Software Evolution: Prototypical Deltas[*]

Eyðun Eli Jacobsen Bent Bruun Kristensen Palle Nowack Torben Worm

The Maersk Mc-Kinney Moller Institute for Production Technology
University of Southern Denmark/Odense University, DK 5230 Odense M, Denmark
e-mail: {jacobsen, bbkristensen, nowack, tworm}@mip.sdu.dk

Abstract

We present a model of the software evolution process. We introduce the notion of a delta, which represents a change in the software's environment, as a key concept for characterizing the software evolution process. A number of prototypical deltas are presented and characterized in terms of the domains, models, and actors involved.

1: Introduction

It is a common observation that almost all software systems change after their inital development and release. We perceive this as software evolution. Software evolution imply several challenges for the software developers and maintainers. The evolution of a software system can have many forms. The objective of this article is to present an abstract view on software evolution in order to expose the commonalities of these forms rather than to discuss the variations. In our view on software evolution we apply a perspective which expose certain properties of the software life-cycle. The properties become the essential elements of the resulting abstract model of software evolution.

We base our work on a previosly developed model of the software *development* process [8]. The model defines a number of participant roles, domains, and models that make up an abstract view on the development process. In this paper the model from [8] is used as the basis for examining the software *evolution* process — evolution is seen as combinations of basic changes to the domains etc. of that model. The overall purpose of the research conducted in this article and in [8] is to expose, understand, and characterize various aspects of the software life-cycle — in [8] the life-cycle seen from a software development perspective, and in this article the life-cycle seen from a software evolution perspective.

We use the term "software development process" for the result of applying a development perspective on the software life-cycle. Similarly we use the term "software evolution process" for the result of applying an evolution perspective on the software life-cycle. The research perspectives *development* and *evolution* are illustrated in Figure 1. The researcher roles apply the development and evolution perspectives on the software life-cycle to describe abstract models corresponding to the perspectives. This article and [8] may be seen as (descriptions of) the two models — they support the understanding of the development and evolution processes by their restricted characterizations of the processes.

[*]This research was supported in part by Danish National Center for IT Research, Project No. 74 and in part by ATV, Project No. EF-659.

For the purpose of describing the research conducted we choose to see the software life cycle system itself as divided into problem and usage domains. Therefore, our models of the software life cycle both have models of both these domains — the problem domain model captures the artifacts created and used during the life cycle, and the usage domain model captures the actors and interactions of the development and evolution processes [1].

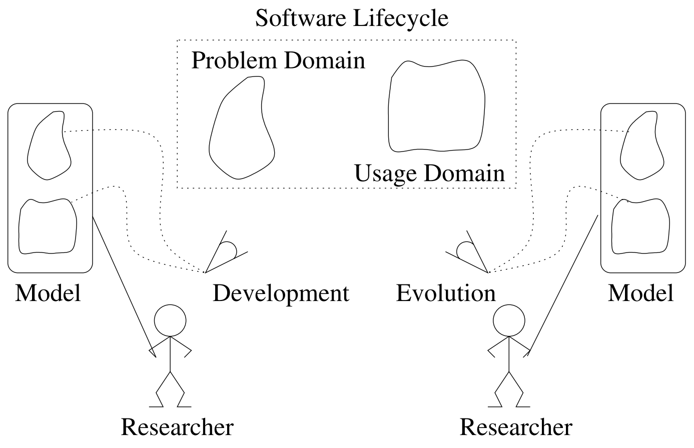


Figure 1. Research Perspectives

The research presented in [8] is seen as the result of an analysis (and a model construction) of the software development process with most emphasis on the problem domain. The focus is on the description of concepts and phenomena that exist during software development and have to be maintained through a usage domain. Consequently, as opposed to that, we see the present article as an analysis (and a model construction) of the software evolution process with most emphasis on the usage domain — and with focus only on evolution aspects. The focus is on the actors in the evolution process and the usage patterns of these actors. The prototypical changes to a software system then become potential usage patterns of the software evolution process. We adapt the problem domain model of the software development process as our problem domain model for the software evolution process.

We believe that there are no "pure" development or evolution processes — any real-world process will contain aspects of both these pure processes. Rephrased in a more positive wording, there are no real-world software cycles where the involved actors can't benefit from applying both perspectives. The use of these restricted perspectives on the software life-cycle is exactly that they allow us to form corresponding abstract models where the characteristics of the perspectives are exposed and characterized. As such the models support our understanding of the software life-cycle. The models can be seen as elements which could be applied as part of methodologies for software development or evolution. It is not the intention of our research to provide concrete techniques or process-steps to be followed. By the restricted perspectives some overall aspects are lost in return for the abstract understanding. We describe the software life-cycle for purpose of understanding — we do not prescribe how to conduct a process.

Paper Organization. In Section 2 we introduce our basic model of software evolution and we relate it to the development model refered to in the introduction. Section 3 introduces the concept

[1]Note that both the problem and usage domains in Figure 1 are defined in relation the the software developers and hence they are not the same as the problem and usage domains described in [8], which are defined in relation the end-user.

a delta. Section 4 describes a number of prototypical deltas. Section 5 analyzes and characterizes the prototypicla deltas. Section 6 contains a brief summary of the major contributions of the work described in the paper.

2: Software Evolution

This section presents a context and the key concepts for characterizing software evolution processes. We present an overall picture of software processes by discussing the domains, the models, and the actors involved in these processes. We continue along the lines in the discussion of research perspective (Section 1) and briefly discuss the two idealized software processes *development* and *evolution* in relation to each other. Finally, we introduce the notion of a delta as the key concept for characterizing software evolution processes.

