*International Journal of Application or Innovation in Engineering & Management (IJAIEM)*
**Web Site: www.ijaiem.org Email: editor@ijaiem.org**
**Volume 3, Issue 12, December 2014 ISSN 2319 - 4847**

# Examination constituent structural design − construction Systems using a Service Oriented for Extreme Programming

**[1]Ch.V.Phani Krishna [2]Mohammad Kemal [3]K.Rajasekhar rao**

[1,2] Associate Professor, Adama University, Ethiopia

[3]Director Sri Prakash Engineering College, India

## ABSTRACT

***Service-oriented architecture for enterprise computing In typical definitions service-oriented architecture appears as a single message and a consistent roadmap for building flexible software system. The natural language metaphors are relatively useless for either fostering communication among technical and non-technical project members or in developing architecture. Service-Oriented Architecture concepts to address different problem areas, i.e., enterprise application integration, business-to-business, business process management, and software productizing. If software architects and software managers are aware of these strands of SOA when talking about SOA in their projects they can avoid Misunderstandings and detours better. some of the author studied the Metaphor as a means of communication among team members and between them and clients.***

## 1.INTRODUCTION

The service-oriented architecture has been a leading metaphor in software architecture, both in industry and academia. This means many software products shipped with the label service-oriented, many technologies were designed around a notion of service orientation, know-how on service orientation was requested by many IT stakeholders from IT consultants in IT and software development projects, many research projects were conducted to elaborate service-oriented concepts and so on. We review the promises and actual usages of service-oriented architecture in the realm of enterprise computing. The metaphor has two purposes. The first is the communication described above. A user ought to have an easier time speaking and giving examples about an "accountant," than about Quicken. A second reason is that the metaphor is supposed to contribute to the team's development of software architecture. Ken Auer and Roy Miller describe the XP metaphor as being "analogous to what most methodologies call architecture"[6]. The purpose of this paper is to study the effectiveness of the metaphor in this light. If software architects and software managers are aware of the concrete strands of SOA when talking about SOA in their projects they can avoid misunderstandings and detours better. The method of the article is to identify the well-distinguishable visions of SOA in the domain of enterprise contributes and to characterize their key objectives and rationales with respect to each of them. With respect to the identification of SOA visions the article's result is that we need to distinguish between SOA as a paradigm for approaching the following problem areas

- Enterprise application integration.
- Business-to-business integration.
- Business process management.
- Software productizing.

A metaphor is meant to be agreed upon by all members of a project as a means of guiding the structure of the architecture [4]. In [Bernstein 1996] Phil Bernstein has identified or envisioned a trend of enterprise computing facilities becoming a kind of utility in enterprise system landscapes. The vision is about decoupling information services from appliances that can consume these services. It is about overcoming silo system landscapes and creating a new degree of freedom in using enterprise applications and databases. The information utility vision is discussed from the viewpoint of an implementing middleware. Similarly, in [Schulte and Natis 1996; Schulte 1996] the concept of a service-oriented architecture is described as being about avoiding the tight coupling of processing logic and data. Later, a concrete pattern of service-oriented architecture emerged with the target to create some kind of enterprise-wide information utility. In another strand of work service-oriented architecture evolved into the vision of creating not only enterprise-wide information utilities but also a world-wide information utility, i.e., an inter-enterprise-wide information technology. In early discussions, service-oriented architecture was rather about the principle of decoupling services from appliances, i.e., the observable trend of some kind of information utilities emerging in large scale enterprise architectures. Some definitions of service-oriented architecture, e.g., the one given in [Gartner Group 2004] boil down to an explanation of modularization. However, the objective of a certain level of decoupling of enterprise applications is a unique selling point of service-oriented architecture one should be aware of. Over time it has become common sense that there are certain design rules and key characteristics that make up service-oriented architecture. To these belong [Brown et al. 2002] coarse-grainedness, interface based design, discoverability of services, single instantiation of components, loose coupling, asynchronous communication, message-based communication. Further such characteristics are strict hierarchical composition, high cohesion, technology independency. SOA principles can add value to enterprise applications, e.g., discoverability can greatly improve reuse, targeting coarse-grainedness can guide code productizing efforts and message orientation is the proven pattern in enterprise application integration. The problem is, however, that the SOA principles are by no means silver bullets [Brooks 1975; 1987] for enterprise application architecture, least of all the combination of them. This means, the circumstances under which the SOA principles add value and how they can be exploited must be analyzed carefully. It is important to recognize that with respect to enterprise system architecture the promise of SOA, i.e., the creation of an information utility, is an objective rather than a feature of the approach. Best practices have to be gathered in order to make SOA work in projects. service-oriented architecture are know from other paradigms like component-based development or object oriented design, others are unique selling points of service-oriented architecture and others explicitly distinguish service-oriented architecture from other approaches. For example, single instantiation of components explicitly distinguishes service orientation from component-orientation. Loose coupling, high cohesion and interface based design are also targeted by object orientation. However, in the context of service-oriented architecture, loose coupling is often understood differently from how it is understood in the object-oriented community. In the object-oriented community it stands for the general pattern of minimizing the overall number of any kind of dependencies between components in a system. In service-oriented architecture it is often understood as the availability of a dynamic binding mechanism. The issue of discoverability of services is often seen as crucial for service-oriented architecture. What is meant is special support for discovering in the form of appropriate registries and tools [Brown 1997]. Discoverability of services touches the assembly and eventually the organization of development and maintenance of services, a strand of work that is currently addressed by SOA governance.

