CARNEGIE MELLON UNIVERSITY

SEMINAL PAPERS IN SOFTWARE ENGINEERING: THE CARNEGIE MELLON CANONICAL COLLECTION

Authors: Mary Shaw (editor), Jonathan Aldrich, Travis D. Breaux, David Garlan, James D. Herbsleb, Christian Kästner, Claire Le Goues, and William L. Scherlis

September 2015

INSTITUTE FOR SOFTWARE RESEARCH
SCHOOL OF COMPUTER SCIENCE
Technical Report#: CMU-ISR-15-107

Seminal Papers in Software Engineering

The Carnegie Mellon Canonical Collection

Fall 2015 revision

Mary Shaw (editor), Jonathan Aldrich, Travis D. Breaux, David Garlan,
Christian Kästner, Claire Le Goues, and William L. Scherlis

To understand the context of current research, it is essential to understand how current results evolved from early fundamental papers. These classic papers develop timeless ideas that transcend technology changes, and the ideas embodied in their solutions often apply to current problems (and, indeed, are reinvented by researchers who are ignorant of the classic literature).

Criteria for selection of the papers in the Canon include

- Papers are selected for their overall significance. The papers should usually be the principal or definitive papers about an idea, rather than expository papers written after the idea has matured (these may appear in other reading lists). A rule of thumb is that the paper in the Canon should be the one you'd cite as the fundamental paper in its area.
- Classic, timeless papers prove their significance over time. It follows that the papers of the Canon will generally be published a minimum of 12-15 years ago.
- Some of the entries may be books. In these cases, the books should be important in their own right, but the entry should highlight one or a few specific passages in the book to focus on. Students should develop a sense of the rest of the book as well.
- Because the size of the Canon is limited, authors will rarely be represented by multiple papers, and redundancy will generally be avoided.
- All students should be familiar with all the papers, and the SE Research Core course should cover them. In practice this limits the size of the Canon to about 25 papers.

The Carnegie Mellon Software Engineering faculty has selected this set of papers as a canonical collection of significant classic papers that balances completeness with reasonable size. All our software engineering students should read and understand these papers (or selections from books). Notes in this document explain the significance of the papers. We will endeavor to cover them in the Core Course in Software Engineering Research, and other courses may feel free to assume that students have read them.

The Canon as of Fall 2015

[AG94] Robert Allen and David Garlan. "Formalizing Architectural Connection." *Proceedings of 16th International Conference on Software Engineering,* 1994, pp. 71-80.

doi: 10.1109/ICSE.1994.296767

This paper helped establish software architecture as a new discipline by focusing on the connectors as first class entities that admitted a formal description and could support various kinds of analysis. When reading this paper pay attention to the kinds of concerns that the formalism addresses – details of the notation and proof of the theorem are less important. [dg]

[Bo88] Barry W. Boehm. "A spiral model of software development and enhancement." *IEEE Computer,* vol. 21, pp. 61-72, May 1988.

doi: 10.1109/2.59

This paper describes an alternative to linear models of software development such as the Waterfall model. The Waterfall model was introduced (by Win Royce and Barry Boehm) some years earlier not to force a linear constraint on process, as is often depicted, but rather as a way to gain earlier attention to software considerations in an overall process of systems engineering. At TRW, where Boehm was employed, the success of waterfall was so significant that it precluded many essential software engineering practices, and particularly the development of experimental prototypes. The spiral model, an early codification of the large family of existing iterative models, was developed to provide an explicit process framework to enable identification of engineering uncertainties early in a process and, through a variety of mechanisms, mitigation of those uncertainties. The uncertainties can relate to a variety of engineering commitments that must be made in a successful software project, including architecture, requirements, performance, user experience, etc.[ja→wls]

[Br95] Frederick P. Brooks. *The Mythical Man-Month: Essays in Software Engineering, 2nd ed.* Addison-Wesley Professional, 1995.In particular, Ch 16 (pp.179-206), the 1986 "No Silver Bullet" paper.