2.1: Software Development Model

This section presents the overall picture of software processes in terms of the involved domains, models, and actors. Figure 2 illustrates the domains, the models, and two actors involved in software processes (cf. [8] for a more detailed illustration). Below we describe the domains and the models (most of these are similar to those in the analysis model of [13]).

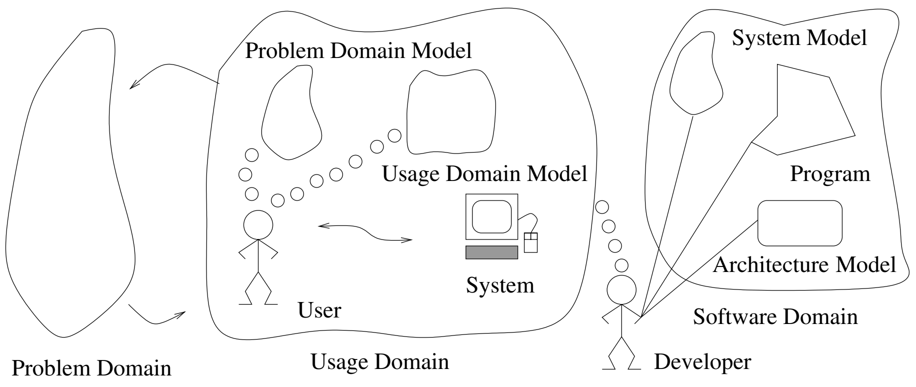


Figure 2. Domains and Models involved Software Processes.

The *Problem Domain* is the part of the surroundings which are is managed, monitored or controlled by a system. The *Problem Domain Model* is an explicit model of the user's understanding of the problem domain — it is developed by the developer through some given process, expressed in some given notation, and seen from some chosen perspective.

The *Usage Domain* is an organization that manages, monitors and controls a problem domain, including the users. The *Usage Domain Model* is an explicit model of the user's understanding of the usage domain — it is developed by the developer through some given process, expressed in some given notation, and seen from some chosen perspective.

The *Software Domain* constitutes the descriptions, which can be partial or complete, of software. The description can design descriptions in various design notations and they can also be descriptions in the a programming language.

The *System Model* forms the developer's conception of the integration of the problem domain models and usage domain models at an abstract level. It is a refined, transformed and enriched

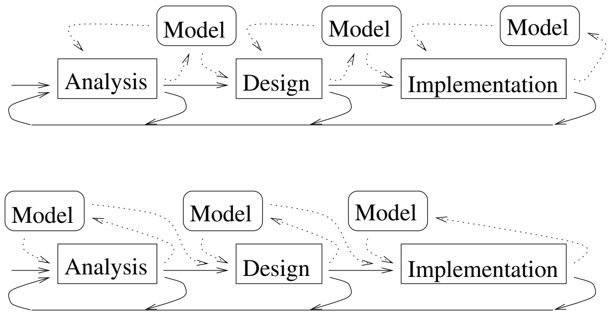


Figure 3. The Development & Evolution Processes

problem domain model. The system model also supports the usage of the system in the sense that it is executable.

The *Architecture Model* forms the developer's conception of the architecture of the system, i.e. the overall structures and their relations and interactions. It is an abstract model over over the system model in the software domain. The model focus on (the organization of) the structure and interaction embedded in the system model. The purpose is to understand the system model from some perspective, to allow the developer to reason about and expose the support for the non-functional requirements, and to map the system model onto the logical platform.

The *Program* is a description of the system in a programming language. We use the term "program" for any collection of source code, i.e a program can also be a set of related programs.

Development Process & Evolution Process When characterizing software development and software evolution, we need to consider the aspects of *iterative* and *incremental* development processes. A software process is iterative and incremental when it consists of a number of iterations. Each iteration is a transformation process where different software parts, models, and domains are considered input and modified software parts, models, and domains are considered output. An increment is any modification of the software parts, models, and domains, which is believed to add to the quality of the end-result. During each iteration the participants in the process perform activities which can be individually categorized as having primarily focus on requirements specification, analysis, design, implementation, and test. The actual blend of these activities is very dependent on the state of the software process: are we in the initial and exploratory phase, or are we in the finalizing phase? This again relies on the context of the software life-cycle: e.g. developer experience, the availability of reusable artifacts etc. We believe that any software process is iterative and incremental. However, the iterations and increments between the starting point and the finishing point can be undocumented or perhaps not even perceived by the developers performing the iterations.

We can investigate a software process from the view-point that it is a development process and from the view-point that it is an evolution process. A real software process will be a mixture of being a development process and an evolution process. The motivation for discussing these two view-points is that they allow us to investigate and expose different aspects of a software process. The development process is treated in more detail in [8].

We illustrate the difference between a development process and an evolution process in Figure 3. Figure 3 illustrates a relation between phases and models in a software process. In a development process the various phases result in various models, and the input to a phase consists primarily of the models from the previous phase. In an evolution process the input to a phase is primarily existing models which resulted from the same phase at an earlier point in time.

