Cognitive Patterns for Software Comprehension: Temporal Details

Adam Murray
armurray@ca.ibm.com
Doctoral Fellow
Center for Advanced Studies, IBM Rational
Ottawa, Ontario, Canada

Timothy C. Lethbridge
tcl@site.uottawa.ca
University of Ottawa
SITE, 800 King Edward Avenue
Ottawa, Ontario, Canada K1N 6N5

1. Introduction

In performing complex tasks, experienced practitioners use cognitive procedures that have much in common with each other. We term these procedures *cognitive patterns.*

In this research, we will focus on the subset of cognitive patterns relevant to software engineering. We propose that the study of and dissemination of these patterns will help tool designers create more useful tools and directly assist developers in working more efficiently and effectively with software. More specifically, the patterns can be applied to help derive features of tools that aid in *understanding* software, whether for design, or some other type of problem solving. Such features will support such cognitive activities as reasoning about and thinking about software artefacts.

A cognitive pattern is a structured textual description of a solution to a recurring cognitive problem in a specific context. A general class of cognitive problems in software engineering is the understanding of the structure and function of an object. More specific problems include: determining the most important aspects of a class diagram; understanding how a specific change is going to affect the system; or coping with cognitive overload due to the amount of detail present in a model.

Cognitive patterns are “patterns”, and hence they are related to the design patterns well known in software engineering. But, whereas a design pattern captures an effective technique for solving a design problem, a cognitive pattern captures a technique that is partly or wholly mental and that is employed by practitioners when trying to perform a complex task. Our intent is to translate cognitive patterns into software features that will facilitate a user’s cognitive abilities: cognitive patterns are therefore more closely related to usability patterns or patterns of user-interface design. An understanding of cognitive patterns helps illuminate the relationship between user and tool.

All patterns ‘balance the forces’ present in the problem and the problem’s context: The designer who uses a design pattern will be interested in the balance between efficiency, reliability, maintainability etc. The person who understands a program will be interested in the balance between cognitive load, correctness of understanding, efficiency of problem solving, and other factors.

Cognitive patterns for software comprehension build on an extensive literature that describes high-level strategies for software comprehension. Well-known strategies include Bottom-up [15], Top-Down [2], Opportunistic [11], As-Needed [12], and Integrated Meta-Model [18]. Each of these may be described as a pattern. The many detailed techniques employed when using each of these can also constitute patterns. In addition, there are issues associated with switching between strategies – for instance, the disruption of the user’s mental model [5] – and solutions to this problem can also constitute patterns. Designers of tools such as SHriMP [16] implicitly recognize these patterns and support user needs through a variety of strategies that can be embodied in patterns.

As we construct our understanding of software, our understanding is affected by time. Mental models and their internal details change over time. In this paper, we describe a high-level pattern for software comprehension[1], entitled *Temporal Details*. The *Temporal Details* pattern illustrates the dynamics of time within the user's mind and helps explain why tools should support these manifest dynamics. Several other patterns contribute to the resolution of forces within the *Temporal Details* pattern. We also present these patterns as a pattern language within this paper.

1.1 Terminology

Before proceeding, we will describe the terminology used throughout this paper:

- **Model**: When discussing a representation of software will use the term "model" as opposed to "diagram". A diagram is a two-dimensional symbolic representation, of processes, features, etc. and employs lines and symbols. Diagrams are found in models, but a model is more: A model is an integrated representation, which has multiple diagrams, each acting as a view of some of the information in the model. The model as a whole will contain information from many dimensions including form, time, and rationale; not all of the information will appear on diagrams.
- **Version**: A recurring theme in this paper is the model that evolves over time. Models may evolve over time because they are developed over time, or because the artefact the model represents changes over time, and therefore the model correspondingly adjusts to match or capture this change. We call the state of a model after a group of changes a "version"; and we are interested in the versions of models under development as well as versions of models representing discrete software releases.
- **Prior** and **final** models: When considering models, we make a distinction between model versions as they existed in the past, which we call "prior models", and the model as it stands at the present moment, which we call the "final model", even though the model may evolve still further in the future.
- **Manipulate**: We use the term "manipulate" to refer to the investigation of design alternatives in models. In particular we consider it possible to manipulate prior versions, that is, the after-the-fact exploration of "what if" scenarios.
- **Transition**: We use the term "transition" to refer to changes between versions.
- **Meaning**: Finally, we use the term "meaning" to refer to the *rationale* for transitions.
- **System**: This paper presents patterns for teaching or understanding an existing complex system through the details of the system history, such as prior states or decisions. When we talk about "the system" we are in general referring to a software system being modelled.

1.2 Target audience and actors

The instantiation of the patterns in this paper will contribute to the understanding of artefacts through models. The patterns may seem intuitive and useful to any practitioner involved in software development (i.e. designers, architects, archaeologists), but our target audience is software developers involved in the software maintenance of large-scale legacy software, or tool developers who build tools to aid such maintenance. By using a tool based on these patterns, a software practitioner may be able to locate undocumented design decisions and, as a result, understand the system in its current form by understanding how the system evolved over time. A side effect of this benefit is less reliance on software archaeologists or software gurus.

We will reference three actors throughout the paper:

- The tool designer, that is, the designer of a modelling tool;
- The model builder, or modeller, who constructs the models; and

[1] A complete set of our cognitive patterns for software comprehension, including the *Baseline Landmark* pattern, may be found online at www.cognitivepatterns.org.

• The tool user, designated by ‘you’ or ‘the user’, who uses features based on the instantiation of the patterns when working with models.

1.3 Case studies woven through the paper

We developed the patterns we present in this paper through three research techniques: a study performed in an industrial setting, the cross-referencing of literature and other field studies in which we have participated. To derive the “known uses” in the patterns below, we studied field experts as they worked with several tools, including:

Grep: Grep is one of the basic tools used by software engineers to learn about source code. We observed software engineers making extensive use of grep as they tried to understand a system component. Some word relevant to the problem would come to mind, and they would issue a grep command to find all the lines of code in the system containing that word. They would then *store* that grep result. This process would be repeated many times, so that in the end they would have effectively built a *model* consisting of numerous search results in files. They would frequently refer to the stored results, perhaps searching inside one set of results to create another.