Available: http://www.amazon.com/

This classic about the nature of software development and software project management is an easy read, and many gems lie inside. The first 15 chapters come from the first edition, and this second edition adds the classic 1986 "No Silver Bullet" paper (chapter 16) together with Brooks' 10-years-after reflections on "No Silver Bullet" (Ch 17) and a retrospective evaluation of the first edition (Ch18, 19). The Canon highlights Chapter 16, "No Silver Bullet", which analyzes the difference between essential and accidental difficulties in software development. Brooks argues that most accidental complexity has already been eliminated from development and that most remaining complexity is unavoidable. The article then discusses potential strategies to address essential complexity. The discussion is valuable to ground and frame research projects with realistic assumptions and expectations. The specific technologies are a quarter century old, so the significance of the paper lies in the nature of this distinction. [ms, ck]

[CES86]E.M. Clarke, E.A. Emerson, A.P. Sistla. "Automatic verification of finite-state concurrent systems using temporal logic specifications." *ACM Transactions on Programming Languages and Systems,* vol. 8, pp. 244-263, Apr.1986.

doi: 10.1145/5397.5399

This is one of the two papers that introduced model checking. Although highly technical in nature, the paper remains quite readable in the sense that the topic is introductory and the terminology has not significantly changed. Since this paper was published, the simple idea it describes has been scaled up to check systems with enormous state spaces, with applications in software engineering that include verifying device drivers and checking security protocols. When reading this paper, focus especially on sections 1-3 and 5.[ja]

[Co68] Melvin E. Conway. "How Do Committees Invent?" *Datamation,* vol. 14, pp. 28–31, Apr. 1968.
Available: http://www.melconway.com/Home/Committees_Paper.html

This paper made the observation that the design of a system reflects the structure of the organization doing the design--an observation that is now called Conway's Law. The essence of the idea, as fleshed out in the paper, is that teams designing two components that interact must inevitably communicate about the interaction, creating an alignment between the structure of the system and the design teams. As more and more software systems are designed by multiple teams, possibly scattered in multiple locations, the principles in this paper are critical for software designers and researchers to understand. [ja]

[CKI88] Bill Curtis, Herb Krasner, and Neil Iscoe. "A Field Study of the Software Design Process for Large Systems." *Communications of the ACM,* vol. 31, pp. 1268-1287, Nov. 1988.
doi: 10.1145/50087.50089

This paper is one of the very first systematic empirical studies of large scale software development. In a turn that surprised many technically trained researchers, as well as behavioral scientists focused on studying individual programmers, this study points to several social and organizational factors as the primary obstacles to effective development. It is the source of a few well-known quotations, e.g., "Writing code isn't the problem, understanding the problem is the problem" and for placing a focus on "great designers" and their social as well as technical abilities. Notice how they structure their inquiry and their results around the organizational levels at which problems manifest themselves. This paper is a stellar example of effectively forging a clear story from a confusing jumble of qualitative results. [jh]

A very early qualitative/empirical account of software engineering teams and projects, with a focus on activities prior to the development and testing of code. This paper highlights a variety of issues related to communication, models, tools, commitment-making, requirements, architecture, etc. Several of the models developed and issues presented foreshadow more modern considerations of process, requirements, teamwork, and communication. [wls]

[DK76] Frank DeRemer and Hans Kron. "Programming-in-the-large versus programming-in-the-small." *IEEE Transactions on Software Engineering,* vol.SE-2, pp. 80-86, Jun. 1976.
doi: 10.1109/TSE.1976.233534

In 1976, most research about software systems focused on the code within modules. This paper introduced the problem of specifying and reasoning about the relations among modules. The notation they introduce may not have stood the test of time, but this was the seminal paper to present the cogent argument that the organization of code modules into systems was worthy of systematic reasoning. [ms]

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. "Architectural Mismatch: Why Reuse Is So Hard." *IEEE Software,* vol.26, pp.17-26, 1995.
doi: 10.1109/52.469757

This paper introduced the term "Architecture Mismatch." It is interesting both because of the concept that it is attempting to name and understand, and also as an example of a paper that attempted to "turn a lemon into lemonade" – or, using a negative experience to try to understand a general class of problems. [dg]

[Ha87] David Harel. "Statecharts: A Visual Formalism For Complex Systems." *Science of Computer Programing,* vol, 8, pp.231-274,1987.

doi: 10.1016/0167-6423(87)90035-9

This paper is the primary early reference for Statecharts. Since this paper was published, a number of variants have been produced, most notably the version incorporated in the UML standard. These all share the basic idea of a visual formalism that attempts to provide a formal but understandable representation of system behavior. Read this paper for the key ideas; judge for yourself how the core concepts contribute to qualities like understandability, compactness, expressiveness, and simplicity (or not). The Canon highlights Sections 1-3 and 4.1 as the most important. [dg]

[Ho72] C.A.R. Hoare. "Proofs of correctness of data representations." *Acta Informatica,* vol.1, pp. 271-281, 1972.

doi: 10.1007/BF00289507

This paper addresses the problem of reasoning about the correspondence between behaviors of two different representations of a system. The setting for the paper is abstract data types, but the approach applies as well to other abstractions, for example transform spaces, simulations, and higher-level languages. Read the paper to understand the reasoning; you need to trace through the formalism only to the extent necessary to do that. This paper complements Chapter 2 of Siewiorek, Bell, and Newell, which emphasizes the models at various levels rather than the proof of correspondence.