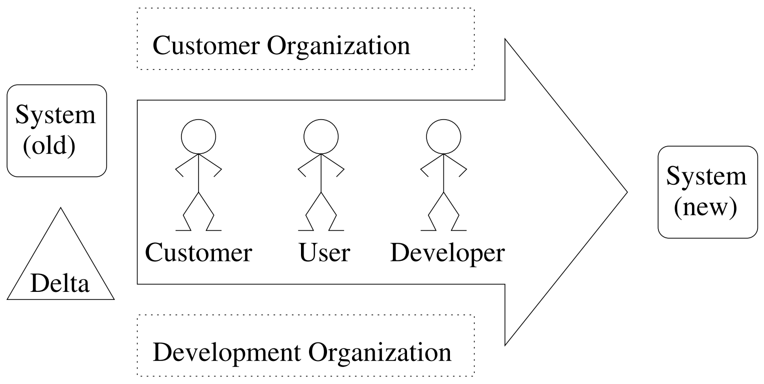


Figure 4. System Evolution

The evolution process is similar to the development process, but is different in essential ways. The evolution process is also iterative — the process changes *complete* models. The models exist and they are the product of other development or evolution processes. The existing models are the product of other developers' work, and the models must be comprehended by the developer role. In short, the evolution process differs from the development process by having complete models from all phases at its beginning and by its focus on changing these models — the development process, in contrast, begins with no models and it focuses on building new models.

2.2: Software Evolution Process

An overall picture of the evolution process is given in Figure 4. We have a customer organization and a development organization. These are both seen as roles, that represent the actual organizations taking part of the evolution process. Also we have the roles of the participants, namely the customer, the user and the developer. These are also roles, that represent the actual persons taking part in the process. To model the evolution process we both refine the roles of the development process and add new roles. As an example we introduce the *documentary* role (not included in Figure 4) in order to emphasize the importance of the existing system. The documenter has knowledge about the existing system — in a sense this role represents the documentation of the system as well as the knowledge that was available during the development of the system. It is obvious that the documenter role can only be approximated because some of this knowledge may be lost, but in this respect the role is similar to the existing roles, which also describe the ideal situation.

The main difference from the development process is that an (old) version of the system exists — an essential part of the evolution process part is the old version of the system. The goal of an *evolution step* is to transform this system into a new version of the system. An evolution step has many forms. We introduce the evolution *delta* to capture the nature of any evolution step of a software system. An evolution delta is characterized by it's associated roles, domains, models and it's consequences. The objective of the introduction of the notion of the delta is precisely to obtain a general characterization of the possible delta's in terms of these elements.

An important aspect of the consequences of a delta is the dependencies among a number of roles, domains and models — given an initial delta how does it spread throughout the model of the organization of the development and evolution process?

We introduce elementary delta's and composite delta's, where a composite delta is defined in terms of other (part-) delta's. A composite delta implies a decomposition into the part-delta's and a subsequent composition of the results from these delta's. We regard these operations on deltas as special cases of the general abstraction processes, the generalization/specialization, and aggregation/decomposition processes, and we obtain a very general and simplified understanding of the elementary change to a software component.

3: Delta Definition

The objective of this section is to characterize the software evolution process in terms of a number of deltas. The overall objective of introducing the notion of a delta is to be able to classify and decompose real life changes by means of standard, intuitive delta types, and to develop a general and applicable framework for the description of new, possibly unique types.

Delta Definition. A delta is *a change to a software system*. The change may be envisioned, be decided, or may implemented — all these and similar situations are covered by our notion of a delta. The actual situation is a matter of the state of the delta.

Delta descriptions can be seen as abstract notions which can be instantiated by determining the actual elements of the delta — i.e. its actual actors, domains and models, and status information. The actual situation is captured by the state of the instance of a delta. A delta description consists of a *collection of actors*, a *collection of domains and models*.

Domains and Models in Deltas. Domains include problem domain, usage domain, functional requirements, non-functional requirements and logical and physical platform. Models include problem domain model, usage domain model, architecture model, system model and program.

Domains and models in general play either a *source* or *destination* role (or both) in a delta:

Definition: *source:* This captures the reasons for addressing the delta. A situation in one or several domains or models makes a given delta relevant for consideration.

Definition: *destination:* This captures the consequences of addressing a delta. A delta can either be limited to its own domain, or it can have consequences for other domains also.

We notice that the program as a model usually plays the role as destination and never the role as source. Also a number of default dependencies exist between various domains and models. These include for example that a change in the system model certainly must have consequences for the program. These (default or derived) dependencies are discussed later in a characterization of domains and models and are not included explicitly in the following examples and discussions. We only discuss the source of a delta and illustrate the destinations in Figures 6(a), 6(b), and 7(a) in Section 5.3.

Actors in Deltas. Actors include customer, end user, domain expert, software developer (sw-developer) and information technology planner (it-planner). We see these as specializations of actor. We see end user and domain expert as specializations of an abstract user actor. We also see sw-developer and it-planner as specializations of an abstract developer actor.

The characteristics of the actors are:

Definition: *Customer:* The customer makes decisions about the evolution (status quo, upgrade, disband etc) of the software system.

Definition: *User:* The end user uses the software system as a tool in this work. The end-user has detailed knowledge about his own work.

Definition: *Domain expert:* . The domain-expert's responsibility is to design business processes in the customer's organization. The domain-expert has knowledge of the whole business domain (the main difference from the end use who has detailed knowledge about only a limited part of the business the domain-expert has overall knowledge of the whole business).

Definition: *Developer:* The sw-developer has competence in software development. A sw-developer builds the software system.

Definition: *IT-planner:* The it-planner has knowledge about the computer technology that the company uses and the technology that is forthcoming on the market.