TkSee: TkSee [7] was developed to assist software engineers in program understanding activities. In some sense, TkSee is a ‘visual grep’ tool: it helps people build, manage and manipulate models consisting of search results.

IBM Rational Software Architect (RSA): RSA enables large-scale team development: many people with different perspectives may work within the same context and the same artefact base and build different views, which may then be synthesized or rationalized into a consistent whole. We examined numerous features, including browse-diagram (a temporary, non-editable diagram that provides a quick way to explore existing relationships among elements), CVS Annotate (a feature within a Configuration Management (CM) System[2]), and Compare-Merge (another CM feature for teams to compare and merge software models).

Whiteboard Think-aloud Study with Mitel and IBM: We asked software engineers explicitly to explain the architecture of a system they were developing. Many of the patterns we describe in this paper have been developed based on these studies. We were particularly interested in the sequence of mental states that prompted participants to describe complex software architectures in a particular way.

Chess system: We developed a tool [17] to allow chess grandmasters to analyse various chess scenarios (games or game fragments known as ‘variations’). Grandmasters analyse chess variations while they are playing (and therefore, entirely in their minds), but we are primarily concerned with the physical consequence of their analysis using tools *after* their games. They may perform analysis for a variety of reasons: to improve their play, to find ‘chess truth’, or for publication purposes. We use this as an example of how many of the patterns apply in a domain other than software comprehension.

2. Temporal Details

In this section we first describe the *Temporal Details* high-level pattern, and then discuss several other patterns that relate to *Temporal Details*.

Figure 1 is a pattern schematic that illustrates the related patterns. The patterns shown in Figure 1 combine to resolve the forces of *Temporal Details*. The circles represent the *Snapshot* pattern (important moments in time). *Snapshots* contain *Meaning*. The boxes represent groupings of *Snapshots.* The arrows represent transitions between *Snapshots*.

[2] RSA supports Rational ClearCase and CVS; however, in our study, we evaluated CVS.

There are several kinds of groupings of *Snapshots*, each shown as a box. This figure shows: a *Longview* (sequential *Snapshots* that tell a story), *Multiple Approaches* (parallel *Snapshots*), a *Snapshot* chosen from a set of *Quick Starts* (the evolutionary origin), and finally, a sample of the ability to *Manipulate History* (which illustrates how we manage evolutionary complexity).

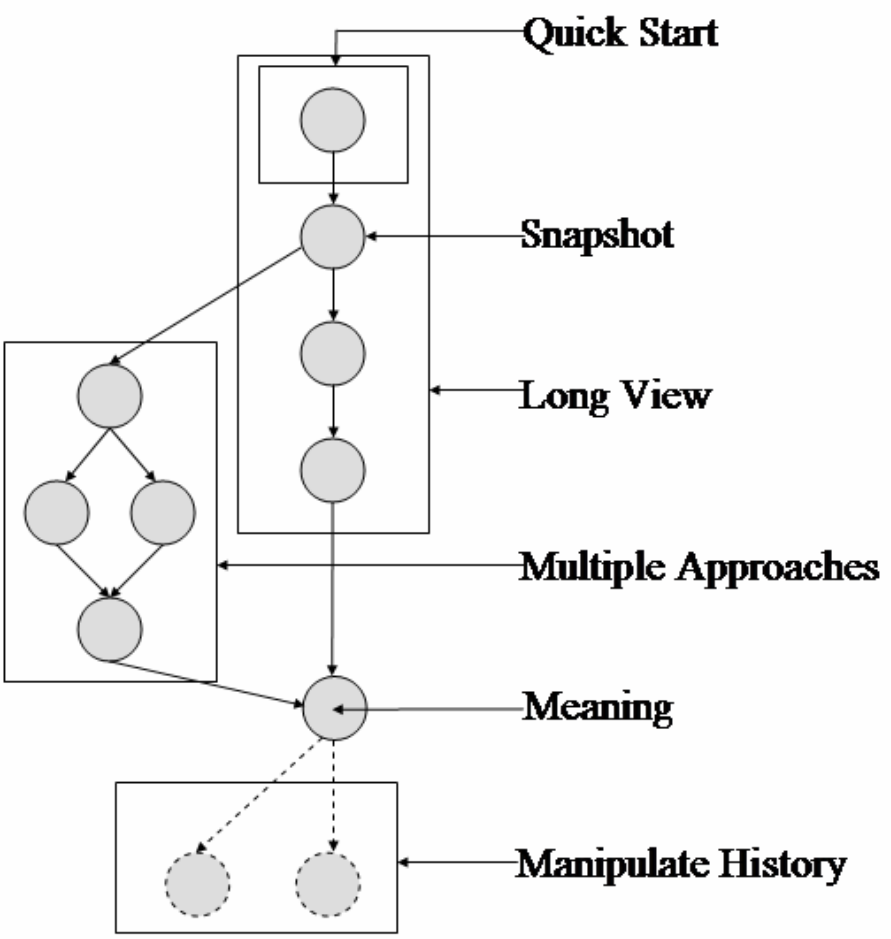

Figure 1: *Temporal Details* Schematic

Temporal Details

Context: **You are dealing with models that evolve over a period of time**. You may be using a tool to explore, reverse engineer, document or explain a large-scale system (e.g. software). Alternatively, you may be doing these activities by drawing diagrams informally on a whiteboard. The models can be diagrammatic or textual.

Problem: After editing a model, you expect that the final version will maximize understanding of some aspect of the system. However **the final model lacks intrinsic details that enable certain kinds of understanding**.

Forces:

- **A final model may be difficult to understand due to its size and complexity**. To understand the way a system works, it may be better to look back to a time when life was simpler. For

example, attempts to make sense of the physical world using our current understanding of the laws of relativity are very difficult. If instead you base your initial understanding on Newton's laws, you will learn much faster, even though Newton's laws are not entirely correct for large speeds and masses. Newton's laws provide a simpler *view* of reality.