## 2. SOFTWARE ARCHITECTURE BY USING XP

We first use an example of Online Ticket Reservation (OTR) that is part of a Ticket system to understand how XP develops software and to describe some of XP's shortcomings related to software architecture.

**Online Ticket Reservation System**

In XP, the customer records the required system functionality at the beginning of each development iteration. The requirements are embodied in user stories.3 The client and the developers together determine the functionality that will be developed for the current iteration during the Planning Game practice.

**Identification of the Requirements**

The customer generally develops the user stories—stories about how the software is to be used or how it will work. User stories also contain descriptions of test situations. For example, user stories from the perspective of the "OTR customer" would include activities such as using the OTR to Ticket Booking, Cancelation. These stories tell us what the user wants, but avoid saying how to do it. Figuring out "how" is left to the engineer. Stories may also show up later as changes affect the system. The customer and developers share information to determine which stories to implement during the Planning Game practice and to produce a schedule. The customer usually goes first; he or she determines the next increment of functionally that will be of value to the user. The stories for this iteration, which usually appear on cards, are selected and given to the developers. The developers then figure out how long each story will take to build, adjusting this estimate with a concept called velocity. you can count on an average of two hours of non-software production activities, your velocity would be approximately six hours per day. Each card is marked with how long the story will take to build. The cards are collated and examined against the length of this iteration (usually about two weeks). The stories that can be done in the time allowed are returned to the customer, who chooses the highest priority ones for implementation. The ones that are too long are returned as well, to be further broken up, possibly for the next round. This is step two of the Planning Game practice. The third step is the return of the highest value items to the developers. In the final step, the developers deliver a schedule to the customer.

**Creating a Design**

The first XP practice that influences the architecture is System Metaphor. The original purpose of this practice was to provide both the customers and developers a simple description of what is in the design and a point on which they could agree to talk. System Metaphor is the least used practice of XP. Using a system metaphor is not a mandatory step of the XP process. If no agreement can be reached with the customers about the metaphor, it is frequently replaced as a practice with the stand-up meeting normally associated with Scrum [Schwaber 02]. Many have reported on the difficulty of coming up with a good system metaphor. Recently, at a major conference, XP originator Kent Beck left it up to a vote as to whether to continue with that task. Jim Tomayko, one of the authors of this report, also shared this experience and reported on the difficulty finding metaphors [Herbsleb 03]. He was about to join those who gave up on metaphors until he found considerable literature on metaphors in other fields and an article in the XP literature that made it easier to define metaphors. David West offers a metaphor for the system architecture as an interpretation of the architecture, which is of Tibetan Buddhist origin [West 02]. XP practice that has some influence on architecture is Simple Design, which means that each component will be easy to understand. Design patterns may be used to create a simple design. The candidates for refactoring are here in the design as well. The design is usually drawn on a whiteboard, so it's easily changeable. Achieving simplicity in the design is a challenge. Simplicity is not limited to the Simple Design practice: it is one of the core values of XP and, as such, is important across all the practices. Thus, simplicity is the motivation for the Simple Design practice and a challenge to many people, especially those who, in complexity, find both comfort and a place to hide things. The developers may elect to perform a spike to explore the implications of implementing the user stories with new technologies. A spike consists of building a simple prototype that allows the team to understand the new technology. Besides functional spikes, spikes related to quality attributes can be performed to guide the design of a deployment view, which supplements the naïve metaphor that represents a logical view. Since the user can check Booking status remotely, the team needs to explore the implications of an N-tier design. A candidate architecture might be posited by the team based on its experience and structured using the three-tiered, client-server, and repository styles. The OTR is, thus, a client of a transaction-processing system.

