debugging with testing. An example of this kind of traceability is shown in Fig. 2.

Fig. 2 – Automated capability to trace runtime errors to the execution path.

- Automated traceability in a systematic analysis of software changes to graphically show version comparison results at the system level, source file level, class and function level, and statement level. This includes identifying which modules are deleted (brown), added (green), changed (red), and unchanged (blue). To a changed module, we can further trace the detailed source code to find which statements are deleted, added, and modified. This type of traceability is very useful for version comparison and debugging, particularly in the maintenance phase when some bugs have been
removed but new bugs are found. An example of this kind of traceability is shown in Fig. 3, provided by Panorama®++ for C/C++.

![Diagram](image.png)

**Fig. 3 – An example of comparing and tracing two versions of a software product.**

- Automated traceability among documents such as those for requirement management as specified in CMMI, including documents for requirement specifications, changes, comprehension, etc. To realize this type of automatic traceability, we use a set of predesigned templates in HTML/XML format. These templates will link together by themselves.

- Automatic traceability through all possible execution paths for each module from a call graph. This kind of traceability is useful in identifying which modules may be affected by a change made within a module.

The two kinds of defect prevention are the following:

1. Defect prevention of previously found defects – a key process in level 5 of CMMI: Optimizing [1]. The purpose of this type of defect prevention is to identify the cause of defects and prevent them from recurring. Usually this process includes
   (a) causal analysis,
   (b) preventive actions,
   (c) kickoff meetings to increase awareness of quality issues,
   (d) data collection and tracking of associated data, and
   (e) improvement of the Defect Prevention Plan.

2. Defect prevention of possible new defects, particularly those related to inconsistent or changed requirement definitions that may contain conflicts, inconsistent designs or design changes, inconsistent coding (such as inconsistencies between function definitions and calling statements), inconsistent code modification, etc.
The steps for preventing defects in code modification to be performed with a module are as follows:

1. Trace the module to be modified to see what requirements, design documents, and test cases will be affected.
2. From the call graph, highlight the module to be modified (getsym) to view all related modules (which may also need to be modified), as shown in Fig. 4, then proceed carefully with any changes.

(3) After modification, use the traceable diagrammed code to check for consistency among all statements calling the modified module (getsym) before compilation, as shown in Fig. 5. Note that a related module may call the modified module in several locations.

Fig. 4 – Highlighting a module to be modified with all the related modules calling or called by it.

Fig. 5 – Using diagrammed and linked code to check for consistency between a modified module and all statements calling it.
3 Component-based development

Components are units of deployment [2]. The DP&T Methodology supports component-based development. As shown in Fig. 6, the rules for implementing each requirement are the following:

1. If there is a reusable subsystem or component in the user’s database that satisfies the requirement or can be easily modified to meet the requirement, then the subsystem or component with higher priority should be used.
2. If there is no reusable subsystem or component in the user’s database suitable to meet the requirement, but there is one on the market that satisfies the requirement or can be easily modified to meet the requirement, then buying the one on the market and using it instead of reproducing it is recommended.
3. If there is no reusable subsystem or component in the user’s database or on the market that meets the requirement, then a new subsystem or component should be built using the DP&T Methodology, with an emphasis on its reuse.

4 Iterative and incremental development

The DP&T Methodology supports iterative and incremental development. The workflows and the production phases with the DP&T Methodology are shown in Fig. 7. The core workflows in the DP&T Methodology are Schedule and Cost Control, Requirement Analysis, Design, Coding, Inspection, Testing, Delivery and Maintenance, and Real-Time Communication using BBS and emails.

All of the artifacts, documents, and source code created through these workflows are traceable forward and backward. An example of tracing a test case to the modules and code branches tested by it is shown in Fig. 8. In this case, the related topic on a BBS is also opened. Here, a BBS on Google is used, but in actual application it will be the user’s private BBS.

![Fig. 6 – Component-based development with the DP&T Methodology.](image-url)
As shown in Fig. 7, there are three production phases:

1. **The "Bone" Build phase** – after prototyping and risk analysis for critical requirements, use the "Bone Coding" technique to decompose the entire system, then use a tool to build the Bone Architecture of the whole system. "Bone Coding" uses modules with an empty body or a body with a list of function call statements without any program logic. The coded program after building a Bone Architecture will be executable after compilation.

2. **The Inception phase** – to build an essential version of the product that satisfies some important and stable requirements determined by the customer. Highlight the key modules and the related execution paths for meeting the selected set of requirements on the Bone Architecture, then assign highest orders to code (unless a module box is used to represent a reusable subsystem or a reusable component) and test all related modules.

3. **The Iterations phase** – for incremental integration of the product.

Fig. 7 – Core workflows and production phases in iterative and incremental development in the DP&T Methodology.

Fig. 8 – An example of tracing a test case to the corresponding source code.
5 Other features

Other features of the DP&T Methodology include

- visual development in the entire life cycle,
- real-time communication,
- responses to requirement changes in real time without waiting for specific milestones, and
- frequent delivery of working software.

6 Conclusion

The new DP&T Methodology is driven by defect prevention and automated traceability. It supports component-based development and incremental and iterative development. It makes all artifacts created in all phases of a software development life cycle traceable for easy requirement validation, consistent code modification, easy defect prevention, and efficient regression testing.

7 References