- **To understand a system you may need to understand it incrementally.**
- **To work with a system concretely you need the final model,** since earlier models will be incomplete and inaccurate. Earlier models have inaccuracies or gaps preventing you from being able to properly understand, develop or find the flaws in the system.
- **The final model contains many levels of detail but abstracts away key historical decisions and their rationale**. A tool supporting a block diagram of a CPU allows you to choose your desired level of detail by drilling down and backing out of different levels, such as sub-blocks, circuits, and transistors; this type of detail we call 'drill-down details'. Drill-down details are needed to tell you how a system works and allow you to use a system concretely, but **they do not help you understand *why* the system is the way it is today**. Drill-down details do not support an understanding of the earlier decisions that constrain the current system.
- We rarely have the luxury of building a system afresh; therefore, **we need to understand and cope with the present constraints**, which are there due to historical decisions and prior constraints. If we built the system afresh today, the constraints may be different, leading to a different design.
- A prior model of a system may be unsound, or incorrect by the standard of today's system, but **understanding prior models is needed to make effective decisions and to avoid making recurring mistakes**.
- **The historical information may be too rich and complex and thus confound the person working with the model**.

Solution: **Support the ability for tool users to view and manipulate aspects of history and the decision-making process that went into the development of the final model**. The user will then be able to more easily understand why certain decisions were made early in the model's development, and will be able to understand the present system as an extension of the simpler system that once existed. The user will always have a choice: to simply view the final form of the model, or to explore the dynamic details of the model's creation. Such details could include the different versions of the model over time, key decisions made and their rationale, and thought processes applied as the model was evolving. Ideally, a tool that supports such details would be transparent, so that users who do not need to look back at this history should never have to contend such a feature. Those users who need historical information can view whatever levels of history they choose.

Known uses: There are a variety of software engineering tools designed to capture design rationale [8, 10, 14]; these originated in the user-interface design community and enable one to store information such as the decision tree leading to the final model, as well as the logic of each decision. Configuration management tools explicitly store states of a system at certain discrete points, and encourage annotation of the changes made each time a version of the system is saved. As an example, corporate meeting rooms often have equipment to create a digital image of the contents of a whiteboard. Also, learning the historical refactorings helps a developer understand the present state of a system. The *Temporal Details* pattern describes how one can capture and illustrate rationale and history in a broader sense than any one of the above tools.

Resulting Context: The application of *Temporal Details* and its related patterns will result in **tools for explaining, exploring and documenting systems** that take advantage of the knowledge embedded in a model's history and **that mesh more closely with the way users think and act**. Full application of *Temporal Details* would result, for example in the ability to *Manipulate History*.

Important downsides of the pattern are that the environment must be built and the historical data must be managed on an ongoing basis. In other words, a tool for working with *Temporal Details* must leave a footprint that is greater than would otherwise be left by a design tool.

Issues that arise when applying *Temporal Details* include deciding which historical information to capture, how to represent it, and how to manage details that accumulate over time. Some of these issues are addressed through the modeling of software evolution by treating history as a first class entity [4]. If these issues are managed well, each user should be able to choose the level of understanding that is appropriate to that user. At the root of the resolution of these issues is the following question: What are the key historical moments, the moments which exemplify key insight? The challenge of this question is addressed in the *Snapshot* pattern.

Snapshot

Also Known As: Short View, Temporal State, Coherent Point, Conceptual Whole, Cohesive Nugget

Context: You are **working with a single evolving model** in the context of *Temporal Details*. Any model is modified by a series of operations, such as adding or removing model elements. The elements could be as small as single lines or characters, or could be larger compositions.

Problem: **With what granularity should you track the evolution** of the model? That is, what are the most useful points to stop and think about the model as it is being built, or look back later to see how the evolution occurred?

Forces:

- **A concept**, even in the context of a complex model, **can often be conveyed simply**.
- **Small but self-contained changes to a model may convey important new meaning**.
- Building a large, complex model before stopping to think about it is likely to result in confusion, and skipping important learning.
- Adding, deleting or replacing information in a model can convey new understanding.
- Tracking evolution with an excessively coarse granularity fails because **humans need smaller increments to evolve their understanding**, and you lose historical perspective.
- **If every gesture** involved in evolving a model **is tracked**, then we are guaranteed that **no historical information is lost**. But **to step through each possible state** of the model **would be slow,** and we would lack guidance about which points are salient.
- **Some states** (e.g. after a single line is drawn) are incoherent or **represent incomplete concepts.** This level of granularity is too fine.
- **When humans try to learn in large numbers of very fine increments, they may not understand the big picture and retain the details.**

- **A model does not have to be complete and accurate before one can pause to think. N**either completeness nor accuracy is needed for incremental understanding; striving for these will be important for some purposes, but not for the gain of understanding.

Solution: **Track and capture evolution at moments when the model is cohesive or conceptually whole**. We call these moments *Snapshots* our research shows that *Snapshots* are naturally present in the development of a model.

By cohesive and conceptually whole, we mean that a concept being conveyed in the model has had sufficient detail added so a person studying the model can understand the concept, but not necessarily perfectly. The granularity of tracking will therefore depend on the strategies used by the person doing the modelling (the *modeller*): Some modellers might add seemingly unrelated elements to a model, and only after considerable time, link them in a way such that the model becomes cohesive. In this case, the *Snapshots* will be far apart in time. Others modellers might add very small increments such that the model is highly cohesive after each increment. In either case, being able to view the model as it existed at *Snapshots* will group potentially large sets of model states into more manageable units.

The *Snapshot* is a coherent step in the process of evolving a model towards its final form. The *Snapshot*, irrespective of the underlying meaning of the model it conveys, will reveal nascent information either independently or in conjunction with other *Snapshots*. It is not generally worth spending a lot of time designating *Snapshots* with a very high level of accuracy; a rough approximation of the set of *Snapshots* will often be sufficient to achieve the objectives of this pattern.

In a tool, a *Snapshot* might be created manually through user actions including 'tagging', saving, annotation, etc. A *Snapshot* might also be identifiable automatically: A modelling tool might recognize *pauses*; i.e. the tool might automatically tag a version as a *Snapshot* when the modeller has made a collection of changes, and then pauses before making more changes. If it is a graphical modelling tool, the tool might recognize the completion of certain diagrammatic 'idioms' (e.g. drawing two boxes with a line between then and labelling the boxes), or even capture the oral explanation made about a diagram. We have used these approaches when manually determining the *Snapshots* in a model's history.

Resulting Context: The application of the *Snapshot* pattern to a tool will enable users to build and present a model in appropriately sized increments, and allow the user to reference and come back to some of those model versions if needed.