Processes in Deltas. When discussing software engineering methodologies from a development perspective a number of phases in the software life-cycle are typically identified [9]: inception, elaboration, construction, and transition. Each cycle goes through these four phases. As mentioned in the Introduction we also typically identify a number of activity types that are applied throughout the process: requirement specification, analysis, design, implementation, and test. The engagement into these activities are distributed heterogenously over the different phases of the software life-cycle.

Both the above mentioned phases and activities can be considered classifications of human activities in relation with the software life-cycle. We propose that when applying a software evolution perspective a different (additional) categorization of human activities is feasible. We describe this in the following.

In the *idea phase* an *initiator role* realizes that a change my be appropriate or neccessary. The initiator perceives a situation as problematical and tracks the problem to one or more aspects of the software system. This reflects that a delta is discovered or realized by someone at some point in time. The initiator has knowledge about the domain(s) which are the source(s) of the delta. The initiator may or may not have a solution proposal for the actual improvement and change.

In the *proposal phase* a *decider role* determines that a delta should be addressed. The decider has the power to decide whether a delta should be implemented or not. Secondly a developer role outlines a solution proposal. The developer takes into account the potential suggestions from the initiator role and the potential constraints imposed by the decider role. The proposal can be of a more or less technical and/or organisational nature, which of course reflects both the nature of the delta and the skills and preferences of the actual person playing the developer role. In general however the proposal is primarily of a technical nature and deals with identified and delimited parts of the system. Several proposal from multiple developers that all deal with the same delta are of course feasible. In this case a developer role (a specialized coordinator role) manages the development and evaluation (see below) of multiple solution proposals.

In the *evaluation phase* a decider role determines whether to implement the proposal.

In the *implementation phase* a developer role implements the delta. The *affects role* are informed about the changes to take place — not only the type of changes, but the actual concrete changes in the use of the system. The delta implies a change for this actor in relation to the software system. Finally, based on input from the affects role, the decider role accepts the revised system.

Considering the people involved in an evolution process we thus can categorize these according to two orthogonal sets of actor roles: (a) the domain-determined roles: customer, user, domain expert, developer, it-planner, and librarian; and (b) the process-determined roles: initiator, decider, and affect.

The idea, proposal, evaluation, and implementation phases described above are generalized to apply to all deltas. It is possible to envision specialized versions of the four phases that would better describe the concrete process of individual type of deltas (see the following section about prototypical deltas). We have not considered such specialized processes.

4 Prototypical Deltas

Below we describe various prototypical deltas by relating them to actors and domains/models. Table 1 summarizes the prototypical deltas. The table describes the deltas in the terms of the actors involved in the initiation and decision, the actors affected by the delta, and the domain in which the delta originates. The domains and models affected by the delta are not shown in the table but discussed in Section 5.3.

Δ#	Delta type	initiator	decider	affects	source
1	*System Combination/ Restructuring*	domain expert	customer	end user	problem domain usage domain
2	*Platform Exchange*	it-planner	customer	end user	physical platform
3	*Usage Addition/ Improvements*	end user domain expert	customer	end user	usage domain
4	*Problem Extension/ Modification*	end user domain expert	customer	end user	problem domain usage domain
5	*Usage-Quality Improvement*	end user domain expert	customer	end user	non-functional requirements logical platform physical platform
6	*Development-Quality Improvement*	sw-developer	customer[a]	sw-developer	non-functional requirements
7	*Administration Extension*	customer it-planner	customer	customer it-planner	usage domain

[a]In this case the customer may be from either the internal development organization, e.g. the product of applying the delta is a framework and the customer is a manager, or he may be from an external organization demanding e.g. certain reusability requirements fulfilled

Table 1. Prototypical Deltas

System Combination/Restructuring. Someone realizes that a system could be improved upon if it worked together with another system, e.g by utilizing the other system's data. This might trigger a delta to both systems so that they should work together. This delta is realizes by the domain-expert and the customer decides that it should be addressed. Since both systems provide each other (or maybe only one way) with data this delta affects the problem domain of the system and also the usage domain since new use cases are related to the new data.

Platform Exchange. This prototypical delta is concerned with a change of technology. We see this as an exchange of the logical/physical platform.

This delta is realized by the it-planner who evaluates the benefits of staying with the old technology and the benefits of changing to the new technology, for example databases or networks. The customer decides that the new technology should be used. The consequences of changing to the new technology depend on the differences between the technologies and the coupling between the software system and the technology. If the technologies are relatively similar and the coupling between the old technology and the software system is low, then the consequences might be limited. On the other hand if the technologies are radically different and the software system is tightly coupled to the old technology, then the consequences of this change are more serious.

This delta arises from technology, and does not affect or trigger any delta in the problem domain nor the usage domain. The spectrum discussed above might also be expressed in terms of the logical platform vs. the physical platform for the system. The delta is a change in the physical platform and the impact of the delta is dependent on how well the changed physical platform conforms to the logical platform.

Usage Additions/Improvements. New features are needed (new use cases) to support that new ways of using the system are envisioned, or realized. There is no new data in the new interactions between the user and the system, but data might be used in new contexts or together with data that it didn't occur with before. An example of this delta could be the situation that new browsing/searching facilities are needed. This kind of delta is realized either by the end-user and the domain-expert. The developer estimates the costs of changing the system and the customer decides what to do. This delta arises in the usage domain, and the derived deltas can have a small or big impact on the software domain. The changes needed in the software domain are of course the inclusion of new interface facilities and possibly further changes are needed. If the data in the system will be used in a very different way than they used to be used, then reorganization or addition to the functional components might be needed. Also reorganization of the database layer might be needed, if the access patterns of the data now are poorly supported by the current database organization.