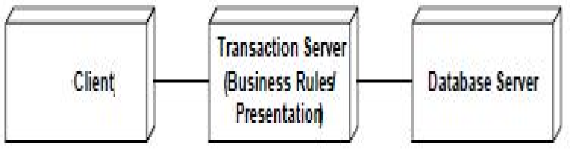

**Fig1:** candidate Architecture deployment view

Creation and Evaluating a Design One quality attribute requirement mentioned earlier is that the system must be easily modifiable to take advantage of new platform capabilities (such as a new database or client) and extensible to allow new functions and business rules to be added. Through the process of the QA, this vague requirement would be refined into several six-part scenarios. For example, the following modifiability scenarios would be typical of an OTR system:

- A developer wants to add a new auditing business rule at design time in 10 person-days without affecting other functionality.
- A developer wants to change the relational schema to add a new view to the database in 30 person-days without affecting other functionality.
- A system administrator wants to employ a new database in 18 person-months without affecting other functionality.
- A developer wants to add a new function to a client menu in 15 person-days without causing any side effects.
- A developer needs to add a Web-based client to the system in 90 person-days without affecting the functionality of the existing OTR client.

To achieve these modifiability requirements, one or more architectural tactics will need to be employed. An architectural tactic is a means of satisfying a quality-attribute-response measure (such as average latency or mean time to failure) by manipulating some aspect of a quality attribute model (such as performance-queuing models or reliability Markov models) through architectural design decisions [Bachmann 02]. In this way, tactics provide a "generate and test" model of architectural design. The ADD method defines a software architecture by basing the design process on the high-priority quality attribute requirements of the system. The ADD approach follows a recursive decomposition process where, at each stage in the decomposition, architectural tactics and patterns are selected to satisfy a chosen set of high-priority quality scenarios. In the case of modifiability, relevant architectural tactics include Localize Changes and Use an Intermediary. The Localize Changes tactic suggests that the business rules, database, and client should be localized into components, and the Use an Intermediary tactic suggests that these components should be separated to insulate them from potential changes in each other. A three-tier client-server model would emerge from the application of the Localize Changes tactic, since this architecture allocates the client, database, and business rules to their own tiers and, hence, localizes the effects of any changes to a single tier. The Use an Intermediary tactic suggests that the communication between the tiers be mediated by some abstract interface (such as a data access layer that uses Open Database Connectivity [ODBC] between the business rules and the database) and a translation layer between the business rules and the client that understands the Extensible Markup Language (XML). The existence of such intermediaries makes it simple to add new databases or clients. For example, a developer can now add a Web-based client and server as a simple addition to the architecture, without affecting the ATM client. To achieve the quality attribute requirement of a "10-second latency on a withdrawal" in the ADD method, a different set of architectural tactics is employed. Performance tactics are divided into three categories: (1) resource demand, (2) resource management, and (3) resource arbitration. Since we cannot control resource demand with an OTR (or, more precisely, because doing so would be bad for business), we must look towards managing and/or arbitrating the use of resources to meet performance goals. Some resource management tactics that are potentially applicable here are Introducing Concurrency, Maintaining Multiple Copies of Either Data or Computations, and Increasing Available Resources. By employing the Introducing Concurrency and Increasing Available Resources tactics, we may choose to deploy additional database servers and business rule servers or to make any of them multithreaded so they can execute multiple requests in parallel. Once we have multiple resources, we need some way of arbitrating among them, so we introduce a new component—a load balancer—that employs one of the resource arbitration tactics such as Fixed-Priority Scheduling or First-In First-Out Scheduling. This component will ensure that the processing load is distributed among the system's resources according to a chosen scheduling policy.