How to identify a *Snapshot* is both an empirical and implementation challenge; what constitutes a conceptually whole moment will be subjective. A tool for working with *Snapshots* will need to allow for this subjectivity and imprecision. Also, the *Snapshot* pattern does not suggest how to organize the entire set of *Snapshots* over time, nor does it suggest how the *Snapshots* ought to be presented. We need *Long Views* and *Multiple Approaches* to organize *Snapshots*, and we need a way to *Manipulate History* for further organization and presentation. When we want to build or recognize the first *Snapshot* of a new model we may need to choose a starting point from a set of *Quick Starts*.

Another problem with *Snapshots* is that although state alone may convey some meaning, the rationale or the decisions made to arrive at a *Snapshot* are not always conveyed in the model – further explicit *Meaning* associated with *Snapshots* may be required. Further to this point, a *Snapshot* does not give you the whole picture; you only get the picture at a moment of time.

Related patterns: The *Speculate about Design* pattern [3] calls for a software engineer engaged in software reengineering to refine their model of a system by checking hypotheses about a design against the source code. In this case, a *Snapshot* can be constructed for each hypothesis. To *Speculate about Design*, an engineer inserts open questions as notes into a software model, then iteratively addresses each question and refines the model accordingly. The reengineering process builds a series of *Snapshots*.

The *Migrate Systems Incrementally* pattern [3] encourages developers to avoid the complexity and risk of "big-bang reengineering" by deploying incremental updates of the system. In this case, each update can be considered a *Snapshot*.

The *Just Enough* pattern [13] tries to ease learners into the more difficult concepts of a new idea by the provision of a brief introduction and dissemination of more information available when the learner is ready for it. In other words, *Just Enough* describes the division of information into coherent units and a way of delivering information from the learner's point of view. *Snapshots* similarly tackle the "right" amount of information and the user's understanding. However, a *Snapshot* is a state of an ongoing process leading towards creating a final model: The emphasis in a *Snapshot* is keeping a point in the history of the model in case it may be useful, whereas the emphasis of Just Enough is active design of an increment in a learning medium. Also, the *Meaning* behind the *Snapshot* is considered separately.

The *Step by Step* pattern [13] encourages people to tackle problems in small increments "with short-term goals, while keeping your long-term vision". Incrementality is therefore a common feature of both *Step by Step* and *Snapshot*; however, in *Snapshot* the idea is to review prior increments, rather than to work forward in increments.

Known Uses: Many tools offer an ability to examine a specific model at a given moment in time, in other words to take a *Snapshot*. This *Snapshot* is not always what the user is looking for in terms of the particular details, but aids understanding.

Grep: Grep produces a model of some aspect of the static state of a system based on a specific set of queries – the effectiveness of the results is directly proportional to the effectiveness of the queries, but rarely will an individual query generate everything the designer needs to solve a specific problem, and never will a grep query represent everything in a design. Nevertheless, individual grep results can be extremely useful *Snapshots* in the user's evolving understanding of a problem. In our studies we observed software engineers print out grep results, store them in files and use them as checklists.

TkSee: TkSee was specifically designed to enable people to explore software and incrementally build models. The models are 'history states' that show certain patterns of relationships that bear on the current problem. As with grep results, software engineers discard these after a short period of time (several hours to several days).

RSA: RSA supports *Snapshots* with the Compare-Merge feature. A *Snapshot* in this context refers to the individual differences between two versions of a software system. If one of the versions is the present system, and the other version is a point in local history, then each *Snapshot* represents an evolutionary development. Figure 2 illustrates a tree structure for navigating the differences, and Figure 3 illustrates the differences visually (e.g. in this particular case, a circle shows the dependency between Class1 and Interface1 has been removed). In addition, CVS versions are *Snapshots* of an entire system in a moment of time, although often with a granularity that is much higher than what we envision for a temporal details tool. RSA also supports individual

browse-diagrams for analysis, these can be seen as *Snapshots* supporting partial visualization of a system in a moment of time.

Studies at Mitel & IBM: In many instances during our videotaped analysis, the participants would produce diagrams on the whiteboard and then begin discussing them. They would not speak until they had made enough changes so the diagram was in a new coherent state: they were thus building *Snapshots*. Based on the questions asked, the participants would then iteratively produce refined *Snapshots* of the system.

Chess system: In chess, a *Snapshot* corresponds to a particular board position. In the mind of the grandmaster, he is examining not only the *Snapshots* that exist on the board, but also various interrelated moves that may occur (*Snapshots* from variations) [9]. A tool to support analysis must support not just the main board positions, but also variations.