This paper was written at the height of the first wave of program verification the beginning of the age of information hiding. It created a bridge between the two, showing how the rigorous reasoning of the former could be connected to the information propagation restrictions of the latter. For our purposes, though, the significance of the paper is the application of Hoare's approach to abstractions of all kinds. For example, when you write a simulator, you should provide assurance that its implementation produces the desired effect in the simulation model. When you engage in model-driven design, you need to have confidence that correct models will lead to correct implementations. Whenever you decide to approach a problem by introducing an abstraction, you incur an obligation to establish consistency between the abstraction and the system is abstracts from. Hoare describes one important approach. In reading this paper, concentrate on the way the correctness argument is constructed. Don't worry about the details of Simula 67, especially the details in Section 9 [ms]

[Hu88] Watts S. Humphrey. "Characterizing the Software Process: A Maturity Framework."*IEEE Software,* vol.5, pp. 73-79,Mar.1988.

doi: 10.1109/52.2014

This paper introduced the Capability Maturity Model (CMM) as a way of evaluating the capability of software organizations to produce software in a predictable way. CMM was created as a means to afford both source-selection evaluators and internal managers a more repeatable metric for organizational capability. CMM has been influential in many parts of the software development ecosystem, and particularly for custom outsourced development. It builds on the early ideas of Taylor and Deming, and extends the metaphor of manufacturing repeatability into design. The CMM led to a number of follow-on models for other disciplines (some far afield, such as service provisioning for call centers, etc.) and also for embedded software systems, including the CMMI family, as developed by the Software Engineering Institute and recently spun out. These models are widely adopted as credentialing mechanisms for the outsourcing community. This paper provides insight into the original purpose, benefits and limitations, and scope. [ja→wls]

[Ja95] Michael Jackson. “The World and the Machine.” Proceedings of the 17th International Conference on Software Engineering, 1995,pp. 283-292.
doi: 10.1145/225014.225041

This paper defines software engineering as an effort to bridge the world and machine through four facets: modeling, interfaces, engineering and the problem. Jackson discusses the shared and imbalanced relationship between these two concerns in terms of software engineering concepts, artifacts and practices; the outcome of this deliberation is often a shift in focus from a purely artifact-based view of engineering to an impact-based view, wherein the success of the artifact depends on how well we couple the artifact to the world through one for the four facets. The challenge of getting this coupling right is a central theme throughout the paper. The paper concludes with Jackson's four denials that can lead a software engineer to deny the very existence of the world. This paper is a broader and deeper reflection on what constitutes requirements than Parnas' four variable model paper, entitled "Functional Documents for Computer Systems," which more traditionally views systems in terms of sensors and actuators. Both are great papers to read together. [tb]

[check on whether this note matches the paper] Jackson’s Problem Frames emerged from his reflections on a long career of software development, during which he created software development methods such as JSP and JSD. He noticed that if he abstracted from the domain information and specific details, many of the problems he had worked on fell into classes that were broadly similar. He developed a framework for describing these classes. In reading about Problem Frames, pay particular attention to his distinction between the problem space and the solution space and to the association of domains through shared phenomena.[ms]

[Ki76] James C. King. Symbolic execution and program testing. CACM 19(7), 385-394, July 1976.
doi:10.1145/360248.360252

This paper describes symbolic execution, a foundational approach in static program analysis. The core idea is to reason about the execution of a program in terms of its effects over classes of (arbitrary) inputs, represented as abstract values, rather than to actually execute that program on concrete inputs. King accurately presents this approach as lying between the two extremes of full, sound formal verification and (unsound) testing in terms of the strengths of the guarantees it can provide about program behavior and correctness. Symbolic execution has been significantly extended since it was first explored in the 70's (King's paper is one of three published more or less simultaneously on related ideas), and now underlies or at least informs a broad range of research techniques in software testing and analysis. It also drove a considerable proportion of the significant advancement in programmatic theorem proving in the intervening decades (King's comment on page two about the impossibility of theorem proving "even for modest programming languages" is effectively no longer true!). [cl]

[KS95] Robert E. Kraut and Lynn A. Streeter. “Coordination in Software Development.” *Communications of the ACM,* vol.38, pp.69-81, Mar.1995.
doi: 10.1145/203330.203345