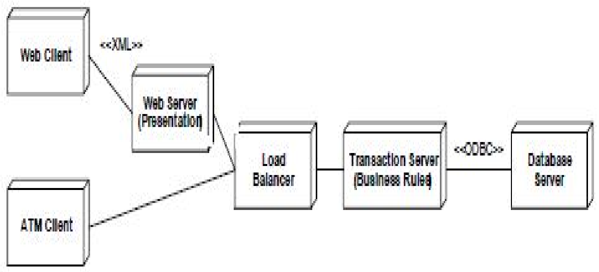

**Fig2.** Candidate Architecture

We have not yet specified the precise degree of replication of any deployed clients or servers, or the size of the thread pool in them. This more detailed specification is the next step in the design process. Once these characteristics have been specified, the latency characteristics of the architecture can be evaluated via a performance-queuing model. However, architectural decisions are complex and interact. For example, the degree to which changes in the database schema will affect the business rules, Web server, or client software also needs to be analyzed. Each abstraction layer (XML and ODBC) will mask some class of changes and expose others. And each layer will impose a performance cost. Similarly, the addition of a load-balancing component will create additional computation and communication overhead but provide the ability to distribute the load among a larger resource pool.

## 3. QA(QUALITY ATTRIBUTE) AND XP

Requirements elicitation, capture, documentation, and analysis are accomplished in an XP project by developers acting as system analysts. After examining user stories at the beginning of each iteration, developers lead and coordinate requirements elicitation and modeling by outlining the system's functionality and delimiting the system. The result is a specification of details for one or more parts of the system's functionality. A QA, which can enhance this process, would be appropriate for the first iteration of an XP project and aid in identifying key system quality attributes. In a one-day workshop format, facilitators who do not play a stakeholder role are best used as QA analysis team members. The stakeholders attending the workshop include the on-site customer and others with an interest in the system (e.g., end users, maintainers, project managers, members of the development team, testers, and integrators). An exemplary sample of scenarios is refined at the workshop. In subsequent iterations, additional scenarios can be elicited and refined as needed by developers in collaboration with the on-site customer. The inputs to the QA include the business drivers and the architectural plan. Business drivers include the business vision, goals, and key system quality attributes. The architectural plan contains information about development including known technical constraints such as an operating system (OS), hardware, or middleware prescribed for use; other systems with which the system must interact; key technical requirements that will drive architectural decisions; and existing context diagrams, high-level system diagrams, and descriptions. If not already documented, that information needs to be elicited from the customer. Some of it may be implicit in the user stories. The outputs from the QA feed into other practices in XP. For example, business goals are elicited and refined during the QA and could be used by the customer to organize existing user stories, inspire additional user stories, or prioritize requirements along the lines of the customer's business needs. The scenarios can help determine what is in and out of the system's scope and can lead to the creation or refinement of the system context diagram or its equivalent. Scenario generation can also lead to the creation of use cases. Typically, customers develop user stories for requirements and then work on acceptance test cases for the end of development. Many customers do not know how to build these test cases. The QA can give them clues, if not encourage them to build the test cases. This way of using the QA fits in with XP's "test first" or "build for the test" philosophy, as test cases are available to test—early in the development process—whether the code implements the requirements. These test cases can be built as code is being built, so the product of the software development test team can be checked at the end of development.