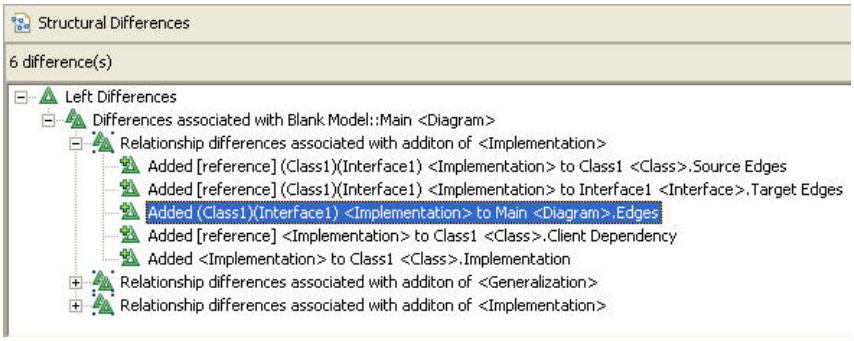

Figure 2: Tree-based navigation of Snapshots in RSA

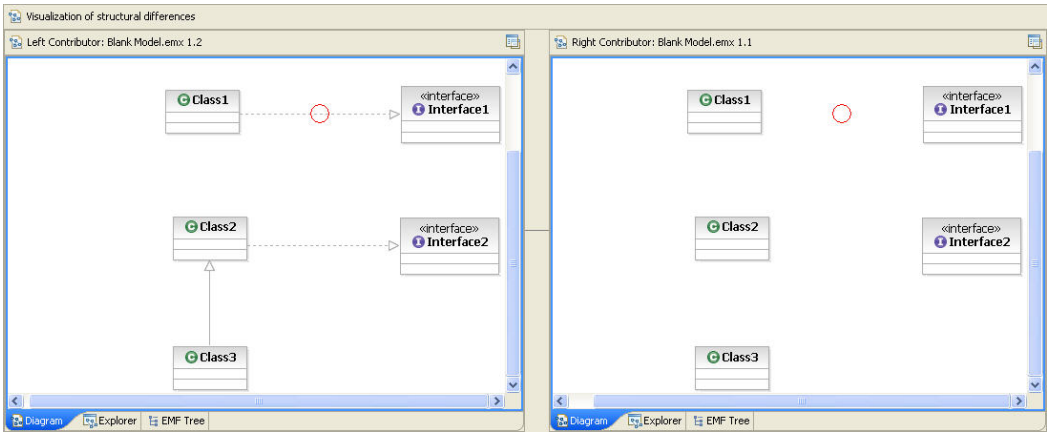

Figure 3: Visual illustration of *Snapshots* in RSA

Quick Start

Also Known As: Starting Templates, Library of Openings

Context: You are creating a model in the context of *Temporal Details*. This might mean you are starting from scratch, or you might have already evolved a model through many *Snapshots*.

Problem: **How do you enable the user to effectively and efficiently create new models?**

Forces:

- **A model will always have at least one diagram** (and hence one starting point), **but will often have many**.
- The start of any task is often undertaken instinctively. However, **people often have difficulty when they start a task**. On the one hand, they may need a catalyst, and on the other hand, they may strive for an immediate, though unobtainable, perfection.
- People may start with anything at all, just to get themselves going. Consider, for example, how the US Marines start with something called a "70% solution" [6] that encourages "high tempo". The idea is that it is better to decide quickly on an imperfect plan than to deliver a perfect plan too late. The Marines find the essence of a complex situation and build upon it quickly. Professional writers also use this technique.
- **People need quick ways to plunge into a new task**, such as modeling, that are not inhibited by start-up costs. These costs might be the need to determine where to save something, the need to make links to existing models, the need to construct a well-known canonical starting point.
- **If people do not have a familiar place from which to start, understanding is inhibited**. But then, if they start with something too large or irrelevant, significant time may be wasted adapting it. People need a familiar or central starting place of an appropriate level of complexity when they begin a process of understanding.
- **An explanation is often best when it contains many interrelated models**, each of which has to be created, and therefore must be started at some point in time.

Solution: **Allow users to quickly start new models by choosing from a small set of existing simple models**. The selection of starting points is more appropriate if they represent a familiar landscape to the user. Each starting point is a *Snapshot* of a model that will likely then evolve through many more *Snapshots*– this first *Snapshot* is neither trivially small, nor is it too complex. As with all *Snapshots* it is a view that is coherent enough to be talked about.

In a tool, this pattern can be implemented using templates of sophisticated, but still very simple designs. For example, it might be useful to start a state diagram with two states linked by a transition: not many useful state diagrams would ever have fewer states than that.

Resulting Context: This pattern encourages speed of exploration or explanation, as well as the creation of multiple models. After you have chosen one of the *Quick Starts*, one of the hardest decisions – where to start – is behind you, at least until you choose to start afresh. A *Quick Start* provides a guide as to where to go next, and how to keep going. As you build a series of new *Snapshots* based on your *Quick Start*, you may need to consider if you intend to build in sequence (*Longview*) or in parallel (*Multiple Approaches*).

In creating a list of suitable *Quick Starts*, you must determine which ones will be appropriate. A downside of starting with simple and imperfect starting points is that you may also need to start again several times. However, this may support incremental improvement of your understanding of a complex system. The real danger comes from the reliance on your set of starting points as definitive, that is, a reluctance to explore other possibilities.

Forcing people to choose a particular starting point would be contrary to the pattern. Users often find they stick exclusively to the templates provided or spend too much time exploring the template set.

Known Uses:

Studies at Mitel & IBM: Our participants start with a single notion, usually a *Baseline Landmark*, and incrementally expand from this starting space. In the telecommunications domain, for example, it is very common to draw a diagram of 'plain old telephone system' (POTS) and explain some new feature by evolving this diagram.

RSA: A software architect may either create new model elements or access existing assets (e.g. requirements code, other models) to build a model. An architect may use search and navigate features to access existing features (*Baseline Landmark).*

Chess system: In chess, grandmasters use "critical positions" to study opening theory. The critical position may be a hotly contested position – perhaps many other professionals reach this position in their play often, or the position is deemed "OK" since a first-class grandmaster played it recently. The grandmaster may prepare for a future opponent by analysing *Snapshots* from the opponent's opening repertoire. Or the grandmaster may wish to broaden his repertoire. In all cases, the grandmaster starts analysis from a historical and relevant position.

Other tools: Word processors and presentation software provide libraries of templates to allow *Quick Starts.*

Long View

Also Known As: Highlight the Story, Higher-Order Snapshot

Context: You are evolving a model through a set of *Snapshots*.

Problem: **How can you tell an effective story** when the complete set of *Snapshots* comprises a rich set of details?

Forces:

- **People appreciate a story** – a tale that evolves over time where linkages are made from step to step.
- **If there is no connection** between historical steps, **then there is no story**.
- An individual *Snapshot* conveys some **concepts**, whereas others **emerge only through a sequence of *Snapshots***.
- **One can lose sight of the forest** (a story) **for the trees** (the individual *Snapshots*).
- **If all historical steps are connected, it is hard to see the key steps**.
- **People have difficulty comprehending a 'big bang' explanation**.
- Explaining or exploring based purely on a series of unrelated *Snapshots* fails because the **sequencing of *Snapshots* helps incrementally build understanding**.