Kraut and Streeter were among the first to clearly point toward the critical role of various forms of coordination -- in particular informal communication when projects are uncertain -- as key to the success of a software project. Also note the several ways in which they measured project success, and how they combined different methods and different kinds of data to answer key questions about what leads to success. Note also that they do not assume there is a single best way to manage every sort of project. [jh]

[MFH02] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. "Two case studies of open source software development: Apache and Mozilla." *ACM Transactions on Software Engineering and Methodology,* vol.11, pp.309-346, Jul.2002.

doi: 10.1145/567793.567795

This paper was one of the early studies of open source projects, and attempts to answer some very basic questions about how open source works. At the time, the popular press had promulgated a number of half-truths about open source that made it difficult to understand how this seemingly freewheeling, chaotic style of development could actually work. The authors recreated project histories using data generated automatically, interpreted in the context of a clear description of the processes followed by the projects. Notice how the research questions pose very basic elements of open source development, and how they were addressed with project data. Also note way that hypotheses -- speculative, but grounded in observations from the first case study -- were validated or modified in light of the evidence from the second case study. Note also all of the difficulties of comparing data from two different open source projects, and several commercial projects, and how the authors attempted to overcome them. [jh]

[NATO68] Peter Naur and Brian Randell. *Software Engineering*. Proceedings of the 1968 NATO Conference on Software Engineering,, in particular chapters 2 and 8.2

http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF

This report documents the event at which the field of software engineering was born. It is important not only for its historical significance but also as a basline reference -- the examples are dated, but the issues are very much with us. Browse through the proceedings to see how familiar the themes are (though the examples are dated). In particular, read chapters 2 and 8.2; the latter is McIlroy's keynote talk on software components, which is probably one of the earliest vision of component markets and mass customization (tradeoff among nonfunctional properties), which lays a good ground for discussing component markets (why did this not happen, what happened instead)and other forms of software reuse and customization. [ms,ck]

[Os87] Leon J. Osterweil. "Software Processes are Software Too." *ICSE '87 Proceedings of the 9th International Conference on Software Engineering*, 1987, pp.2-13.

http://dl.acm.org/citation.cfm?id=41765.41766

From the observation that the workflow of software development involves (possibly iterative) sequences of steps with branch points, the author argues that the development activity can be described precisely and algorithmically. Although the original presentation is marred by the failure to consider the types supported by a language (which led to the suggestion that any language would be fine), this paper set the expectation that development processes could be described precisely. [ms]

[Pa72] D .L. Parnas. "On the criteria to be used in decomposing systems into modules." *Communications of the ACM*, vol. 15, pp.1053-1058, 1972.

doi: 10.1145/361598.361623

This is the seminal paper that described the idea of information hiding. The idea of hiding data representation is very conventional by now, but Parnas has a broader idea in mind. They key idea in information hiding is to hide design decisions that are likely to change within modules. These design decisions are often data representation, but may include algorithm choice, dependencies on system context, etc. Parnas' motivation, in classic software engineering style, flows primarily from the need to support software evolution. This is in marked contrast to work that motivates encapsulation for the purposes of verification. [ja]

[PCW84] D.L. Parnas, P.C. Clements, and D.M. Weiss. "The Modular Structure of Complex Systems." *ICSE'84 Proceedings of the 7th International Conference on Software Engineering,* 1984, pp.408-417.

Available: http://dl.acm.org/citation.cfm?id=800054.801999

This is one of Parnas' early papers on information hiding. After a decade of development of the ideas introduced in 1972, he applied his techniques to write a specification of the existing A7E avionics system. This paper provides an overview of the A7E System. Focus on the ways in which the authors chose to represent the system, and the criteria that they used for decomposition. Pay particular attention to the abstraction achieved by the Data Banker module. The specific structures of the A7E system are not particularly important. [dg,ms]

[RR85] Samuel T. Redwine, Jr. and William Riddle. " Software Technology Maturation." *ICSE '85 Proceedings of the 8th International Conference on Software Engineering*, 1985, pp.189-200.