Problem Extension/Modification. New data must be administered, probably in new ways. The addition of new data represents a change in the problem domain of the system - the system is extended to administer different kind of data. This change is realized by the domain-expert or the end user. If the suggestion comes from the domain-expert then the data plays a role in relation to several of the systems of the organization. If the suggestion comes from the end-user, then the data is an important part in the specific job-function of the end-user.

This delta arises in the problem domain and also in the usage domain. The changes triggered are the addition of functionality, model components, and database organization. This change can on the one extreme just be an add-on which does not tightly cooperate with existing parts of the system, or it can on the other extreme be an add-on that "blends" with existing parts of the system which have to be changed to accommodate for the new add-on.

Usage-Quality Improvement. Various usage-qualities could be improved upon such as for example performance, scalability, deployment and stability — all examples on non-functional requirements. These kind of changes are "under the hood" changes - the surface (or functionality) of the system is unchanged.

Performance is increased by paying attention to how the system is used and then to use technologies that fit the usage pattern better than the current technologies do. This delta primarily takes place in the software domain. The functional component, model component, and database of the system might be reorganized. Specific technologies, such as databases, networks and deployment hardware or auxiliary hardware, might be replaced. Performance problems are realized by the end-user, who finds the software unsatisfactory (from being annoying to being useless) in his job. Depending on the degree of satisfaction the customer decides how to address this delta. Performance problems can also be realized by the end-user of a system which is dependent on this system.

Scalability is increased by using more appropriate components (databases, hardware, logical organization (architecture)). Addressing the scalability of the software increases the price (more expensive databases etc, and because the software developer can not use a trivial organization of the software. Scalability is the question of how well the software can handle assignments of different sizes. This delta is realized by the domain-expert. The domain-expert recognizes that a new assignment is conceptually similar to an existing assignment for which the organization has a software system. Should the existing system be changed, or should a new system be built? This is estimated by the developer and the customer makes a decision.

Deployment in many places has varying consequences depending on how similar the deployment places are to each other. If the systems are very similar then the change needed to the software is generalization. If the systems are very different then it might be necessary with separate systems for each deployment place. In terms of the logical platform we can cast this issue as a question of whether the deployment platforms can meaningfully be described by the same logical platform or whether they have separate logical platform. Also (and more important) fundamental changes might be needed to the function component and model component and the databases component because of the distribution aspect (if it is changed to a distributed application).

Stability of the system is addressed by explicitly making assumptions about the failure of services on which the system depends, and by introducing multiple instances of the same service in order to ensure redundancy in the system. This need is recognized by the end-user who is bothered by system crashes triggered by events which are natural in his environment.

Examples might be a situation where some auxiliary hardware is malfunctioning (a sensor of some kind) or that a specific service is unavailable. System-crashes are accepted to varying degrees depending of the importance (from web surfing to life support) of the system.

Development-Quality Improvement. Various development-qualities could be improved upon such as for example modifiability, design with reuse, design for reuse — all development related non-functional requirements. The system is changed in order to better accommodate changes etc. This delta is recognized by the sw-developer. The motivation for this delta can be of three kinds.

The first kind of motivation is that the sw-developer wants to be better able to modify the software system on the request of the customer. Regular request for changes occur, and therefore the system is structured and generalized in a way such that new modifications of a certain kinds are easier to execute. This delta can affect all parts of the software, from the interface structuring to the structure of the model component.

The second and third kind of motivation come from the developer's organization and are related to design with reuse and design for reuse. Design with reuse might be a motivation if the outcome of a new developments in the developers organization might be re-used in this software system. Also, if similar software systems are being developed in the organization then it might be a good idea to develop some components that both systems can use. The third motivation is that an outcome of this project might be useful in another project and therefore a specific component in this project is thoroughly developed such that it can be used in the other project as well.

Administration Extensions. The system is extended with various logging/statistics capabilities — some kind of side functionality might be added to the system. This functionality is related to the operation of the system. Examples include usage profiles such as end-user profiles, usage in relation to time of day and various other statistics.

This delta can trigger deltas with high impact all over the system depending on which parts of the system should be monitored. This delta can be recognized by the customer or by the it-planner. The customer wants this facility if he wants to monitor his employees. The it-planner wants this kind of facility if the operation (technical aspects) of the software systems should be monitored.

5: Characterization of Deltas

In the previous subsection we have listed a number of prototypical deltas. We find such prototypes more important than possible canonical prototypes that are non intuitive, but still relevant. However, we assume that it is possible to describe a number of more or less conceivable basic deltas. We also assume that any real life delta can be seen as a combinations of such basic deltas — the real life delta is a composition of basic deltas. Similarly, and more relevant because of the intuitive nature of the prototypical deltas, we find that many real life deltas can be composed of prototypical deltas.

5.1: Basic and Composite Deltas.

In this section we present a view on how to decompose the prototypical deltas into basic deltas and how to compose and abstract these basic deltas into composite deltas. The model used to reason about the abstraction over deltas uses the familiar concepts *aggregation*, *decomposition*, *specialization*, and *generalization*.

If we look at the application of deltas as stated above we may assume that deltas are applied in a sequential manner, i.e. $\Delta_1, \Delta_2, \ldots, \Delta_n$ and thus the new system is a result of applying these deltas in the specific order. Because each delta affects specific models in the system depending of the delta type we cannot assume that we can apply the deltas in arbitrary order and still end op with the same result.