Solution: **Allow users to look at the history of model evolution and designate a coherent sequence of *Snapshots* to have particular significance**. The sequence tells a story that would otherwise be hidden. The end-points of the sequence become, in some sense, higher-order *Snapshots*. A *Long View* is a view that shows how something evolved. How something evolved from one *Snapshot* to another is a story. Sometimes you cannot grasp a concept unless you have more of the small units. Sometimes you cannot grasp one of the small units unless you grasp another one. By seeing the individual *Snapshots* in context you may be able to understand a larger component of the entire system or understand some otherwise unintelligible *Snapshot.*

Resulting Context: You may start a new *Long View* to explore a new aspect of the system; you may also start a new *Long View* if another was less fruitful than expected or resulted in only a partial understanding. A tool that implements the *Long View* pattern would allow the explanation and exploration of the history of a model through sequences, rather than just a simple presentation of *Snapshots* without any organization. Someone understanding a design would be able to understand more about the thought processes of designers. However, just as all *Snapshots* need not be retained, the user needs to remain flexible as to which sequences will be stored for later reference – and needs to remain in control of those choices. Tools to help guide the user through a series of *Snapshot*s may also use "relevance feedback". You may still need a mechanism to *Manipulate History* to clean up and organize *Long Views,* as well as to add *Meaning* to them.

Known Uses:

Grep: We have observed software engineers doing two things to evolve and group their query results: one is to edit their previous grep command lines; the other is to pipe one set of grep results through another grep query. The notion of revisiting previous queries is poorly supported by tools; with native UNIX environments the user does however have access to buffered history of terminal output, and the history of commands. Users can also save queries to text files that can be concatenated, or run through other grep filtering steps. Taken together, these facilities allow a rudimentary ability to create a story or *Long View* from query results.

TkSee: TkSee contains multiple indented trees of queries. The user can easily refer to previous queries, and group several of them together as a *Long View*. Users can also save query result sets to files and bring them back.

RSA: The Compare-Merge feature supports a series of *Snapshots*, which collectively form a *Long View*. If the software architect clicks on consecutive items in the tree shown in Figure 2, RSA illustrates the *Long View* visually, as in Figure 3. RSA compiles a list of versions in the "CVS Resource History" which illustrates the *Long View* as an evolution of a system across versions.

Studies at Mitel & IBM: In our whiteboard video sessions, the participants would iteratively refine diagrams representing their knowledge of a software system. Because of limited whiteboard space, they would erase (cull) less important aspects of the model to provide more relevant artefacts to address the questions. History was difficult; to refer to previous versions they would use verbal comments or redraw previous diagram components.

Chess system: While analysing a position, a grandmaster first compiles a list of candidate moves, the *Quick Starts* for branches of forced (through a series of checks or threats) and unforced variations. The grandmaster subsequently steps through each candidate move in turn, building a *Long View* for each candidate move. The *Long View* is a series of moves that ends in an evaluation (white is winning, the position is equal, the position is unclear, etc.). For many positions within the variations, the grandmaster must compile a new set of candidate moves; thus, the grandmaster is building a tree of variations – the topic of the *Multiple Approaches* pattern.

Multiple Approaches

Context: You will be modeling or understanding using a set of *Snapshots*.

Problem: **How do you support the representation of non-linear patterns of evolution of a model**?

Forces:

- **People** may not fully understand one explanation, and **may need an alternative perspective.**
- **Different people may need to approach understanding using different strategies.**
- **There may be different** but perfectly **valid ways to model something** or solve a problem.
- Allowing only a single path fails because it does not recognize the alternative perspectives, or else forces the perspectives to be considered in a less useful order.

Solution: **Allow branching and re-joining in the network of connected *Snapshots***. After a branch point, the *Snapshots* or sequences of *Snapshots* (*Long Views*) in either branch can be used for different purposes. A user may use different sequences to explain or explore the various aspects of a system. Alternatively, the user may use multiple sequences to approach the same aspect in different ways, either developing an alternative understanding, or a more complete understanding. Sequences may split and merge at arbitrary points.

Resulting Context: A user will be able to designate and explore multiple paths for understanding and exploration and the user should see the path structure so he or she can compare paths and learn from different perspectives. One obvious downside to *Multiple Approaches* is the generation of a complex network of model versions. The user may need to *Manipulate History* to cope with the large amount of information and to filter important details.

Related Patterns: The *Multiple Approaches* pattern allows a person in an organization to make a compelling argument from different viewpoints as to how an idea may meet the *Tailor Made* [13] needs of an organization and the *Personal Touch* [13] that people require to see the personal value that an idea may bring them.

The *Multiple Approaches* pattern depends upon branching. A group of patterns which addresses branching from the perspective of software configuration management (SCM) is "*Streamed Lines*: Branching Patterns for Parallel Software Development" [1]. The SCM patterns describe how to support parallel development through project management, organizational structures and technology. The implementation of the SCM patterns help address problems of communication, visibility, project planning, and risk management and resolve some of the technical challenges associated with the capture and organization of *Snapshot* networks.

Known Uses:

TkSee: We studied users and evolved TkSee with features to build and explore hierarchies of exploration paths incrementally: each branch is a separate approach, throw away exploration paths or sub trees, save exploration trees to a file and reload them and, switch among trees. In fact, TkSee supports a hierarchy of exploration hierarchies.

RSA: A software architect may analyse *Multiple Approaches* with browse-diagrams (system *Snapshots)* with the intent of building deeper understanding through multiple perspectives. RSA is not limited to browse diagrams – a software architect may create many typical UML diagrams (*Snapshots*), and the tool allows the architect to link them all. Thus the tool supports multiple branches and joins between *Snapshots*. In addition, RSA supports *Multiple Approaches* through CVS features such as branching (that is, you retain the baseline while you work on different versions).

Studies at Mitel & IBM: In our whiteboard sessions, the participants digressed into discussion of seemingly unrelated parts of the system, and later conjoined concepts to deepen understanding.

Chess system: As has been discussed, during analysis grandmasters built a tree of variations for candidate moves, or *Snapshots.* The right pane in Figure 4 illustrates support for *Multiple Approaches* through a visual hierarchy of clickable moves. When a grandmaster clicks on a move, the system illustrates the *Snapshot* in the left pane.

Other examples: Multiple product lines, multiple configurations, achieving the same result by applying different methods.