## 4. DESIGNING SERVICES AS A BASIS FOR REUSE

Software productizing refers to various extra initiatives that can be performed to make software a product, e.g., well-documented, well-tested and therefore deliverable to buyers, and more general in the sense of being prepared for adaptation to different contexts and platforms. We therefore use the term software productizing here for the extra effort needed to make a piece of software more generally usable, i.e., to make it reusable for other software products than the one it is currently developed for. If, at the time development, a developer already has some other concrete applications in mind, for example, some applications that are also currently under development or applications that are planned, the extra efforts in generalizing a piece of software can lead to a quick win. Otherwise, software productizing is about anticipating potential future software development. Then, software productizing must involve analyzing which kind of future applications the several pieces of currently developed software could be useful for. This involves an analysis of how the design of these applications will probably look or it involves efforts for this family of future potential applications. There is an obvious tradeoff between these software productizing efforts and the costs saved by reusing the resulting generalized pieces of software. Software productizing in the discussed sense is somehow conceptually opposite to the current trend of agile software development, with its courage-driven design [Beck 2000; Draheim 2006; 2003; Guta et al. 2009] and continuous refactoring. SOA governance [Holley et al. 2006; Ziebermayr 2010] is the entirety of efforts that contribute towards making the promises of service-oriented architecture a reality [Holley et al. 2006]. Definitions of SOA governance contain high-level descriptions of the problems encountered in projects that try to make service-oriented concepts a reality in an enterprise. Typical SOA governance definitions put a focus onto the operations of software services, i.e., monitoring, controlling and measuring services [Holley et al. 2006]. However, in SOA governance projects it is often expected that a SOA governance expert gives some advise on how to organize the IT development to enable better reuse [Ziebermayr et al. 2007] of existing software across all project boundaries. This heavily affects software process and development team organization issues. For example, initiatives like the Smart SOA [IBM Corporation 2007] best practices catalogue − which by the way must not be mixed up with SEI’s SMART (Service-Oriented Migration and Reuse Technique) approach − try to address these issues. In practice, there is often a very straightforward understanding of SOA

## 5. CONCLUSION

In this paper highlight an apparent gap between the SOA of programming and Extreme Programming. Whilst experience reports may prove useful in highlighting some of the potential problems with the introduction and on-going use of XP in a variety of companies and projects, there is a need for some over-arching understanding about the use of Extreme Programming practices on a psychological level. To the author’s knowledge this work has not yet begun, and the handful of empirical studies which have taken place have done so in an academic environment, or have attempted to answer questions about whether or not a practice is appropriate, without taking any more than an educated guess at why this may be the case. We identified four well-distinguishable visions for service-oriented architecture, i.e., the enterprise application integration vision, the business-to-business-vision, the flexible processes vision and eventually the software productizing vision. We have identified two different styles of service-oriented architecture for enterprise application architecture which are basically distinguished from each other by whether the service tier implements business logic and holds persistent data and coin the terms fat hub resp. thin hub hub-and-spoke architecture for these architectural styles. We characterized SOA governance as an approach to massive software reuse. We have thoroughly motivated SOA governance as an approach to the maintenance of software product variants.

## REFERENCES

[1] AGRAWAL, A. 2007. WS-BPEL Extension for People (BPEL4People), version 1.0. Tech. rep., Active
[2] Endpoints, Adobe Systems, BEA Systems, IBM, Oracle, SAP. June.
[3] ATKINSON, C., BOSTAN, P., HUMMEL, O., AND STOLL, D. 2007. A Practical Approach to Web Service
[4] Discovery and Retrieval. In Proceedings of ICWS 2007 − the 5th IEEE International
[5] Conference on Web Services. IEEE Press.
[6] ATKINSON, C. AND HUMMEL, O. 2007. Supporting Agile Reuse Through Extreme Harvesting. In
[7] Proceedings of XP 2007 − the 8th International Conference on Agile Processes in Software
[8] Engineering and Extreme Programming, Lecture Notes in Computer Science 4536. Springer.
[9] BECK, K. 2000. Extreme Programming Explained − Embrace Change. Addison-Wesley.
[10] BEISIEGEL, M. 2005. Service Component Architecture − Building Systems using a Service Oriented
[11] Architecture. Tech. Rep. Joint Whitepaper, version 0.9, BEA, IBM, Interface21, IONA,
[12] Oracle, SAP, Siebel, Sybase. November.
[13] BEISIEGEL, M. 2007. ASCA Policy Framework, SCA Version 1.00. Tech. rep., BEA, Cape Clear,
[14] IBM, Interface21, IONA, Oracle, Primeton, Progress, Red Hat, Rogue Wave, SAP, Siemens,
[15] Software AG, Sun, Sybase, TIBCO. March.
[16] BERNSTEIN, P. 1996. Middleware: a Model for Distributed System Services. Communications of the
[17] ACM 39, 2 (February), 86−98.
[18] BOX, D. 2000. Simple Object Access Protocol (SOAP) 1.1 − W3C Note. Tech. rep. May.
[19] BROOKS, F. 1975. The Mythical Man-month − Essays on Software Engineering. Addison-Wesley.