This have some implications if we wish to compose a number of basic deltas into a composite and complex delta as shown in Figure 5. The figure shows the application of the same deltas in two different ways. In the upper half of the figure the three deltas are applied in sequence and in the lower half of the figure the two first deltas are combined into one composite delta and applied before the third delta. These two approaches may be equivalent if the composed delta consists only of the concatenation of the deltas but they may also be different if the processes of the two deltas are intertwined.

A delta has a number of properties depicted in the figure by the little “handles” on the deltas. The description of these properties are shown in the figure as the little “blob” at the bottom of the delta. An important property to consider when combining deltas is the process property because the order of application of deltas is important; the other properties may be a trivial combination of the properties of the basic deltas, i.e. aggregation of the properties.

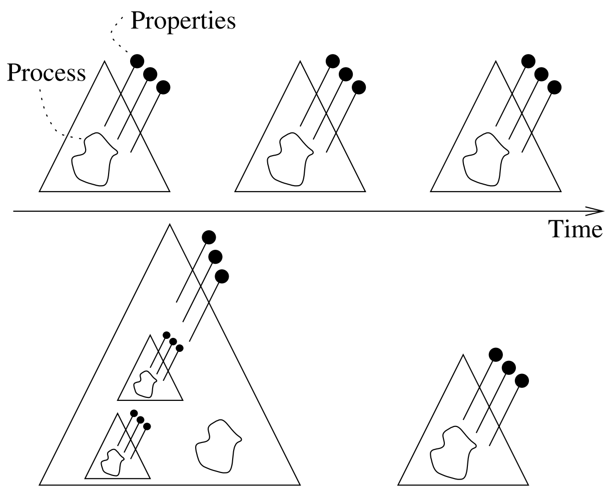


Figure 5. Delta Abstraction

Because the deltas stated in this article are prototypical the properties of the deltas might not span all possible properties and thus a model for abstraction over deltas must consider the addition of more properties. This calls for a model equivalent to the model the object-oriented paradigm is based upon.

The introduction of this model implies that the delta concept is a concept it is possible to build abstractions with in a manner similar to the familiar object-oriented approach. It is e.g. possible to build hierarchies of delta types by means of specialization and generalization as well as to aggregate and decompose them.

If we introduce the notion of basic deltas from which more complicated deltas can be built, e.g. the prototypical deltas described in this article, then we can either decompose these deltas in their constituent parts or build new prototypical deltas from these deltas and other (basic) deltas.

We could describe a combination of say the prototypical delta *Usage Additions/Improvements:* and a new (rather artificial, but still intuitive) basic delta *Pure Problem Extension:* to obtain the prototypical delta *Problem Extension/Modification:* as follows:

Usage Addition/ Improvements	end user domain expert	customer	end user	usage domain	usage domain
Pure Problem Extension	end user domain expert	customer	end user	problem domain	problem domain
Problem Extension/ Modification	end user domain expert	customer	end user	problem domain usage domain	problem domain usage domain

In this example the new delta is purely derived as a simple composition of the properties from the two constituent deltas but the same effect could have been obtained by using the "Pure Problem Extension" delta as a generic delta, and then specialize this delta to the "Problem Extension/Modification" delta by adding the properties of the "Usage Addition/Improvement" different from the generic type.

It is not the purpose of this article to explore these aspects of deltas thoroughly, but to introduce the aspects for further study.

	Roles		
Δ#	init	decide	affects
domain expert	1,3–5	–	–
customer	7	1–7	7
end user	3–5	–	1,3–5
it-planner	2,7	–	7
sw-developer	6	–	6

(a) Actors and Roles

	Domains					
Δ#	PD	UD	FR	NFR	LP	PP
domain expert	1,4	1,3,4	–	5	5	5
customer	1,4	1,3,4,7	–	5,6	5	2,5
end user	1,4	1,3,4	–	5	5	5
it-planner	–	–	–	–	–	2
sw-developer	–	7	–	6	–	–

(b) Actors and Domains

Table 2. Characterization of actors

5.2: Characterization of Actors.

According to the definitions previously given, an actor plays one or more of the roles: *initiator*, *decider*, and *affects* in the process of applying a delta. Furthermore the actors play other roles in the development process as shown in the first column of Table 2(a) and (b).

We are treating prototypical deltas and thus we cannot infer broad generalizations from table 2(a) and (b), but we may extract some tendencies.

The actors come from two organizations (actually organization roles), the customer organization and the developer organization, but the actors are independent of these organizations in the sense that any actor can come from any of these organizations.

We characterize actors according to the prototypical deltas. Actors seem to play certain roles in the prototypical deltas more often than others. We characterize the actors according to the roles that they play in the deltas of the previous subsection.

Actors seem to be engaged in certain domains and models in the prototypical deltas more often than others. We characterize the actors according to the domains and models in which they are engaged in the deltas of the previous subsection.

Table 2(a) shows in which deltas the actors assume which roles. If we e.g. look at the domain expert we see that he only plays the role of initiator in the prototypical deltas we present here. This is not to say that he may not assume the role of either decider or affects in another prototypical delta.