Available: http://dl.acm.org/citation.cfm?id=319568.319624&coll=DL&dl=GUIDE&CFID=107042429&CFTOKEN=92850089

This paper discusses the stages in the evolution of technology ideas from bright idea to supported product. The examples are quite dated, so pay attention to the evolution model and the variability among the examples rather than to the particulars of the examples. See how the model holds up by identifying the stages of development for current software engineering topics. [ms]

[RW73] Horst W.J. Rittel and Melvin M. Webber. "Dilemmas in a general theory of planning)." *Policy Sciences,* vol. 4, pp. 155-169, 1973.

doi: 10.1007/BF01405730

This is the paper that introduced the idea of "wicked problems": when problems are deeply embedded in social systems, they cannot be specified in such a way that scientific analysis will lead to solutions. The demands and uncertainties of the problem setting are intrinsically incompatible with the precise specifications of, for example, goals and success criteria, that are required for scientific approaches. Read the paper for the cautionary tale about the limits of professionalism and, more significantly, for the properties that distinguish tame from wicked problems. The details of the examples are important only to the extent that they help you understand the properties. The idea of "wicked problem" has become widely misinterpreted as referring to software that is complicated because of implementation issues, project complexity, or even bad design (that is, by Brooks' accidental difficulties); you will see that is not at all what the authors intend. [ms]

[Ro70] W.W. Royce. "Managing the development of large software systems: concepts and techniques." *Proceedings of IEEE WESCON*, August 1970, pp1-9. Reprinted in *ICSE '87 Proceedings of the 9th international conference on Software Engineering*, 1987, pp. 328-338.

Available: http://dl.acm.org/citation.cfm?id=41765.41801&coll=DL&dl=ACM

This is the original "waterfall" paper. It was originally published in WESCON, in August 1970. The reprint in ICSE '87 is much more accessible, but it's important to remember that the paper was written in 1970. [ms]

[SG96] Mary Shaw and David Garlan. "*Software Architecture: Perspectives on an Emerging Discipline*." Prentice-Hall, Inc., 1996.

Available: http://www.amazon.com/

This was one of the first books published on software architecture, and it helped to define the field. Focus on the first two chapters of the book and the early sections of Chapter 3. Pay attention to the arguments given for architecture-level abstractions and the identification of architectural styles as a key element of the emerging discipline. Other chapters are also interesting with respect to requirements for architecture description languages, formal representation and analysis of software architectures, and the use of multiple architectural views to describe the same system. [dg]

[SBN87] Daniel P. Siewiorek, C. Gordon Bell, and Allen Newell, "Chapter 2, Levels and Abstraction," in *Computer Structures*. McGraw Hill, 1987, pp.9-16.

Available: https://research.microsoft.com/en-us/um/people/gbell/Computer_Structures_Principles_and_Examples/csp0025.htm

This is the revised version of Bell and Newell's Computer Structures, which was published in 1971, roughly at the same time as [Ho72]. The first edition was the first great compilation of computer architectures, made all the more great by its introduction of a systematic framework for comparing these architectures, complete with notations corresponding to the design levels. Indeed, the PMS level was one of the inspirations for the component/connector model of software architecture. Our interest is in the structure of the taxonomy, not all the details of the hardware. [ms]

[Si96] Herbert A. Simon. *The Sciences of the Artificial. 3rd edition*, MIT Press, 1996. Ch 1 (pp.1-17) and Ch 5 (pp. 111-125)

Available: http://www.amazon.com/

In the passage from Ch 1, Simon distinguishes natural from artificial objects and phenomena, with an emphasis on artificial phenomena that are represented symbolically. Creation for a purpose implies a purpose for the artifact, hence a distinction between the internal environment and the external environment and the need for a clear interface between the two. This sets the stage for purposeful creation of artificial phenomena, that is, for engineering, and for simulation as a means for understanding phenomena.

In Ch 5, Simon begins by arguing that design can and should be rooted in fundamental principles, but it is not exclusively about these fundamentals. Note his definition of design at the beginning. He returns to the distinction between the inner and the outer world, saying that designers create artifacts for the purpose of conforming the inner to the outer and that there can and should be a science of design. Pay particular attention to the use of optimization and search – the former for designs where alternatives are already given, and the latter for cases where the alternatives must be discovered during design. [ms]

[TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M Sutton. "N Degrees of Separation: Multi-Dimensional Separation of Concerns." *ICSE'99 Proceedings of the 21st International Conference on Software Engineering,* 1999, pp. 107-119.

doi: 10.1145/302405.302457

The paper discusses that crosscutting is unavoidable when implementing a software system using hierarchical decomposition mechanisms. While almost everything can be modularized, any chosen modularization will induce scattering of other concerns that do not align with the dominant decomposition. The paper illustrates the limits of traditional information hiding. While the proposed solution may not withstand the test of time, the paper introduces the key concept "tyranny of the dominant decomposition" and laid a conceptual foundation and motivation for many attempts to manage or encapsulate crosscutting concerns in software systems. [ck]