[20] BROOKS, F. P. 1987. No Silver Bullet − Essence and Accidents of Software Engineering. IEEE
[21] Computer 20, 4 (April).
[22] BROWN, A. 1997. CASE in the 21st Century − Challenges Facing Existing Case Vendors. In
[23] Proceedings of STEP'97 − the 8th International Workshop on Software Technology and
[24] Enginering Practice. IEEE Press.
[25] BROWN, A., JOHNSTON, S., AND KELLY, K. 2002. Using Service-Oriented Architecture and
[26] Component-Based Development to Build Web Service Applications. Tech. rep., Santa Clara,
[27] CA: Rational Software Corporation.
[28] BUSINESS PROCESS PROJECT TEAM. 2001. ebXML Business Process Specification Schema, Version
[29] Tech. rep., UN/CEFACT, OASIS.
[30] CHOW, L., MEDLEY, C., AND RICHARDSON, C. 2007. BPM and Service-Oriented Archtiecture
[31] Teamed Togehter: A Pathway to Success for an Agile Government. In 2007 BPM and
[32] Workflow Handbook, L. Fischer, Ed. Workflow Management Coalition, 33−54.
[33] COLAN, M. 2004. Service-Oriented Architecture expands the Vision of Web Services −
[34] Characteristics of Service-Oriented Architecture. Tech. rep., IBM Corporation. April.
[35] COMPUTER, C. AND AGENCY, T. 2000. IT Infrastructure Library − Service Support. Tech. rep.,
[36] Renouf.
[37] DEREMER, F. AND KRON, H. 1975. Programming-in-the-Large Versus Programming-in-the-Small.
[38] In Proceedings of the International Conference on Reliable Software. ACM Press, 114−121.
[39] DERLER, P. AND WEINREICH, R. 2006. Models and Tools for SOA Governance. In Proceedings of
[40] TEAA 2006 − International Conference on Trends in Enterprise Application Architecture,
[41] Lecture Notes in Computer Science 4473, D. Draheim and G. Weber, Ed. Springer.
[42] DRAHEIM, D. 2003. A CSCW and Project Management Tool for Learning Software Engineering. In
[43] Proceedings of FIE 2003 − Frontiers in Education: Engineering as a Human Endeavor. IEEE
[44] Press.
[45] DRAHEIM, D. 2006. Learning Software Engineering with EASE. In Informatics and the Digital
[46] Society, Tom J. van Weert and Robert K. Munro, Ed. Kluwer Academic Publishers. 2003.
[47] DRAHEIM, D. AND KOPTEZKY, T. 2007. Workflow Management and Service-Oriented Architecture.
[48] [Bachmann 02] Bachmann, F.; Bass, L.; & Klein, M. Illuminating the Fundamental
[49] Contributors to Software Architecture Quality (CMU/SEI-2002-TR-
[50] 025, ADA407778). Pittsburgh, PA: Software Engineering Institute,
[51] Carnegie Mellon University, 2002.
[52] http://www.sei.cmu.edu/publications/documents/02.reports
[53] /02tr025.html
[54] [Barbacci 03] Barbacci, M. R.; Ellison, R.; Lattanze, A. J.; Stafford, J. A.;
[55] Weinstock, C. B.; & Wood, W. G. Quality Attribute Workshops
[56] (QAWs), Third Edition (CMU/SEI-2003-TR-016, ADA418428).
[57] Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon
[58] University, 2003. http://www.sei.cmu.edu/publications/documents
[59] /03.reports/03tr016.html
[60] [Bass 03] Bass, L.; Clements, P.; & Kazman, R. Software Architecture in
[61] Practice, Second Edition. Boston, MA: Addison-Wesley, 2003.
[62] http://www.sei.cmu.edu/publications/books/engineering
[63] /sw-arch-practice-second-edition.html
[64] [Beck 04] Beck, K. Extreme Programming Explained: Embrace Change,
[65] Second Edition. Boston, MA: Addison-Wesley, 2004.
[66] [Boehm 04] Boehm, B. & Turner, R. Balancing Agility and Discipline: A Guide
[67] for the Perplexed. Boston, MA: Addison-Wesley, 2004.
[68] [Clements 00] Clements, P. Active Reviews for Intermediate Designs (CMU/SEI-
[69] 2000-TN-009, ADA383775). Pittsburgh, PA: Software Engineering
[70] Institute, Carnegie Mellon University, 2000.
[71] http://www.sei.cmu.edu/publications/documents/00.reports
[72] /00tn009.html