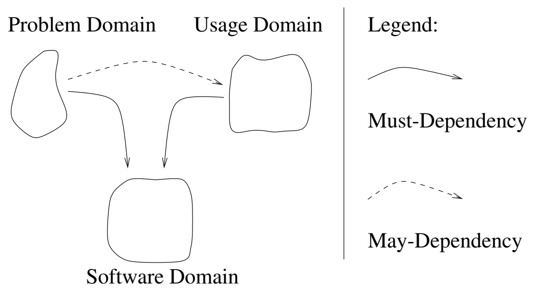
It is clear that the customer always is the decider and that the customer seldom directly initiates or is affected by a delta. This interpretation of the table should not be taken too literally because the table shows an ideal world. In the real world the decision is more likely to be based on interaction between the customer and the other roles.

It is interesting to note that the role initiating a delta not necessarily participates in the evaluation of the delta in the affects role.

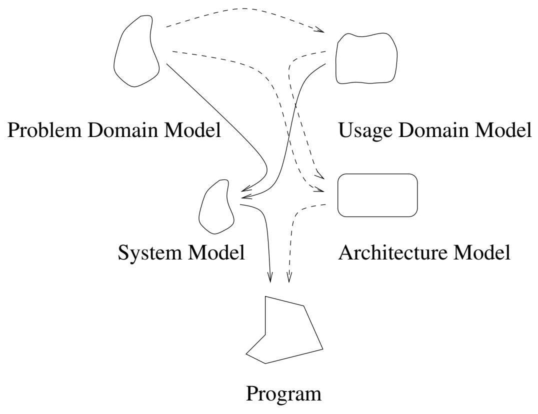
Table 2(b) shows which roles and domains are involved in a given delta.

5.3: Change Dependency Graph.

We characterize the domains and models in terms of the inherently given dependencies between these. A domain or model depends on a change in other domains or models. This dependency graph is general in the sense that it specifies valid change dependencies independent of the (prototypical or actual) delta. This means that to understand the consequences of a delta, not the delta alone but also the dependency graph are necessary. The dependency graph specifies areas for derived potential or necessary changes.



(a) Domains



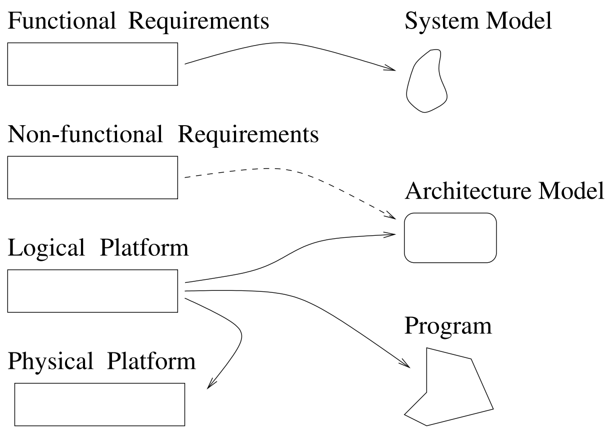
(b) Models

Figure 6. Dependency Graphs

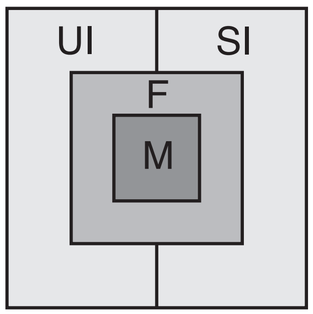
In the following we describe the dependency graph in logical portions. In Figure 6(a) we focus on the dependencies among the domains only. A change in the problem or usage domain must have influence on the software domain — because the various elements in the software domain are derived from the models of these domains any change in these domains must be reflected in the software. Also a change in the problem domain may have influence on the usage domain — because the problem domain determines what is taken care of by the system any change here may be reflected in how we take care of things as captured by the usage domain. We distinguish between *must* dependencies and *may* dependencies to reflect the fact that in some cases a change somewhere must necessarily imply some change in some other place, whereas in other cases a change may or may not imply another change. In Figure 6(a) we illustrate these dependencies between the domains — must dependencies by solid arrows and may dependencies by dotted arrows. The relation between the problem domain model and the problem domain is a must dependency — similarly for the usage domain model and the usage domain. These inherently given dependencies are not illustrated.

In Figure 6(b) we focus on the dependencies among the models. The may dependency between the problem and usage domain is inherently reflected directly in an identical dependency between the problem and usage domain models. The software domain includes three models between which certain dependencies exist. Any change in either of the problem and usage domain models must have effect on the system model, because the system model is an explicit reflection of these models. Similarly the program is an explicit transformation' from the system models — also an example of a must dependency. When we turn from the system model to the architecture model the same dependency structure is valid, however the must dependencies are replaced by may dependencies. This is due to the nature of the architecture model, which may or may not be influenced by the changes dependent on the type of the changes. Similarly, the program may not be influenced by a change in the architecture model if this is of logical nature only.

Figure 7(a) illustrates the dependencies between, one one hand, the requirement and platform domains and, on the other hand, the software domain. The functional and the non-functional re-



(a) Dependency Graph: Requirements and Platforms



(b) Consequences of Deltas

quirements are independent — these are two dimensions in the requirement universe. The system model has a must dependency to the functional requirements, because the system model is the place where the functional requirements are implemented. A change in the non-functional requirements may not be reflected in the architecture model, because the same architecture may support a variety of such requirements. A change in the logical platform must imply a change in the physical platform, because the physical platform is the realization of the logical platform. A change in the logical platform must influence both the architecture model and the program, because the program (together with the physical platform) includes the realization of the logical platform. We see the architecture model so closely related to the logical platform that a must dependency between these is valid — the logical platform is mirrored in the architecture model.