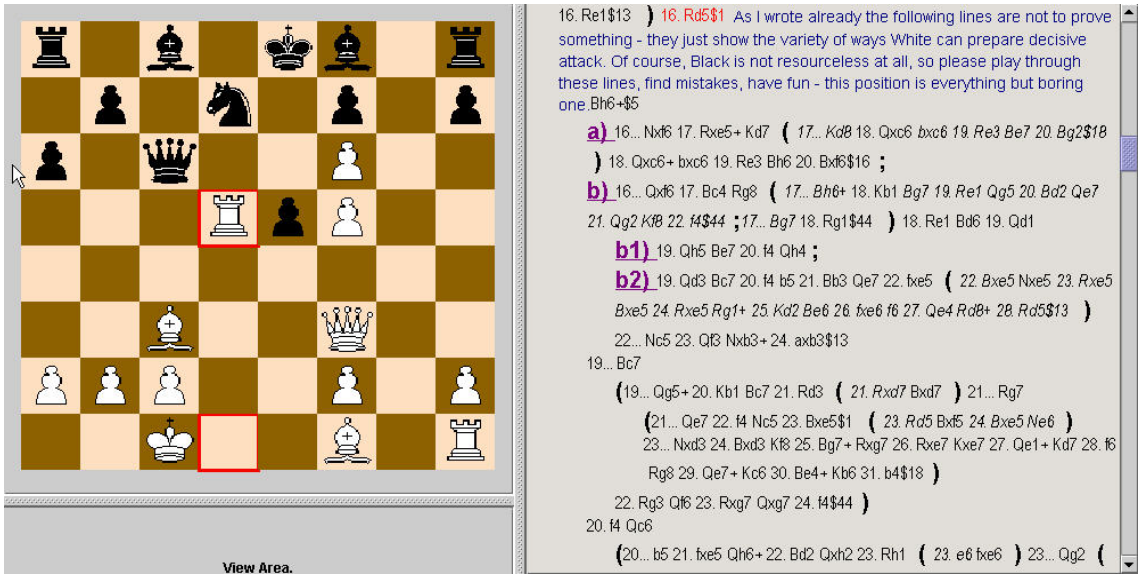

Figure 4: ChessLink Analysis Feature, *Multiple Approaches* as chess variations

Manipulate History

Also Known As: Superman rewinds time to save Lois Lane

Context: You have a historical record of model evolution in the context of *Temporal Details*

Problem: **The network of *Snapshots* may not be good enough for users to learn from**.

Forces:

- A, then B is the way things happened, but B then A may make a more comprehensible story.
- **History is in the eye of the historian and the reader of a history**: No two historians will tell a story the same way, and no historian will ever know exactly what happened.
- Historical fiction can help one understand history by making it more comprehensible. No harm is done as long as one realizes that some fictionalization is involved.

Solution: **Allow for the network itself to be rearranged and adjusted** so you can revisit your understanding process by following previously followed paths. But as you do this, retain the previous network. The result can then become a network of networks.

Resulting Context: You will be able to edit not just the models, but also the networks of models. This pattern builds on Snapshot, Long View and Multiple Approaches: Those patterns allow you to

designate points, sequences and branches in the history of a model's evolution. Manipulate History allows you to adjust that history itself.

Known Uses:

TkSee: Supports the manipulation of multiple hierarchies of historical queries. For example: *Snapshots* are the results of queries or other operations on the model; *Long Views* are the sequences of queries that form a hierarchy whose results can be saved as an exploration and revisited later; these explorations can themselves be edited to refine the user's understanding, and saved again.

RSA: Using the Compare-Merge feature, developers can retrace the meaning of decisions by interpreting iterative changes. Furthermore, UML notes, comments and documentation attributes may provide insight into the decision process.

Chess system: After a chess game, grandmasters investigate what-if scenarios for the critical moments of games to unlock the 'secrets of the position'. The primary goal of this is to improve their thinking, though they may also uncover improvements in their games that they can use in later games. The entire basis of modern chess opening theory is continual reflection on revision of historical games [9].

Meaning

Also Known As: Annotation, Metadata

Context: You are working with a network of *Snapshots*.

Problem: **A model cannot inform you why the decision to transition from *Snapshot* to *Snapshot* was made, yet you often need such deeper understanding**

Forces:

- **A deeper understanding of history can be derived if you know why something** happened, not just what happened. The "why" is, however, often lost in the mist of time.
- A *Snapshot* only captures state and will often not even imply the rationale for the state.
- Tying rationale for a *Snapshot* to the *Snapshot* itself may be inappropriate as it is may be difference between two *Snapshots* that is of most interest – and one may later on want to *Manipulate History*, which would seriously confuse rationale tied to a single *Snapshot*.
- Rationale attached to a *Quick Start* may facilitate its use.
- The need for annotation is proportional to the size of a network of historical models.
- People often want to make use of information about information (meta-information) that may be valuable.

Solution: **Allow annotation or other mechanisms for recording knowledge about any of the *Snapshots* or transitions between *Snapshots***. Annotation features in tools may be one step towards retaining *Meaning*. Clearly developers need to indicate significant information that cannot be represented in diagram form. Perhaps notations need to be designed to represent this kind of information; or at the very least, structured documentation formats could be developed to capture this information.

Resulting Context: Following use of the *Meaning* pattern, the tool designer can build tools to allow annotations of all types. For example a tool that stores several states of an evolving explanation could allow the user to record why new details are added or replaced to create a new *Snapshot*. Design rationale is an important area of study in software engineering. Tools should allow the user to flow more easily from one design task to another while storing design decisions. Downsides to this pattern occur because people often disdain documentation. More annotation implies three more work tasks, one task is the annotation step, the second is the maintenance of previous annotations, and the third is reading annotations. Transparency is a very important aspect of this feature: do not enforce any of the three prior works tasks, but support them to an appropriate degree. Simply marking *Snapshot* moments may be sufficient *Meaning.*

Known Uses:

RSA: The developer may commit changes to CVS with comments annotating their rationale (illustrated in Figure 5). Furthermore, a developer may use the "CVS Annotate" feature (illustrated in Figure 6) to associate changes and comments with a particular version. In addition, RSA has a traceability feature supporting traceable design decisions across artefacts.

Studies at Mitel & IBM: The participants described their rationale for drawing. In other words, they described why they were about to start explaining something; why they were erasing something or why they were adding new details.

Chess system: Grandmasters analyse their own games to determine the "truth" behind the decisions they made over the board. They record these analyses in both variations and textual annotations, particularly when the games are to be published. The right pane in Figure 4 illustrates a sample annotation.