5.4: System-Internal Effects of Deltas

We regard a software system to consist of an end-user interface, a system interface, a function part, and a model part as depicted in Figure 7(b).These parts of a software system are abstract and their presence do not imply any specific architecture of the software system. They can be perceived as logical classification categories for aspects of software systems.

Changes that result from a delta as described previously in this section can affect one or more of the four logical parts. Once a part has been changed it is informative to consider which other parts need to change.

In general the partitioning of a system into the four logical parts is the expression of a separation of concerns. The model part is considered most static as it is developed on the basis of the problem domain which tends to be fairly static for long periods of time. The interface parts are considered most fragile as they deal with presentation and interchange of data. External interfaces are not possible to control inside a project and they are likely to change. In between the interface and model parts, the function part captures the system aspects that connects the interface (presentation) and the model (data). The functions are considered to change with a frequency in between those of the interfaces and the model.

These arguments for the division into the four parts already suggest that the effects of change is sought to be logically localized and thus (ideally) globally minimized. In practice however the effects of change is very dependent on the actual (architectural) design: how is the four logical parts made manifest in the system. Nonetheless we believe that the following two observations are characteristic of systems design:

Deltas affecting the problem domain implies a change to the model part of the system. Depending on the nature of the delta the change typically propagates into the function part of the system, and into the interfaces of the system. Deltas affecting the usage domain implies a change to the function part and/or user interface parts of the system. In general it should not be necessary to change the model part of the system.

6: Summary

Related Work. Our model of the software development process is only one out of many. In fact each object-oriented software development methodology including [2], [10], [13], [15], [18] has its own underlying, implicit or explicit, model. The intention of our model is that it to a large extent subsumes these methodologies when seen abstractly from the perspective of the involved domains and models. However, none of the methodologies include substantial elements of the evolution aspect of the software life cycle.

The architecture model of the software development model is the least well understood. Our understanding of architecture as abstractions over the software domain is presented in [9]. The framework (for example [11]) and pattern ([4], [5], [12], [14], [17]) technologies both support aspects of software architecture. The description of software architecture is also supported by means of architecture languages or dedicated architectural patterns ([1],[3], [6], [7], [16]).

Characterization. The article is based on experience from real life development and evolution processes as well as from our experience as supervisors of student projects. The empirical foundation for the analysis in the article are the prototypical cases — the cases are seen as a an informal summary of our experience, and they serve as inspiration for the analysis. The accuracy of our results relies heavily on the soundness and completeness of these cases.

The article presents a model for the evolution process in terms of the notion of delta's. The delta's can serve both as a theoretical model for understanding the fundamental ingredients in the evolution process, but also as a means of describing and performing actual instances of software evolution.

The most important results of the article are:

- The investigation of the pure evolution process separate from the development process as a matter of research strategy, and the resulting understanding of evolution as the more general and fundamental process — leaving the development process as the (usually discussed and presented) special case.
- A definition of the notion of a delta, and the description of prototypical cases as delta's.
- The characterization of delta's including 1) the possibility of aggregation and classification hierarchies for delta's, 2) the characterization of delta's according to their participating roles and source domains, 3) the inherently valid dependency graph of derived domains and models for potential changes, 4) the consequences of delta's to logical interface, function, and model parts of systems.

References

[1] L. Bass, P. Clements, K. Kazman: Software Architecture in Practice. Addison-Wesley, 1998.

[2] G. Booch, J. Rumbaugh, I. Jacobson. The Unified Modeling Language User Guide. Addison Wesley, 1998.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. Pattern-Oriented Software Architecture: A System of Patterns. Wiley & Sons, 1996.

[4] J. O. Coplien, D. C. Schmidt: Pattern Languages of Program Design. Addison-Wesley, 1995.

[5] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[6] D. Garlan, M. Shaw: An Introduction to Software Architecture. In: Advances in Software Engineering and Knowledge Engineering, (ed. V. Ambriola, G. Tortora), World Scientific Publishing Company, 1993.

[7] D. Garlan, D. E. Perry: Introduction to the Special Issue on Software Architecture. IEEE Transactions on Software Engineering, Vol. 21, No. 4, 1995.

[8] E. E. Jacobsen, B. B. Kristensen, P. Nowack: Models, Domains and Abstraction in Software Development. Proceedings of International Conference on Technology of Object-Oriented Languages and Systems, 1998.

[9] E. E. Jacobsen, B. B. Kristensen, P. Nowack: Characterizing Architecture as Abstractions over the Software Domain. To be published, 1999.

[10] I. Jacobson, G. Booch, J. Rumbaugh. The Unified Software Development Process. Addison Wesley, 1998.

[11] R. E. Johnson, B. Foote: Designing Reusable Classes. Journal of Object-Oriented Programming, 1988.

[12] R. Martin, D. Riehle, F. Buschmann: Pattern Language of Program Design, 3. Addison-Wesley, 1997.

[13] L. Mathiassen, A. Munk-Madsen, P. A. Nielsen, J. Stage: Objektorienteret Analyse og Design. (In Danish) Marko 1997.

[14] W. Pree: Design Patterns for Object-Oriented Software Development Addison-Wesley 1995.

[15] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen: Object-Oriented Modeling and Design. Prentice Hall 1991.

[16] M. Shaw, D. Garlan: Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, 1996.

[17] J. M. Vlissides, J. O. Coplien, N. L. Kerth: Pattern Languages of Program Design, 2. Addison-Wesley, 1996.

[18] R. Wirfs-Brock, B. Wilkerson, L. Wiener: Designing Object-Oriented Software. Prentice Hall, 1990.