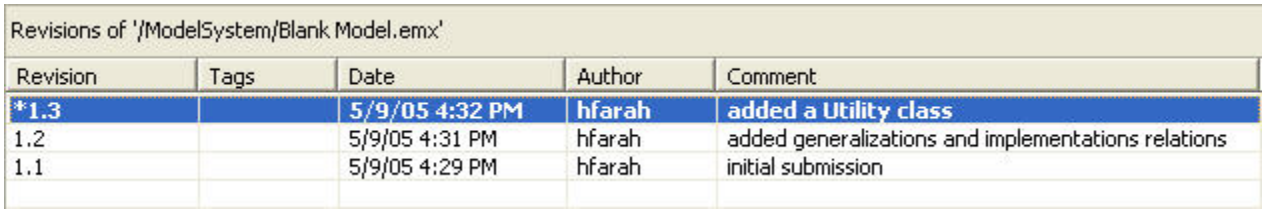

Figure 5: CVS commenting in RSA, one form of *Meaning*

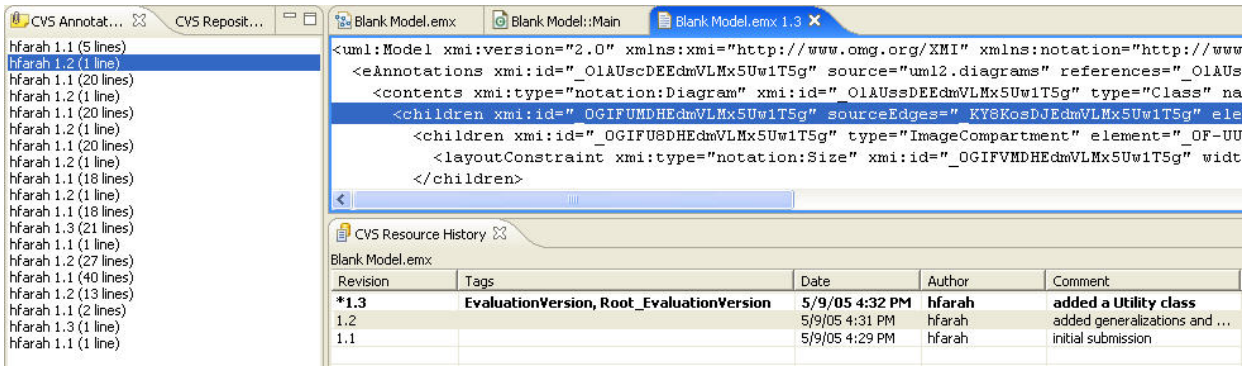

Figure 6: CVS Annotate feature for visualizing changes and comments with versions

To sum up then: we can leverage the patterns now expressed to gain a deeper realisation of their benefits in forthcoming tools.

Acknowledgements

Our warm thanks to our shepherd, Linda Rising. We learned much about patterns through your tutelage, Linda. Thanks to Richard Gabriel and the participants of our writer's workshop – your feedback was invaluable. Thanks also to Dwight Deugo for his early review of this paper and introduction to pattern philosophy. Thanks to IBM for their financial and resource support in this research. And thanks to the industrial participants in our empirical studies, without whom we would not have discovered *Temporal Details*.

References

[1] Appleton, B., Berczuk, S., Cabrera, R., and Orenstein, R. Streamed Lines: Branching Patterns for Parallel Software Development. *Proc.* Pattern Languages of Programs (PLoP '98), Monticello, Illinois, USA, 1998.

[2] Brooks, R. Towards a theory of the comprehension of computer programs. *Int. J. of Man-Machine Studies*, vol. 18:543-554, 1983.

[3] Demeyer, S., Ducasse, S., and Nierstrasz, O. *Object-Oriented Reengineering Patterns*: Morgan Kaufmann and DPunkt, 2002.

[4] Ducasse, S., Girba, T., and J.-M., F. Modeling Software Evolution by Treating History as a First Class Entity. *Proc.* Workshop on Software Evolution Through Transformation (SETra 2004), pp. 71-82, 2004.

[5] Eades, P., Lai, W., Misue, K., and Sugiyama, K. Preserving the mental map of a diagram. *Proc.* Compugraphics 91, pp. 24-33, Springer LNCS/AI, 1991.

[6] Freedman, D. *Corps Business: The 30 Management Principles of the U.S. Marines*. London: Collins, 2001.

[7] Herrera, F. *A Usability Study of the "TkSee" Software Exploration Tool*. M.Sc., Computer Science, University of Ottawa, Ottawa, 1999.

[8] Jarczyk, A., Loffler, P., and Shipman, F. Design Rationale for Software Engineering: A Survey. *Proc.* Int. Conf. on System Sciences, Hawaii, USA, pp. 577-586, IEEE Computer Society Press, 1992.

[9] Kotov, A. *Think Like a Grandmaster*. London: Batsford Chess Books, 1971.

[10] Kunz, W. and Rittel, H. Issues as Elements of Information Systems. Working Paper No 131, University of California, Berkeley 1970.

[11] Letovsky, S. Cognitive processes in program comprehension. *Proc.* Empirical Studies of Programmers: Papers presented at the First Workshop (ESP'86), pp. 58-79, 1986.

[12] Littman, D., Pinto, J., Letovsky, S., and Soloway, E. Mental models and software maintenance. *Proc.* Empirical Studies of Programmers: Papers presented at the First Workshop (ESP'86), pp. 80-98, 1986.

[13] Manns, M. L. and Rising, L. *Fearless Change: Patterns for Introducing New Ideas*. Boston, MA: Addison-Wesley Professional, 2004.

[14] McCall, R. J. PHI: A Conceptual Foundation for Design Hypermedia. *Design Studies*, vol. 12:30-41, 1991.

[15] Shneiderman, B. *Software Psychology: Human Factors in Computer and Information Systems*: Winthrop Publishers, Inc., 1980.

[16] Storey, M.-A. D., Fracchia, F. D., and Muller, H. A. Cognitive design elements to support the construction of a mental model during software exploration. *Software Systems (special issue on Program Comprehension)*, vol. 44:171-185, 1999.

[17] Thizy, D. and Murray, A. On the refactoring of a distributed chess analysis tool. University of Ottawa Computer Science Technical report TR-2002-12, 2002.

[18] von Mayrhauser, A. and Vans, A. Program comprehension during software maintenance and evolution, in *IEEE Computer*, pp. 44-55, 1995.